1. Introduction

Modern schemes for text compression use explicit models to help them predict what characters will come next (Bell *et al.*, 1990). The actual next characters are coded with respect to these predictions, resulting in compression of information. Indeed, one of the most important advances in the theory of data compression over the last decade is the insight, cogently expressed by Rissanen and Langdon (1981), that the process can be split into *modeling* and *coding*, the first assigning probabilities to symbols and the second translating these probabilities to a sequence of bits.

Text compression models are generally formed adaptively by the encoder as it works through the text. The model is transmitted implicitly, so that the decoder can build and maintain an identical one that is exactly in step with the encoder's. Although the models involved may be very large, the method of adaptive model formation avoids the need to transmit them explicitly. This results in a very effective text compression strategy that outperforms others both in theory (Cleary & Witten, 1983) and practice (Bell *et al.*, 1990). Adaptive modeling applies equally well to the *storage* of text—the bit-stream that would have been transmitted is simply saved instead and can be read and decoded later by an interpreter that builds its model dynamically from the text as decoding proceeds. However, this procedure does not permit random access to the text, for the model changes constantly and can only be reconstructed by reading the text from the beginning. Other modeling methods must be developed for full-text retrieval systems.

The technique that modern compression schemes use for the actual encoding operation is called "arithmetic coding" (Witten *et al.*, 1987). Like the older and better-known method of Huffman coding, this takes the next character and encodes it with respect to a probability distribution that is supplied by the model, producing a bit-stream that can be decoded correctly by a decoder with access to the same probability distribution. Unlike Huffman coding, it is provably optimal for arbitrary probability distributions (whereas Huffman coding is only optimal when the probabilities are integer powers of 1/2). Although the best schemes for text compression use adaptive models, arithmetic coding is suitable for use with any kind of model.

This paper explores the application of arithmetic coding to full-text retrieval systems. These involve the storage of a large body of text, along with a lexicon that lists the words it contains and a concordance that indicates the exact locations at which each word can be found. A typical query might seek all sentences that contain a particular word or combination of words. The random-access requirement means that adaptive modeling cannot be used. Of course, the fact that the whole text is available before compression means that a model could be formed in advance from the text itself and then used to compress it. However, it is uneconomic to store models created by standard adaptive modeling techniques because of their large size.

1.1 SUMMARY OF MODELS DEVELOPED

A number of different kinds of model have been developed for different parts of a full-text retrieval system and are presented and evaluated in this paper. They include

- a model that predicts the size of compressed text from its uncompressed size;
- a static model that uses words as the unit to be compressed;
- a model for compressing sorted lists that must be searched efficiently;
- a Bernouilli model of inter-word gaps.

The first model is used to encode pointers into the main text, and is described in Section 4. The number of words that intervene between successive pointers is known, and is used to predict the size of the corresponding compressed text and hence the actual difference between pointers. The second model, which is suitable for the main text, is described in Section 5. A key technique is the insertion of synchronization points into the text to allow it to be accessed randomly, and the overhead that this incurs is found to be negligible. The third model, which is suitable for storing the lexicon, is described in Section 6. It capitalizes upon the fact that the data is sorted, and also divides the information into fixed-length blocks to permit binary searching. The fourth model, which can be used to store concordance pointers efficiently, is described in Section 7. It is an analytical model of word distribution that predicts the difference between consecutive concordance pointers for a particular word. This is a rare instance of the successful use of an analytical model (rather than an observed frequency distribution) for arithmetic coding.

Section 2, which precedes these descriptions of modeling methods, contains a brief review of salient properties of arithmetic coding—the technique used to actually encode predictions from the models. Following that is an introduction to the requirements and architecture of full-text retrieval systems.

1.2 THE DESIGN SPACE

The intent of this paper is to explore design trade-offs, not to promote a particular system design. A number of new modeling techniques have been investigated, and in fact a system that incorporates them all has been implemented for evaluation purposes. However, implementation decisions will depend on properties of the application envisaged—size of text, retrieval mechanisms needed, response time requirements, processing power and storage capacity available.

We assume that the most important factors are the storage space required for the database, the number of disk accesses necessary for retrieval, and the scalability of the design to ever larger databases. A number of authors (e.g. Cichocki and Ziemer, 1988; Klein *et al.*, 1989) argue that advances in mass storage technology like CD-ROMs have certainly not eliminated the need for compression, and may actually increase it because of the desirability of accommodating huge databases on a single disk. Whether subsidiary data structures such as the lexicon should be (a) stored uncompressed, or (b) stored compressed but expanded when read into main memory, or (c) kept always compressed, is clearly a classic "store *vs* compute" trade-off that depends on the circumstances of the application. We describe techniques that favor compression over speed, on the assumption that system designers will be able to make any necessary compromises themselves. In the future faster processors will encourage greater compression.

2. Overview of arithmetic coding

A symbol that is expected to occur with probability p can be represented in no less than $-\log p$ bits[1] on average; this is Shannon's celebrated source coding theorem (1948). In this manner a symbol with a high probability is coded in few bits, while an unlikely one requires many bits. We can obtain the expected length of a code by averaging over all possible symbols, giving the formula

$$-\sum p_i \log p_i \; .$$

This value is called the *entropy* of the probability distribution, because it is a measure of the amount of order (or disorder) in the symbols.

The task of representing a symbol with probability p in approximately $-\log p$ bits is called *coding*. This is a narrow sense of the term—we will use "compression" to refer to the wider activity. An encoder is given a set of probabilities that represent the predicted distribution of the next character, and the next character itself. It produces a stream of bits from which the actual next character can be decoded, given the same predicted distribution that the encoder used. The probabilities may differ from one point in the text to the next.

2.1 HUFFMAN CODING

The best-known method of coding is Huffman's algorithm (1952), which is surveyed in detail by Lelewer and Hirschberg (1987). However, this technique is not suitable for our purposes because it must approximate $-\log p$ with an integer number of bits. This is particularly inappropriate when one symbol is highly probable (which is desirable, and is often the case with sophisticated models). The smallest code that Huffman's method can generate is one bit, yet we frequently wish to use less than this.

It is possible to overcome these problems by blocking symbols into large groups, making the error relatively small when distributed over the group. However, this introduces its own problems, since the alphabet is now considerably larger (it is the set of all possible blocks).

2.2 ARITHMETIC CODING

An approach that is conceptually simpler and much more attractive than blocking is a recent technique called *arithmetic coding*. Witten *et al.* (1987) presents a full description and evaluation, and includes a complete

[1]Throughout this paper the base of logarithms is 2 and the unit of information is bits.

implementation in the C language. The most important properties of arithmetic coding are:

- it is able to code a symbol with probability p in a number of bits arbitrarily close to $-\log p$;
- the symbol probabilities may be different at each step;
- it requires very little memory;
- it is very fast.

With arithmetic coding a symbol may add a fractional number of bits to the output. In practice, of course, the output has to be an integral number of bits; what happens is that several high-probability symbols together end up adding a single bit to the output. Symbols can be encoded with only a small number of fixed-point arithmetic operations.

One complication of arithmetic coding is that it works with a *cumulative* probability distribution, which means that some ordering should be placed on the symbols, and the cumulative probability associated with a symbol is the sum of the probabilities of all preceding symbols. Although many of the models developed in this paper store plain frequency counts, they include an efficient way of accumulating these to determine the cumulative distribution required by the arithmetic coder.

Since arithmetic coding effectively encodes each symbol into a fractional number of bits that corresponds to its entropy, some overhead is inevitable when terminating the compressed form of a message. In fact, this is never greater than two bits—in other words, a sequence of k symbols $s_1\ s_2 \ldots s_k$ that are encoded with probabilities $p_1\ p_2 \ldots p_k$ will occupy at most

$$-\sum_{i=1}^{k} \log p_i \ + 2$$

bits. If the coded stream is padded to round out a byte, the overhead increases by an average of half a byte to 6 bits. The small size of this overhead is crucial to the success of the methods we describe.

2.3 Computational requirements

Most implementations of arithmetic coding use three 16-bit registers internally. They perform arithmetic on 16-bit integers, and use 32-bit intermediate values. In such implementations the frequency counts are represented as 14 bits, which means that individual probabilities may not fall below 2^{-14} (see Witten *et al*, 1987, for further discussion). This is adequate for most, but not all, of the encodings required in our models.

An example of where it is inadequate occurs when coding the main text. As described more fully in Section 5, we code words individually based on their frequency. With more than 2^{14} different words, it is not possible to represent their cumulative frequencies as different 14-bit values, even approximately. Consequently it is necessary to use higher precision within the arithmetic coder.

Using a 32-bit arithmetic coder (which involves 32-bit operations and 64-bit intermediate values) means that counts can be represented as 30 bits. Then the maximum number of words that can be accommodated is 2^{30}, which is about 10^9. A 32-bit coder will accommodate texts that fit on a few 550 Mbyte CD-ROMs; for larger texts it will be necessary to use greater integer precision. Note that the use of a higher-precision arithmetic coding does not imply that coding is less efficient. Its only drawback is increased execution time for decoding (and also encoding, which is less important in the present context).

Our implementation uses 16-bit arithmetic coding, with 14-bit counts, wherever possible. The implementation is a mildly-optimized version of the C code described by Witten *et al.* (1987). (The nature of the optimizations are discussed in that paper.) In cases where higher precision is required, a 31-bit coder (not 32 for unimportant technical reasons) with 29-bit counts is used.

3. Full-text retrieval: requirements and architecture

Full-text retrieval systems divide the main text into lexical units such as book, chapter, paragraph, and sentence. They aim to provide efficient means of answering queries that involve retrieving all lexical units of a certain type (e.g. all sentences) containing

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

While the main text includes enough information to answer such queries without any auxiliary data structures, it is infeasible to scan it all and therefore a concordance must be used. Moreover, since mass storage media like CD-ROMs tend to have rather slow random access times, it is important to minimize the use of large data structures such as text and concordance. In the scheme we describe, the relevant lexical units are identified without ever reading the main text, and the concordance need only be accessed once for each word involved in the query.

3.1 THE SAMPLE TEXT

For testing, the present work uses a sample text that contains three versions of the Bible: the King James Version, the New International Version, and the Revised Standard Version. Together they comprise 198 books. Although this sample is very small in comparison to large-scale full-text retrieval systems, it is nevertheless adequate for experimentation and testing. We believe that scaling up will not materially affect any of the design trade-offs discussed here so long as main memory is large enough to accommodate the appropriate indexes, in compressed form. If this is not the case some re-engineering will be needed to reduce disk accesses, but the compression methods and models will still be applicable.

Table 1 compares the sample with other free-text databases. Word *tokens* are individual words of the text, counting duplicates; word *types* are lexically distinct tokens, that is, words in the lexicon. The first column gives sizes that are characteristic of very large document retrieval systems (Bird *et al.*, 1978, as reported by Bratley and Choueka, 1982). The second shows the size of the Responsa Retrieval Project which collects a large body of running text, written mainly in Hebrew and Aramaic (figures are from 1981, reported by Choueka *et al.*, 1988). The third is for the *Trésor de la Langue Française*, which covers the literature, history, and science of France from the eighteenth century to the present (Klein *et al.*, 1989).

3.2 Lexical preprocessing

Although the sample text is represented as a sequence of letters, the compression scheme and concordance are based on words. There are many different possible ways of defining what a "word" is (Witten and Bell, 1990). We have adopted a simple approach that splits text into words at a space or before a punctuation mark: thus, for example, the most frequent "word" is the comma character. It was decided to omit from the concordance words made up of punctuation characters alone. Table 2 gives some details of the sample text, including its size and the composition of the lexicon.

The sample has a four-level lexical hierarchy comprising version, book, chapter, and verse; the number of each is given in the Table. Sentences are not appropriate for this text: sometimes a verse includes several sentences, but sometimes two or more verses span a single sentence. In fact the "verse" level is very like the "sentence" level in most other documents. As this illustrates, the appropriate lexical hierarchy will depend on the kind of material being stored. Lexical units end with a special marker in the text; these count as words.

An important lexical issue is the treatment of upper- and lower-case characters. In many situations it is possible to predict the case of a letter. For example, Pike (1981) presented a simple and reversible method for transposing most upper-case characters to lower case. He inverted the case of a letter if

- it is the first letter in the document;
- it is the first letter after a period;
- the last two letters were upper-case;
- it is the the letter "I" flanked by spaces.

The original text can be recovered by re-applying the same rules once more—that is why they are expressed in terms of "inverting" case rather than changing to lower case. The transformation captures some of the rules of English explicitly, and its potential payoff is greater compression and a smaller lexicon.

In order to decide how to treat case, some preliminary experiments were performed on the sample. First, the compression potential was investigated. Using a standard technique (PPMC, see below), compression of the main text of 8.87 bit/word was reduced to 8.86 bit/word when all letters were folded to lower case. This gain in performance is insignificant, and in practice, if case is folded in a way that permits the original to be recovered, compression will likely increase (for example, using Pike's method it increases to 8.89 bit/word).

Second, the potential for reducing the size of the text's lexicon was assessed. About 6000 of the 24,300 words had initial capitals. However, only 1200 of these appear in uncapitalized form—the rest were mostly proper names. Consequently if case could be predicted perfectly, the lexicon would shrink from about 24,300 words to about 23,100 words—a saving of only 5%. In fact, Pike's method catches just over half of these 1200 words but increases the number of mixed-case words, for a net shrinkage in the lexicon of only 1%.

Because of the small potential savings the full text was used. Capitalized words were treated as completely different tokens from their lower-case equivalents.

4. Compressing the lexical hierarchy and disk addresses

Elements at the lowest level of the lexical hierarchy are the smallest addressable units in the main text database. For example, in our case each verse is addressable on disk. Consequently it is necessary to store a disk pointer for every one. At 4 bytes each (enough to address seven 550 Mbyte CD-ROMs), the pointers for the sample text consume 366 Kbyte. Access will be quicker if they can be kept in main store, and so we now study how to compress them.

It is also advantageous to keep a tree structure showing the word number at which each level of the lexical hierarchy begins. This makes it simple to convert a position in the text which is expressed as *word number*, or distance in words from the start of the text, to one expressed as a *coordinate* in the lexical hierarchy—in our case, version number, book-within-version, chapter-within-book, and verse-within-chapter. Again, the data structure involved may be quite large—3 bytes for each of 94,000 verses gives 274 Kbyte for the sample text. Again, however, it can be compressed—and in fact, as we shall see, it helps considerably in representing the disk pointers.

4.1 The lexical hierarchy

Figure 1 illustrates a suitable structure to record the lexical hierarchy of the sample text. Compression is only worthwhile for the lowest level because the exponential growth in index sizes makes gains at higher levels relatively insignificant.

For each unit at the lowest level of the lexical hierarchy, two quantities are stored:

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

Naturally these are highly correlated, and we take advantage of this. In effect it means that, since disk pointers are needed anyway, little further storage is required for the lexical hierarchy.

If the number of words in a unit is stored literally, allowance must be made for the largest unit. Choueka *et al.* (1988) cite a situation where a "sentence" occupies 676 words, and in our case the longest verse is 90 words: thus 10 and 7 bits respectively would be needed to represent unit lengths. However, storage requirements can be reduced by creating a model which gives, for each length that occurs, the number of units that share that length, and coding the individual lengths with respect to the model. For the sample text this yields an entropy of only 5.37 bit/verse. To store the model requires just a few bytes for each length that occurs.

4.2 Disk addresses

The number of bytes a unit occupies on disk can be predicted from the number of words in it. In our case the average figure is 1.34 byte/word (taken over the 2,272,697 "real" words—i.e. excluding those comprising punctuation alone). Using this to predict the space occupied, and representing differences from the predicted value with respect to their empirically-observed distribution, leads to very economical storage. For the sample text it represents the number of bytes in only 3.77 bit/verse on average (compare with the 4 bytes mentioned above for uncompressed storage of disk pointers). In fact, the predicted size is accurate to within one byte in 37% of cases, and the prediction error assumes only 44 different values ranging from –24 to +21 bytes. Again, storing the model consumes just a few bytes for each of these 44 values.

4.3 Efficient conversion

The data structure of Figure 1 permits efficient conversion from the coordinates of a word to its actual word number. It is also necessary to be able to convert word numbers efficiently to hierarchical coordinate specifications, and to disk addresses. A binary search on word number can be performed on the lowest uncompressed index in the lexical hierarchy (the chapter index in Figure 1). Once the appropriate unit at that level (chapter) is found, the corresponding block at the lowest level (verse) can be decompressed and scanned linearly to find the unit (verse) containing the target word.

Our text has an average of 26.3 verses per chapter, so only 13 entries in the verse table need to be decoded on average. The longest chapter contains 176 verses, although this is an outlier (next longest is 89 verses); if it were necessary to limit worst case access time these large chapters should be further subdivided in the chapter index.

The structure of Figure 1 has the further advantage that if the lowest lexical index (verse) is so large that it must be stored on disk, the price paid can be reduced to just one disk access by a slight reorganization—namely, moving the word number of the first verse of each chapter from the verse index to the chapter index.

5. Compressing the main text

The main text is the principal data structure and it is important that it is compressed effectively and can be decoded efficiently. It is also necessary that individual elements at the lowest unit of the lexical hierarchy can be accessed independently.

5.1 Compression by different methods

Using a modern adaptive coding method such as PPMC (Moffat, 1988; Bell *et al.*, 1990), the sample can be compressed to occupy an average of 8.87 bit/word. However, it can then only be read sequentially by a decoder that constructs its model adaptively from the text decoded. The alternative of storing the model explicitly is rather expensive. PPMC is designed to use a maximum of 500 Kbyte of memory, and when this is exhausted it clears the model, primes it from the last few hundred characters and continues coding. When compressing the sample text it will certainly have made use of the full allocation. Amortized over the words in the text, this corresponds to an additional 1.51 bit/word.

Instead, we have investigated the use of schemes that parse the text into words for encoding. For these "word models" a lexicon must be stored, but this presents little additional overhead since full-text retrieval systems require a lexicon anyway. A "zero-order" model encodes each word individually according to its frequency. If the model is "static" the word frequencies are precomputed and stored rather than being accumulated on the fly.

The entropy of the sample text using a static zero-order word model is 8.75 bit/word—slightly less than with the state-of-the-art adaptive encoder PPMC. For a fair comparison the size of the model should be added. Using a straightforward method of representing the words in the lexicon (storing them as character strings), when amortized over the words in the text, contributes an overhead of 0.60 bit/word, while storage of the word counts (3 byte/count) accounts for a further 0.21 bit/word. This brings the entropy of the word model to considerably more than that of PPMC. However, compressing the lexicon as described in the next section reduces the overhead considerably, to 0.20 bit/word (0.16 bits for the words themselves and 0.04 bits for the counts), bringing the total to 8.95 bit/word—competitive with PPMC. These figures are summarized in Table 3. Moreover, for a full-text retrieval system only the space for the counts should be charged against the compression scheme because the words in the lexicon need to be stored anyway.

First-order word models, which condition each word's probability on its predecessor, were investigated too. Not including the lexicon, a first-order static model consumes only 5.76 bit/word on average. The lexicon will be considerably larger than for a zero-order model, since it must store the frequency of each consecutive pair of words that occur. Effective compression of a first-order model's lexicon in a way that permits decoding to take place reasonably efficiently involves an intricate trade-off between space and speed. Consequently the efficient implementation of first-order models has not been investigated in detail, and the zero-order word model is assumed in the remainder of the paper.

For interest, Table 3 also shows the performance of adaptive word models, both zero- and first-order. These are considerably inferior to the corresponding static models, even (in the zero-order case) when lexicon storage is taken into account. It is reassuring to find that the opportunity presented by static modeling to pre-scan the whole text yields an appreciable advantage in compression performance.

5.2 Synchronization points

The zero-order word model calculates the probability of each word and passes this distribution, along with the actual next word, to an arithmetic coder which produces a continuous stream of bits for storage. In order to provide random access to lexical units of the text it is necessary to terminate the arithmetic coding process at the end of each unit and begin it again for the next. We call this process "synchronization."

Each synchronization point incurs a termination overhead. There is a trade-off between synchronizing on small units (e.g. verses) and larger ones (e.g. chapters): at the expense of processing time a large unit can be completely decoded to provide access to its individual constituents. We assume that the text is synchronized according to the smallest indexed unit (verse).

Terminating arithmetic coding incurs a 2-bit overhead. However, further overhead may occur because of the resolution of disk addressing. In general, lower resolution gives smaller pointers but more wasted space on disk. If bit addressing is used, there is no further overhead—but disk pointers are 3 bits larger than they would otherwise be. On the other hand, byte addressing incurs an additional byte-padding overhead of 4 bits on average. Despite its 1-bit disadvantage, byte addressing is preferred on grounds of simplicity. Then each synchronization unit involves a 6-bit overhead on average.

Verses in the sample text average 259.5 bits each in compressed form, and so synchronization imposes a 2.3% overhead on the compressed text.

6. Compressing the lexicon

The lexicon serves a dual purpose: through it the main text is decoded and the concordance is accessed. It contains an entry for every word type which records

- the word itself;
- the word's occurrence count;
- a pointer to its concordance entry.

When decoding the main text, the lexicon is accessed via the occurrence count. Each arithmetic decoding operation yields the cumulative probability of the next word, and the word with that probability is sought in the lexicon. When responding to user requests to locate all occurrences of a particular word, the lexicon is accessed via the word itself to locate its entry in the concordance, which is then decoded and the appropriate units of text read off.

Since the lexicon is used for every word decoded, it is essential that access be fast. For this reason entries are divided into fixed-length blocks of compressed information, so that a binary search can be used to locate the block that contains the target word or occurrence count. Once found, the block is decoded and searched linearly for the target word or count. If enough main memory were available to allow the entire lexicon to be decoded before being searched access time would be reduced further, for the linear search could be avoided.

We first describe the coding of each of the three components of a word's entry in the lexicon, and then outline the data structure in which the blocks are represented. Finally we describe the storage of a permuted lexicon which allows partial word matches to be sought.

6.1 WORDS

The sample text's lexicon comprises 24,323 words, with an average of 7.37 characters each (8.37 if a terminator is included). Uncompressed, these consume a total of 199 Kbyte. The result of compressing them by several different methods is shown in the first part of Table 4.

The benchmark method PPMC yields substantial compression, although being adaptive it is certainly not appropriate for the lexicon since it does not permit random access. Also, the lexicon is sorted into alphabetic order, and PPMC does not take advantage of this. A method called "front compression" is based on the observation that most words in a sorted list share the same initial letters as their predecessor. The repeated letters are simply replaced with a number representing the size of this common prefix. Figure 2 shows part of a lexicon and its front compression coding.

Front coding eliminates prefix repetition but does not otherwise compress the words. That considerable opportunity remains for further compression is shown by applying PPMC to the front-coded list. Since this is inappropriate for random-access data structures a simpler method is used. Separate models are made of prefix length, suffix length (to avoid the need for a terminator), and characters in the lexicon, and each component is coded with respect to the appropriate model. This produces a figure of 17.7 bit/word which corresponds to 2.12 bit/character—excellent performance considering the simplicity of the models used.

The models themselves, for prefix length, suffix length, and characters, consume negligible storage. Twenty different prefix lengths occur in the sample text, and if the occurrence frequency of each is stored as a 4-byte number the prefix-length model consumes 80 bytes. The suffix-length model is similar. For the character model, assuming 128 different characters (in fact, only 81 actually occur in the text) and 4-byte occurrence counts, around 0.5 Kbyte is required.

6.2 COUNTS

In the sample text's lexicon, counts range from 1 (for *hapax legomena*, or words that only occur once) to 175,513 (for the "word" comprising the comma character—the next most frequent word was "the", with 172,946 occurrences). However, for arithmetic coding *cumulative* counts are required, and the total cumulative count is, of course, the number of words in the sample text, namely 2,717,553. Literal storage would need 22 bit/count (middle section of Table 4).

It is obviously more economical to store actual rather than cumulative counts, and rely on the data structure for the lexicon to provide a reasonably efficient way of determining cumulative values (see below, Section 6.4). This reduces the storage requirement to 18 bit/count.

Elias (1975) has developed several ways of representing variable-length integers, one of which—the γ code—is particularly suitable. This represents the number i as $\lfloor \log i \rfloor$ zeros followed by the shortest binary code for i (which begins with a 1). This code gives an average of 5.5 bit/count.

Although the largest count is 175,513, the number of different counts is quite small—891 in fact. Consequently is is feasible to make an exact model by storing the values of 891 different counts, along with the number of times each occurs. This gives an average of 5.0 bit/count. Although only slightly better than the Elias code this representation is preferred since it involves arithmetic coding and this enables different kinds of lexicon data (words, counts, concordance pointers) to be coded in a single stream, without having to terminate and re-start the arithmetic coding process. The greatest number of times a count occurs is 6,452, the number of *hapax legomena* in the text. Allowing 4 bytes for each (clearly lavish!) gives a model size of around 7 Kbyte.

6.3 Locating concordance entries

Almost all words in the lexicon have a corresponding entry in the concordance, which is pointed to from the lexicon. Since the concordance is a large data structure, it is worth considering how best to represent these pointers.

The difference between successive concordance pointers is the size of a word's concordance entry, in bytes. The simplest model would record a distribution of the size of concordance entries and code each one with respect to the distribution. Using this method for the sample text, concordance entries would be coded in 5.55 bits each. However, we can do even better.

Not surprisingly, the size of the concordance entry for a word is correlated with its occurrence count, and so the latter can be used as a predictor by measuring the average ratio of concordance entry size to occurrence count. Testing this on the sample text, the concordance entry size is predicted to within one byte from the occurrence count in 63% of cases. It is only necessary to store the discrepancy between the actual size of a word's concordance entry and that predicted from its occurrence count. Recording all these discrepancies, with their frequencies, provides a model according to which they can be coded.

Using this method on the sample text gives an average of 3.89 bits for each concordance entry. The discrepancy assumes 831 different values—the largest is 349,732, occurring once only, while the most frequent is the value –1, occurring 12,020 times. Allowing four bytes to store each number (again, lavish) gives a storage requirement of around 6 Kbyte for the model.

6.4 Data structure

The average size of a lexicon entry, which comprises the word itself, its occurrence count and concordance pointer, is 17.7+5.0+3.9=26.6 bits, giving a total size for 24,323 words of 79 Kbyte. The size of the models involved is approximately 14 Kbyte. Of course they could be compressed dramatically (by modeling the models!) if this were thought worthwhile.

The lexicon data structure is sketched in Figure 3. It is divided into fixed-length blocks: 256 bytes is a suitable size. Larger blocks would increase access time because more words would have to be decoded and checked once the relevant block was found by a binary search. Smaller ones would increase the overhead of the block header and the dead space at the end of each block.

Because both counts and concordance pointers are coded differentially, their initial values for a block are given literally at the beginning of the block. Since words are front coded, and also to expedite searching by words, the first word of each block is also stored unencoded in the block header. Following the header is a sequence of lexicon entries, arithmetically coded as a single unit.

Each lexicon entry contains the prefix length, suffix length, suffix characters, occurrence count, and concordance pointer, encoded as described above.

In fact, given the data structure used, the lexicon turns out to occupy 85 Kbyte. This represents a 7% overhead on the 79 Kbyte calculated above, which corresponds to approximately 15 bytes of header and 2 bytes of wastage in each 256-byte block. This is the price paid for random access to the compressed lexicon.

6.5 Allowing partial word matches

It is frequently desirable to retrieve units of a text based on incomplete word matches, such as prefix, suffix, and word stem matches. Bratley and Choueka (1982) describe an elegant method for finding query terms of the forms X, $X*$, $*X$, $*X*$, $X*Y$ where X and Y are specified strings of characters and * matches any string. 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 4 shows a permuted version of the lexicon of Figure 2. Just as all words of the form abas* can be found easily from the sorted list at the left of Figure 2, all words of the form *on* (i.e. "abalone" and "abandon") can be found by seeking entries beginning with the pattern "on" in Figure 4 (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. For further details, see Bratley and Choueka (1982).

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 minus 1. For the sample text this is $24{,}196 \times 6.40 = 154{,}948$. The average length of each is 9.27 (10.27 if a terminating character is included)—greater than the average length of lexicon entries since long words have more permutations—and 1.5 Mbyte is needed for uncompressed storage.

The final block of Table 4 shows the effect of the various compression techniques used for the words of the lexicon. Their performance appears worse because of the longer average word; when normalized for this it is similar. Using front coding with models of prefix length, suffix length and characters, yields a total size of permuted index for our sample text of 25.6 × 154,948 bits, or 484 Kbyte. As before, the size of the models involved is negligible.

Like the lexicon, the permuted dictionary must be efficiently searchable. Using a block structure similar to that of Figure 3 but without counts and concordance pointers increases the size of the permuted index by 4% to 503 Kbyte.

7. Compressing the concordance

The concordance contains, for each word type appearing in the text, a list of pointers to all occurrences or "tokens" of that word. Since it includes a pointer for every token in the text, it will consume about as much space as the text itself. Indeed, when both text and concordance are compressed the latter can even be larger, because the pointers are all different whereas words are chosen from a restricted vocabulary.

For example, the concordance for the sample text contains 2,272,697 entries, each pointing to a particular word in the text. Since the full text contains that number of words, 22 bits are needed to specify a particular position, and if this is rounded to 3 bytes the uncompressed concordance occupies 6.5 Mbyte. The space can be reduced somewhat if concordance entries indicate the beginning of a higher lexical unit rather than the word itself; but this will compromise performance on queries such as finding occurrences of words within a certain distance of each other, since the main text will have to be consulted to determine the exact distance between words from different lexical units. Consequently this section addresses the design of a concordance that holds full word pointers.

7.1 Representing the concordance

The most pressing design issue for the concordance is whether to store pointers as the distance in words from the start of the text or as coordinates with a component for each level in the lexical hierarchy.

Choueka *et al.* (1988) use hierarchical coordinates and compress them with an *ad hoc* compression scheme. In their corpus of 38×10^6 words a plain word number requires 26 bits, while its coordinates, when compressed by their best method, occupy only 24 bits on average. Klein *et al.* (1989) also use compressed hierarchical coordinates. Moreover, their concordance is reduced by eliminating references to the 100 most frequent words. Although this decreases concordance size by half, it affects performance since some queries are no longer possible—and although it is rare to encounter a query that contains only very frequent words, it does happen (Klein *et al.*, 1989).

In our case it is possible to convert between word numbers and coordinates using the data structure described in Section 4 and illustrated in Figure 1, and so the penalty for using word numbers in the concordance need not be large.

The crucial factor that favors the use of word pointers for the concordance is that an excellent, and tractable, model is available for encoding them. Since the concordance accounts for about half the space occupied by a full-text retrieval system, an efficient representation is of paramount importance. The model roughly halves the storage required for the concordance, at the cost of increased computation time to convert word numbers to hierarchical coordinates (if these are indeed required). The increase in computation time is fairly small since only a few binary searches are involved—one for each level of the hierarchy.

7.2 A BERNOUILLI MODEL FOR INTER-WORD GAPS

For each word in the lexicon, pointers in the concordance record successive occurrences in the text. The difference between one pointer and the next is conveniently modelled by a geometric distribution of word occurrence.

Consider the occurrences of a particular word. Their number is known: it is just the occurrence count in the word's lexicon entry. Dividing this by the size of the text gives the word's occurrence probability p. Model the occurrences as a Bernouilli process with probability p. The chance of an inter-word gap of size k is the probability of having k non-occurrences followed by one occurrence, or $(1-p)^k p$, for $k = 0, 1, 2, \ldots$. This is called the "geometric" distribution.

The cumulative probabilities required by arithmetic coding can easily be calculated by summing this distribution, and the probability of having up to k non-occurrences followed by one occurrence is simply $1 - (1-p)^{k+1}$. This can be used directly by an arithmetic encoder and decoder to represent inter-word gaps.

Now consider the question of precision in the arithmetic coder. Problems will arise when the difference between two cumulative probabilities drops below the minimum non-zero frequency that can be represented by a count. With the 31-bit coder used, this occurs when the inter-word gap k is so large that $(1-p)^k p < 2^{-29}$. For example, the most frequent word indexed in our concordance is "the," which occurs 172,946 times (p=0.071). When k is 221 or greater, the geometric distribution gives a probability of less than 2^{-29}. Clearly the chance of a gap of more than 221 words between successive occurrences of "the" is very small, but it does happen (15 times in the sample text).

Fortunately, the solution is simple. When the inter-word gap exceeds the maximum value that can be accommodated, it is split into two parts (or more if necessary), one with probability corresponding to the maximum gap that can be represented by the 29-bit count, and the other representing the balance. These parts are coded in two separate arithmetic encoding operations, and the probability associated with the second is automatically scaled into the range given by the tail of the geometric distribution. This ensures that no bits are wasted (actually this is not a significant advantage since the two-part encoding scheme is rarely necessary—0.7% of cases for the sample text). Although it sounds complicated, the implementation is straightforward (a few lines of C).

The cumulative probability formula $1 - (1-p)^{k+1}$ must be inverted when decoding to determine k, and inverted exactly in order for the decoder to proceed correctly. Inversion gives a floating-point approximation to k; the true value is found by testing the rounded-down version and, if it fails, rounding it up instead.

7.3 Performance of the Bernouilli model

Table 5 shows the number of bits occupied by each entry in the concordance using various different compression methods. The exact distribution of inter-word gaps, which would occupy a great deal of storage since there are 122,847 different gap lengths ranging from 1 to 2,270,586, has an entropy of 10.47 bits. The geometric distribution gives an entropy of 10.58 bits. Some wastage occurs when each concordance block—and there are 24,192 of them, one for each different word indexed—is terminated and padded to the nearest byte. In practice the actual space occupied by the concordance averages 10.63 bits per pointer.

The difference between this value and the theoretical one of 10.58 bits is surprisingly low and deserves elaboration. The expected amount of wasted space is 6 bits per block, and a block appears for each word indexed. Thus the expected wastage is only 18 Kbyte in a concordance of compressed size 2.9 Mbyte (2,272,697 pointers at 10.58 bits each). This wastage of 0.6% corresponds to the difference between the actual and theoretical values.

It is interesting that the common words, which are often omitted from concordances because of the inordinate number of pointers associated with them, contribute little to the size of the compressed concordance. The word "the" occurs 172946 times (7.6% of all concordance entries) but only contributes 10,304 bytes (0.34% of total concordance size)—an average of 0.48 bits per pointer! A *hapax legomena*, on the other hand, will contribute 24 bits: its probability is $(1-p)^k p$ where k indicates the word's position in the text (between 0 and 2,272,696) and $p = 1/2,272,697$; the negative logarithm of this lies between 21.1 and 22.6 bits, and byte-padding brings it up to 3 bytes.

Consequently the payoff for omitting common words from the concordance is smaller than might be expected. For example, the 100 most common words in the sample account for 76% of references in the concordance but only 44% of its compressed size. However, they include many words that one might reasonably wish to index on (e.g. Lord, God, Israel). The 15 most common words are more obvious candidates for omission, but although they account for 43% of concordance entries they occupy only 19% of its compressed size.

8. Speed

With the data structures discussed above, the procedure of finding and printing all verses in the sample text that contain a given word is shown in Figure 6.

The time to perform this procedure is dominated by the process of searching the compressed lexicon (steps 1 and 3c). Each 256-byte block contains about 72 entries. On average, half of these entries must be decoded to find a given word (step 1b) or cumulative count (step 3c(ii)). Taking into account prefix and suffix lengths, suffix characters, count, and concordance pointer, each word decoded requires an average of 6.8 arithmetic decoding operations (the mean number of suffix characters is 2.8). Thus every word presented on the screen involves about 245 arithmetic decodes.

Witten *et al.* (1987) report an arithmetic decoding time of 58 μs for an optimized assembly language implementation on a SUN-3/75. However, this figure includes an adaptive model update, which is not needed in the present case. From it a presentation time of 14 ms/word can be calculated, ignoring everything except the arithmetic decoding operations in step 3c.

The actual implementation has been timed on a much faster SUN SPARCstation, and (coincidentally) runs at 15 ms/word, or 68 word/s, which corresponds to a character rate of 570 char/s—a little faster than a 4800 baud line. Hardware arithmetic decoding (e.g. Mitchell and Pennebaker, 1987; Langdon *et al.*, 1988) could of course speed things up greatly, as could storing the lexicon in uncompressed form. For example, when the lexicon in stored uncompressed, avoiding the linear decode-and-search of steps 1b and 3c(ii), speed increased by an order of magnitude to 50 Kbit/s. No doubt these figures could be improved considerably by optimizing the code. They could also be improved (at the expense of space) by using smaller blocks.

9. Conclusions

This paper has shown how to use a variety of different models to address the problem of compression in full-text retrieval systems. The overall result for the sample text is summarized in Table 6, which shows the compression achieved on the main data structures required. The miscellaneous minor data structures, which are mostly models for compression, are summarized in Table 7: they could easily be compressed further but account for only 0.4% of space occupied. The total compressed size of 6151 Kbytes is less than half the size of the original text alone.

This is a considerable improvement on earlier work. For example, Klein *et al.* (1989) report a total compressed size to 72% of the original text (503 Mbyte compressed, 700 uncompressed). They achieve compression to 35%, 40%, and 60% (expressed in the terms of Table 6) on text, dictionary, and concordance respectively—but is it not clear that these figures are directly comparable because of possibly different ways of measuring the original size, and of their omission of the 100 most common words from the concordance.

There are several opportunities for further improvement. The prime targets are the main text and concordance, between them accounting for almost 90% of space occupied. As noted in Table 3, a first-order model for the main text has the potential to reduce storage requirements substantially, from 8.75 to 5.76 bit/word, at the cost of a dictionary of word pairs and some additional complexity.

Turning to the concordance, the Bernouilli model of inter-word gaps could be improved. Although Table 5 shows that it leads to an entropy which is close to that for the exact distribution of gap lengths, it should be possible to do much better because, unlike the exact distribution, the model is sensitive to the frequency of the word whose gaps are being considered. One source of inaccuracy is that the model gives the highest probability to a gap of 0, whereas consecutive appearances of the same word are rare. While this could be fixed, a potentially more productive approach is to take account of the recency effect—the fact that occurrences of a particular word tend to be clustered rather than evenly distributed throughout the text.

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.

Bird, R.M., Newsbaum, J.B. and Trefftzs, J.L. (1989) Text file inversion: an evaluation," *Proc. Fourth Workshop for Computer Architecture for Non-Numeric Processing*: 42–50; Syracuse University; August.

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.

Cleary, J.G. and Witten, I.H. (1984) "A comparison of enumerative and adaptive codes," *IEEE Trans. Information Theory IT–30*(2): 306–315; March.

Elias, P. (1975) "Universal codeword sets and representations of the integers," *IEEE Trans Information Theory IT–21*(2): 194–203; March.

Huffman, D.A. (1952) "A method for the construction of minimum-redundancy codes," *Proc. IERE 40*(9): 1098–1101; September.

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.

Langdon, G.G., Pennebaker, W.B., Mitchell, J.L., Arps, R.M. and Rissanen, J.J. (1988) "A tutorial on the adaptive Q-coder," IBM Research Report RJ5736, San Jose, CA; June.

Lelewer, D.A. and Hirschberg, D.S. (1987) "Data compression," *Computing Surveys 13*(3): 261–296; September.

Mitchell, J.L. and Pennebaker, W.B. (1987) "Optimal hardware and software coding procedures for the Q-Coder," IBM Research Report RC12660, San Jose, CA; April.

Moffat, A. (1988) "A note on the PPM data compression algorithm," Research Report 88/7, Department of Computer Science, University of Melbourne, Parkville, Victoria, Australia.

Pike, J. (1981) "Text compression using a 4 bit coding system," *Computer J. 24*(4).

Rissanen, J. and Langdon, G.G. (1981) "Universal modeling and coding," *IEEE Trans. Information Theory IT–27*(1): 12–23; January.

Shannon, C.E. (1948) "A mathematical theory of communication," *Bell System Technical J. 27*: 398–403; July.

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

	Bird *et al.*, 1978	RRP (Choueka *et al.*, 1988)	TLF (Klein *et al.*, 1989)	sample text
Number of documents	1 000 000	37 500	2 600	200
Number of word-tokens	$2\,000 \times 10^6$	38×10^6	112×10^6	2.7×10^6
Number of word-types	500 000	?	360 000	22 000

Table 1 Size of some free text databases

Tokens (number of words)	2 717 553	
Types (size of lexicon)	24 323	
Concordanced words	24 192	99.5%
all lower case	15 258	62.7%
capitalized lower-case	1 228	5.0%
other capitalized word	4 761	19.6%
all upper case	42	0.2%
other mixed-case	1	0.0%
all numeric	258	1.1%
other mixed characters	2 644	10.9%
Unconcordanced words	131	0.5%
all punctuation	127	0.5%
hierarchy markers	4	0.0%
Lexical hierarchy		
versions	3	
books	198	
chapters	3 567	
verses	93 654	

Table 2 Particulars of the sample text used for experiments

	bit/word	
Zero-order static word modeling		
Text with respect to model	8.75	
Model	*uncompressed*	*compressed*
storage of words	0.60	0.16
storage of counts	0.21	0.04
Total	9.56	8.95
Other methods, for comparison		
PPMC	8.87	
PPMC, accounting for model storage	10.34	
First-order static word modeling	5.76	
Zero-order adaptive word modeling	10.98	
First-order adaptive word modeling	9.33	

Table 3 Performance of different modeling techniques on the main text

For the words	8 bit/char	67.0	bit/word
	PPMC	24.4	bit/word
	front coding	38.8	bit/word
	front coding with PPMC	13.9	bit/word
	front coding, individual models	17.7	bit/word
For the counts	cumulative counts	22	bit/count
	non-cumulative counts	18	bit/count
	Elias γ code	5.5	bit/count
	exact model of counts	5.0	bit/count
For permuted words	8 bit/char	82.2	bit/word
	PPMC	27.7	bit/word
	front coding	49.7	bit/word
	front coding with PPMC	21.1	bit/word
	front coding, individual models	25.6	bit/word

Table 4 Compression of words and counts in lexicon and permuted dictionary

	bits
Without compression	22
Exact distribution of gap lengths	10.47
Entropy of geometric distribution	10.58
Actual space using geometric distribution	10.63

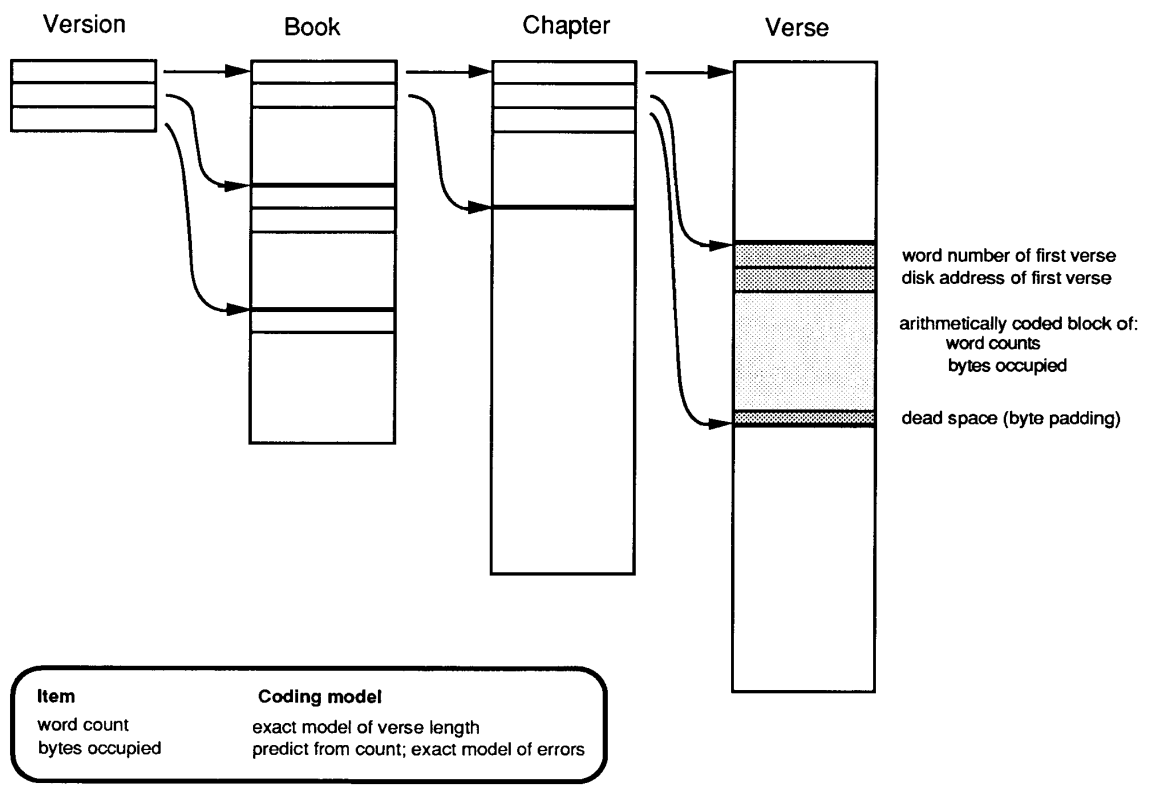
Table 5 Space occupied by pointers in the concordance

	Full size (Kbyte)	Compressed size (Kbyte)	Compression
Text	12 570	2 967	24%
Lexicon	199	85	43%
Permuted dictionary	1 554	503	32%
Concordance	6 658	2 949	44%
Word number/disk address for smallest lexical unit	640	104	16%
Miscellaneous	—	27	
Total		6 151	

Table 6 Overall compression of the sample text

		bytes	*see section*
Top levels of lexical hierarchy			*4.1*
	version to book number	9	
	book to chapter number	507	
	chapter number to verse	10 512	
Models for verse information			*4.1, 4.2*
	words in verse	360	
	prediction error for bytes occupied	200	
Models for lexicon			*6.1, 6.2, 6.3*
	prefix lengths	120	
	suffix lengths	120	
	characters	1 024	
	counts	7 136	
	concordance pointers	6 656	
Models for permuted dictionary			*6.5*
	prefix lengths	120	
	suffix lengths	120	
	characters	1 024	
Model for concordance			*7.2*
	geometric model	0	
	Total	27	Kbyte

Table 7 Miscellaneous storage requirements for the sample text

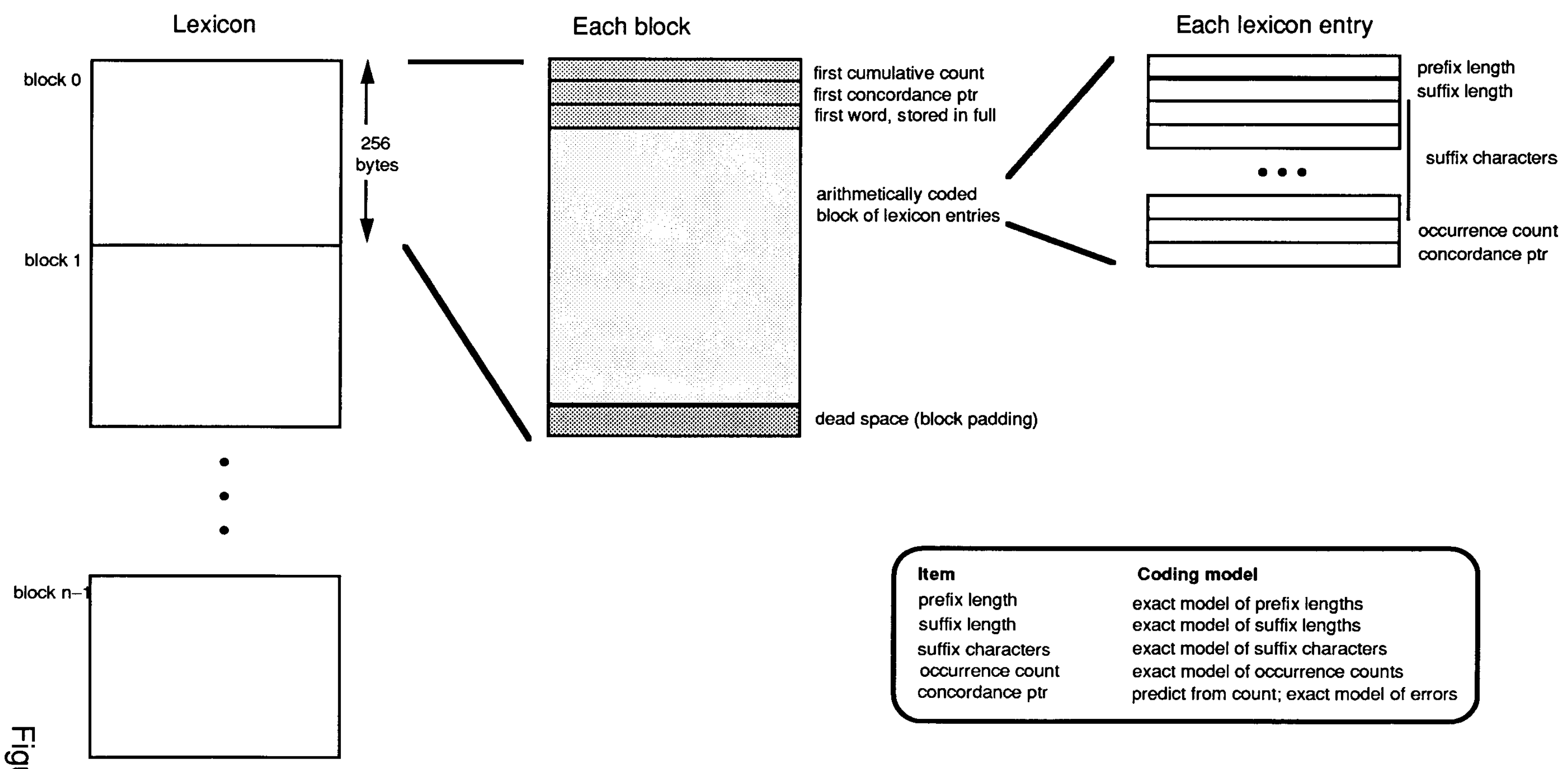

Version
Book
Chapter
Verse
word number of first verse
disk address of first verse
arithmetically coded block of:
word counts
bytes occupied
dead space (byte padding)

Item	Coding model
word count	exact model of verse length
bytes occupied	predict from count; exact model of errors

Figure 1

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

Figure 2

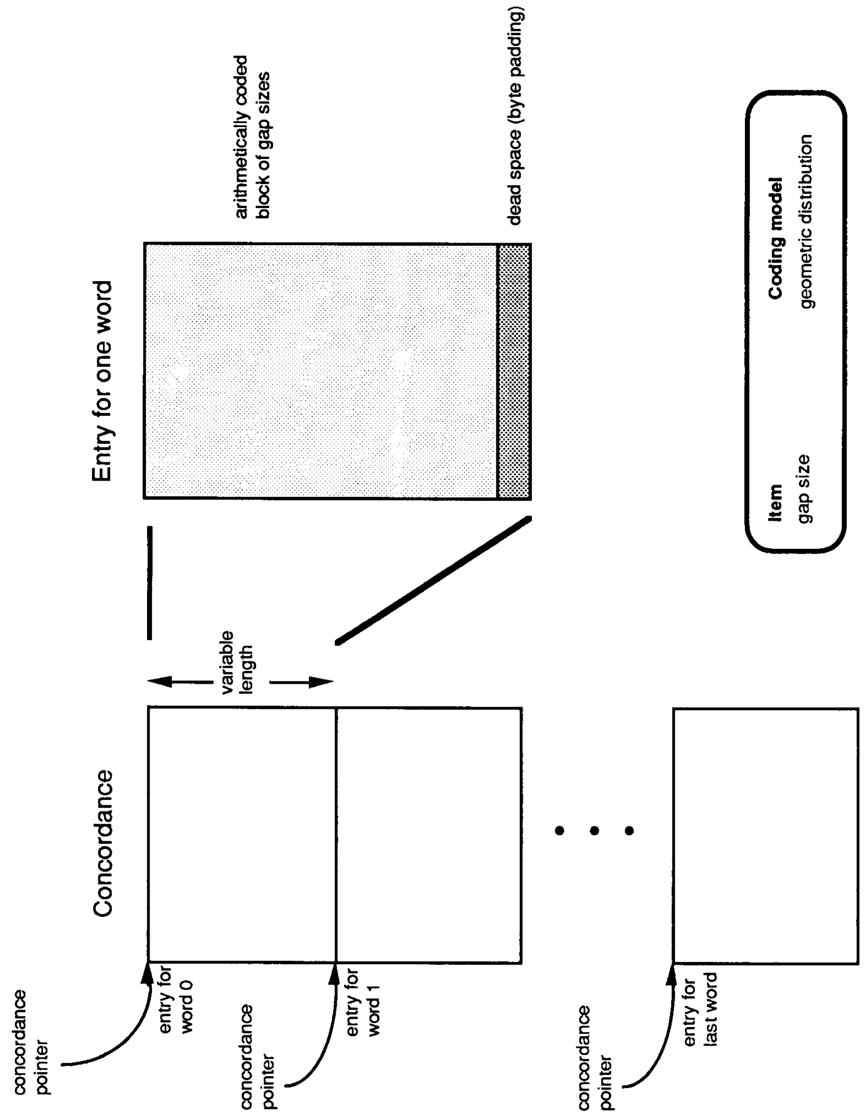


Item	**Coding model**
prefix length	exact model of prefix lengths
suffix length	exact model of suffix lengths
suffix characters	exact model of suffix characters
occurrence count	exact model of occurrence counts
concordance ptr	predict from count; exact model of errors

Figure 3

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 4



Entry for one word
arithmetically coded block of gap sizes
dead space (byte padding)
Item
gap size
Coding model
geometric distribution
Concordance
variable length
concordance pointer
entry for word 0
concordance pointer
entry for word 1
concordance pointer
entry for last word

Figure 5

```
1.  Find word in lexicon
    a.  Binary search (on word) to find block
    b.  Decode block until word is found (linear search)
        (Fail on encountering
          • lexicographically greater word
          • end of block)
    Result: pointer to the concordance entries for the given word

2.  Decode the concordance entries

3.  For each concordance entry, print the verse containing it
        a. Translate word number into the verse's disk address
            (i)   Binary search to find block of verse table
            (ii)  Decode block to find verse (linear search)
        b. Binary search chapter, book, version tables for hierarchical
                  coordinate (for display only)
        c. Read, decode, and print words to the end-of-verse marker.
          Decoding each word involves finding it in the lexicon:
            (i)   Binary search (on cumulative count) to find block
            (ii)  Decode block for word with appropriate count (linear
                  search).
```

Figure 6