6 Index Compression

An inverted index for a given text collection can be quite large, especially if it contains full positional information about all term occurrences in the collection. In a typical collection of English text there is approximately one token for every 6 bytes of text (including punctuation and whitespace characters). Hence, if postings are stored as 64-bit integers, we may expect that an uncompressed positional index for such a collection consumes between 130% and 140% of the uncompressed size of the original text data. This estimate is confirmed by Table 6.1, which lists the three example collections used in this book along with their uncompressed size, their compressed size, and the sizes of an uncompressed and a compressed schema-independent index. An uncompressed schema-independent index for the TREC45 collection, for instance, consumes about 331 MB, or 122% of the raw collection size.[1]

Table 6.1 Collection size, uncompressed and compressed (`gzip --best`), and size of schema-independent index, uncompressed (64-bit integers) and compressed (vByte), for the three example collections.

	Collection Size		Index Size	
	Uncompressed	**Compressed**	**Uncompressed**	**Compressed**
Shakespeare	7.5 MB	2.0 MB	10.5 MB (139%)	2.7 MB (36%)
TREC45	1904.5 MB	582.9 MB	2331.1 MB (122%)	533.0 MB (28%)
GOV2	425.8 GB	79.9 GB	328.3 GB (77%)	62.1 GB (15%)

An obvious way to reduce the size of the index is to not encode each posting as a 64-bit integer but as a $\lceil \log(n) \rceil$-bit integer, where n is the number of tokens in the collection. For TREC45 ($\lceil \log(n) \rceil = 29$) this shrinks the index from 2331.1 MB to 1079.1 MB — 57% of the raw collection size. Compared to the naïve 64-bit encoding, this is a major improvement. However, as you can see from Table 6.1, it is not even close to the 533 MB that can be achieved if we employ actual index compression techniques.

Because an inverted index consists of two principal components, the dictionary and the postings lists, we can study two different types of compression methods: dictionary compression and postings list compression. Because the size of the dictionary is typically very small compared

[1] The numbers for GOV2 are lower than for the other two collections because its documents contain a nontrivial amount of JavaScript and other data that are not worth indexing.

to the total size of all postings lists (see Table 4.1 on page 106), researchers and practitioners alike usually focus on the compression of postings lists. However, dictionary compression can sometimes be worthwhile, too, because it decreases the main memory requirements of the search engine and allows it to use the freed resources for other purposes, such as caching postings lists or search results.

The remainder of this chapter consists of three main parts. In the first part (Sections 6.1 and 6.2) we provide a brief introduction to general-purpose, symbolwise data compression techniques. The second part (Section 6.3) treats the compression of postings lists. We discuss several compression methods for inverted lists and point out some differences between the different types of inverted indices. We also show how the effectiveness of these methods can be improved by applying document reordering techniques. The last part (Section 6.4) covers compression algorithms for dictionary data structures and shows how the main memory requirements of the search engine can be reduced substantially by storing the in-memory dictionary entries in compressed form.

6.1 General-Purpose Data Compression

In general, a data compression algorithm takes a chunk of data A and transforms it into another chunk of data B, such that B is (hopefully) smaller than A, that is, it requires fewer bits to be transmitted over a communication channel or to be stored on a storage medium. Every compression algorithm consists of two components: the *encoder* (or *compressor*) and the *decoder* (or *decompressor*). The encoder takes the original data A and outputs the compressed data B. The decoder takes B and produces some output C.

A particular compression method can either be *lossy* or *lossless*. With a lossless method, the decoder's output, C, is an exact copy of the original data A. With a lossy method, C is not an exact copy of A but an approximation that is somewhat similar to the original version. Lossy compression is useful when compressing pictures (e.g., JPEG) or audio files (e.g., MP3), where small deviations from the original are invisible (or inaudible) to the human senses. However, in order to estimate the loss of quality introduced by those small deviations, the compression algorithm needs to have some a priori knowledge about the structure or the meaning of the data being compressed.

In this chapter we focus exclusively on lossless compression algorithms, in which the decoder produces an exact copy of the original data. This is mainly because it is not clear what the value of an approximate reconstruction of a given postings list would be (except perhaps in the case of positional information, where it may sometimes be sufficient to have access to approximate term positions). When people talk about *lossy index compression*, they are often not referring to data compression methods but to index pruning schemes (see Section 5.1.5).

6.2 Symbolwise Data Compression

When compressing a given chunk of data A, we are usually less concerned with A's actual appearance as a bit string than with the *information* contained in A. This information is called the *message*, denoted as M. Many data compression techniques treat M as a sequence of *symbols* from a symbol set $\mathcal{S}$ (called the *alphabet*):

$$M = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle, \quad \sigma_i \in \mathcal{S}. \tag{6.1}$$

Such methods are called *symbolwise* or *statistical*. Depending on the specific method and/or the application, a symbol may be an individual bit, a byte, a word (when compressing text), a posting (when compressing an inverted index), or something else. Finding an appropriate definition of what constitutes a symbol can sometimes be difficult, but once a definition has been found, well-known statistical techniques can be used to re-encode the symbols in M in order to decrease their overall storage requirements. The basic idea behind symbolwise data compression is twofold:

1. Not all symbols in M appear with the same frequency. By encoding frequent symbols using fewer bits than infrequent ones, the message M can be represented using a smaller number of bits in total.
2. The i-th symbol in a symbol sequence $\langle \sigma_1, \sigma_2, \ldots, \sigma_i, \ldots, \sigma_n \rangle$ sometimes depends on the previous symbols $\langle \ldots, \sigma_{i-2}, \sigma_{i-1} \rangle$. By taking into account this interdependence between the symbols, even more space can be saved.

Consider the task of compressing the text found in the Shakespeare collection (English text with XML markup), stored in ASCII format and occupying 8 bytes per character. Without any special effort we can decrease the storage requirements from 8 to 7 bits per character because the collection contains only 86 distinct characters. But even among the characters that do appear in the collection, we can see that there is a large gap between the frequencies with which they appear. For example, the six most frequent and the six least frequent characters in the Shakespeare collection are:

1. " ":	742,018	4. "⟨":	359,452		81. "(":	2	84. "8":	2
2. "E":	518,133	5. "⟩":	359,452	...	82. ")":	2	85. "$":	1
3. "e":	410,622	6. "t":	291,103		83. "5":	2	86. "7":	1

If the most frequent character (the whitespace character) is re-encoded using 1 bit less (6 instead of 7), and the two least frequent characters are re-encoded using 1 bit more (8 instead of 7), a total of 742,016 bits is saved.

Moreover, there is an interdependence between consecutive characters in the text collection. For example, the character "u", appearing 114,592 times in the collection, is located somewhere in the middle of the overall frequency range. However, every occurrence of the character "q" is followed by a "u". Thus, every "u" that is preceded by a "q" can be encoded using 0 bits because a "q" is never followed by anything else!

6.2.1 Modeling and Coding

Symbolwise compression methods usually work in two phases: modeling and coding. In the modeling phase a probability distribution $\mathcal{M}$ (also referred to as the *model*) is computed that maps symbols to their probability of occurrence. In the coding phase the symbols in the message M are re-encoded according to a code $\mathcal{C}$. A code is simply a mapping from each symbol σ to its codeword $\mathcal{C}(\sigma)$, usually a sequence of bits. $\mathcal{C}(\sigma)$ is chosen based on σ's probability according to the compression model $\mathcal{M}$. If $\mathcal{M}(\sigma)$ is small, then $\mathcal{C}(\sigma)$ will be long; if $\mathcal{M}(\sigma)$ is large, then $\mathcal{C}(\sigma)$ will be short (where the length of a codeword σ is measured by the number of bits it occupies).

Depending on how and when the modeling phase takes place, a symbolwise compression method falls into one of three possible classes:

- In a **static method** the model $\mathcal{M}$ is independent of the message M to be compressed. It is assumed that the symbols in the message follow a certain predefined probability distribution. If they don't, the compression results can be quite disappointing.
- **Semi-static methods** perform an initial pass over the message M and compute a model $\mathcal{M}$ that is then used for compression. Compared with static methods, semi-static methods have the advantage that they do not blindly assume a certain distribution. However, the model $\mathcal{M}$, computed by the encoder, now needs to be transmitted to the decoder as well (otherwise, the decoder will not know what to do with the encoded symbol sequence) and should therefore be as compact as possible. If the model itself is too large, a semi-static method may lose its advantage over a static one.
- The encoding procedure of an **adaptive compression method** starts with an initial static model and gradually adjusts this model based on characteristics of the symbols from M that have already been coded. When the compressor encodes σ_i, it uses a model $\mathcal{M}_i$ that depends only on the previously encoded symbols (and the initial static model):

  $$\mathcal{M}_i = f(\sigma_1, \ldots, \sigma_i - 1).$$

  When the decompressor needs to decode σ_i, it has already seen $\sigma_1, \ldots, \sigma_i - 1$ and thus can apply the same function f to reconstruct the model $\mathcal{M}_i$. Adaptive methods, therefore, have the advantage that no model ever needs to be transmitted from the encoder to the decoder. However, they require more complex decoding routines, due to the necessity to continuously update the compression model. Decoding, therefore, is usually somewhat slower than with a semi-static method.

The probabilities in the compression model $\mathcal{M}$ do not need to be unconditional. For example, it is quite common to choose a model in which the probability of a symbol σ depends on the $1, 2, 3, \ldots$ previously coded symbols (we have seen this in the case of the Shakespeare collection, where the occurrence of a "q" is enough information to know that the next character is going to be a "u"). Such models are called *finite-context* models or *first-order*, *second-order*, *third-order*, ... models. A model $\mathcal{M}$ in which the probability of a symbol is independent of the previously seen symbols is called a *zero-order* model.

Compression models and codes are intimately connected. Every compression model $\mathcal{M}$ has a code (or a family of codes) associated with it: the code (or codes) that minimizes the average codeword length for a symbol sequence generated according to $\mathcal{M}$. Conversely, every code has a corresponding probability distribution: the distribution for which the code is optimal. For example, consider the zero-order compression model $\mathcal{M}_0$:

$$\mathcal{M}_0(\text{“a”}) = 0.5,\ \mathcal{M}_0(\text{“b”}) = 0.25,\ \mathcal{M}_0(\text{“c”}) = 0.125,\ \mathcal{M}_0(\text{“d”}) = 0.125. \tag{6.2}$$

A code $\mathcal{C}_0$ that is optimal with respect to the model $\mathcal{M}_0$ has the following property:

$$|\mathcal{C}_0(\text{“a”})| = 1,\ |\mathcal{C}_0(\text{“b”})| = 2,\ |\mathcal{C}_0(\text{“c”})| = 3,\ |\mathcal{C}_0(\text{“d”})| = 3 \tag{6.3}$$

(where $|\mathcal{C}_0(X)|$ denotes the bit length of $\mathcal{C}_0(X)$). The following code meets this requirement:[2]

$$\mathcal{C}_0(\text{“a”}) = \overline{0},\ \mathcal{C}_0(\text{“b”}) = \overline{11},\ \mathcal{C}_0(\text{“c”}) = \overline{100},\ \mathcal{C}_0(\text{“d”}) = \overline{101}. \tag{6.4}$$

It encodes the symbol sequence "aababacd" as

$$\mathcal{C}_0(\text{“aababacd”}) = \overline{00110110100101}. \tag{6.5}$$

An important property of the code $\mathcal{C}_0$ is that it is *prefix-free*, that is, there is no codeword $\mathcal{C}_0(x)$ that is a prefix of another codeword $\mathcal{C}_0(y)$. A prefix-free code is also referred to as a *prefix code*. Codes that are not prefix-free normally cannot be used for compression (there are some exceptions to this rule; see Exercise 6.3). For example, consider the alternative code $\mathcal{C}_1$, with

$$\mathcal{C}_1(\text{“a”}) = \overline{1},\ \mathcal{C}_1(\text{“b”}) = \overline{01},\ \mathcal{C}_1(\text{“c”}) = \overline{101},\ \mathcal{C}_1(\text{“d”}) = \overline{010}. \tag{6.6}$$

Based on the lengths of the codewords, this code also appears to be optimal with respect to $\mathcal{M}_0$. However, the encoded representation of the sequence "aababacd" now is

$$\mathcal{C}_1(\text{“aababacd”}) = \overline{11011011101010}. \tag{6.7}$$

[2] To avoid confusion between the numerals "0" and "1" and the corresponding bit values, we denote the bit values as $\overline{0}$ and $\overline{1}$, except where it is obvious that the latter meaning is intended.

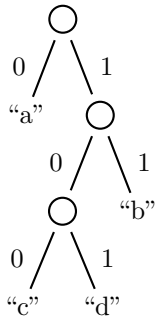

Figure 6.1 Binary tree associated with the prefix code $\mathcal{C}_0$. Codewords: $\mathcal{C}_0(\text{“a”}) = \overline{0}$, $\mathcal{C}_0(\text{“b”}) = \overline{11}$, $\mathcal{C}_0(\text{“c”}) = \overline{100}$, $\mathcal{C}_0(\text{“d”}) = \overline{101}$.

When the decoder sees this sequence, it has no way of knowing whether the original symbol sequence was “aababacd” or “accaabd”. Hence, the code is ambiguous and cannot be used for compression purposes.

A prefix code $\mathcal{C}$ can be thought of as a binary tree in which each leaf node corresponds to a symbol σ. The labels encountered along the path from the tree's root to the leaf associated with σ define the symbol's codeword, $\mathcal{C}(\sigma)$. The depth of a leaf equals the length of the codeword for the respective symbol.

The code tree for the code $\mathcal{C}_0$ is shown in Figure 6.1. The decoding routine of a compression algorithm can translate a bit sequence back into the original symbol sequence by following the edges of the tree, outputting a symbol — and jumping back to the root node — whenever it reaches a leaf, thus essentially using the tree as a binary decision diagram. The tree representation of a binary code also illustrates why the prefix property is so important: In the tree for a code that is not prefix-free, some codewords are assigned to internal nodes; when the decoder reaches such a node, it does not know whether it should output the corresponding symbol and jump back to the root, or keep following edges until it arrives at a leaf node.

Now consider a compression model $\mathcal{M}$ for a set of symbols $\{\sigma_1, \ldots, \sigma_n\}$ such that the probability of each symbol is an inverse power of 2:

$$\mathcal{M}(\sigma_i) = 2^{-\lambda_i} \;; \quad \lambda_i \in \mathbb{N} \;\text{ for } 1 \leq i \leq n. \tag{6.8}$$

Because $\mathcal{M}$ is a probability distribution, we have

$$\sum_{i=1}^{n} \mathcal{M}(\sigma_i) \;=\; \sum_{i=1}^{n} 2^{-\lambda_i} \;=\; 1. \tag{6.9}$$

Let us try to find an optimal code tree for this distribution. Every node in the tree must be either a leaf node (and have a codeword associated with it) or an internal node with exactly two children. Such a tree is called a *proper binary tree*. If the tree had an internal node with

only one child, then it would be possible to improve the code by removing that internal node, thus reducing the depths of its descendants by 1. Hence, the code would not be optimal.

For the set $\mathcal{L} = \{L_1, \ldots, L_n\}$ of leaf nodes in a proper binary tree, the following equation holds:

$$\sum_{i=1}^{n} 2^{-d(L_i)} = 1, \tag{6.10}$$

where $d(L_i)$ is the depth of node L_i and is also the length of the codeword assigned to the symbol associated with L_i. Because of the similarity between Equation 6.9 and Equation 6.10, it seems natural to assign codewords to symbols in such a way that

$$|\mathcal{C}(\sigma_i)| = d(L_i) = \lambda_i = -\log_2(\mathcal{M}(\sigma_i)) \quad \text{for } 1 \leq i \leq n. \tag{6.11}$$

The resulting tree represents an optimal code for the given probability distribution $\mathcal{M}$. Why is that? Consider the average number of bits per symbol used by $\mathcal{C}$ if we encode a sequence of symbols generated according to $\mathcal{M}$:

$$\sum_{i=1}^{n} \Pr[\sigma_i] \cdot |\mathcal{C}(\sigma_i)| \;=\; -\sum_{i=1}^{n} \mathcal{M}(\sigma_i) \cdot \log_2(\mathcal{M}(\sigma_i)) \tag{6.12}$$

because $|\mathcal{C}(\sigma_i)| = -\log_2(\mathcal{M}(S_i))$. According to the following theorem, this is the best that can be achieved.

Source Coding Theorem (Claude Shannon, 1948)
Given a symbol source S, emitting symbols from an alphabet $\mathcal{S}$ according to a probability distribution $\mathcal{P}_S$, a sequence of symbols cannot be compressed to consume less than

$$\mathcal{H}(S) \;=\; -\sum_{\sigma \in \mathcal{S}} \mathcal{P}_S(\sigma) \cdot \log_2(\mathcal{P}_S(\sigma))$$

bits per symbol on average. $\mathcal{H}(S)$ is called the *entropy* of the symbol source S.

By applying Shannon's theorem to the probability distribution defined by the model $\mathcal{M}$, we see that the chosen code is in fact optimal for the given model. Therefore, if our initial assumption that all probabilities are inverse powers of 2 (see Equation 6.8) does in fact hold, then we know how to quickly find an optimal encoding for symbol sequences generated according to $\mathcal{M}$. In practice, of course, this will rarely be the case. Probabilities can be arbitrary values from the interval $[0, 1]$. Finding the optimal prefix code $\mathcal{C}$ for a given probability distribution $\mathcal{M}$ is then a little more difficult than for the case where $\mathcal{M}(\sigma_i) = 2^{-\lambda_i}$.

6.2.2 Huffman Coding

One of the most popular bitwise coding techniques is due to Huffman (1952). For a given probability distribution $\mathcal{M}$ on a finite set of symbols $\{\sigma_1, \ldots, \sigma_n\}$, Huffman's method produces a prefix code $\mathcal{C}$ that minimizes

$$\sum_{i=1}^{n} \mathcal{M}(\sigma_i) \cdot |\mathcal{C}(\sigma_i)|. \tag{6.13}$$

A Huffman code is guaranteed to be optimal among all codes that use an integral number of bits per symbol. If we drop this requirement and allow codewords to consume a fractional number of bits, then there are other methods, such as arithmetic coding (Section 6.2.3), that may achieve better compression.

Suppose we have a compression model $\mathcal{M}$ with $\mathcal{M}(\sigma_i) = \Pr[\sigma_i]$ (for $1 \leq i \leq n$). Huffman's method constructs an optimal code tree for this model in a bottom-up fashion, starting with the two symbols of smallest probability. The algorithm can be thought of as operating on a set of trees, where each tree is assigned a probability mass. Initially there is one tree T_i for each symbol σ_i, with $\Pr[T_i] = \Pr[\sigma_i]$. In each step of the algorithm, the two trees T_j and T_k with minimal probability mass are merged into a new tree T_l. The new tree is assigned a probability mass $\Pr[T_l] = \Pr[T_j] + \Pr[T_k]$. This procedure is repeated until there is only a single tree T_{Huff} left, with $\Pr[T_{\text{Huff}}] = 1$.

Figure 6.2 shows the individual steps of the algorithm for the symbol set $\mathcal{S} = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ with associated probability distribution

$$\Pr[\sigma_1] = 0.18,\ \Pr[\sigma_2] = 0.11,\ \Pr[\sigma_3] = 0.31,\ \Pr[\sigma_4] = 0.34,\ \Pr[\sigma_5] = 0.06. \tag{6.14}$$

After the tree has been built, codewords may be assigned to symbols in a top-down fashion by a simple traversal of the code tree. For instance, σ_3 would receive the codeword $\overline{01}$; σ_2 would receive the codeword $\overline{110}$.

Optimality

Why are the codes produced by this method optimal? First, note that an optimal prefix code $\mathcal{C}_{opt}$ must satisfy the condition

$$\Pr[x] < \Pr[y] \Rightarrow |\mathcal{C}_{opt}(x)| \geq |\mathcal{C}_{opt}(y)| \quad \text{for every pair of symbols } (x, y) \tag{6.15}$$

(otherwise, we could simply swap the codewords for x and y, thus arriving at a better code). Furthermore, because an optimal prefix code is always represented by a proper binary tree, the codewords for the two least likely symbols must always be of the same length d (their nodes are siblings, located at the lowest level of the binary tree that represents $\mathcal{C}_{opt}$; if they are not siblings, we can rearrange the leaves at the lowest level in such a way that the two nodes become siblings).

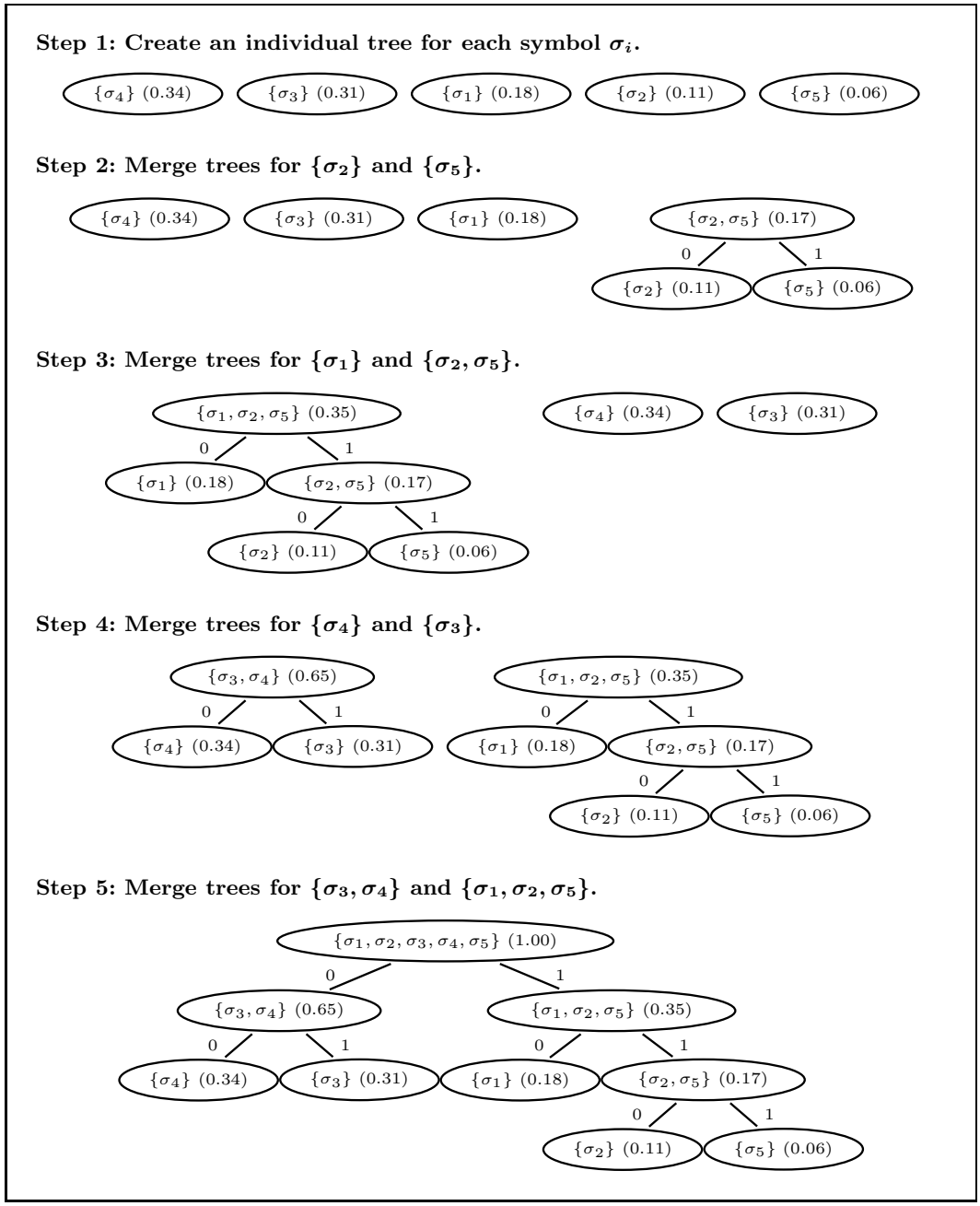

Figure 6.2 Building a Huffman code tree for the symbol set $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ with associated probability distribution $\Pr[\sigma_1] = 0.18$, $\Pr[\sigma_2] = 0.11$, $\Pr[\sigma_3] = 0.31$, $\Pr[\sigma_4] = 0.34$, $\Pr[\sigma_5] = 0.06$.

We can now prove the optimality of Huffman's algorithm by induction on the number of symbols n for which a prefix code is built. For $n = 1$, the method produces a code tree of height 0, which is clearly optimal. For the general case consider a symbol set $\mathcal{S} = \{\sigma_1, \ldots, \sigma_n\}$. Let σ_j and σ_k denote the two symbols with least probability ($j < k$ w.l.o.g.). Their codewords have the same length d. Thus, the expected codeword length for a sequence of symbols from $\mathcal{S}$ is

$$\mathrm{E}[\mathrm{Huff}(\mathcal{S})] \;=\; d \cdot (\Pr[\sigma_j] + \Pr[\sigma_k]) + \sum_{x \in (\mathcal{S} \setminus \{\sigma_j, \sigma_k\})} \Pr[x] \cdot |\mathcal{C}(x)| \tag{6.16}$$

(bits per symbol). Consider the symbol set

$$\mathcal{S}' = \{\sigma_1, \ldots, \sigma_{j-1}, \sigma_{j+1}, \ldots, \sigma_{k-1}, \sigma_{k+1}, \ldots, \sigma_n, \sigma'\} \tag{6.17}$$

that we obtain by removing σ_j and σ_k and replacing them by a new symbol σ' with

$$\Pr[\sigma'] = \Pr[\sigma_j] + \Pr[\sigma_k]. \tag{6.18}$$

Because σ' is the parent of σ_j and σ_k in the Huffman tree for $\mathcal{S}$, its depth in the tree for $\mathcal{S}'$ is $d - 1$. The expected codeword length for a sequence of symbols from $\mathcal{S}'$ then is

$$\mathrm{E}[\mathrm{Huff}(\mathcal{S}')] \;=\; (d-1) \cdot (\Pr[\sigma_j] + \Pr[\sigma_k]) + \sum_{x \in (\mathcal{S}' \setminus \{\sigma'\})} \Pr[x] \cdot |\mathcal{C}(x)|. \tag{6.19}$$

That is, $\mathrm{E}[\mathrm{Huff}(\mathcal{S}')] = \mathrm{E}[\mathrm{Huff}(\mathcal{S})] - \Pr[\sigma_j] - \Pr[\sigma_k]$.

By induction we know that the Huffman tree for $\mathcal{S}'$ represents an optimal prefix code (because $\mathcal{S}'$ contains $n - 1$ elements). Now suppose the Huffman tree for $\mathcal{S}$ is not optimal. Then there must be an alternative, optimal tree with expected cost

$$\mathrm{E}[\mathrm{Huff}(\mathcal{S})] - \varepsilon \quad (\text{for some } \varepsilon > 0). \tag{6.20}$$

By collapsing the nodes for σ_j and σ_k in this tree, as before, we obtain a code tree for the symbol set $\mathcal{S}'$, with expected cost

$$\mathrm{E}[\mathrm{Huff}(\mathcal{S})] - \varepsilon - \Pr[\sigma_j] - \Pr[\sigma_k] \;=\; \mathrm{E}[\mathrm{Huff}(\mathcal{S}')] - \varepsilon \tag{6.21}$$

(this is always possible because the nodes for σ_j and σ_k are siblings in the optimal code tree, as explained before). However, this contradicts the assumption that the Huffman tree for $\mathcal{S}'$ is optimal. Hence, a prefix code for $\mathcal{S}$ with cost smaller than $\mathrm{E}[\mathrm{Huff}(\mathcal{S})]$ cannot exist. The Huffman tree for $\mathcal{S}$ must be optimal.

Complexity

The second phase of Huffman's algorithm, assigning codewords to symbols by traversing the tree built in the first phase, can trivially be done in time $\Theta(n)$, since there are $2n - 1$ nodes

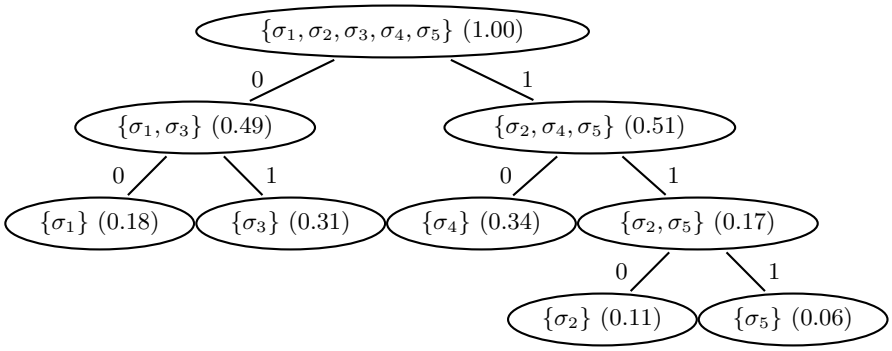

Figure 6.3 Canonical version of the Huffman code from Figure 6.2. All symbols at the same level in the tree are sorted in lexicographical order.

in the tree. The first phase, building the tree, is slightly more complicated. It requires us to continually keep track of the two trees $T_j, T_k \in \mathcal{T}$ with minimum probability mass. This can be realized by maintaining a priority queue (e.g., a min-heap; see page 141) that always holds the minimum-probability tree at the beginning of the queue. Such a data structure supports INSERT and EXTRACT-MIN operations in time $\Theta(\log(n))$. Because, in total, Huffman's algorithm needs to perform $2(n-1)$ EXTRACT-MIN operations and $n-1$ INSERT operations, the running time of the first phase of the algorithm is $\Theta(n \log(n))$. Hence, the total running time of the algorithm is also $\Theta(n \log(n))$.

Canonical Huffman codes

When using Huffman coding for data compression, a description of the actual code needs to be prepended to the compressed symbol sequence so that the decoder can transform the encoded bit sequence back into the original symbol sequence. This description is known as the *preamble*. It usually contains a list of all symbols along with their codewords. For the Huffman code from Figure 6.2, the preamble could look as follows:

$$\langle(\sigma_1, \overline{10}), (\sigma_2, \overline{110}), (\sigma_3, \overline{01}), (\sigma_4, \overline{00}), (\sigma_5, \overline{111})\rangle. \tag{6.22}$$

Describing the Huffman code in such a way can be quite costly in terms of storage space, especially if the actual message that is being compressed is relatively short.

Fortunately, it is not necessary to include an exact description of the actual code; a description of certain aspects of it suffices. Consider again the Huffman tree constructed in Figure 6.2. An equivalent Huffman tree is shown in Figure 6.3. The length of the codeword for each symbol σ_i is the same in the original tree as in the new tree. Thus, there is no reason why the code from Figure 6.2 should be preferred over the one from Figure 6.3.

Reading the symbols σ_i from left to right in Figure 6.3 corresponds to ordering them by the length of their respective codewords (short codewords first); ties are broken according to

the natural ordering of the symbols (e.g., σ_1 before σ_3). A code with this property is called a *canonical Huffman code*. For every possible Huffman code $\mathcal{C}$, there is always an equivalent canonical Huffman code $\mathcal{C}_{can}$.

When describing a canonical Huffman code, it is sufficient to list the bit lengths of the codewords for all symbols, not the actual codewords. For example, the tree in Figure 6.3 can be described by

$$\langle\,(\sigma_1,2),(\sigma_2,3),(\sigma_3,2),(\sigma_4,2),(\sigma_5,3)\,\rangle\,. \tag{6.23}$$

Similarly, the canonicalized version of the Huffman code in Figure 6.1 is

$$\langle\,(\text{“a”},\overline{0}),(\text{“b”},\overline{10}),(\text{“c”},\overline{110}),(\text{“d”},\overline{111})\,\rangle\,. \tag{6.24}$$

It can be described by

$$\langle\,(\text{“a”},1),(\text{“b”},2),(\text{“c”},3),(\text{“d”},3)\,\rangle\,. \tag{6.25}$$

If the symbol set $\mathcal{S}$ is known by the decoder a priori, then the canonical Huffman code can be described as $\langle 1,2,3,3\rangle$, which is about twice as compact as the description of an arbitrary Huffman code.

Length-limited Huffman codes

Sometimes it can be useful to impose an upper bound on the length of the codewords in a given Huffman code. This is mainly for performance reasons, because the decoder can operate more efficiently if codewords are not too long (we will discuss this in more detail in Section 6.3.6).

We know that, for an alphabet of n symbols, we can always find a prefix code in which no codeword consumes more than $\lceil\log_2(n)\rceil$ bits. There are algorithms that, given an upper bound L on the bit lengths of the codewords ($L \geq \lceil\log_2(n)\rceil$), produce a prefix code $\mathcal{C}_L$ that is optimal among all prefix codes that do not contain any codewords longer than L. Most of those algorithms first construct an ordinary Huffman code $\mathcal{C}$ and then compute $\mathcal{C}_L$ by performing some transformations on the binary tree representing $\mathcal{C}$. Technically, such a code is not a Huffman code anymore because it lost its universal optimality (it is optimal only among all length-limited codes). However, in practice the extra redundancy in the resulting code is negligible.

One of the most popular methods for constructing length-limited prefix codes is Larmore and Hirschberg's (1990) Package-Merge algorithm. It produces an optimal length-limited prefix code in time $O(nL)$, where n is the number of symbols in the alphabet. Since L is usually small (after all, this is the purpose of the whole procedure), Package-Merge does not add an unreasonable complexity to the encoding part of a Huffman-based compression algorithm. Moreover, just as in the case of ordinary Huffman codes, we can always find an equivalent canonical code for any given length-limited code, thus allowing the same optimizations as before.

6.2.3 Arithmetic Coding

The main limitation of Huffman coding is its inability to properly deal with symbol distributions in which one symbol occurs with a probability that is close to 1. For example, consider the following probability distribution for a two-symbol alphabet $\mathcal{S} = \{\text{“a”}, \text{“b”}\}$:

$$\Pr[\text{“a”}] = 0.8,\ \Pr[\text{“b”}] = 0.2. \tag{6.26}$$

Shannon's theorem states that symbol sequences generated according to this distribution cannot be encoded using less than

$$-\Pr[\text{“a”}] \cdot \log_2(\Pr[\text{“a”}]) - \Pr[\text{“b”}] \cdot \log_2(\Pr[\text{“b”}]) \ \approx\ 0.2575 + 0.4644 \ =\ 0.7219 \tag{6.27}$$

bits per symbol on average. With Huffman coding, however, the best that can be achieved is 1 bit per symbol, because each codeword has to consume an integral number of bits. Thus, we increase the storage requirements by 39% compared to the lower bound inferred from Shannon's theorem.

In order to improve upon the performance of Huffman's method, we need to move away from the idea that each symbol is associated with a separate codeword. This could, for instance, be done by combining symbols into m-tuples and assigning a codeword to each unique m-tuple (this technique is commonly known as *blocking*). For the example above, choosing $m = 2$ would result in the probability distribution

$$\Pr[\text{“aa”}] = 0.64,\ \Pr[\text{“ab”}] = \Pr[\text{“ba”}] = 0.16,\ \Pr[\text{“bb”}] = 0.04. \tag{6.28}$$

A Huffman code for this distribution would require 1.56 bits per 2-tuple, or 0.78 bits per symbol on average (see Exercise 6.1). However, grouping symbols into blocks is somewhat cumbersome and inflates the size of the preamble (i.e., the description of the Huffman code) that also needs to be transmitted from the encoder to the decoder.

Arithmetic coding is a method that solves the problem in a more elegant way. Consider a sequence of k symbols from the set $\mathcal{S} = \{\sigma_1, \ldots, \sigma_n\}$:

$$\langle s_1, s_2, \ldots, s_k \rangle \ \in\ \mathcal{S}^k. \tag{6.29}$$

Each such sequence has a certain probability associated with it. For example, if we consider k to be fixed, then the above sequence will occur with probability

$$\Pr[\langle s_1, s_2, \ldots, s_k \rangle] = \prod_{i=1}^{k} \Pr[s_i]. \tag{6.30}$$

Obviously, the sum of the probabilities of all sequences of the same length k is 1. Hence, we may think of each such sequence x as an interval $[x_1, x_2)$, with $0 \leq x_1 < x_2 \leq 1$ and $x_2 - x_1 = \Pr[x]$.

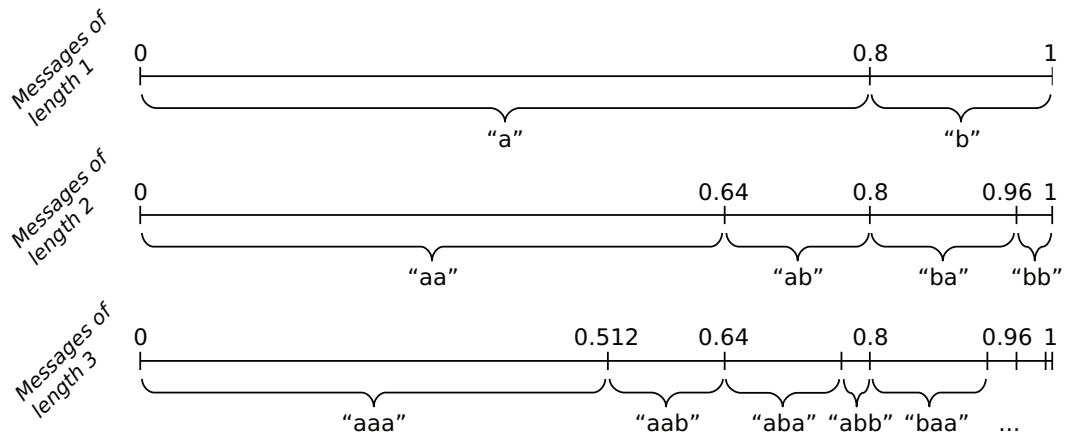

Figure 6.4 Arithmetic coding: transforming symbol sequences into subintervals of the interval $[0, 1)$. A message M (sequence of symbols) is encoded as a binary subinterval of the interval corresponding to M. Probability distribution assumed for the example: Pr["a"] = 0.8, Pr["b"] = 0.2.

These intervals are arranged as subintervals of the interval $[0, 1)$, in lexicographical order of the respective symbol sequences, and form a partitioning of $[0, 1)$.

Consider again the distribution from Equation 6.26. Under the interval representation, the symbol sequence "aa" would be associated with the interval $[0, 0.64)$; the sequence "ab" with the interval $[0.64, 0.80)$; the sequence "ba" with $[0.80, 0.96)$; and, finally, "bb" with $[0.96, 1.0)$. This method can be generalized to symbol sequences of arbitrary length, as shown in Figure 6.4.

With this mapping between messages and intervals, a given message can be encoded by encoding the associated interval $\mathcal{I}$ instead of the message itself. As it might be difficult to encode $\mathcal{I}$ directly, however, arithmetic coding instead encodes a smaller interval $\mathcal{I}'$ that is contained in the interval $\mathcal{I}$ (i.e., $\mathcal{I}' \subseteq \mathcal{I}$) and that is of the special form

$$\mathcal{I}' = [x, x + 2^{-q}), \quad \text{with } x = \sum_{i=1}^{q} a_i \cdot 2^{-i} \quad (a_i \in \{0, 1\}). \tag{6.31}$$

We call this a *binary interval*. A binary interval can be encoded in a straightforward fashion, as a bit sequence $\langle a_1, a_2, \ldots, a_q \rangle$. For instance, the bit sequence $\overline{0}$ represents the interval $[0, 0.5)$; the bit sequence $\overline{01}$ represents the interval $[0.25, 5)$; and the bit sequence $\overline{010}$ represents the interval $[0.25, 0.375)$.

The combination of these two steps — (1) transforming a message into an equivalent interval $\mathcal{I}$ and (2) encoding a binary interval within $\mathcal{I}$ as a simple bit sequence — is called *arithmetic coding*. When presented the message "aab", an arithmetic coder would find the corresponding interval $\mathcal{I} = [0.512, 0.64)$ (see Figure 6.4). Within this interval it would identify the binary

subinterval

$$\mathcal{I}' \;=\; [0.5625, 0.625) \;=\; [x, x + 2^{-4}), \tag{6.32}$$

with $x = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$. Thus, the message "aab" would be encoded as $\overline{1001}$.

It turns out that $\overline{1001}$ with its 4 bits is actually 1 bit longer than the 3-bit sequence of a Huffman code for the same probability distribution. On the other hand, the code sequence for the message "aaa" would require 3 bits with a Huffman code but only a single bit ($\overline{0}$) if arithmetic coding is used. In general, for a message with probability p corresponding to an interval $\mathcal{I} = [y, y + p)$, we need to find a binary interval $[x, x + 2^{-q})$ with

$$y \leq x < x + 2^{-q} \leq y + p. \tag{6.33}$$

It can be shown that such an interval always exists for every q with $2^{-q} \leq p/2$. The smallest q that meets this requirement is

$$q = \lceil -\log_2(p) \rceil + 1. \tag{6.34}$$

Thus, in order to encode a message

$$M = \langle s_1, s_2, \ldots, s_k \rangle \tag{6.35}$$

with associated probability $p = \prod_{i=1}^{k} \Pr[s_i]$, we need no more than $\lceil -\log_2(p) \rceil + 1$ bits. For messages that contain only a few symbols, the "+1" term may sometimes make arithmetic coding less space-efficient than Huffman coding. For $k \to \infty$, however, the average number of bits per symbol approaches the theoretical optimum, as given by Shannon's source coding theorem (see Section 6.2.2). Thus, under the assumption that the compression model $\mathcal{M}$ perfectly describes the statistical properties of the symbol sequence to be compressed, arithmetic coding is asymptotically optimal.

Two convenient properties of arithmetic coding are:

- A different model may be used for each position in the input sequence (subject to the constraint that it may depend only on information already accessible to the decoder), thus making arithmetic coding the method of choice for adaptive compression methods.
- The length of the message need not be known to the decoder ahead of time. Instead, it is possible to add an artificial end-of-message symbol to the alphabet and treat it like any other symbol in the alphabet (in particular, to continually update its probability in the compression model so that it consumes less and less probability mass as the message gets longer).

For the purpose of this chapter, we may safely ignore these rather specific aspects of arithmetic coding. If you are interested in the details, you can find them in any standard textbook on data compression (see Section 6.6).

Implementation

It might seem from our description of arithmetic coding that the encoding and decoding procedures require floating-point operations and are susceptible to rounding errors and other problems that would arise from such an implementation. However, this is not the case. Witten et al. (1987), for instance, show how arithmetic coding can be implemented using integer arithmetic, at the cost of a tiny redundancy increase (less than 10^{-4} bits per symbol).

Huffman coding versus arithmetic coding

The advantage, in terms of bits per symbol, that arithmetic coding has over Huffman codes depends on the probability distribution $\mathcal{M}$. Obviously, if all symbol probabilities are inverse powers of 2, then Huffman is optimal and arithmetic coding has no advantage. For an arbitrary model $\mathcal{M}$, on the other hand, it is easy to see that the redundancy of a Huffman code (i.e., the number of bits wasted per encoded symbol), compared with the optimal arithmetic code, is at most 1 bit. This is true because we can trivially construct a prefix code with codeword lengths $|\mathcal{C}(\sigma_i)| = \lceil -\log_2(p_i) \rceil$. Such a code has a redundancy of less than 1 bit per symbol, and because it is a prefix code, it cannot be better than $\mathcal{M}$'s Huffman code.

In general, the redundancy of the Huffman code depends on the characteristics of the symbol set's probability distribution. Gallager (1978), for instance, showed that, for a given compression model $\mathcal{M}$, the redundancy of a Huffman code is always less than $\mathcal{M}(\sigma_{max}) + 0.0861$, where σ_{max} is the most likely symbol. Horibe (1977) showed that the redundancy is at most $1 - 2 \cdot \mathcal{M}(\sigma_{min})$, where σ_{min} is the least likely symbol.

Because the compression rates achieved by Huffman coding are usually quite close to the theoretical optimum and because decoding operations for Huffman codes can be realized more efficiently than for arithmetic codes, Huffman coding is often preferred over arithmetic coding. This is especially true in the context of index compression, where query processing performance, and not the size of the inverted index, is typically the main criterion when developing a search engine. Length-limited canonical Huffman codes can be decoded very efficiently, much faster than arithmetic codes.

6.2.4 Symbolwise Text Compression

What you have just learned about general-purpose symbolwise data compression can be applied directly to the problem of compressing text. For example, if you want to decrease the storage requirements of a given text collection, you can write a semi-static, Huffman-based compression algorithm that compresses the text in a two-pass process. In the first pass it collects frequency information about all characters appearing in the collection, resulting in a zero-order compression model. It then constructs a Huffman code $\mathcal{C}$ from this model, and in a second pass replaces each character σ in the text collection with its codeword $\mathcal{C}(\sigma)$.

If you implement this algorithm and run it on an arbitrary piece of English text, you will find that the compressed version of the text requires slightly more than 5 bits per character. This

Table 6.2 Text compression rates (in bits per character) for the three example collections. For Huffman and arithmetic coding, numbers do not include the size of the compression model (e.g., Huffman tree) that also needs to be transmitted to the decoder.

Collection	Huffman			Arithmetic Coding			Other	
	0-order	1-order	2-order	0-order	1-order	2-order	gzip	bzip2
Shakespeare	5.220	2.852	2.270	5.190	2.686	1.937	2.155	1.493
TREC45	5.138	3.725	2.933	5.105	3.676	2.813	2.448	1.812
GOV2	5.413	3.948	2.832	5.381	3.901	2.681	1.502	1.107

is in line with the conventional wisdom that the zero-order entropy of English text is around 5 bits per character. The exact numbers for our three example collections are shown in Table 6.2.

You can refine this algorithm and make it use a first-order compression model instead of a zero-order one. This requires you to construct and store/transmit up to 256 Huffman trees, one for each possible context (where the context is the previous byte in the uncompressed text), but it pays off in terms of compression effectiveness, by reducing the space requirements to less than 4 bits per character. This method may be extended to second-order, third-order, ... models, resulting in better and better compression rates. This shows that there is a strong interdependency between consecutive characters in English text (for example, every "q" in the Shakespeare collection is followed by a "u"). By exploiting this interdependency we can achieve compression rates that are much better than 5 bits per character.

Regarding the relative performance of Huffman coding and arithmetic coding, the table shows that the advantage of arithmetic coding over Huffman is very small. For the zero-order model the difference is less than 0.04 bits per character for all three collections. For the higher-order models this difference becomes larger (e.g., 0.12–0.33 bits for the second-order model) because the individual probability distributions in those models become more and more skewed, thus favoring arithmetic coding.

Note, however, that the numbers in Table 6.2 are slightly misleading because they do not include the space required by the compression model (the Huffman trees) itself. For example, with a third-order compression model we may need to transmit $256^3 = 16.8$ million Huffman trees before we can start encoding the actual text data. For a large collection like GOV2 this may be worthwhile. But for a smaller collection like Shakespeare this is clearly not the case. Practical compression algorithms, therefore, use adaptive methods when working with finite-context models and large alphabets. Examples are PPM (prediction by partial matching; Cleary and Witten, 1984) and DMC (dynamic Markov compression; Cormack and Horspool, 1987).

Starting with an initial model (maybe a zero-order one), adaptive methods gradually refine their compression model $\mathcal{M}$ based on the statistical information found in the data seen so far. In practice, such methods lead to very good results, close to those of semi-static methods, but with the advantage that no compression model needs to be transmitted to the decoder (making them effectively better than semi-static approaches).

Finally, the table compares the relatively simple compression techniques based on Huffman coding or arithmetic coding with more sophisticated techniques such as Ziv-Lempel (Ziv and Lempel, 1977; `gzip`[3]) and Burrows-Wheeler (Burrows and Wheeler, 1994; `bzip2`[4]). At heart, these methods also rely on Huffman or arithmetic coding, but they do some extra work before they enter the coding phase. It can be seen from the table that this extra work leads to greatly increased compression effectiveness. For example, `bzip2` (with parameter `--best`) can compress the text in the three collections to between 1.1 and 1.8 bits per character, an 80% improvement over the original encoding with 8 bits per character. This is consistent with the general assumption that the entropy of English text is between 1 and 1.5 bits per character — bounds that were established more than half a century ago (Shannon, 1951).

6.3 Compressing Postings Lists

Having discussed some principles of general-purpose data compression, we can apply these principles to the problem of compressing an inverted index, so as to reduce its potentially enormous storage requirements. As mentioned in Chapter 4 (Table 4.1 on page 106), the vast majority of all data in an inverted index are postings data. Thus, if we want to reduce the total size of the index, we should focus on ways to compress the postings lists before dealing with other index components, such as the dictionary.

The exact method that should be employed to compress the postings found in a given index index depends on the type of the index (docid, frequency, positional, or schema-independent), but there is some commonality among the different index types that allows us to use the same general approach for all of them.

Consider the following sequence of integers that could be the beginning of a term's postings list in a docid index:

$$L = \langle 3, 7, 11, 23, 29, 37, 41, \ldots \rangle .$$

Compressing L directly, using a standard compression method such as Huffman coding, is not feasible; the number of elements in the list can be very large and each element of the list occurs only a single time. However, because the elements in L form a monotonically increasing sequence, the list can be transformed into an equivalent sequence of differences between consecutive elements, called Δ-*values*:

$$\Delta(L) = \langle 3, 4, 4, 12, 6, 8, 4, \ldots \rangle .$$

[3] gzip (`www.gzip.org`) is actually not based on the Ziv-Lempel compression method but on the slightly different DEFLATE algorithm, since the use of the Ziv-Lempel method was restricted by patents during gzip's initial development.

[4] bzip2 (`www.bzip.org`) is a freely available, patent-free data compression software based on the Burrows-Wheeler transform.

The new list $\Delta(L)$ has two advantages over the original list L. First, the elements of $\Delta(L)$ are smaller than those of L, meaning that they can be encoded using fewer bits. Second, some elements occur multiple times in $\Delta(L)$, implying that further savings might be possible by assigning codewords to Δ-values based on their frequency in the list.

The above transformation obviously works for docid indices and schema-independent indices, but it can also be applied to lists from a document-centric positional index with postings of the form $(d, f_{t,d}, \langle p_1, \ldots, p_{f_{t,d}} \rangle)$. For example, the list

$$L = \langle (3, 2, \langle 157, 311 \rangle), (7, 1, \langle 212 \rangle), (11, 3, \langle 17, 38, 133 \rangle), \ldots \rangle$$

would be transformed into the equivalent Δ-list

$$\Delta(L) = \langle (3, 2, \langle 157, 154 \rangle), (4, 1, \langle 212 \rangle), (4, 3, \langle 17, 21, 95 \rangle), \ldots \rangle .$$

That is, each docid is represented as a difference from the previous docid; each within-document position is represented as a difference from the previous within-document position; and the frequency values remain unchanged.

Because the values in the three resulting lists typically follow very different probability distributions (e.g., frequency values are usually much smaller than within-document positions), it is not uncommon to apply three different compression methods, one to each of the three sublists of the Δ-transformed postings list.

Compression methods for postings in an inverted index can generally be divided into two categories: *parametric* and *nonparametric* codes. A nonparametric code does not take into account the actual Δ-gap distribution in a given list when encoding postings from that list. Instead, it assumes that all postings lists look more or less the same and share some common properties — for example, that smaller gaps are more common than longer ones. Conversely, prior to compression, parametric codes conduct an analysis of some statistical properties of the list to be compressed. A parameter value is selected based on the outcome of this analysis, and the codewords in the encoded postings sequence depend on the parameter.

Following the terminology from Section 6.2.1, we can say that nonparametric codes correspond to static compression methods, whereas parametric codes correspond to semi-static methods. Adaptive methods, due to the complexity associated with updating the compression model, are usually not used for index compression.

6.3.1 Nonparametric Gap Compression

The simplest nonparametric code for positive integers is the *unary code*. In this code a positive integer k is represented as a sequence of $k - 1$ $\overline{0}$ bits followed by a single $\overline{1}$ bit. The unary code is optimal if the Δ-values in a postings list follow a geometric distribution of the form

$$\Pr[\Delta = k] = \left(\frac{1}{2}\right)^k , \tag{6.36}$$

that is, if a gap of length $k+1$ is half as likely as a gap of length k (see Section 6.2.1 for the relationship between codes and probability distributions). For postings in an inverted index this is normally not the case, except for the most frequent terms, such as "the" and "and", that tend to appear in almost every document. Nonetheless, unary encoding plays an important role in index compression because other techniques (such as the γ code described next) rely on it.

Elias's γ code

One of the earliest nontrivial nonparametric codes for positive integers is the γ code, first described by Elias (1975). In γ coding the codeword for a positive integer k consists of two components. The second component, the *body*, contains k in binary representation. The first component, the *selector*, contains the length of the body, in unary representation. For example, the codewords of the integers 1, 5, 7, and 16 are:

k	selector(k)	body(k)
1	1	1
5	001	101
7	001	111
16	00001	10000

You may have noticed that the body of each codeword in the above table begins with a $\overline{1}$ bit. This is not a coincidence. If the selector for an integer k has a value selector(k) $= j$, then we know that $2^{j-1} \leq k < 2^j$. Therefore, the j-th least significant bit in the number's binary representation, which happens to be the first bit in the codeword's body, must be $\overline{1}$. Of course, this means that the first bit in the body is in fact redundant, and we can save one bit per posting by omitting it. The γ codewords for the four integers above then become $\overline{1}$ (1), $\overline{001\ 01}$ (5), $\overline{001\ 11}$ (7), and $\overline{00001\ 0000}$ (16).

The binary representation of a positive integer k consists of $\lfloor \log_2(k) \rfloor + 1$ bits. Thus, the length of k's codeword in the γ code is

$$|\gamma(k)| \ = \ 2 \cdot \lfloor \log_2(k) \rfloor + 1 \text{ bits.} \tag{6.37}$$

Translated back into terms of gap distribution, this means that the γ code is optimal for integer sequences that follow the probability distribution

$$\Pr[\Delta = k] \ \approx \ 2^{-2 \cdot \log_2(k) - 1} \ = \ \frac{1}{2 \cdot k^2}. \tag{6.38}$$

δ and ω codes

γ coding is appropriate when compressing lists with predominantly small gaps (say, smaller than 32) but can be quite wasteful if applied to lists with large gaps. For such lists the δ code, a

variant of γ, may be the better choice. δ coding is very similar to γ coding. However, instead of storing the selector component of a given integer in unary representation, δ encodes the selector using γ. Thus, the δ codewords of the integers 1, 5, 7, and 16 are:

k	selector(k)	body(k)
1	1	
5	01 1	01
7	01 1	11
16	001 01	0000

(omitting the redundant $\overline{1}$ bit in all cases). The selector for the integer 16 is $\overline{001\ 01}$ because the number's binary representation requires 5 bits, and the γ codeword for 5 is $\overline{001\ 01}$.

Compared with γ coding, where the length of the codeword for an integer k is approximately $2 \cdot \log_2(k)$, the δ code for the same integer consumes only

$$|\delta(k)| \;=\; \lfloor \log_2(k) \rfloor + 2 \cdot \lfloor \log_2(\lfloor \log_2(k) \rfloor + 1) \rfloor + 1 \text{ bits}, \tag{6.39}$$

where the first component stems from the codeword's body, while the second component is the length of the γ code for the selector. δ coding is optimal if the gaps in a postings list are distributed according to the following probability distribution:

$$\Pr[\Delta = k] \;\approx\; 2^{-\log_2(k) - 2 \cdot \log_2(\log_2(k)) - 1} \;=\; \frac{1}{2k \cdot (\log_2(k))^2}. \tag{6.40}$$

For lists with very large gaps, δ coding can be up to twice as efficient as γ coding. However, such large gaps rarely appear in any realistic index. Instead, the savings compared to γ are typically somewhere between 15% and 35%. For example, the γ codewords for the integers 2^{10}, 2^{20}, and 2^{30} are 21, 41, and 61 bits long, respectively. The corresponding δ codewords consume 17, 29, and 39 bits (−19%, −29%, −36%).

The idea to use γ coding to encode the selector of a given codeword, as done in δ coding, can be applied recursively, leading to a technique known as ω coding. The ω code for a positive integer k is constructed as follows:

1. Output $\overline{0}$.
2. If $k = 1$, then stop.
3. Otherwise, encode k as a binary number (including the leading $\overline{1}$) and prepend this to the bits already written.
4. $k \leftarrow \lfloor \log_2(k) \rfloor$.
5. Go back to step 2.

For example, the ω code for $k = 16$ is

$$\overline{10\ 100\ 10000\ 0} \tag{6.41}$$

Table 6.3 Encoding positive integers using various nonparametric codes.

Integer	γ Code	δ Code	ω Code
1	1	1	0
2	01 0	01 0 0	10 0
3	01 1	01 0 1	11 0
4	001 00	01 1 00	10 100 0
5	001 01	01 1 01	10 101 0
6	001 10	01 1 10	10 110 0
7	001 11	01 1 11	10 111 0
8	0001 000	001 00 000	11 1000 0
16	00001 0000	001 01 0000	10 100 10000 0
32	000001 00000	001 10 00000	10 101 100000 0
64	0000001 000000	001 11 000000	10 110 1000000 0
127	0000001 111111	001 11 111111	10 110 1111111 0
128	00000001 0000000	0001 000 0000000	10 111 10000000 0

because $\overline{10000}$ is the binary representation of 16, $\overline{100}$ is the binary representation of 4 ($= \lfloor \log_2(16) \rfloor$), and $\overline{10}$ is the binary representation of 2 ($= \lfloor \log_2(\lfloor \log_2(16) \rfloor) \rfloor$). The length of the ω codeword for an integer k is approximately

$$|\omega(k)| \;=\; 2 + \log_2(k) + \log_2(\log_2(k)) + \log_2(\log_2(\log_2(k))) + \cdots \tag{6.42}$$

Table 6.3 lists some integers along with their γ, δ, and ω codewords. The δ code is more space-efficient than γ for all integers $n \geq 32$. The ω code is more efficient than γ for all integers $n \geq 128$.

6.3.2 Parametric Gap Compression

Nonparametric codes have the disadvantage that they do not take into account the specific characteristics of the list to be compressed. If the gaps in the given list follow the distribution implicitly assumed by the code, then all is well. However, if the actual gap distribution is different from the implied one, then a nonparametric code can be quite wasteful and a parametric method should be used instead.

Parametric compression methods can be divided into two classes: *global* methods and *local* methods. A global method chooses a single parameter value that is used for all inverted lists in the index. A local method chooses a different value for each list in the index, or even for each small chunk of postings in a given list, where a chunk typically comprises a few hundred or a few thousand postings. You may recall from Section 4.3 that each postings list contains a

list of synchronization points that help us carry out random access operations on the list. Synchronization points fit naturally with chunkwise list compression (in fact, this is the application from which they initially got their name); each synchronization point corresponds to the beginning of a new chunk. For heterogeneous postings lists, in which different parts of the list have different statistical characteristics, chunkwise compression can improve the overall effectiveness considerably.

Under most circumstances local methods lead to better results than global ones. However, for very short lists it is possible that the overhead associated with storing the parameter value outweighs the savings achieved by choosing a local method, especially if the parameter is something as complex as an entire Huffman tree. In that case it can be beneficial to choose a global method or to employ a *batched* method that uses the same parameter value for all lists that share a certain property (e.g., similar average gap size).

Because the chosen parameter can be thought of as a brief description of a compression model, the relationship between local and global compression methods is a special instance of a problem that we saw in Section 6.1 when discussing the relationship between modeling and coding in general-purpose data compression: When to stop modeling and when to start coding? Choosing a local method leads to a more precise model that accurately describes the gap distribution in the given list. Choosing a global method (or a batched method), on the other hand, makes the model less accurate but also reduces its (amortized) size because the same model is shared by a large number of lists.

Golomb/Rice codes

Suppose we want to compress a list whose Δ-values follow a *geometric distribution*, that is, the probability of seeing a gap of size k is

$$\Pr[\Delta = k] \;=\; (1-p)^{k-1} \cdot p, \tag{6.43}$$

for some constant p between 0 and 1. In the context of inverted indices this is not an unrealistic assumption. Consider a text collection comprising N documents and a term T that appears in N_T of them. The probability of finding an occurrence of T by randomly picking one of the documents in the collection is N_T/N. Therefore, under the assumption that all documents are independent of each other, the probability of seeing a gap of size k between two subsequent occurrences is

$$\Pr[\Delta = k] \;=\; \left(1 - \frac{N_T}{N}\right)^{k-1} \cdot \frac{N_T}{N}. \tag{6.44}$$

That is, after encountering an occurrence of T, we will first see $k-1$ documents in which the term does not appear (each with probability $1 - \frac{N_T}{N}$), followed by a document in which it does appear (probability $\frac{N_T}{N}$). This gives us a geometric distribution with $p = \frac{N_T}{N}$.

Figure 6.5 shows the (geometric) gap distribution for a hypothetical term with $p = 0.01$. It also shows the distribution that emerges if we group Δ-values into buckets according to their bit

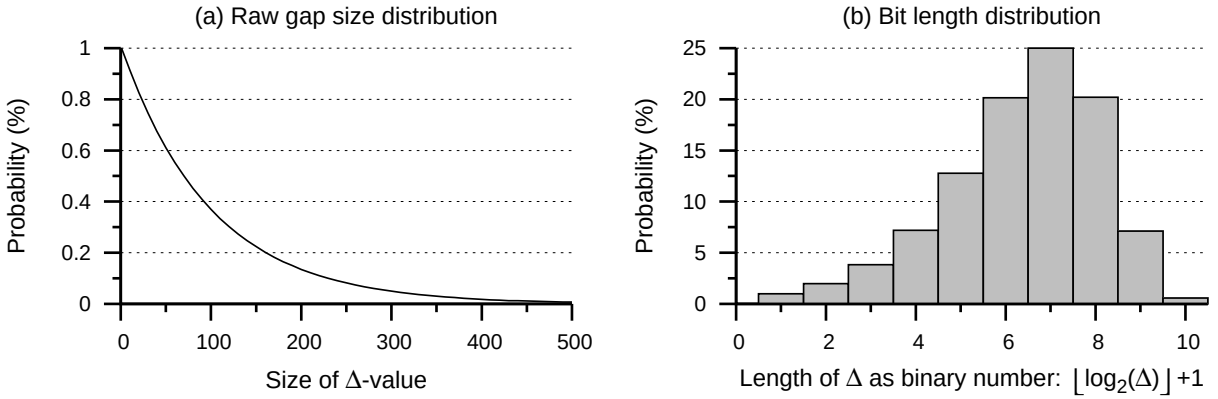

Figure 6.5 Distribution of Δ-gaps in a hypothetical postings list with $N_T/N = 0.01$ (i.e., term T appears in 1% of all documents). (a) distribution of raw gap sizes; (b) distribution of bit lengths (i.e., number of bits needed to encode each gap as a binary number).

length $\text{len}(k) = \lfloor \log_2(k) \rfloor + 1$, and compute the probability of each bucket. In the figure, 65% of all Δ-values fall into the range $6 \leq \text{len}(k) \leq 8$. This motivates the following encoding:

1. Choose an integer M, the *modulus*.
2. Split each Δ-value k into two components, the quotient $q(k)$ and the remainder $r(k)$:

$$q(k) = \lfloor (k-1)/M \rfloor, \quad r(k) = (k-1) \bmod M.$$

3. Encode k by writing $q(k)+1$ in unary representation, followed by $r(k)$ as a $\lfloor \log_2(M) \rfloor$-bit or $\lceil \log_2(M) \rceil$-bit binary number.

For the distribution shown in Figure 6.5 we might choose $M = 2^7$. Few Δ-values are larger than 2^8, so the vast majority of the postings require less than 3 bits for the quotient $q(k)$. Conversely, few Δ-values are smaller than 2^5, which implies that only a small portion of the 7 bits allocated for each remainder $r(k)$ is wasted.

The general version of the encoding described above, with an arbitrary modulus M, is known as *Golomb coding*, after its inventor, Solomon Golomb (1966). The version in which M is a power of 2 is called *Rice coding*, also after its inventor, Robert Rice (1971).[5]

[5] Strictly speaking, Rice did not invent Rice codes — they are a subset of Golomb codes, and Golomb's research was published half a decade before Rice's. However, Rice's work made Golomb codes attractive for practical applications, which is why Rice is often co-credited for this family of codes.

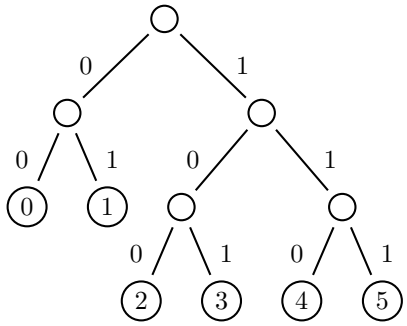

Figure 6.6 Codewords of the remainders $0 \leq r(k) < 6$, for the Golomb code with parameter $M = 6$.

Let us look at a concrete example. Suppose we have chosen the modulus $M = 2^7$, and we want to encode a gap of size $k = 345$. Then the resulting Rice codeword is

$$\text{Rice}_{M=2^7}(345) \;=\; \overline{001}\;1011000,$$

since $q(345) = \lfloor (345-1)/2^7 \rfloor = 2$, and $r(345) = (345-1) \bmod 2^7 = 88$.

Decoding a given codeword is straightforward. For each posting in the original list, we look for the first $\overline{1}$ bit in the code sequence. This gives us the value of $q(k)$. We then remove the first $q(k)+1$ bits from the bit sequence, extract the next λ bits (for $\lambda = \lceil \log_2(M) \rceil$), thus obtaining $r(k)$, and compute k as

$$k \;=\; q(k) \cdot 2^{\lambda} + r(k) + 1. \tag{6.45}$$

For efficiency, the multiplication with 2^{λ} is usually implemented as a bit shift.

Unfortunately, the simple encoding/decoding procedure described above works only for Rice codes and cannot be applied to general Golomb codes where M is not a power of 2. Since we allocate $\lceil \log_2(M) \rceil$ bits to each remainder $r(k)$, but there are only $M < 2^{\lceil \log_2(M) \rceil}$ possible remainders, some parts of the code space would be left unused, leading to a suboptimal code. In fact, if we used $\lceil \log_2(M) \rceil$ bits for each remainder regardless of the value of M, the resulting Golomb code would always be inferior to the corresponding Rice code with parameter $M' = 2^{\lceil \log_2(M) \rceil}$.

The solution is to reclaim the unused part of the code space by encoding some of the remainders using $\lfloor \log_2(M) \rfloor$ bits and others using $\lceil \log_2(M) \rceil$ bits. Which remainders should receive the shorter codewords, and which the longer ones? Since Golomb coding is based on the assumption that the Δ-values follow a geometric distribution (Equation 6.44), we expect that smaller gaps have a higher probability of occurrence than larger ones. The codewords for the remainders $r(k)$ are therefore assigned in the following way:

- Values from the interval $[0, 2^{\lceil \log_2(M) \rceil} - M - 1]$ receive $\lfloor \log_2(M) \rfloor$-bit codewords.
- Values from the interval $[2^{\lceil \log_2(M) \rceil} - M, M - 1]$ receive $\lceil \log_2(M) \rceil$-bit codewords.

Table 6.4 Encoding positive integers (e.g., Δ-gaps) using parameterized Golomb/Rice codes. The first part of each codeword corresponds to the quotient $q(k)$. The second part corresponds to the remainder $r(k)$.

Integer	Golomb Codes			Rice Codes	
	$M = 3$	$M = 6$	$M = 7$	$M = 4$	$M = 8$
1	1 0	1 00	1 00	1 00	1 000
2	1 10	1 01	1 010	1 01	1 001
3	1 11	1 100	1 011	1 10	1 010
4	01 0	1 101	1 100	1 11	1 011
5	01 10	1 110	1 101	01 00	1 100
6	01 11	1 111	1 110	01 01	1 101
7	001 0	01 00	1 111	01 10	1 110
8	001 10	01 01	01 00	01 11	1 111
9	001 11	01 100	01 010	001 00	01 000
31	00000000001 0	000001 00	00001 011	00000001 10	0001 110

Figure 6.6 visualizes this mechanism for the Golomb code with modulus $M = 6$. The remainders $0 \leq r(k) < 2$ are assigned codewords of length $\lfloor \log_2(6) \rfloor = 2$; the rest are assigned codewords of length $\lceil \log_2(6) \rceil = 3$. Looking at this approach from a slightly different perspective, we may say that the codewords for the M different remainders are assigned according to a canonical Huffman code (see Figure 6.3 on page 184). Table 6.4 shows Golomb/Rice codewords for various positive integers k and various parameter values M.

Compared with the relatively simple Rice codes, Golomb coding achieves slightly better compression rates but is a bit more complicated. While the $r(k)$-codewords encountered by a Rice decoder are all of the same length and can be decoded using simple bit shifts, a Golomb decoder needs to distinguish between $r(k)$-codewords of different lengths (leading to branch mispredictions) and needs to perform relatively costly integer multiplication operations, since M is not a power of 2. As a consequence, Rice decoders are typically between 20% and 40% faster than Golomb decoders.

Finding the optimal Golomb/Rice parameter value

One question that we have not yet addressed is how we should choose the modulus M so as to minimize the average codeword length. Recall from Section 6.2.1 that a code $\mathcal{C}$ is optimal with respect to a given probability distribution $\mathcal{M}$ if the relationship

$$|\mathcal{C}(\sigma_1)| \;=\; |\mathcal{C}(\sigma_2)| + 1, \tag{6.46}$$

for two symbols σ_1 and σ_2, implies

$$\mathcal{M}(\sigma_1) \;=\; \frac{1}{2} \cdot \mathcal{M}(\sigma_2). \tag{6.47}$$

We know that the Golomb codeword for the integer $k + M$ is 1 bit longer than the codeword for the integer k (because $q(k + M) = q(k) + 1$). Therefore, the optimal parameter value M^* should satisfy the following equation:

$$\begin{aligned} \Pr[\Delta = k + M^*] = \frac{1}{2} \cdot \Pr[\Delta = k] \quad &\Leftrightarrow \quad (1 - N_T/N)^{k+M^*-1} = \frac{1}{2} \cdot (1 - N_T/N)^{k-1} \\ &\Leftrightarrow \quad M^* = \frac{-\log(2)}{\log(1 - N_T/N)}. \end{aligned} \tag{6.48}$$

Unfortunately, M^* is usually not an integer. For Rice coding, we will have to choose between $M = 2^{\lfloor \log_2(M^*) \rfloor}$ and $M = 2^{\lceil \log_2(M^*) \rceil}$. For Golomb coding, we may choose between $M = \lfloor M^* \rfloor$ and $M = \lceil M^* \rceil$. In general, it is not clear, which one is the better choice. Sometimes one produces the better code, sometimes the other does. Gallager and van Voorhis (1975) proved that

$$M_{opt} \;=\; \left\lceil \frac{\log(2 - N_T/N)}{-\log(1 - N_T/N)} \right\rceil \tag{6.49}$$

always leads to the optimal code.

As an example, consider a term T that appears in 50% of all documents. We have $N_T/N = 0.5$, and thus

$$M_{opt} \;=\; \lceil -\log(1.5)/\log(0.5) \rceil \approx \lceil 0.585/1.0 \rceil \;=\; 1. \tag{6.50}$$

The optimal Golomb/Rice code in this case, perhaps as expected, turns out to be the unary code.

Huffman codes: LLRUN

If the gaps in a given list do not follow a geometric distribution — for example, because the document independence assumption does not hold or because the list is not a simple docid list — then Golomb codes may not lead to very good results.

We already know a compression method that can be used for lists with arbitrary distribution and that yields optimal compression rates: Huffman coding (see Section 6.2.2). Unfortunately, it is difficult to apply Huffman's method directly to a given sequence of Δ-values. The difficulty stems from the fact that the set of distinct gaps in a typical postings list has about the same size as the list itself. For example, the term "aquarium" appears in 149 documents in the TREC45 collection. Its docid list consists of 147 different Δ-values. Encoding this list with Huffman is unlikely to decrease the size of the list very much, because the 147 different Δ-values still need to be stored in the preamble (i.e., the Huffman tree) for the decoder to know which Δ-value a particular codeword in the Huffman-encoded gap sequence corresponds to.

Instead of applying Huffman coding to the gap values directly, it is possible to group gaps of similar size into buckets and have all gaps in the same bucket share a codeword in the Huffman code. For example, under the assumption that all gaps from the interval $[2^j, 2^{j+1} - 1]$ have approximately the same probability, we could create buckets $B_0, B_1, \ldots$ with $B_j = [2^j, 2^{j+1} - 1]$. All Δ-values from the same bucket B_j then share the same Huffman codeword w_j. Their encoded representation is w_j, followed by the j-bit representation of the respective gap value as a binary number (omitting the leading $\overline{1}$ bit that is implicit from w_j).

The resulting compression method is called *LLRUN* and is due to Fraenkel and Klein (1985). It is quite similar to Elias's γ method, except that the selector value (the integer j that defines the bucket in which a given Δ-value lies) is not encoded in unary, but according to a minimum-redundancy Huffman code. LLRUN's advantage over a naïve application of Huffman's method is that the bucketing scheme dramatically decreases the size of the symbol set for which the Huffman tree is created. Δ-values, even in a schema-independent index, are rarely greater than 2^{40}. Thus, the corresponding Huffman tree will have at most 40 leaves. With a canonical, length-limited Huffman code and a limit of $L = 15$ bits per codeword (see Section 6.2.2 for details on length-limited Huffman codes), the code can then be described in $4 \times 40 = 160$ bits. Moreover, the assumption that gaps from the same bucket B_j appear with approximately the same probability is usually reflected by the data found in the index, except for very small values of j. Hence, compression effectiveness is not compromised very much by grouping gaps of similar size into buckets and having them share the same Huffman codeword.

6.3.3 Context-Aware Compression Methods

The methods discussed so far all treat the gaps in a postings list independently of each other. In Section 6.1, we saw that the effectiveness of a compression method can sometimes be improved by taking into account the context of the symbol that is to be encoded. The same applies here. Sometimes consecutive occurrences of the same term form clusters (many occurrences of the same term within a small amount of text). The corresponding postings are very close to each other, and the Δ-values are small. A compression method that is able to properly react to this phenomenon can be expected to achieve better compression rates than a method that is not.

Huffman codes: Finite-context LLRUN

It is straightforward to adjust the LLRUN method from above so that it takes into account the previous Δ-value when encoding the current one. Instead of using a zero-order model, as done in the original version of LLRUN, we can use a first-order model and have the codeword w_j for the current gap's selector value depend on the value of the previous gap's selector, using perhaps 40 different Huffman trees (assuming that no Δ-value is bigger than 2^{40}), one for each possible predecessor value.

The drawback of this method is that it would require the encoder to transmit the description of 40 different Huffman trees instead of just a single one, thus increasing the storage requirements

```
encodeLLRUN-2 (⟨L[1], ..., L[n]⟩, ϑ, output) ≡
1   let Δ(L) denote the list ⟨L[1], L[2] − L[1], ..., L[n] − L[n − 1]⟩
2   Δ_max ← max_i{Δ(L)[i]}
3   initialize the array bucketFrequencies[0..1][0..⌊log_2(Δ_max)⌋] to zero
4   c ← 0   // the context to be used for finite-context modeling
5   for i ← 1 to n do   // collect context-specific statistics
6      b ← ⌊log_2(Δ(L)[i])⌋   // the bucket of the current Δ-value
7      bucketFrequencies[c][b] ← bucketFrequencies[c][b] + 1
8      if b < ϑ then c ← 0 else c ← 1
9   for i ← 0 to 1 do   // build two Huffman trees, one for each context
10     T_i ← buildHuffmanTree(bucketFrequencies[i])
11  c ← 0   // reset the context
12  for i ← 1 to n do   // compress the postings
13     b ← ⌊log_2(Δ(L)[i])⌋
14     write b's Huffman codeword, according to the tree T_c, to output
15     write Δ(L)[i] to output, as a b-bit binary number (omitting the leading 1)
16     if b < ϑ then c ← 0 else c ← 1
17  return
```

Figure 6.7 Encoding algorithm for LLRUN-2, the finite-context variant of LLRUN with two different Huffman trees. The threshold parameter ϑ defines a binary partitioning of the possible contexts.

of the compressed list. In practice, however, using 40 different models has almost no advantage over using just a few, say two or three. We refer to this revised variant of the LLRUN method as LLRUN-k, where k is the number of different models (i.e., Huffman trees) employed by the encoder/decoder.

Figure 6.7 shows the encoding algorithm for LLRUN-2. The threshold parameter ϑ, provided explicitly in the figure, can be chosen automatically by the algorithm, by trying all possible values $0 < \vartheta < \log_2(\max_i\{\Delta(L)[i]\})$ and selecting the one that minimizes the compressed size of the list. Note that this does not require the encoder to actually compress the whole list $\Theta(\log(\max_i\{\Delta(L)[i]\})$ times but can be done by analyzing the frequencies with which Δ-values from each bucket B_j follow Δ-values from each other bucket $B_{j'}$. The time complexity of this analysis is $\Theta(\log(\max_i\{\Delta(L)[i]\})^2)$.

Interpolative coding

Another context-aware compression technique is the *interpolative coding* method invented by Moffat and Stuiver (2000). Like all other list compression methods, interpolative coding uses the fact that all postings in a given inverted list are stored in increasing order, but it does so in a slightly different way.

Consider the beginning of the postings list L for the term "example" in a docid index for the TREC45 collection:

$$L = \langle 2, 9, 12, 14, 19, 21, 31, 32, 33 \rangle .$$

```
encodeInterpolative (⟨L[1], ..., L[n]⟩, output) ≡
1   encodeGamma (n)
2   encodeGamma (L[1], output)
3   encodeGamma (L[n] − L[1], output)
4   encodeInterpolativeRecursively (⟨L[1], ..., L[n]⟩, output)

encodeInterpolativeRecursively (⟨L[1], ..., L[n]⟩, output) ≡
5   if n < 3 then
6     return
7   middle ← ⌈n/2⌉
8   firstPossible ← L[1] + (middle − 1)
9   lastPossible ← L[n] + (middle − n)
10  k ← ⌈log2(lastPossible − firstPossible + 1)⌉
11  write (L[middle] − firstPossible) to output, as a k-bit binary number
12  encodeInterpolativeRecursively (⟨L[1], ..., L[middle]⟩, output)
13  encodeInterpolativeRecursively (⟨L[middle], ..., L[n]⟩, output)
```

Figure 6.8 Compressing a postings list $\langle L[1], \ldots, L[n] \rangle$ using interpolative coding. The resulting bit sequence is written to *output*.

The interpolative method compresses this list by encoding its first element $L[1] = 2$ and its last element $L[9] = 33$ using some other method, such as γ coding. It then proceeds to encode $L[5] = 19$. However, when encoding $L[5]$, it exploits the fact that, at this point, $L[1]$ and $L[9]$ are already known to the decoder. Because all postings are stored in strictly increasing order, it is guaranteed that

$$2 = L[1] < L[2] < \ldots < L[5] < \ldots < L[8] < L[9] = 33.$$

Therefore, based on the information that the list contains nine elements, and based on the values of $L[1]$ and $L[9]$, we already know that $L[5]$ has to lie in the interval $[6, 29]$. As this interval contains no more than $2^5 = 32$ elements, $L[5]$ can be encoded using 5 bits. The method then proceeds recursively, encoding $L[3]$ using 4 bits (because, based on the values of $L[1]$ and $L[5]$, it has to be in the interval $[4, 17]$), encoding $L[2]$ using 4 bits (because it has to be in $[3, 11]$), and so on.

Figure 6.8 gives a more formal definition of the encoding procedure of the interpolative method. The bit sequence that results from compressing the list L according to interpolative coding is shown in Table 6.5. It is worth pointing out that $L[8] = 32$ can be encoded using 0 bits because $L[7] = 31$ and $L[9] = 33$ leave only one possible value for $L[8]$.

As in the case of Golomb coding, the interval defined by *firstPossible* and *lastPossible* in Figure 6.8 is rarely a power of 2. Thus, by encoding each possible value in the interval using k bits, some code space is wasted. As before, this deficiency can be cured by encoding $2^k - (\mathit{lastPossible} - \mathit{firstPossible} + 1)$ of all possible values using $k - 1$ bits and the remaining

Table 6.5 The result of applying interpolative coding to the first nine elements of the docid list for the term "example" (data taken from TREC45).

Postings (orig. order)	**Postings** (visitation order)	**Compressed Bit Sequence**	
$(n = 9)$	$(n = 9)$	0001001	(γ codeword for $n = 9$)
2	2	010	(γ codeword for 2)
9	33	000011111	(γ codeword for $31 = 33 - 2$)
12	19	01101	($13 = 19 - 6$ as 5-bit binary number)
14	12	1000	($8 = 12 - 4$ as 4-bit binary number)
19	9	0110	($6 = 9 - 3$ as 4-bit binary number)
21	14	001	($1 = 14 - 13$ as 3-bit binary number)
31	31	1010	($10 = 31 - 21$ as 4-bit binary number)
32	21	0001	($1 = 21 - 20$ as 4-bit binary number)
33	32		($0 = 32 - 32$ as 0-bit binary number)

values using k bits. For simplicity, this mechanism is not shown in Figure 6.8. The details can be found in the material on Golomb coding in Section 6.3.2.

Unlike in the case of Golomb codes, however, we do not know for sure which of the $(\mathit{lastPossible} - \mathit{firstPossible} + 1)$ possible values are more likely than the others. Hence, it is not clear which values should receive the k-bit codewords and which values should receive $(k-1)$-bit codewords. Moffat and Stuiver (2000) recommend giving the shorter codewords to the values in the middle of the interval $[\mathit{firstPossible}, \mathit{lastPossible}]$, except when the interval contains only a single posting (corresponding to $n = 3$ in the function **encodeInterpolativeRecursively**), in which case they recommend assigning the shorter codewords to values on both ends of the interval so as to exploit potential clustering effects. In experiments this strategy has been shown to save around 0.5 bits per posting on average.

6.3.4 Index Compression for High Query Performance

The rationale behind storing postings lists in compressed form is twofold. First, it decreases the storage requirements of the search engine's index. Second, as a side effect, it decreases the disk I/O overhead at query time and thus potentially improves query performance. The compression methods discussed in the previous sections have in common that they are targeting the first aspect, mainly ignoring the complexity of the decoding operations that need to be carried out at query time.

When choosing between two different compression methods A and B, aiming for optimal query performance, a good rule of thumb is to compare the decoding overhead of each method with the read performance of the storage medium that contains the index (e.g., the computer's hard drive). For example, a hard drive might deliver postings at a rate of 50 million bytes

(= 400 million bits) per second. If method A, compared to B, saves 1 bit per posting on average, and A's relative decoding overhead per posting, compared to that of method B, is less than 2.5 ns (i.e., the time needed to read a single bit from the hard disk), then A should be preferred over B. Conversely, if the relative overhead is more than 2.5 ns, then B should be preferred.

2.5 ns is not a lot of time, even for modern microprocessors. Depending on the clock frequency of the CPU, it is equivalent to 2–10 CPU cycles. Therefore, even if method A can save several bits per posting compared to method B, its decoding routine still needs to be extremely efficient in order to have an advantageous effect on query performance.

The above rule of thumb does not take into account the possibility that compressed postings can be decoded in parallel with ongoing disk I/O operations or that postings lists may be cached in memory, but it serves as a good starting point when comparing two compression methods in terms of their impact on the search engine's query performance. As we shall see in Section 6.3.6, the methods discussed so far do not necessarily lead to optimal query performance, as their decoding routines can be quite complex and time-consuming. Two methods that were specifically designed with high decoding throughput in mind are presented next.

Byte-aligned codes

Byte-aligned gap compression is one of the simplest compression methods available. Its popularity is in parts due to its simplicity, but mostly due to its high decoding performance. Consider the beginning of the postings list for the term "aligned", extracted from a docid index for the GOV2 collection:

$$\begin{aligned} L &= \langle\, 1624, 1650, 1876, 1972, \ldots \rangle\,, \\ \Delta(L) &= \langle\, 1624, 26, 226, 96, 384, \ldots \rangle\,. \end{aligned} \tag{6.51}$$

In order to avoid expensive bit-fiddling operations, we want to encode each Δ-value using an integral number of bytes. The easiest way to do this is to split the binary representation of each Δ-value into 7-bit chunks and prepend to each such chunk a *continuation flag* — a single bit that indicates whether the current chunk is the last one or whether there are more to follow. The resulting method is called *vByte* (for *variable-byte* coding). It is used in many applications, not only search engines. The vByte-encoded representation of the above docid list is

1 1011000 0 0001100 0 0011010 1 1100010 0 0000001 0 1100000 1 0000000 0 0000011 ...

(spaces inserted for better readability).

For example, the body of the first chunk ($\overline{1011000}$) is the representation of the integer 88 as a 7-bit binary number. The body of the second chunk ($\overline{0001100}$) is the integer 12 as a 7-bit binary number. The $\overline{0}$ at the beginning of the second chunk indicates that this is the end of the current codeword. The decoder, when it sees this, combines the two numbers into $88 + 12 \times 2^7 = 1624$, yielding the value of the first list element.

```
   encodeVByte (⟨L[1], ..., L[n]⟩, outputBuffer) ≡
1      previous ← 0
2      for i ← 1 to n do
3          delta ← L[i] − previous
4          while delta ≥ 128 do
5              outputBuffer.writeByte(128 + (delta & 127))
6              delta ← delta ≫ 7
7          outputBuffer.writeByte(delta)
8          previous ← L[i]
9      return

   decodeVByte (inputBuffer, ⟨L[1], ..., L[n]⟩) ≡
10     current ← 0
11     for i ← 1 to n do
12         shift ← 0
13         b ← inputBuffer.readByte()
14         while b ≥ 128 do
15             current ← current + ((b & 127) ≪ shift)
16             shift ← shift + 7
17             b ← inputBuffer.readByte()
18         current ← current + (b ≪ shift)
19         L[i] ← current
20     return
```

Figure 6.9 Encoding and decoding routine for vByte. Efficient multiplication and division operations are realized through bit-shift operations ("≪": left-shift; "≫": right-shift; "&": bit-wise AND).

The encoding and decoding routines of the vByte method are shown in Figure 6.9. When you look at the pseudo-code in the figure, you will notice that vByte is obviously not optimized for maximum compression. For example, it reserves the codeword $\overline{00000000}$ for a gap of size 0. Clearly, such a gap cannot exist, since the postings form a strictly monotonic sequence. Other compression methods (γ, LLRUN, ...) account for this fact by not assigning any codeword to a gap of size 0. With vByte, however, the situation is different, and it makes sense to leave the codeword unused. vByte's decoding routine is highly optimized and consumes only a few CPU cycles per posting. Increasing its complexity by adding more operations (even operations as simple as $+1/-1$) would jeopardize the speed advantage that vByte has over the other compression methods.

Word-aligned codes

Just as working with whole bytes is more efficient than operating on individual bits, accessing entire machine words is normally more efficient than fetching all its bytes separately when decoding a compressed postings list. Hence, we can expect to obtain faster decoding routines if we enforce that each posting in a given postings list is always stored using an integral number

Table 6.6 Word-aligned postings compression with Simple-9. After reserving 4 out of 32 bits for the selector value, there are 9 possible ways of dividing the remaining 28 bits into equal-size chunks.

Selector	0	1	2	3	4	5	6	7	9
Number of Δ's	1	2	3	4	5	7	9	14	28
Bits per Δ	28	14	9	7	5	4	3	2	1
Unused bits per word	0	0	1	0	3	0	1	0	0

of 16-bit, 32-bit, or 64-bit machine words. Unfortunately, doing so would defeat the purpose of index compression. If we encode each Δ-value as a 32-bit integer, we might as well choose an even simpler encoding and store each posting directly as an uncompressed 32-bit integer.

The following idea leads us out of this dilemma: Instead of storing each Δ-value in a separate 32-bit machine word, maybe we can store several consecutive values, say n of them, in a single word. For example, whenever the encoder sees three consecutive gaps $k_1 \ldots k_3$, such that $k_i \leq 2^9$ for $1 \leq i \leq 3$, then it may store all three of them in a single machine word, assigning 9 bits within the word to each k_i, thus consuming 27 bits in total and leaving 5 unused bits that can be used for other purposes.

Anh and Moffat (2005) discuss several word-aligned encoding methods that are based on ideas similar to those described above. Their simplest method is called *Simple-9*. It works by inspecting the next few Δ-values in a postings sequence, trying to squeeze as many of them into a 32-bit machine word as possible. Of course, the decoder, when it sees a 32-bit word, supposedly containing a sequence of Δ-values, does not know how many bits in this machine word were reserved for each Δ. Therefore, Simple-9 reserves the first 4 bits in each word for a selector: a 4-bit integer that informs the decoder about the split used within the current word. For the remaining 28 bits in the same word, there are nine different ways of dividing them into chunks of equal size (shown in Table 6.6). This is how the method received its name.

For the same Δ-sequence as before (docid list for the term "aligned" in the GOV2 collection), the corresponding code sequence now is

$$0001\ 00011001011000\ 00000000011001\ 0010\ 011100001\ 001011111\ 101111111\ \mathrm{U}, \qquad (6.52)$$

where $\overline{0010}$ (=2) is the selector used for the second machine word and $\overline{011100001}$ is the integer 225 (= 1876 − 1650 − 1) as a 9-bit binary number. "U" represents an unused bit.

For the example sequence, Simple-9 does not allow a more compact representation than vByte. This is because the term "aligned" is relatively rare and its postings list has rather large gaps. For frequent terms with small gaps, however, Simple-9 has a clear advantage over vByte because it is able to encode a postings list using as little as 1 bit per gap (plus some overhead for the selectors). Its decoding performance, on the other hand, is almost as high as that of vByte, because the shift-mask operations necessary to extract the n $\lfloor \frac{28}{n} \rfloor$-bit integers from a given machine word can be executed very efficiently.

Efficiently decoding unaligned codes

Even though the unaligned methods from the previous sections, unlike vByte and Simple-9, were not designed with the specific target of achieving high decoding performance, most of them still allow rather efficient decoding. However, in order to attain this goal, it is imperative that expensive bit-by-bit decoding operations be avoided whenever possible.

We illustrate this through an example. Consider the γ code from Section 6.3.1. After transforming the postings list

$$L = \langle 7, 11, 24, 26, 33, 47 \rangle \tag{6.53}$$

into an equivalent sequence of Δ-values

$$\Delta(L) = \langle 7, 4, 13, 2, 7, 14 \rangle, \tag{6.54}$$

the bit sequence produced by the γ coder is

$$\gamma(L) = \overline{001\ 11\ 001\ 00\ 0001\ 101\ 01\ 0\ 001\ 11\ 0001\ 110}. \tag{6.55}$$

In order to determine the bit length of each codeword in this sequence, the decoder repeatedly needs to find the first occurrence of a $\overline{1}$ bit, indicating the end of the codeword's selector component. It could do this by processing $\gamma(L)$ in a bit-by-bit fashion, inspecting every bit individually and testing whether it is $\overline{0}$ or $\overline{1}$. However, such a decoding procedure would not be very efficient. Not only would each code bit require at least one CPU operation, but the conditional jumps associated with the decision "*Is the current bit a* $\overline{0}$ *bit?*" are likely to result in a large number of branch mispredictions, which will flush the CPU's execution pipeline and slow down the decoding process dramatically (see the appendix for a brief introduction to the concepts of high-performance computing; or see Patterson and Hennessy (2009) for more detailed coverage of the topic).

Suppose we know that no element of $\Delta(L)$ is larger than $2^4 - 1$ (as is the case in the above example). Then this implies that none of the selectors in the code sequence are longer than 4 bits. Hence, we may construct a table T containing $2^4 = 16$ elements that tells us the position of the first $\overline{1}$ bit in the sequence, given the next 4 bits:

$$\begin{array}{llll} T[\,\overline{0000}\,] = 5, & T[\,\overline{0001}\,] = 4, & T[\,\overline{0010}\,] = 3, & T[\,\overline{0011}\,] = 3, \\ T[\,\overline{0100}\,] = 2, & T[\,\overline{0101}\,] = 2, & T[\,\overline{0110}\,] = 2, & T[\,\overline{0111}\,] = 2, \\ T[\,\overline{1000}\,] = 1, & T[\,\overline{1001}\,] = 1, & T[\,\overline{1010}\,] = 1, & T[\,\overline{1011}\,] = 1, \\ T[\,\overline{1100}\,] = 1, & T[\,\overline{1101}\,] = 1, & T[\,\overline{1110}\,] = 1, & T[\,\overline{1111}\,] = 1 \end{array} \tag{6.56}$$

(where "5" indicates "not among the first 4 bits").

We can use this table to implement an efficient decoding routine for the γ code, as shown in Figure 6.10. Instead of inspecting each bit individually, the algorithm shown in the figure loads them into a 64-bit *bit buffer*, 8 bits at a time, and uses the information stored in the lookup table T, processing up to 4 bits in a single operation. The general approach followed

```
   decodeGamma (inputBuffer, T[0..2^k − 1], ⟨L[1], ..., L[n]⟩) ≡
1     current ← 0
2     bitBuffer ← 0
3     bitsInBuffer ← 0
4     for i ← 1 to n do
5        while bitsInBuffer + 8 ≤ 64 do
6           bitBuffer ← bitBuffer + (inputBuffer.readByte() ≪ bitsInBuffer)
7           bitsInBuffer ← bitsInBuffer + 8
8        codeLength ← T[bitBuffer & (2^k − 1)]
9        bitBuffer ← bitBuffer ≫ (codeLength − 1)
10       mask ← (1 ≪ codeLength) − 1
11       current ← current + (bitBuffer & mask)
12       bitBuffer ← bitBuffer ≫ codeLength
13       bitsInBuffer ← bitsInBuffer − 2 × codeLength − 1
14       L[i] ← current
15    return
```

Figure 6.10 Table-driven γ decoder. Bit-by-bit decoding operations are avoided through the use of a bit buffer and a lookup table T of size 2^k that is used for determining the length of the current codeword. Efficient multiplication and division operations are realized through bit-shift operations ("≪": left shift; "≫": right shift; "&": bitwise AND).

by the algorithm is referred to as *table-driven decoding*. It can be applied to γ, δ, Golomb, and Rice codes as well as the Huffman-based LLRUN method. In the case of LLRUN, however, the contents of the table T depend on the specific code chosen by the encoding procedure. Thus, the decoder needs to process the Huffman tree in order to initialize the lookup table before it can start its decoding operations.

Table-driven decoding is the reason why length-limited Huffman codes (see Section 6.2.2) are so important for high-performance query processing: If we know that no codeword is longer than k bits, then the decoding routine requires only a single table lookup (in a table of size 2^k) to figure out the next codeword in the given bit sequence. This is substantially faster than explicitly following paths in the Huffman tree in a bit-by-bit fashion.

6.3.5 Compression Effectiveness

Let us look at the compression rates achieved by the various methods discussed so far. Table 6.7 gives an overview of these rates, measured in bits per posting, on different types of lists (docids, TF values, within-document positions, and schema-independent) for the three example collections used in this book. For the Shakespeare collection, which does not contain any real *documents*, we decided to treat each `<SPEECH>` $\cdots$ `</SPEECH>` XML element as a document for the purpose of the compression experiments.

The methods referred to in the table are: γ and δ coding (Section 6.3.1); Golomb/Rice and the Huffman-based LLRUN (Section 6.3.2); interpolative coding (Section 6.3.3); and the two

Table 6.7 Compression effectiveness of various compression methods for postings lists, evaluated on three different text collections. All numbers represent compression effectiveness, measured in bits per posting. Bold numbers indicate the best result in each row.

	List Type	γ	δ	Golomb	Rice	LLRUN	Interp.	vByte	S-9
Shakesp.	Document IDs	8.02	7.44	6.48	6.50	6.18	**6.18**	9.96	7.58
	Term frequencies	1.95	2.08	2.14	2.14	1.98	**1.70**	8.40	3.09
	Within-doc pos.	8.71	8.68	6.53	6.53	**6.29**	6.77	8.75	7.52
	Schema-indep.	15.13	13.16	10.54	10.54	**10.17**	10.49	12.51	12.75
TREC45	Document IDs	7.23	6.78	5.97	6.04	5.65	**5.52**	9.45	7.09
	Term frequencies	2.07	2.27	1.99	2.00	1.95	**1.71**	8.14	2.77
	Within-doc pos.	12.69	11.51	8.84	8.89	**8.60**	8.83	11.42	10.89
	Schema-indep.	17.03	14.19	12.24	12.38	**11.31**	11.54	13.71	15.37
GOV2	Document IDs	8.02	7.47	5.98	6.07	5.98	**5.97**	9.54	7.46
	Term frequencies	2.74	2.99	2.78	2.81	2.56	**2.53**	8.12	3.67
	Within-doc pos.	11.47	10.45	9.00	9.13	**8.02**	8.34	10.66	10.10
	Schema-indep.	13.69	11.81	11.73	11.96	**9.45**	9.98	11.91	n/a

performance-oriented methods vByte and Simple-9 (Section 6.3.4). In all cases, postings lists were split into small chunks of about 16,000 postings each before applying compression. For the parametric methods (Golomb, Rice, and LLRUN), compression parameters were chosen separately for each such chunk (*local parameterization*), thus allowing these methods to take into account small distributional variations between different parts of the same postings list — an ability that will come in handy in Section 6.3.7.

As you can see from the table, interpolative coding consistently leads to the best results for docids and TF values on all three text collections, closely followed by LLRUN and Golomb/Rice. Its main advantage over the other methods is its ability to encode postings using less than 1 bit on average if the gaps are predominantly of size 1. This is the case for the docid lists of the most frequent terms (such as "the", which appears in 80% of the documents in GOV2 and 99% of the documents in TREC45), but also for lists of TF values: When picking a random document D and a random term T that appears in D, chances are that T appears only a single time.

For the other list types (*within-document positions* and *schema-independent*), the methods' roles are reversed, with LLRUN taking the lead, followed by interpolative coding and Golomb/Rice. The reason why Golomb/Rice codes perform so poorly on these two list types is that the basic assumption on which Golomb coding is based (i.e., that Δ-gaps follow a geometric distribution) does not hold for them.

Under the document independence assumption, we know that the gaps in a given docid list roughly follow a distribution of the form

$$\Pr[\Delta = k] \;=\; (1-p)^{k-1} \cdot p \tag{6.57}$$

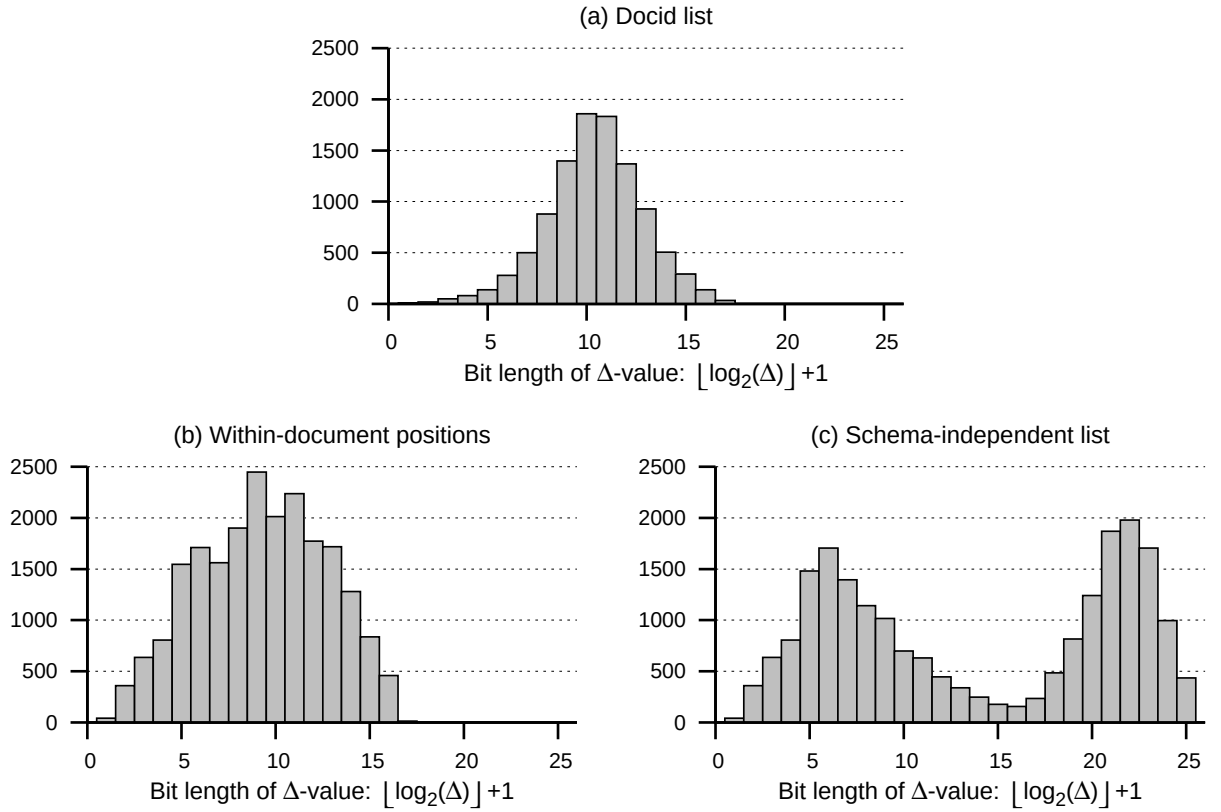

Figure 6.11 Gap distributions in different postings lists for the term "huffman" in the GOV2 collection. Vertical axis: Number of Δ-values of the given size. Only the gaps in the docid list follow a geometric distribution.

(see Equation 6.43). For within-document positions and schema-independent lists, this assumption does not hold. If a random term T appears toward the beginning of a document, then it is more likely to appear again in the same document than if we first see it toward the end of the document. Therefore, the elements of these lists are not independent of each other, and the gaps do not follow a geometric distribution.

The occurrence pattern of the term "huffman" in the GOV2 collection is a good example of this phenomenon. Figure 6.11 shows the gap distribution exhibited by the term's docid list, its document-centric positional list, and its schema-independent list. After a logarithmic transformation, putting all Δ-gaps of the same length $\text{len}(\Delta) = \lfloor \log_2(\Delta) \rfloor + 1$ into the same bucket, plot (a) shows the curve that is characteristic for a geometric distribution, with a clear peak around $\text{len}(\Delta) \approx 10$. Plot (b), for the list of within-document positions, also has a peak, but it is not as clearly distinguished as in (a). Finally, the schema-independent list in plot (c) does not follow a geometric distribution at all. Instead, we can see two peaks: one corresponding to two

Table 6.8 Zero-order LLRUN versus first-order LLRUN. Noteworthy improvements are achieved only for the two positional indices (within-document positions and schema-independent) built from GOV2. All other indices either are too small to absorb the storage overhead (due to the more complex model) or their list elements do not exhibit much interdependency.

List type	TREC45			GOV2		
	LLRUN	LLRUN-2	LLRUN-4	LLRUN	LLRUN-2	LLRUN-4
Within-doc positions	8.60	8.57	8.57	8.02	7.88	7.82
Schema-independent	11.31	11.28	11.28	9.45	9.29	9.23

or more occurrences of "huffman" in the same document ($\text{len}(\Delta) \approx 6$), the other corresponding to consecutive occurrences in different documents ($\text{len}(\Delta) \approx 22$). Obviously, if a method is tailored toward geometric distributions, it will not do very well on a postings list that follows a clearly nongeometric distribution, such as the one shown in Figure 6.11(c). This is why LLRUN's performance on positional postings lists (document-centric or schema-independent) is so much better than that of Golomb/Rice — up to 2.5 bits per posting.

A method that was specifically designed with the interdependencies between consecutive postings in mind is the LLRUN-k method (Section 6.3.3). LLRUN-k is very similar to LLRUN, except that it utilizes a first-order compression model, materialized in k different Huffman trees instead of just a single one. Table 6.8 shows the compression rates attained by LLRUN-k in comparison with the original, zero-order LLRUN method.

For the two smaller collections, the method does not achieve any substantial savings, because the slightly better coding efficiency is compensated for by the larger that precedes the actual codeword sequence. For the two positional indices built from GOV2, however, we do in fact see a small reduction of the storage requirements of the inverted lists. In the case of within-document positions, LLRUN-2/LLRUN-4 leads to a reduction of 0.14/0.20 bits per posting (-1.7%/-2.5%); for the schema-independent index, it saves 0.16/0.22 bits per posting (-1.7%/-2.3%). Whether these savings are worthwhile depends on the application. For search engines they are usually not worthwhile, because the 2% size reduction is likely to be outweighed by the more complex decoding procedure.

6.3.6 Decoding Performance

As pointed out in Section 6.3.4, when motivating byte- and word-aligned compression methods, reducing the space consumed by the inverted index is only one reason for applying compression techniques. Another, and probably more important, reason is that a smaller index may also lead to better query performance, at least if the postings lists are stored on disk. As a rule of thumb, an index compression method may be expected to improve query performance if the decoding overhead (measured in nanoseconds per posting) is lower than the disk I/O time saved by storing postings in compressed form.

Table 6.9 Cumulative disk I/O and list decompression overhead for a docid index built from GOV2. The cumulative overhead is calculated based on a sequential disk throughput of 87 MB/second ($\widehat{=}$ 1.37 nanoseconds per bit).

Compression Method	**Compression** (bits per docid)	**Decoding** (ns per docid)	**Cumulative Overhead** (decoding + disk I/O)
γ	4.94	7.67	14.44 ns
Golomb	4.10	10.82	16.44 ns
Rice	4.13	6.45	12.11 ns
LLRUN	4.09	7.04	12.64 ns
Interpolative	4.19	27.21	32.95 ns
vByte	8.77	1.35	13.36 ns
Simple-9	5.32	2.76	10.05 ns
Uncompressed (32-bit)	32.00	0.00	43.84 ns
Uncompressed (64-bit)	64.00	0.00	87.68 ns

Table 6.9 shows compression effectiveness (for docid lists), decoding efficiency, and cumulative overhead for various compression methods. All values were obtained by running the 10,000 queries from the efficiency task of the TREC Terabyte Track 2006 and decompressing the postings lists for all query terms (ignoring stopwords). Note that the compression effectiveness numbers shown in the table ("bits per docid") are quite different from those in the previous tables, as they do not refer to the entire index but only to terms that appear in the queries. This difference is expected: Terms that are frequent in the collection tend to be frequent in the query stream, too. Since frequent terms, compared to infrequent ones, have postings lists with smaller Δ-gaps, the compression rates shown in Table 6.9 are better than those in Table 6.7.

To compute the cumulative decoding + disk I/O overhead of the search engine, we assume that the hard drive can deliver postings data at a rate of 87 MB per second ($\widehat{=}$ 1.37 ns per bit; see the appendix). vByte, for instance, requires 8.77 bits (on average) per compressed docid value, which translates into a disk I/O overhead of $8.77 \times 1.37 = 12.01$ ns per posting. In combination with its decoding overhead of 1.35 ns per posting, this results in a cumulative overhead of 13.36 ns per posting.

Most methods play in the same ballpark, exhibiting a cumulative overhead of 10–15 ns per posting. Interpolative coding with its rather complex recursive decoding routine is an outlier, requiring a total of more than 30 ns per posting. In comparison with the other techniques, the byte-aligned vByte proves to be surprisingly mediocre. It is outperformed by three other methods, including the relatively complex LLRUN algorithm. Simple-9, combining competitive compression rates with very low decoding overhead, performed best in the experiment.

Of course, the numbers shown in the table are only representative of docid indices. For a positional or a schema-independent index, for example, the results will look more favorable for vByte. Moreover, the validity of the simplistic performance model applied here (calculating

the total overhead as a sum of disk I/O and decoding operations) may be questioned in the presence of caching and background I/O. Nonetheless, the general message is clear: Compared to an uncompressed index (with either 32-bit or 64-bit postings), every compression method leads to improved query performance.

6.3.7 Document Reordering

So far, when discussing various ways of compressing the information stored in the index's postings lists, we have always assumed that the actual information stored in the postings lists is fixed and may not be altered. Of course, this is an oversimplification. In reality, a document is not represented by a numerical identifier but by a file name, a URL, or some other textual description of its origin. In order to translate the search results back into their original context, which needs to be done before they can be presented to the user, each numerical document identifier must be transformed into a textual description of the document's location or context. This transformation is usually realized with the help of a data structure referred to as the *document map* (see Section 4.1).

This implies that the docids themselves are arbitrary and do not possess any inherent semantic value. We may therefore reassign numerical identifiers as we wish, as long as we make sure that all changes are properly reflected by the document map. We could, for instance, try to reassign document IDs in such a way that index compressibility is maximized. This process is usually referred to as *document reordering*. To see its potential, consider the hypothetical docid list

$$L = \langle 4, 21, 33, 40, 66, 90 \rangle \tag{6.58}$$

with γ-code representation

$$\overline{001\ 00\ 00001\ 0001\ 0001\ 100\ 001\ 11\ 00001\ 1010\ 000001\ 00010} \tag{6.59}$$

(46 bits). If we were able to reassign document identifiers in such a way that the list became

$$L = \langle 4, 6, 7, 45, 51, 120 \rangle \tag{6.60}$$

instead, then this would lead to the γ-code representation

$$\overline{001\ 00\ 01\ 0\ 1\ 000001\ 00110\ 001\ 10\ 0000001\ 000101} \tag{6.61}$$

(36 bits), reducing the list's storage requirements by 22%.

It is clear from the example that we don't care so much about the average magnitude of the gaps in a postings list (which is 17.2 in Equation 6.58 but 23.2 in Equation 6.60) but are instead interested in minimizing the average codeword length, which is a value closely related to the

logarithm of the length of each gap. In the above case, the value of the sum

$$\sum_{i=1}^{5} \lceil \log_2(L[i+1] - L[i]) \rceil \tag{6.62}$$

is reduced from 22 to 18 (−18%).

In reality, of course, things are not that easy. By reassigning document identifiers so as to reduce the storage requirements of one postings list, we may increase those of another list. Finding the optimal assignment of docids, the one that minimizes the total compressed size of the index, is believed to be computationally intractable (i.e., NP-complete). Fortunately, there are a number of document reordering heuristics that, despite computing a suboptimal docid assignment, typically lead to considerably improved compression effectiveness. Many of those heuristics are based on clustering algorithms that require substantial implementation effort and significant computational resources. Here, we focus on two particular approaches that are extremely easy to implement, require minimal computational resources, and still lead to pretty good results:

- The first method reorders the documents in a text collection based on the number of distinct terms contained in each document. The idea is that two documents that each contain a large number of distinct terms are more likely to share terms than are a document with many distinct terms and a document with few distinct terms. Therefore, by assigning docids so that documents with many terms are close together, we may expect a greater clustering effect than by assigning docids at random.
- The second method assumes that the documents have been crawled from the Web (or maybe a corporate Intranet). It reassigns docids in lexicographical order of URL. The idea here is that two documents from the same Web server (or maybe even from the same directory on that server) are more likely to share common terms than two random documents from unrelated locations on the Internet.

The impact that these two document reordering heuristics have on the effectiveness of various index compression methods is shown in Table 6.10. The row labeled "Original" refers to the official ordering of the documents in the GOV2 collections and represents the order in which the documents were crawled from the Web. The other rows represent random ordering, documents sorted by number of unique terms, and documents sorted by URL.

For the docid lists, we can see that document reordering improves the effectiveness of every compression method shown in the table. The magnitude of the effect varies between the methods because some methods (e.g., interpolative coding) can more easily adapt to the new distribution than other methods (e.g., Golomb coding). But the general trend is the same for all of them. In some cases the effect is quite dramatic. With interpolative coding, for example, the average space consumption can be reduced by more than 50%, from 5.97 to 2.84 bits per posting. This is mainly because of the method's ability to encode sequences of very small gaps using less than

Table 6.10 The effect of document reordering on compression effectiveness (in bits per list entry). All data are taken from GOV2.

	Doc. Ordering	γ	Golomb	Rice	LLRUN	Interp.	vByte	S-9
Doc. IDs	Original	8.02	6.04	6.07	5.98	5.97	9.54	7.46
	Random	8.80	6.26	6.28	6.35	6.45	9.86	8.08
	Term count	5.81	5.08	5.15	4.60	4.26	9.18	5.63
	URL	3.88	5.10	5.25	3.10	2.84	8.76	4.18
TF values	Original	2.74	2.78	2.81	2.56	2.53	8.12	3.67
	Random	2.74	2.86	2.90	2.60	2.61	8.12	3.87
	Term count	2.74	2.59	2.61	2.38	2.31	8.12	3.20
	URL	2.74	2.63	2.65	2.14	2.16	8.12	2.83

1 bit on average. The other methods need at least 1 bit for every posting (although, by means of blocking (see Section 6.2.3), LLRUN could easily be modified so that it requires less than 1 bit per posting for such sequences).

What might be a little surprising is that document reordering improves not only the compressibility of docids, but also that of TF values. The reason for this unexpected phenomenon is that, in the experiment, each inverted list was split into small chunks containing about 16,000 postings each, and compression was applied to each chunk individually instead of to the list as a whole. The initial motivation for splitting the inverted lists into smaller chunks was to allow efficient random access into each postings list, which would not be possible if the entire list had been compressed as one atomic unit (see Section 4.3). However, the welcome side effect of this procedure is that the compression method can choose different parameter values (i.e., apply different compression models) for different parts of the same list (except for γ coding and vByte, which are nonparametric). Because the basic idea behind document reordering is to assign numerically close document identifiers to documents that are similar in content, different parts of the same inverted lists will have different properties, which leads to better overall compression rates. When using LLRUN, for example, the average number of bits per TF value decreases from 2.56 to 2.14 (-16%).

6.4 Compressing the Dictionary

In Chapter 4, we showed that the dictionary for a text collection can be rather large. Not quite as large as the postings lists, but potentially still too large to fit into main memory. In addition, even if the dictionary is small enough to fit into RAM, decreasing its size may be a good idea because it allows the search engine to make better use of its memory resources — for example, by caching frequently accessed postings lists or by caching search results for frequent queries.

The goal here is to reduce the size of the dictionary as much as possible, while still providing low-latency access to dictionary entries and postings lists. Because dictionary lookups represent one of the main bottlenecks in index construction, and because the merge-based indexing algorithm presented in Section 4.5.3 is largely independent of the amount of main memory available to the indexing process, there is usually no reason to apply dictionary compression at indexing time. We therefore restrict ourselves to the discussion of compression techniques that can be used for the query-time dictionary, after the index has been built.

Recall from Figure 4.1 (page 107) that a sort-based in-memory dictionary is essentially an array of integers (the *primary* array) in which each integer is a pointer to a variable-size term descriptor (i.e., dictionary entry), an element of the *secondary* array. Each term descriptor contains the term itself and a file pointer indicating the location of the term's inverted list in the postings file. Dictionary entries in this data structure are accessed through binary search on the primary array, following the pointers into the secondary array to perform string comparisons. The pointers in the primary array usually consume 32 bits each (assuming that the secondary array is smaller than 2^{32} bytes).

A sort-based dictionary contains three sources of memory consumption: the pointers in the primary array, the file pointers in the term descriptors, and the terms themselves. We can take care of all three components at the same time by combining multiple consecutive term descriptors into groups and compressing the contents of each group, taking into account the fact that all data in the secondary array are sorted.

Dictionary groups

The fundamental insight is that it is not necessary (and in fact not beneficial) to have a pointer in the primary array for every term in the dictionary. Suppose we combine g consecutive terms into a group (g is called the *group size*) and keep a pointer only for the first term in each group (called the *group leader*). If $g = 1$, then lookups are realized in the way we are used to — by performing a binary search on the list of all $|\mathcal{V}|$ term descriptors. For $g > 1$, a lookup requires a binary search on the $\lceil |\mathcal{V}|/g \rceil$ group leaders, followed by a sequential scan of the remaining $g - 1$ members of the group identified through binary search.

The revised version of the sort-based dictionary data structure is shown in Figure 6.12 (for a group size of 3). When searching for the term "shakespeareanism" using the dictionary shown in the figure, the implementation would first identify the group led by "shakespeare" as potentially containing the term. It would then process all terms in that group in a linear fashion, in order to find the dictionary entry for "shakespeareanism". Conceptually, this lookup procedure is quite similar to the dictionary interleaving technique discussed in Section 4.4.

Let us calculate the number of string comparisons performed during a lookup for a single term T. For simplicity, we assume that T does in fact appear in the dictionary, that the dictionary contains a total of $|\mathcal{V}| = 2^n$ terms, and that the group size is $g = 2^m$, for some positive integers m and n (with $m < n$ and $n \gg 1$).

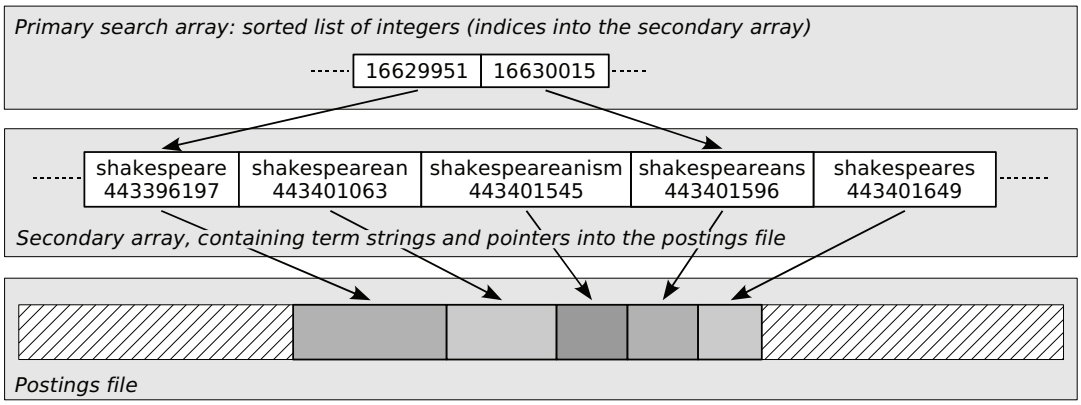

Figure 6.12 Sort-based dictionary with grouping: reducing the pointer overhead in the primary array by combining dictionary entries into groups.

- If $g = 1$, then a lookup requires approximately $n - 1$ string comparisons on average. The "-1" stems from the fact that we can terminate the binary search as soon as we encounter T. There is a 50% chance that this happens after n comparisons, a 25% chance that it happens after $n-1$ comparisons, a 12.5% chance that it happens after $n-2$ comparisons, and so forth. Thus, on average we need $n - 1$ string comparisons.
- For $g > 1$, we have to distinguish between two cases:
  1. The term is one of the $|\mathcal{V}|/g$ group leaders. The probability of this event is $1/g$, and it requires $n - m - 1$ string comparisons on average.
  2. The term is not a group leader. The probability of this happening is $(g-1)/g$. It requires $(n - m) + g/2$ string comparisons on average, where the first summand corresponds to a full binary search on the $|\mathcal{V}|/g$ group leaders (which cannot be terminated early because T is not a group leader), while the second summand corresponds to the linear scan of the $g-1$ non-leading members of the final group. The scan is terminated as soon as the matching term is found.

  Combining 1 and 2, the expected number of comparison operations is

$$\frac{1}{g}(n - m - 1) + \frac{g-1}{g}\left(n - m + \frac{g}{2}\right) \;=\; \log_2(|\mathcal{V}|) - \log_2(g) - \frac{1}{g} + \frac{g-1}{2}.$$

Table 6.11 lists the average number of comparisons for a dictionary of size $|\mathcal{V}| = 2^{20}$, using various group sizes. It shows that choosing $g = 2$ does not increase the number of comparisons per lookup at all. Selecting $g = 8$ increases them by about 7%. At the same time, however, it substantially reduces the pointer overhead in the primary array and improves the dictionary's cache efficiency, because fewer memory pages have to be accessed.

Table 6.11 Average number of string comparisons for a single-term lookup on a sort-based dictionary containing $|\mathcal{V}| = 2^{20}$ terms.

Group size	1	2	4	8	16	32	64
String comparisons	19.0	19.0	19.3	20.4	23.4	30.5	45.5

Front coding

Once we have combined dictionary entries into groups, we can compress the term descriptors by using the set of group leaders as synchronization points. Because the term descriptors within a given group have to be traversed sequentially anyway, regardless of whether we store them compressed or uncompressed, we may as well compress them in order to reduce the storage requirements and the amount of data that needs to be accessed during a lookup.

Term strings in a sort-based dictionary are usually compressed by using a technique known as *front coding*. Because the terms are sorted in lexicographical order, consecutive terms almost always share a common prefix. This prefix can be quite long. For example, all five terms shown in Figure 6.12 share the prefix "shakespeare".

In front coding we omit the prefix that is implicitly given by the term's predecessor in the dictionary and store only the length of the prefix. Thus, the front-coded representation of a term is a triplet of the form

$$(\textit{prefixLength},\ \textit{suffixLength},\ \textit{suffix}). \tag{6.63}$$

For ease of implementation, it is common to impose an upper limit of 15 characters on the length of both prefix and suffix, as this allows the dictionary implementation to store both *prefixLength* and *suffixLength* in a single byte (4 bits each). Cases in which the suffix is longer than 15 characters can be handled by storing an escape code $(\textit{prefix}, \textit{suffix}) = (*, 0)$ and encoding the string in some other way.

The front-coded representation of the terms shown in Figure 6.12 is

$$\langle\ \text{“shakespeare”}, (11, 2, \text{“an”}), (13, 3, \text{“ism”})\ \rangle,\ \langle\ \text{“shakespeareans”}, (11, 1, \text{“s”}), \ldots\ \rangle.$$

As mentioned earlier, the first element of each group is stored in uncompressed form, serving as a synchronization point that can be used for binary search.

In a last step, after compressing the term strings, we can also reduce the file pointer overhead in the dictionary. Because the lists in the postings file are sorted in lexicographical order, in the same order as the terms in the dictionary, the file pointers form a monotonically increasing sequence of integers. Thus, we can reduce their storage requirements by applying any of the Δ-based list compression techniques described in Section 6.3. For example, using the byte-aligned

Table 6.12 The effect of dictionary compression on a sort-based dictionary for GOV2, containing 49.5 million entries. Dictionary size and average term lookup time are given for different group sizes and different compression methods.

Group Size	No Compression	Front Coding	Front Coding +vByte	Front Coding +vByte+LZ
1 term	1046 MB / 2.8 μs	n/a	n/a	n/a
2 terms	952 MB / 2.7 μs	807 MB / 2.6 μs	643 MB / 2.6 μs	831 MB / 3.1 μs
4 terms	904 MB / 2.6 μs	688 MB / 2.5 μs	441 MB / 2.4 μs	533 MB / 2.9 μs
16 terms	869 MB / 2.7 μs	598 MB / 2.2 μs	290 MB / 2.3 μs	293 MB / 4.4 μs
64 terms	860 MB / 3.9 μs	576 MB / 2.4 μs	252 MB / 3.2 μs	195 MB / 8.0 μs
256 terms	858 MB / 9.3 μs	570 MB / 4.8 μs	243 MB / 7.8 μs	157 MB / 29.2 μs

vByte method, the compressed representation of the dictionary group led by "shakespeare" is

$$\begin{aligned} & \langle\, (\langle 101, 96, 54, 83, 1\rangle, \text{“shakespeare”}), (\langle 2, 38\rangle, 11, 2, \text{“an”}), (\langle 98, 3\rangle, 13, 3, \text{“ism”}) \,\rangle \\ \equiv\ & \langle\, (443396197, \text{“shakespeare”}), (4866, 11, 2, \text{“an”}), (482, 13, 3, \text{“ism”}) \,\rangle, \end{aligned} \tag{6.64}$$

where the first element of each dictionary entry is the Δ-encoded file pointer (vByte in the first line, simple Δ-value in the second line). As in the case of front coding, the file pointer of each group leader is stored in uncompressed form.

The effect of the methods described above, on both dictionary size and average lookup time, is shown in Table 6.12. By arranging consecutive dictionary entries into groups but not applying any compression, the size of the dictionary can be decreased by about 18% (858 MB vs. 1046 MB). Combining the grouping technique with front coding cuts the dictionary's storage requirements approximately in half. If, in addition to grouping and front coding, file pointers are stored in vByte-encoded form, the dictionary can be shrunk to approximately 25% of its original size.

Regarding the lookup performance, there are two interesting observations. First, arranging term descriptors in groups improves the dictionary's lookup performance, despite the slightly increased number of string comparisons, because reducing the number of pointers in the primary array makes the binary search more cache-efficient. Second, front-coding the terms makes dictionary lookups faster, regardless of the group size g, because fewer bytes need to be shuffled around and compared with the term that is being looked up.

The last column in Table 6.12 represents a dictionary compression method that we have not described so far: combining grouping, front coding, and vByte with a general-purpose Ziv-Lempel compression algorithm (as implemented in the `zlib`[6] compression library) that is run on each group after applying front coding and vByte. Doing so is motivated by the fact that

[6] `www.zlib.net`

Table 6.13 Total size of in-memory dictionary when combining dictionary compression with the interleaved dictionary organization from Section 4.4. Compression method: FC+vByte+LZ. Underlying index data structure: frequency index (docids + TF values) for GOV2.

		Index Block Size (in bytes)						
		512	1,024	2,048	4,096	8,192	16,384	32,768
Group Size	1 term	117.0 MB	60.7 MB	32.0 MB	17.1 MB	9.3 MB	5.1 MB	9.9 MB
	4 terms	68.1 MB	35.9 MB	19.2 MB	10.4 MB	5.7 MB	3.2 MB	1.8 MB
	16 terms	43.8 MB	23.4 MB	12.7 MB	7.0 MB	3.9 MB	2.3 MB	1.3 MB
	64 terms	32.9 MB	17.8 MB	9.8 MB	5.4 MB	3.1 MB	1.8 MB	1.0 MB
	256 terms	28.6 MB	15.6 MB	8.6 MB	4.8 MB	2.7 MB	1.6 MB	0.9 MB

even a front-coded dictionary exhibits a remarkable degree of redundancy. For example, in a front-coded dictionary for the TREC collection (containing 1.2 million distinct terms) there are 171 occurrences of the suffix "ation", 7726 occurrences of the suffix "ing", and 722 occurrences of the suffix "ville". Front coding is oblivious to this redundancy because it focuses exclusively on the terms' prefixes. The same is true for the Δ-transformed file pointers. Because many postings lists (in particular, lists that contain only a single posting) are of the same size, we will end up with many file pointers that have the same Δ-value. We can eliminate this remaining redundancy by applying standard Ziv-Lempel data compression on top of front coding + vByte. Table 6.12 shows that this approach, although it does not work very well for small group sizes, can lead to considerable savings for $g \geq 64$ and can reduce the total size of the dictionary by up to 85% compared to the original, uncompressed index.

Combining dictionary compression and dictionary interleaving

In Section 4.4, we discussed a different way to reduce the memory requirements of the dictionary. By interleaving dictionary entries with the on-disk postings lists and keeping only one dictionary entry in memory for every 64 KB or so of index data, we were able to substantially reduce the dictionary's memory requirements: from a gigabyte down to a few megabytes. It seems natural to combine dictionary compression and dictionary interleaving to obtain an even more compact representation of the search engine's in-memory dictionary.

Table 6.13 summarizes the results that can be obtained by following this path. Depending on the index block size B (the maximum amount of on-disk index data between two subsequent in-memory dictionary entries — see Section 4.4 for details) and the in-memory dictionary group size g, it is possible to reduce the total storage requirements to under 1 MB (B=32,768; g=256). The query-time penalty resulting from this approach is essentially the I/O overhead caused by having to read an additional 32,768 bytes per query term: less than 0.5 ms for the hard drives used in our experiments.

Even more remarkable, however, is the dictionary size resulting from the configuration B=512, g=256: less than 30 MB. Choosing an index block size of B=512 does not lead to any measurable degradation in query performance, as 512 bytes (=1 sector) is the minimum transfer unit of most hard drives. Nevertheless, the memory requirements of the dictionary can be decreased by more than 97% compared to a data structure that does not use dictionary interleaving or dictionary compression.

6.5 Summary

The main points of this chapter are:

- Many compression methods treat the message to be compressed as a sequence of symbols and operate by finding a code that reflects the statistical properties of the symbols in the message by giving shorter codewords to symbols that appear more frequently.
- For any given probability distribution over a finite set of symbols, Huffman coding produces an optimal prefix code (i.e., one that minimizes the average codeword length).
- Arithmetic coding can attain better compression rates than Huffman coding because it does not assign an integral number of bits to each symbol's codeword. However, Huffman codes can usually be decoded much faster than arithmetic codes and are therefore given preference in many applications (e.g., search engines).
- Compression methods for inverted lists are invariably based on the fact that postings within the same list are stored in increasing order. Compression usually takes place after transforming the list into an equivalent sequence of Δ-values.
- Parametric methods usually produce smaller output than nonparametric methods. When using a parametric method, the parameter should be chosen on a per-list basis (*local parameterization*).
- If the Δ-gaps in a postings list follow a geometric distribution, then Golomb/Rice codes lead to good compression results.
- If the Δ-gaps are very small, then interpolative coding achieves very good compression rates, due to its ability to encode postings using less than 1 bit on average (arithmetic coding has the same ability but requires more complex decoding operations than interpolative coding).
- For a wide range of gap distributions, the Huffman-based LLRUN method leads to excellent results.
- Front coding has the ability to reduce the memory requirements of the search engine's dictionary data structure significantly. In combination with dictionary interleaving, it can reduce the size of the in-memory dictionary by more than 99% with virtually no degradation in query performance.

6.6 Further Reading

An excellent overview of general-purpose data compression (including text compression) is given by Bell et al. (1990) and, more recently, by Sayood (2005) and Salomon (2007). An overview of compression techniques for postings lists in inverted files is provided by Witten et al. (1999). Zobel and Moffat (2006) present a survey of research carried out in the context of inverted files, including an overview of existing approaches to inverted list compression.

Huffman codes were invented by David Huffman in 1951, while he was a student at MIT. One of his professors, Robert Fano, had asked his students to come up with a method that produces an optimal binary code for any given (finite) message source. Huffman succeeded and, as a reward, was exempted from the course's final exam. Huffman codes have been studied extensively over the past six decades, and important results have been obtained regarding their redundancy compared to the theoretically optimal (arithmetic) codes (Horibe, 1977; Gallager, 1978; Szpankowski, 2000).

Arithmetic codes can overcome the limitations of Huffman codes that are caused by the requirement that an integral number of bits be used for each symbol. Arithmetic coding was initially developed by Rissanen (1976) and Rissanen and Langdon (1979), and by Martin (1979) under the name *range encoding*, but did not experience widespread adoption until the mid-1980s, when Witten et al. (1987) published their implementation of an efficient arithmetic coder. Many modern text compression algorithms, such as DMC (Cormack and Horspool, 1987) and PPM (Cleary and Witten, 1984), are based upon variants of arithmetic coding.

γ, δ, and ω codes were introduced by Elias (1975), who also showed their *universality* (a set of codewords is called *universal* if, by assigning codewords C_i to symbols σ_i in such a way that higher-probability symbols receive shorter codewords, the expected number of bits per encoded symbol is within a constant factor of the symbol source's entropy). Golomb codes were proposed by Golomb (1966) in a very entertaining essay about "Secret Agent 00111" playing a game of roulette. Interpolative coding is due to Moffat and Stuiver (2000). In addition to the method's application in inverted-file compression, they also point out other applications, such as the efficient representation of the (i.e., description of the code used by the encoder) in Huffman coding or the encoding of move-to-front values used in compression algorithms based on the Burrows-Wheeler transform.

Byte-aligned compression methods such as vByte (Section 6.3.4) were first explored academically by Williams and Zobel (1999) but had been in use long before. Trotman (2003), in the context of on-disk indices, explores a number of compression effectiveness versus decoding performance trade-offs, showing that variable-byte encoding usually leads to higher query performance than bitwise methods. Scholer et al. (2002) come to the same conclusion but also show that a byte-aligned compression scheme can sometime even lead to improvements over an uncompressed *in-memory* index. In a recent study, Büttcher and Clarke (2007) show that this is true even if the search engine's index access pattern is completely random.

Reordering documents with the goal of achieving better compression rates was first studied by Blandford and Blelloch (2002), whose method is based on a clustering approach that tries to assign nearby docids to documents with similar content. In independent research, Shieh et al. (2003), present a similar method that reduces document reordering to the traveling salesperson problem (TSP). The idea to improve compressibility by sorting documents according to their URLs is due to Silvestri (2007).

6.7 Exercises

Exercise 6.1 Consider the symbol set $\mathcal{S} = \{\text{“a”}, \text{“b”}\}$ with associated probability distribution Pr[“a”] = 0.8, Pr[“b”] = 0.2. Suppose we group symbols into blocks of $m = 2$ symbols when encoding messages over $\mathcal{S}$ (resulting in the new distribution Pr[“aa”] = 0.64, Pr[“ab”] = 0.16, etc.). Construct a Huffman tree for this new distribution.

Exercise 6.2 Show that the γ code is prefix-free (i.e., that there is no codeword that is a prefix of another codeword).

Exercise 6.3 Give an example of an unambiguous binary code that is not prefix-free. What is the probability distribution for which this code is optimal? Construct an optimal prefix-free code for the same distribution.

Exercise 6.4 Write down the decoding procedure for ω codes (see Section 6.3.1).

Exercise 6.5 Find the smallest integer n_0 such that the ω code for all integers $n \geq n_0$ is shorter than the respective δ code. Be prepared for n_0 to be very large ($> 2^{100}$).

Exercise 6.6 What is the expected number of bits per codeword when using a Rice code with parameter $M = 2^7$ to compress a geometrically distributed postings list for a term T with $N_T/N = 0.01$?

Exercise 6.7 Consider a postings list with geometrically distributed Δ-gaps and average gap size $N/N_T = 137$ (i.e., $p = 0.0073$). What is the optimal parameter M^* in the case of Golomb coding? What is the optimal Rice parameter M^*_{Rice}? How many bits on average are wasted if the optimal Rice code is used instead of the Golomb code? How many bits on average are wasted by using the optimal Golomb instead of arithmetic coding?

Exercise 6.8 In Section 6.2.2 we discussed how to construct a Huffman tree in time $\Theta(n \log(n))$, where $n = |\mathcal{S}|$ is the size of the input alphabet. Now suppose the symbols in $\mathcal{S}$ have already been sorted by their respective probability when they arrive at the encoder. Design an algorithm that builds a Huffman tree in time $\Theta(n)$.

Exercise 6.9 The Simple-9 compression method from Section 6.3.4 groups sequences of Δ-values into 32-bit machine words, reserving 4 bits for a selector value. Consider a similar method, Simple-14, that uses 64-bit machine words instead. Assuming that 4 bits per 64-bit word are reserved for the selector value, list all possible splits of the remaining 60 bits. Which of the two methods do you expect to yield better compression rates on a typical docid index? Characterization the type of docid lists on which Simple-9 will lead to better/worse results than Simple-14.

Exercise 6.10 One of the document reordering methods from Section 6.3.7 assigns numerical document identifiers according to the lexicographical ordering of the respective URLs. Is there another method, also based on the documents' URLs, that might achieve even better compression rates? Consider the individual components of a URL and how you could use them.

Exercise 6.11 (project exercise) Add index compression to your existing implementation of the inverted index data structure. Implement support for the byte-aligned vByte method from Section 6.3.4 as well as your choice of γ coding (Section 6.3.1) or Simple-9 (Section 6.3.4). Your new implementation should store all postings lists in compressed form.

6.8 Bibliography

Anh, V. N., and Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166.

Bell, T. C., Cleary, J. G., and Witten, I. H. (1990). *Text Compression*. Upper Saddle River, New Jersey: Prentice-Hall.

Blandford, D. K., and Blelloch, G. E. (2002). Index compression through document reordering. In *Data Compression Conference*, pages 342–351. Snowbird, Utah.

Burrows, M., and Wheeler, D. (1994). *A Block-Sorting Lossless Data Compression Algorithm*. Technical Report SRC-RR-124. Digital Systems Research Center, Palo Alto, California.

Büttcher, S., and Clarke, C. L. A. (2007). Index compression is good, especially for random access. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, pages 761–770. Lisbon, Portugal.

Cleary, J. G., and Witten, I. H. (1984). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402.

Cormack, G. V., and Horspool, R. N. S. (1987). Data compression using dynamic Markov modelling. *The Computer Journal*, 30(6):541–550.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203.

Fraenkel, A. S., and Klein, S. T. (1985). Novel compression of sparse bit-strings. In Apostolico, A., and Galil, Z., editors, *Combinatorial Algorithms on Words*, pages 169–183. New York: Springer.

Gallager, R. G. (1978). Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674.

Gallager, R. G., and Voorhis, D. C. V. (1975). Optimal source codes for geometrically distributed integer alphabets. *IEEE Transactions on Information Theory*, 21(2):228–230.

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

Horibe, Y. (1977). An improved bound for weight-balanced tree. *Information and Control*, 34(2):148–151.

Larmore, L. L., and Hirschberg, D. S. (1990). A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464–473.

Martin, G. N. N. (1979). Range encoding: An algorithm for removing redundancy from a digitised message. In *Proceedings of the Conference on Video and Data Recording*. Southampton, England.

Moffat, A., and Stuiver, L. (2000). Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47.

Patterson, D. A., and Hennessy, J. L. (2009). *Computer Organization and Design: The Hardware/Software Interface* (4th ed.). San Francisco, California: Morgan Kaufmann.

Rice, R. F., and Plaunt, J. R. (1971). Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Commununication Technology*, 19(6):889–897.

Rissanen, J. (1976). Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203.

Rissanen, J., and Langdon, G. G. (1979). Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162.

Salomon, D. (2007). *Data Compression: The Complete Reference* (4th ed.). London, England: Springer.

Sayood, K. (2005). *Introduction to Data Compression* (3rd ed.). San Francisco, California: Morgan Kaufmann.

Scholer, F., Williams, H. E., Yiannis, J., and Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229. Tampere, Finland.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656.

Shannon, C. E. (1951). Prediction and entropy of printed English. *Bell System Technical Journal*, 30:50–64.

Shieh, W. Y., Chen, T. F., Shann, J. J. J., and Chung, C. P. (2003). Inverted file compression through document identifier reassignment. *Information Processing & Management*, 39(1):117–131.

Silvestri, F. (2007). Sorting out the document identifier assignment problem. In *Proceedings of the 29th European Conference on IR Research*, pages 101–112. Rome, Italy.

Szpankowski, W. (2000). Asymptotic average redundancy of Huffman (and other) block codes. *IEEE Transactions on Information Theory*, 46(7):2434–2443.

Trotman, A. (2003). Compressing inverted files. *Information Retrieval*, 6(1):5–19.

Williams, H. E., and Zobel, J. (1999). Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201.

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd ed.). San Francisco, California: Morgan Kaufmann.

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

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

Zobel, J., and Moffat, A. (2006). Inverted files for text search engines. *ACM Computing Surveys*, 38(2):1–56.