5 *Index compression*

Chapter 1 introduced the dictionary and the inverted index as the central data structures in information retrieval (IR). In this chapter, we employ a number of compression techniques for dictionary and inverted index that are essential for efficient IR systems.

One benefit of compression is immediately clear. We need less disk space. As we will see, compression ratios of 1:4 are easy to achieve, potentially cutting the cost of storing the index by 75%.

There are two more subtle benefits of compression. The first is increased use of caching. Search systems use some parts of the dictionary and the index much more than others. For example, if we cache the postings list of a frequently used query term t, then the computations necessary for responding to the one-term query t can be entirely done in memory. With compression, we can fit a lot more information into main memory. Instead of having to expend a disk seek when processing a query with t, we instead access its postings list in memory and decompress it. As we will see below, there are simple and efficient decompression methods, so that the penalty of having to decompress the postings list is small. As a result, we are able to decrease the response time of the IR system substantially. Because memory is a more expensive resource than disk space, increased speed owing to caching – rather than decreased space requirements – is often the prime motivator for compression.

The second more subtle advantage of compression is faster transfer of data from disk to memory. Efficient decompression algorithms run so fast on modern hardware that the total time of transferring a compressed chunk of data from disk and then decompressing it is usually less than transferring the same chunk of data in uncompressed form. For instance, we can reduce input/output (I/O) time by loading a much smaller compressed postings list, even when you add on the cost of decompression. So, in most cases, the retrieval system runs faster on compressed postings lists than on uncompressed postings lists.

If the main goal of compression is to conserve disk space, then the speed

of compression algorithms is of no concern. But for improved cache utilization and faster disk-to-memory transfer, decompression speeds must be high. The compression algorithms we discuss in this chapter are highly efficient and can therefore serve all three purposes of index compression.

POSTING

In this chapter, we define a *posting* as a docID in a postings list. For example, the postings list (6; 20, 45, 100), where 6 is the termID of the list's term, contains three postings. As discussed in Section 2.4.2 (page 41), postings in most search systems also contain frequency and position information; but we will only consider simple docID postings here. See Section 5.4 for references on compressing frequencies and positions.

This chapter first gives a statistical characterization of the distribution of the entities we want to compress – terms and postings in large collections (Section 5.1). We then look at compression of the dictionary, using the dictionary-as-a-string method and blocked storage (Section 5.2). Section 5.3 describes two techniques for compressing the postings file, variable byte encoding and γ encoding.

5.1 Statistical properties of terms in information retrieval

As in the last chapter, we use Reuters-RCV1 as our model collection (see Table 4.2, page 70). We give some term and postings statistics for the collection in Table 5.1. "Δ%" indicates the reduction in size from the previous line. "T%" is the cumulative reduction from unfiltered.

The table shows the number of terms for different levels of preprocessing (column 2). The number of terms is the main factor in determining the size of the dictionary. The number of nonpositional postings (column 3) is an indicator of the expected size of the nonpositional index of the collection. The expected size of a positional index is related to the number of positions it must encode (column 4).

In general, the statistics in Table 5.1 show that preprocessing affects the size of the dictionary and the number of nonpositional postings greatly. Stemming and case folding reduce the number of (distinct) terms by 17% each and the number of nonpositional postings by 4% and 3%, respectively. The

RULE OF 30

treatment of the most frequent words is also important. The *rule of 30* states that the 30 most common words account for 30% of the tokens in written text (31% in the table). Eliminating the 150 most common words from indexing (as stop words; cf. Section 2.2.2, page 27) cuts 25% to 30% of the nonpositional postings. But, although a stop list of 150 words reduces the number of postings by a quarter or more, this size reduction does not carry over to the size of the compressed index. As we will see later in this chapter, the postings lists of frequent words require only a few bits per posting after compression.

The deltas in the table are in a range typical of large collections. Note,

► **Table 5.1** The effect of preprocessing on the number of terms, nonpositional postings, and tokens for Reuters-RCV1. "Δ%" indicates the reduction in size from the previous line, except that "30 stop words" and "150 stop words" both use "case folding" as their reference line. "T%" is the cumulative ("total") reduction from unfiltered. We performed stemming with the Porter stemmer (Chapter 2, page 33).

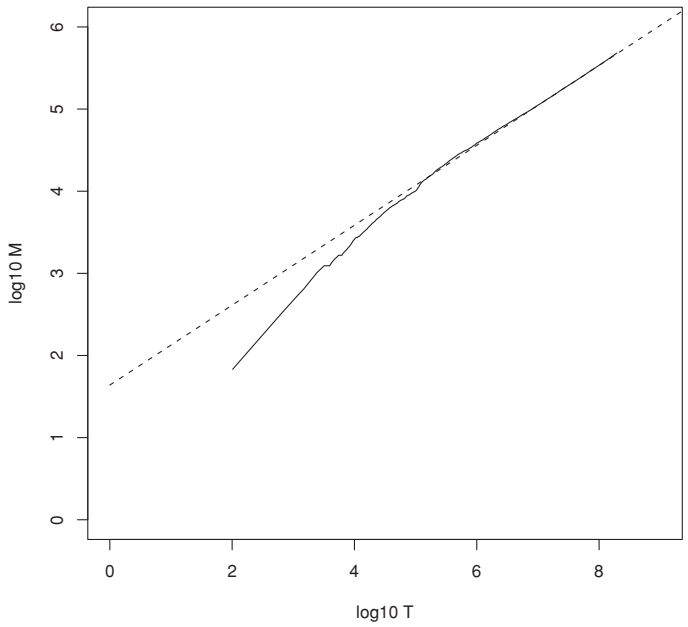
	(distinct) terms			nonpositional postings			tokens (= number of positi entries in postings)		
	number	Δ%	T%	number	Δ%	T%	number	Δ%	T%
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	−2	−2	100,680,242	−8	−8	179,158,204	−9	−9
case folding	391,523	−17	−19	96,969,056	−3	−12	179,158,204	−0	−9
30 stop words	391,493	−0	−19	83,390,443	−14	−24	121,857,825	−31	−38
150 stop words	391,373	−0	−19	67,001,847	−30	−39	94,516,599	−47	−52
stemming	322,383	−17	−33	63,812,300	−4	−42	94,516,599	−0	−52

however, that the percentage reductions can be very different for some text collections. For example, for a collection of web pages with a high proportion of French text, a lemmatizer for French reduces vocabulary size much more than the Porter stemmer does for an English-only collection because French is a morphologically richer language than English.

LOSSLESS
LOSSY COMPRESSION

The compression techniques we describe in the remainder of this chapter are *lossless*, that is, all information is preserved. Better compression ratios can be achieved with *lossy compression*, which discards some information. Case folding, stemming, and stop word elimination are forms of lossy compression. Similarly, the vector space model (Chapter 6) and dimensionality reduction techniques like latent semantic indexing (Chapter 18) create compact representations from which we cannot fully restore the original collection. Lossy compression makes sense when the "lost" information is unlikely ever to be used by the search system. For example, web search is characterized by a large number of documents, short queries, and users who only look at the first few pages of results. As a consequence, we can discard postings of documents that would only be used for hits far down the list. Thus, there are retrieval scenarios where lossy methods can be used for compression without any reduction in effectiveness.

Before introducing techniques for compressing the dictionary, we want to estimate the number of distinct terms M in a collection. It is sometimes said that languages have a vocabulary of a certain size. The second edition of the *Oxford English Dictionary* (OED) defines more than 600,000 words. But the vocabulary of most large collections is much larger than the OED. The OED does not include most names of people, locations, products, or scientific



► **Figure 5.1** Heaps' law. Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1. For these data, the dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least-squares fit. Thus, $k = 10^{1.64} \approx 44$ and $b = 0.49$.

entities like genes. These names need to be included in the inverted index, so our users can search for them.

5.1.1 Heaps' law: Estimating the number of terms

HEAPS' LAW

A better way of getting a handle on M is *Heaps' law*, which estimates vocabulary size as a function of collection size:

$$M = kT^b \tag{5.1}$$

where T is the number of tokens in the collection. Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$. The motivation for Heaps' law is that the simplest possible relationship between collection size and vocabulary size is linear in log–log space and the assumption of linearity is usually born out in practice as shown in Figure 5.1 for Reuters-RCV1. In this case, the fit is excellent for $T > 10^5 = 100{,}000$, for the parameter values $b = 0.49$ and $k = 44$. For example, for the first 1,000,020 tokens Heaps' law

predicts 38,323 terms:

$$44 \times 1{,}000{,}020^{0.49} \approx 38{,}323.$$

The actual number is 38,365 terms, very close to the prediction.

The parameter k is quite variable because vocabulary growth depends a lot on the nature of the collection and how it is processed. Case-folding and stemming reduce the growth rate of the vocabulary, whereas including numbers and spelling errors increase it. Regardless of the values of the parameters for a particular collection, Heaps' law suggests that (i) the dictionary size continues to increase with more documents in the collection, rather than a maximum vocabulary size being reached, and (ii) the size of the dictionary is quite large for large collections. These two hypotheses have been empirically shown to be true of large text collections (Section 5.4). So dictionary compression is important for an effective information retrieval system.

5.1.2 Zipf's law: Modeling the distribution of terms

We also want to understand how terms are distributed across documents. This helps us to characterize the properties of the algorithms for compressing postings lists in Section 5.3.

ZIPF'S LAW

A commonly used model of the distribution of terms in a collection is *Zipf's law*. It states that, if t_1 is the most common term in the collection, t_2 is the next most common, and so on, then the collection frequency cf_i of the ith most common term is proportional to $1/i$:

$$\text{cf}_i \propto \frac{1}{i}. \tag{5.2}$$

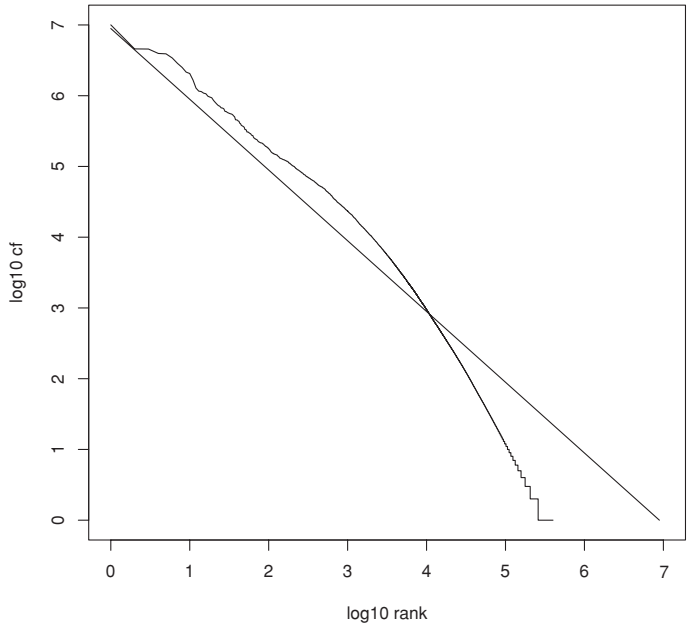
So if the most frequent term occurs cf_1 times, then the second most frequent term has half as many occurrences, the third most frequent term a third as many occurrences, and so on. The intuition is that frequency decreases very rapidly with rank. Equation (5.2) is one of the simplest ways of formalizing such a rapid decrease and it has been found to be a reasonably good model.

Equivalently, we can write Zipf's law as $\text{cf}_i = ci^k$ or as $\log \text{cf}_i = \log c + k \log i$ where $k = -1$ and c is a constant to be defined in Section 5.3.2. It

POWER LAW

is therefore a *power law* with exponent $k = -1$. See Chapter 19, page 426, for another power law, a law characterizing the distribution of links on web pages.

The log–log graph in Figure 5.2 plots the collection frequency of a term as a function of its rank for Reuters-RCV1. A line with slope –1, corresponding to the Zipf function $\log \text{cf}_i = \log c - \log i$, is also shown. The fit of the data to the law is not particularly good, but good enough to serve as a model for term distributions in our calculations in Section 5.3.



► **Figure 5.2** Zipf's law for Reuters-RCV1. Frequency is plotted as a function of frequency rank for the terms in the collection. The line is the distribution predicted by Zipf's law (weighted least-squares fit; intercept is 6.95).

? **Exercise 5.1** [⋆]

Assuming one machine word per posting, what is the size of the uncompressed (non-positional) index for different tokenizations based on Table 5.1? How do these numbers compare with Table 5.6?

5.2 Dictionary compression

This section presents a series of dictionary data structures that achieve increasingly higher compression ratios. The dictionary is small compared with the postings file as suggested by Table 5.1. So why compress it if it is responsible for only a small percentage of the overall space requirements of the IR system?

One of the primary factors in determining the response time of an IR system is the number of disk seeks necessary to process a query. If parts of the dictionary are on disk, then many more disk seeks are necessary in query evaluation. Thus, the main goal of compressing the dictionary is to fit it in main memory, or at least a large portion of it, to support high query through-

	term	document frequency	pointer to postings list
	a	656,265	⟶
	aachen	65	⟶
	...	...	...
	zulu	221	⟶
space needed:	20 bytes	4 bytes	4 bytes

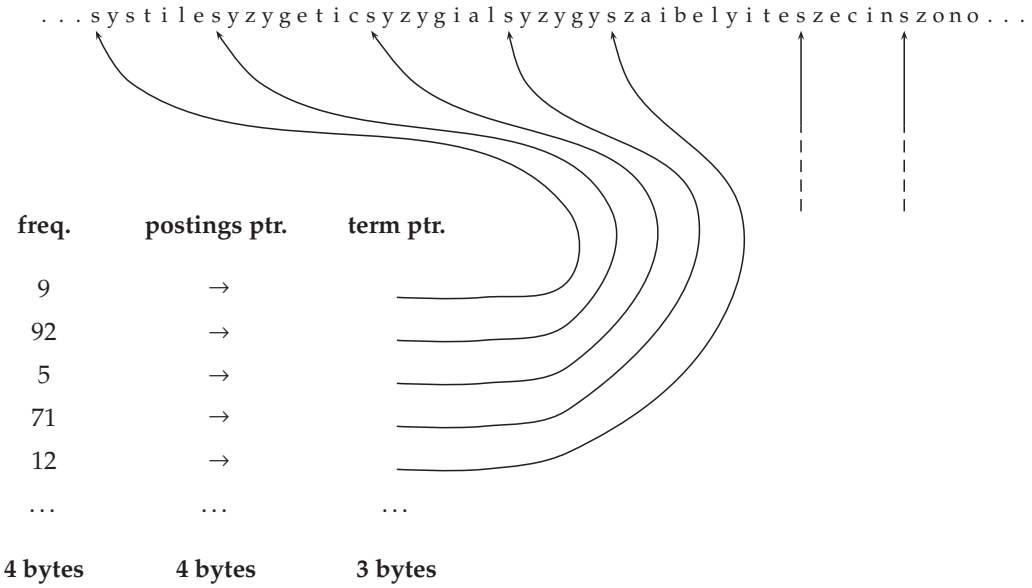
► **Figure 5.3** Storing the dictionary as an array of fixed-width entries.

put. Although dictionaries of very large collections fit into the memory of a standard desktop machine, this is not true of many other application scenarios. For example, an enterprise search server for a large corporation may have to index a multiterabyte collection with a comparatively large vocabulary because of the presence of documents in many different languages. We also want to be able to design search systems for limited hardware such as mobile phones and onboard computers. Other reasons for wanting to conserve memory are fast startup time and having to share resources with other applications. The search system on your PC must get along with the memory-hogging word processing suite you are using at the same time.

5.2.1 Dictionary as a string

The simplest data structure for the dictionary is to sort the vocabulary lexicographically and store it in an array of fixed-width entries as shown in Figure 5.3. We allocate 20 bytes for the term itself (because few terms have more than twenty characters in English), 4 bytes for its document frequency, and 4 bytes for the pointer to its postings list. Four-byte pointers resolve a 4 gigabytes (GB) address space. For large collections like the web, we need to allocate more bytes per pointer. We look up terms in the array by binary search. For Reuters-RCV1, we need $M \times (20 + 4 + 4) = 400{,}000 \times 28 =$ 11.2megabytes (MB) for storing the dictionary in this scheme.

Using fixed-width entries for terms is clearly wasteful. The average length of a term in English is about eight characters (Table 4.2, page 70), so on average we are wasting twelve characters in the fixed-width scheme. Also, we have no way of storing terms with more than twenty characters like hydrochlorofluorocarbons and supercalifragilisticexpialidocious. We can overcome these shortcomings by storing the dictionary terms as one long string of characters, as shown in Figure 5.4. The pointer to the next term is also used to demarcate the end of the current term. As before, we locate terms in the data structure by way of binary search in the (now smaller) table. This scheme saves us 60% compared to fixed-width storage – 12 bytes on average of the



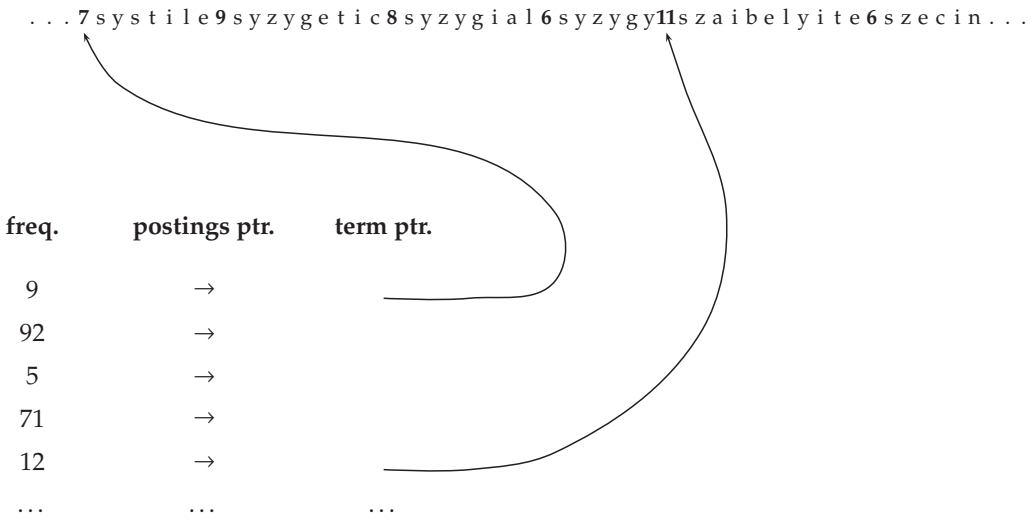
▶ **Figure 5.4** Dictionary-as-a-string storage. Pointers mark the end of the preceding term and the beginning of the next. For example, the first three terms in this example are systile, syzygetic, and syzygial.

20 bytes we allocated for terms before. However, we now also need to store term pointers. The term pointers resolve 400,000 $\times$ 8 $=$ 3.2 $\times$ 10^6 positions, so they need to be $\log_2 3.2 \times 10^6 \approx 22$ bits or 3 bytes long.

In this new scheme, we need 400,000 $\times$ $(4 + 4 + 3 + 8)$ $=$ 7.6 MB for the Reuters-RCV1 dictionary: 4 bytes each for frequency and postings pointer, 3 bytes for the term pointer, and 8 bytes on average for the term. So we have reduced the space requirements by one third from 11.2 to 7.6 MB.

5.2.2 Blocked storage

We can further compress the dictionary by grouping terms in the string into blocks of size k and keeping a term pointer only for the first term of each block (Figure 5.5). We store the length of the term in the string as an additional byte at the beginning of the term. We thus eliminate $k - 1$ term pointers, but need an additional k bytes for storing the length of each term. For $k = 4$, we save $(k - 1) \times 3 = 9$ bytes for term pointers, but need an additional $k = 4$ bytes for term lengths. So the total space requirements for the dictionary of Reuters-RCV1 are reduced by 5 bytes per four-term block, or a total of 400,000 $\times$ 1/4 $\times$ 5 $=$ 0.5 MB, bringing us down to 7.1 MB.

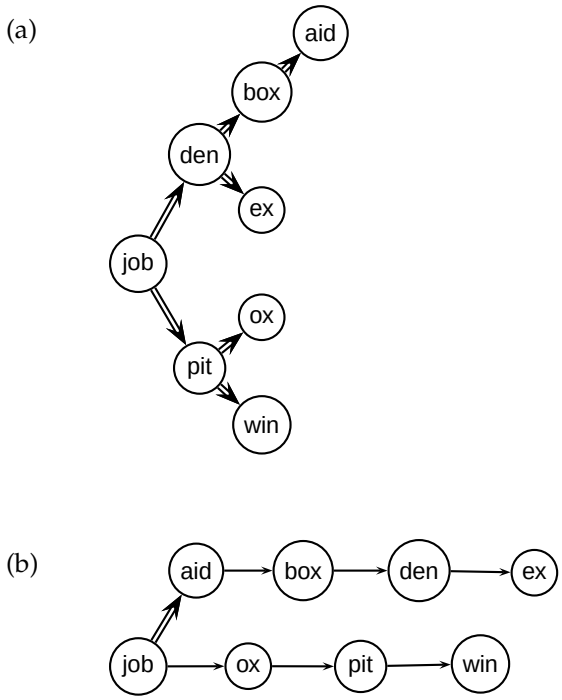


▶ **Figure 5.5** Blocked storage with four terms per block. The first block consists of systile, syzygetic, syzygial, and syzygy with lengths of seven, nine, eight, and six characters, respectively. Each term is preceded by a byte encoding its length that indicates how many bytes to skip to reach subsequent terms.

By increasing the block size k, we get better compression. However, there is a tradeoff between compression and the speed of term lookup. For the eight-term dictionary in Figure 5.6, steps in binary search are shown as double lines and steps in list search as simple lines. We search for terms in the uncompressed dictionary by binary search (a). In the compressed dictionary, we first locate the term's block by binary search and then its position within the list by linear search through the block (b). Searching the uncompressed dictionary in (a) takes on average $(0+1+2+3+2+1+2+2)/8 \approx 1.6$ steps, assuming each term is equally likely to come up in a query. For example, finding the two terms, aid and box, takes three and two steps, respectively. With blocks of size $k = 4$ in (b), we need $(0+1+2+3+4+1+2+3)/8 = 2$ steps on average, $\approx 25\%$ more. For example, finding den takes one binary search step and two steps through the block. By increasing k, we can get the size of the compressed dictionary arbitrarily close to the minimum of $400{,}000 \times (4+4+1+8) = 6.8$ MB, but term lookup becomes prohibitively slow for large values of k.

FRONT CODING

One source of redundancy in the dictionary we have not exploited yet is the fact that consecutive entries in an alphabetically sorted list share common prefixes. This observation leads to *front coding* (Figure 5.7). A common prefix



► **Figure 5.6** Search of the uncompressed dictionary (a) and a dictionary compressed by blocking with $k = 4$ (b).

One block in blocked compression ($k = 4$) ...
8automata**8**automate**9**automatic**10**automation

⇓

...further compressed with front coding.
8automat∗a**1**⋄e**2** ⋄ ic**3**⋄ion

► **Figure 5.7** Front coding. A sequence of terms with identical prefix ("automat") is encoded by marking the end of the prefix with ∗ and replacing it with ⋄ in subsequent terms. As before, the first byte of each entry encodes the number of characters.

► **Table 5.2** Dictionary compression for Reuters-RCV1.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
∼, with blocking, $k = 4$	7.1
∼, with blocking & front coding	5.9

is identified for a subsequence of the term list and then referred to with a special character. In the case of Reuters, front coding saves another 1.2 MB, as we found in an experiment.

Other schemes with even greater compression rely on minimal perfect hashing, that is, a hash function that maps M terms onto $[1, \ldots, M]$ without collisions. However, we cannot adapt perfect hashes incrementally because each new term causes a collision and therefore requires the creation of a new perfect hash function. Therefore, they cannot be used in a dynamic environment.

Even with the best compression scheme, it may not be feasible to store the entire dictionary in main memory for very large text collections and for hardware with limited memory. If we have to partition the dictionary onto pages that are stored on disk, then we can index the first term of each page using a B-tree. For processing most queries, the search system has to go to disk anyway to fetch the postings. One additional seek for retrieving the term's dictionary page from disk is a significant, but tolerable increase in the time it takes to process a query.

Table 5.2 summarizes the compression achieved by the four dictionary data structures.

? **Exercise 5.2**

Estimate the space usage of the Reuters-RCV1 dictionary with blocks of size $k = 8$ and $k = 16$ in blocked dictionary storage.

Exercise 5.3

Estimate the time needed for term lookup in the compressed dictionary of Reuters-RCV1 with block sizes of $k = 4$ (Figure 5.6, b), $k = 8$, and $k = 16$. What is the slowdown compared with $k = 1$ (Figure 5.6, a)?

5.3 Postings file compression

Recall from Table 4.2 (page 70) that Reuters-RCV1 has 800,000 documents, 200 tokens per document, six characters per token, and 100,000,000 postings where we define a posting in this chapter as a docID in a postings list, that is, excluding frequency and position information. These numbers

► **Table 5.3** Encoding gaps instead of document IDs. For example, we store gaps 107, 5, 43, ..., instead of docIDs 283154, 283159, 283202, ... for computer. The first docID is left unchanged (only shown for arachnocentric).

	encoding	postings list							
the	docIDs	...		283042		283043		283044	283045
	gaps				1		1	1	
computer	docIDs	...		283047		283154		283159	283202
	gaps				107		5	43	
arachnocentric	docIDs	252000		500100					
	gaps	252000	248100						

correspond to line 3 ("case folding") in Table 5.1. Document identifiers are $\log_2 800{,}000 \approx 20$ bits long. Thus, the size of the collection is about $800{,}000 \times 200 \times 6$ bytes $= 960$ MB and the size of the uncompressed postings file is $100{,}000{,}000 \times 20/8 = 250$ MB.

To devise a more efficient representation of the postings file, one that uses fewer than 20 bits per document, we observe that the postings for frequent terms are close together. Imagine going through the documents of a collection one by one and looking for a frequent term like computer. We will find a document containing computer, then we skip a few documents that do not contain it, then there is again a document with the term and so on (see Table 5.3). The key idea is that the *gaps* between postings are short, requiring a lot less space than 20 bits to store. In fact, gaps for the most frequent terms such as the and for are mostly equal to 1. But the gaps for a rare term that occurs only once or twice in a collection (e.g., arachnocentric in Table 5.3) have the same order of magnitude as the docIDs and need 20 bits. For an economical representation of this distribution of gaps, we need a *variable encoding* method that uses fewer bits for short gaps.

To encode small numbers in less space than large numbers, we look at two types of methods: bytewise compression and bitwise compression. As the names suggest, these methods attempt to encode gaps with the minimum number of bytes and bits, respectively.

5.3.1 Variable byte codes

VARIABLE BYTE ENCODING

CONTINUATION BIT

Variable byte (VB) encoding uses an integral number of bytes to encode a gap. The last 7 bits of a byte are "payload" and encode part of the gap. The first bit of the byte is a *continuation bit*.It is set to 1 for the last byte of the encoded gap and to 0 otherwise. To decode a variable byte code, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1. We then extract and concatenate the 7-bit parts. Figure 5.8 gives pseudocode

```
VBENCODENUMBER(n)
1  bytes ← ⟨⟩
2  while true
3  do PREPEND(bytes, n mod 128)
4     if n < 128
5        then BREAK
6     n ← n div 128
7  bytes[LENGTH(bytes)] += 128
8  return bytes

VBENCODE(numbers)
1  bytestream ← ⟨⟩
2  for each n ∈ numbers
3  do bytes ← VBENCODENUMBER(n)
4     bytestream ← EXTEND(bytestream, bytes)
5  return bytestream

VBDECODE(bytestream)
1  numbers ← ⟨⟩
2  n ← 0
3  for i ← 1 to LENGTH(bytestream)
4  do if bytestream[i] < 128
5        then n ← 128 × n + bytestream[i]
6        else n ← 128 × n + (bytestream[i] − 128)
7             APPEND(numbers, n)
8             n ← 0
9  return numbers
```

► **Figure 5.8** VB encoding and decoding. The functions div and mod compute integer division and remainder after integer division, respectively. PREPEND adds an element to the beginning of a list, for example, PREPEND($\langle 1,2\rangle, 3$) = $\langle 3,1,2\rangle$. EXTEND extends a list, for example, EXTEND($\langle 1,2\rangle, \langle 3,4\rangle$) = $\langle 1,2,3,4\rangle$.

► **Table 5.4** VB encoding. Gaps are encoded using an integral number of bytes. The first bit, the continuation bit, of each byte indicates whether the code ends with this byte (1) or not (0).

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

► **Table 5.5** Some examples of unary and γ codes. Unary codes are only shown for the smaller numbers. Commas in γ codes are for readability only and are not part of the actual codes.

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	1111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110,0000000001

for VB encoding and decoding and Table 5.4 an example of a VB-encoded postings list. [1]

With VB compression, the size of the compressed index for Reuters-RCV1 is 116 MB as we verified in an experiment. This is a more than 50% reduction of the size of the uncompressed index (see Table 5.6).

NIBBLE

The idea of VB encoding can also be applied to larger or smaller units than bytes: 32-bit words, 16-bit words, and 4-bit words or *nibbles*. Larger words further decrease the amount of bit manipulation necessary at the cost of less effective (or no) compression. Word sizes smaller than bytes get even better compression ratios at the cost of more bit manipulation. In general, bytes offer a good compromise between compression ratio and speed of decompression.

For most IR systems variable byte codes offer an excellent tradeoff between time and space. They are also simple to implement – most of the alternatives referred to in Section 5.4 are more complex. But if disk space is a scarce resource, we can achieve better compression ratios by using bit-level encodings, in particular two closely related encodings: γ codes, which we will turn to next, and δ codes (Exercise 5.9).

5.3.2 γ codes

VB codes use an adaptive number of *bytes* depending on the size of the gap. Bit-level codes adapt the length of the code on the finer grained *bit* level. The

1. Note that the origin is 0 in the table. Because we never need to encode a docID or a gap of 0, in practice the origin is usually 1, so that 10000000 encodes 1, 10000101 encodes 6 (not 5 as in the table), and so on.

UNARY CODE

simplest bit-level code is *unary code*. The unary code of n is a string of n 1s followed by a 0 (see the first two columns of Table 5.5). Obviously, this is not a very efficient code, but it will come in handy in a moment.

How efficient can a code be in principle? Assuming the 2^n gaps G with $1 \leq G \leq 2^n$ are all equally likely, the optimal encoding uses n bits for each G. So some gaps ($G = 2^n$ in this case) cannot be encoded with fewer than $\log_2 G$ bits. Our goal is to get as close to this lower bound as possible.

γ ENCODING

A method that is within a factor of optimal is γ *encoding*. γ codes implement variable-length encoding by splitting the representation of a gap G into a pair of *length* and *offset*. *Offset* is G in binary, but with the leading 1 removed.[2] For example, for 13 (binary 1101) *offset* is 101. *Length* encodes the length of *offset* in unary code. For 13, the length of *offset* is 3 bits, which is 1110 in unary. The γ code of 13 is therefore 1110101, the concatenation of length 1110 and offset 101. The right hand column of Table 5.5 gives additional examples of γ codes.

A γ code is decoded by first reading the unary code up to the 0 that terminates it, for example, the four bits 1110 when decoding 1110101. Now we know how long the offset is: 3 bits. The offset 101 can then be read correctly and the 1 that was chopped off in encoding is prepended: $101 \rightarrow 1101 = 13$.

The length of *offset* is $\lfloor \log_2 G \rfloor$ bits and the length of *length* is $\lfloor \log_2 G \rfloor + 1$ bits, so the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits. γ codes are always of odd length and they are within a factor of 2 of what we claimed to be the optimal encoding length $\log_2 G$. We derived this optimum from the assumption that the 2^n gaps between 1 and 2^n are equiprobable. But this need not be the case. In general, we do not know the probability distribution over gaps a priori.

ENTROPY

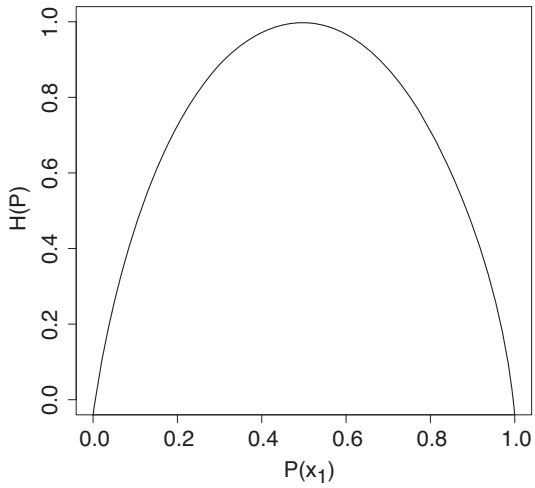
The characteristic of a discrete probability distribution[3] P that determines its coding properties (including whether a code is optimal) is its *entropy* $H(P)$, which is defined as follows:

$$H(P) = -\sum_{x \in X} P(x) \log_2 P(x)$$

where X is the set of all possible numbers we need to be able to encode (and therefore $\sum_{x \in X} P(x) = 1.0$). Entropy is a measure of uncertainty as shown in Figure 5.9 for a probability distribution P over two possible outcomes, namely, $X = \{x_1, x_2\}$. Entropy is maximized ($H(P) = 1$) for $P(x_1) = P(x_2) = 0.5$ when uncertainty about which x_i will appear next is largest; and

2. We assume here that G has no leading 0s. If there are any, they are removed before deleting the leading 1.

3. Readers who want to review basic concepts of probability theory may want to consult Rice (2006) or Ross (2006). Note that we are interested in probability distributions over integers (gaps, frequencies, etc.), but that the coding properties of a probability distribution are independent of whether the outcomes are integers or something else.



▶ **Figure 5.9** Entropy $H(P)$ as a function of $P(x_1)$ for a sample space with two outcomes x_1 and x_2.

minimized ($H(P) = 0$) for $P(x_1) = 1, P(x_2) = 0$ and for $P(x_1) = 0, P(x_2) = 1$ when there is absolute certainty.

It can be shown that the lower bound for the expected length $E(L)$ of a code L is $H(P)$ if certain conditions hold (see the references). It can further be shown that for $1 < H(P) < \infty$, γ encoding is within a factor of 3 of this optimal encoding, approaching 2 for large $H(P)$:

$$\frac{E(L_\gamma)}{H(P)} \leq 2 + \frac{1}{H(P)} \leq 3.$$

What is remarkable about this result is that it holds for any probability distribution P. So without knowing anything about the properties of the distribution of gaps, we can apply γ codes and be certain that they are within a factor of ≈ 2 of the optimal code for distributions of large entropy. A code like γ code with the property of being within a factor of optimal for an arbitrary distribution P is called *universal*.

UNIVERSAL CODE

In addition to universality, γ codes have two other properties that are useful for index compression. First, they are *prefix free*, namely, no γ code is the prefix of another. This means that there is always a unique decoding of a sequence of γ codes – and we do not need delimiters between them, which would decrease the efficiency of the code. The second property is that γ codes are *parameter free*. For many other efficient codes, we have to fit the parameters of a model (e.g., the binomial distribution) to the distribution

PREFIX FREE

PARAMETER FREE

of gaps in the index. This complicates the implementation of compression and decompression. For instance, the parameters need to be stored and retrieved. And in dynamic indexing, the distribution of gaps can change, so that the original parameters are no longer appropriate. These problems are avoided with a parameter-free code.

How much compression of the inverted index do γ codes achieve? To answer this question we use Zipf's law, the term distribution model introduced in Section 5.1.2. According to Zipf's law, the collection frequency cf_i is proportional to the inverse of the rank i, that is, there is a constant c' such that:

$$\text{cf}_i = \frac{c'}{i}. \tag{5.3}$$

We can choose a different constant c such that the fractions c/i are relative frequencies and sum to 1 (that is, $c/i = \text{cf}_i/T$):

$$1 = \sum_{i=1}^{M} \frac{c}{i} = c \sum_{i=1}^{M} \frac{1}{i} \quad = \quad c\, H_M \tag{5.4}$$

$$c = \frac{1}{H_M} \tag{5.5}$$

where M is the number of distinct terms and H_M is the Mth harmonic number. [4] Reuters-RCV1 has $M = 400{,}000$ distinct terms and $H_M \approx \ln M$, so we have

$$c = \frac{1}{H_M} \approx \frac{1}{\ln M} = \frac{1}{\ln 400{,}000} \approx \frac{1}{13}.$$

Thus the ith term has a relative frequency of roughly $1/(13i)$, and the expected average number of occurrences of term i in a document of length L is:

$$L\frac{c}{i} \approx \frac{200 \times \frac{1}{13}}{i} \approx \frac{15}{i}$$

where we interpret the relative frequency as a term occurrence probability. Recall that 200 is the average number of tokens per document in Reuters-RCV1 (Table 4.2).

Now we have derived term statistics that characterize the distribution of terms in the collection and, by extension, the distribution of gaps in the postings lists. From these statistics, we can calculate the space requirements for an inverted index compressed with γ encoding. We first stratify the vocabulary into blocks of size $Lc = 15$. On average, term i occurs $15/i$ times per

4. Note that, unfortunately, the conventional symbol for both entropy and harmonic number is H. Context should make clear which is meant in this chapter.

	N documents
Lc most frequent terms	N gaps of 1 each
Lc next most frequent terms	$N/2$ gaps of 2 each
Lc next most frequent terms	$N/3$ gaps of 3 each
...	...

► **Figure 5.10** Stratification of terms for estimating the size of a γ encoded inverted index.

document. So the average number of occurrences $\overline{f}$ per document is $1 \leq \overline{f}$ for terms in the first block, corresponding to a total number of N gaps per term. The average is $\frac{1}{2} \leq \overline{f} < 1$ for terms in the second block, corresponding to $N/2$ gaps per term, and $\frac{1}{3} \leq \overline{f} < \frac{1}{2}$ for terms in the third block, corresponding to $N/3$ gaps per term, and so on. (We take the lower bound because it simplifies subsequent calculations. As we will see, the final estimate is too pessimistic, even with this assumption.) We will make the somewhat unrealistic assumption that all gaps for a given term have the same size as shown in Figure 5.10. Assuming such a uniform distribution of gaps, we then have gaps of size 1 in block 1, gaps of size 2 in block 2, and so on.

Encoding the N/j gaps of size j with γ codes, the number of bits needed for the postings list of a term in the jth block (corresponding to one row in the figure) is:

$$\begin{aligned} \textit{bits-per-row} &= \frac{N}{j} \times (2 \times \lfloor \log_2 j \rfloor + 1) \\ &\approx \frac{2N \log_2 j}{j}. \end{aligned}$$

To encode the entire block, we need $(Lc) \cdot (2N \log_2 j)/j$ bits. There are $M/(Lc)$ blocks, so the postings file as a whole will take up:

$$\sum_{j=1}^{\frac{M}{Lc}} \frac{2NLc \log_2 j}{j}. \tag{5.6}$$

► **Table 5.6** Index and dictionary compression for Reuters-RCV1. The compression ratio depends on the proportion of actual text in the collection. Reuters-RCV1 contains a large amount of XML markup. Using the two best compression schemes, γ encoding and blocking with front coding, the ratio compressed index to collection size is therefore especially small for Reuters-RCV1: $(101 + 5.9)/3600 \approx 0.03$.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9
collection (text, xml markup etc)	3600.0
collection (text)	960.0
term incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ encoded	101.0

For Reuters-RCV1, $\frac{M}{Lc} \approx 400{,}000/15 \approx 27{,}000$ and

$$\sum_{j=1}^{27{,}000} \frac{2 \times 10^6 \times 15 \log_2 j}{j} \approx 224 \text{ MB}. \tag{5.7}$$

So the postings file of the compressed inverted index for our 960 MB collection has a size of 224 MB, one fourth the size of the original collection.

When we run γ compression on Reuters-RCV1, the actual size of the compressed index is even lower: 101 MB, a bit more than one tenth of the size of the collection. The reason for the discrepancy between predicted and actual value is that (i) Zipf's law is not a very good approximation of the actual distribution of term frequencies for Reuters-RCV1 and (ii) gaps are not uniform. The Zipf model predicts an index size of 251 MB for the unrounded numbers from Table 4.2. If term frequencies are generated from the Zipf model and a compressed index is created for these artificial terms, then the compressed size is 254 MB. So to the extent that the assumptions about the distribution of term frequencies are accurate, the predictions of the model are correct.

Table 5.6 summarizes the compression techniques covered in this chapter. The term incidence matrix (Figure 1.1, page 4) for Reuters-RCV1 has size $400{,}000 \times 800{,}000 = 40 \times 8 \times 10^9$ bits or 40 GB.

γ codes achieve great compression ratios – about 15% better than variable byte codes for Reuters-RCV1. But they are expensive to decode. This is because many bit-level operations – shifts and masks – are necessary to decode a sequence of γ codes as the boundaries between codes will usually be

somewhere in the middle of a machine word. As a result, query processing is more expensive for γ codes than for variable byte codes. Whether we choose variable byte or γ encoding depends on the characteristics of an application, for example, on the relative weights we give to conserving disk space versus maximizing query response time.

The compression ratio for the index in Table 5.6 is about 25%: 400 MB (uncompressed, each posting stored as a 32-bit word) versus 101 MB (γ) and 116 MB (VB). This shows that both γ and VB codes meet the objectives we stated in the beginning of the chapter. Index compression substantially improves time and space efficiency of indexes by reducing the amount of disk space needed, increasing the amount of information that can be kept in the cache, and speeding up data transfers from disk to memory.

Exercise 5.4 [⋆]

Compute variable byte codes for the numbers in Tables 5.3 and 5.5.

Exercise 5.5 [⋆]

Compute variable byte and γ codes for the postings list $\langle 777, 17743, 294068, 31251336 \rangle$. Use gaps instead of docIDs where possible. Write binary codes in 8-bit blocks.

Exercise 5.6

Consider the postings list $\langle 4, 10, 11, 12, 15, 62, 63, 265, 268, 270, 400 \rangle$ with a corresponding list of gaps $\langle 4, 6, 1, 1, 3, 47, 1, 202, 3, 2, 130 \rangle$. Assume that the length of the postings list is stored separately, so the system knows when a postings list is complete. Using variable byte encoding: (i) What is the largest gap you can encode in 1 byte? (ii) What is the largest gap you can encode in 2 bytes? (iii) How many bytes will the above postings list require under this encoding? (Count only space for encoding the sequence of numbers.)

Exercise 5.7

A little trick is to notice that a gap cannot be of length 0 and that the stuff left to encode after shifting cannot be 0. Based on these observations: (i) Suggest a modification to variable byte encoding that allows you to encode slightly larger gaps in the same amount of space. (ii) What is the largest gap you can encode in 1 byte? (iii) What is the largest gap you can encode in 2 bytes? (iv) How many bytes will the postings list in Exercise 5.6 require under this encoding? (Count only space for encoding the sequence of numbers.)

Exercise 5.8 [⋆]

From the following sequence of γ-coded gaps, reconstruct first the gap sequence and then the postings sequence: 1110001110101011111101101111011.

Exercise 5.9

δ CODES

γ codes are relatively inefficient for large numbers (e.g., 1025 in Table 5.5) as they encode the length of the offset in inefficient unary code. *δ codes* differ from γ codes in that they encode the first part of the code (*length*) in γ code instead of unary code. The encoding of *offset* is the same. For example, the δ code of 7 is 10,0,11 (again, we add commas for readability). 10,0 is the γ code for *length* (2 in this case) and the encoding of *offset* (11) is unchanged. (i) Compute the δ codes for the other numbers

► **Table 5.7** Two gap sequences to be merged in blocked sort-based indexing

γ encoded gap sequence of run 1	1110110111111001011111111110100011111001
γ encoded gap sequence of run 2	1111101000011111100010001111111001000011111010101

in Table 5.5. For what range of numbers is the δ code shorter than the γ code? (ii) γ code beats variable byte code in Table 5.6 because the index contains stop words and thus many small gaps. Show that variable byte code is more compact if larger gaps dominate. (iii) Compare the compression ratios of δ code and variable byte code for a distribution of gaps dominated by large gaps.

Exercise 5.10

Go through the above calculation of index size and explicitly state all the approximations that were made to arrive at Equation (5.6).

Exercise 5.11

For a collection of your choosing, determine the number of documents and terms and the average length of a document. (i) How large is the inverted index predicted to be by Equation (5.6)? (ii) Implement an indexer that creates a γ-compressed inverted index for the collection. How large is the actual index? (iii) Implement an indexer that uses variable byte encoding. How large is the variable byte encoded index?

Exercise 5.12

To be able to hold as many postings as possible in main memory, it is a good idea to compress intermediate index files during index construction. (i) This makes merging runs in blocked sort-based indexing more complicated. As an example, work out the γ-encoded merged sequence of the gaps in Table 5.7. (ii) Index construction is more space efficient when using compression. Would you also expect it to be faster?

Exercise 5.13

(i) Show that the size of the vocabulary is finite according to Zipf's law and infinite according to Heaps' law. (ii) Can we derive Heaps' law from Zipf's law?

5.4 References and further reading

Heaps' law was discovered by Heaps (1978). See also Baeza-Yates and Ribeiro-Neto (1999). A detailed study of vocabulary growth in large collections is (Williams and Zobel 2005). Zipf's law is due to Zipf (1949). Witten and Bell (1990) investigate the quality of the fit obtained by the law. Other term distribution models, including K mixture and two-poisson model, are discussed by Manning and Schütze (1999, Chapter 15). Carmel et al. (2001), Büttcher and Clarke (2006), Blanco and Barreiro (2007), and Ntoulas and Cho (2007) show that lossy compression can achieve good compression with no or no significant decrease in retrieval effectiveness.

Dictionary compression is covered in detail by Witten et al. (1999, Chapter 4), which is recommended as additional reading.

Subsection 5.3.1 is based on (Scholer et al. 2002). The authors find that variable byte codes process queries two times faster than either bit-level compressed indexes or uncompressed indexes with a 30% penalty in compression ratio compared with the best bit-level compression method. They also show that compressed indexes can be superior to uncompressed indexes not only in disk usage, but also in query processing speed. Compared with VB codes, "variable nibble" codes showed 5% to 10% better compression and up to one third worse effectiveness in one experiment (Anh and Moffat 2005). Trotman (2003) also recommends using VB codes unless disk space is at a premium. In recent work, Anh and Moffat (2005; 2006a) and Zukowski et al. (2006) have constructed word-aligned binary codes that are both faster in decompression and at least as efficient as VB codes. Zhang et al. (2007) investigate the increased effectiveness of caching when a number of different compression techniques for postings lists are used on modern hardware.

δ codes (Exercise 5.9) and γ codes were introduced by Elias (1975), who proved that both codes are universal. In addition, δ codes are asymptotically optimal for $H(P) \rightarrow \infty$. δ codes perform better than γ codes if large numbers (greater than 15) dominate. A good introduction to information theory, including the concept of entropy, is (Cover and Thomas 1991). While Elias codes are only asymptotically optimal, arithmetic codes (Witten et al. 1999, Section 2.4) can be constructed to be arbitrarily close to the optimum $H(P)$ for any P.

PARAMETERIZED CODE

GOLOMB CODES

Several additional index compression techniques are covered by Witten et al. (1999; Sections 3.3 and 3.4 and Chapter 5). They recommend using *parameterized codes* for index compression, codes that explicitly model the probability distribution of gaps for each term. For example, they show that *Golomb codes* achieve better compression ratios than γ codes for large collections. Moffat and Zobel (1992) compare several parameterized methods, including LLRUN (Fraenkel and Klein 1985).

The distribution of gaps in a postings list depends on the assignment of docIDs to documents. A number of researchers have looked into assigning docIDs in a way that is conducive to the efficient compression of gap sequences (Moffat and Stuiver 1996; Blandford and Blelloch 2002; Silvestri et al. 2004; Blanco and Barreiro 2006; Silvestri 2007). These techniques assign docIDs in a small range to documents in a cluster where a cluster can consist of all documents in a given time period, on a particular web site, or sharing another property. As a result, when a sequence of documents from a cluster occurs in a postings list, their gaps are small and can be more effectively compressed.

Different considerations apply to the compression of term frequencies and word positions than to the compression of docIDs in postings lists. See Scholer et al. (2002) and Zobel and Moffat (2006). Zobel and Moffat (2006) is recommended in general as an in-depth and up-to-date tutorial on inverted

indexes, including index compression.

This chapter only looks at index compression for Boolean retrieval. For ranked retrieval (Chapter 6), it is advantageous to order postings according to term frequency instead of docID. During query processing, the scanning of many postings lists can then be terminated early because smaller weights do not change the ranking of the highest ranked k documents found so far. It is not a good idea to precompute and store weights in the index (as opposed to frequencies) because they cannot be compressed as well as integers (see Section 7.1.5, page 140).

Document compression can also be important in an efficient information retrieval system. de Moura et al. (2000) and Brisaboa et al. (2007) describe compression schemes that allow direct searching of terms and phrases in the compressed text, which is infeasible with standard text compression utilities like gzip and compress.

Exercise 5.14 [⋆]

We have defined unary codes as being "10": sequences of 1s terminated by a 0. Interchanging the roles of 0s and 1s yields an equivalent "01" unary code. When this 01 unary code is used, the construction of a γ code can be stated as follows: (1) Write G down in binary using $b = \lfloor \log_2 j \rfloor + 1$ bits. (2) Prepend $(b - 1)$ 0s. (i) Encode the numbers in Table 5.5 in this alternative γ code. (ii) Show that this method produces a well-defined alternative γ code in the sense that it has the same length and can be uniquely decoded.

Exercise 5.15 [⋆ ⋆ ⋆]

Unary code is not a universal code in the sense defined above. However, there exists a distribution over gaps for which unary code is optimal. Which distribution is this?

Exercise 5.16

Give some examples of terms that violate the assumption that gaps all have the same size (which we made when estimating the space requirements of a γ-encoded index). What are general characteristics of these terms?

Exercise 5.17

Consider a term whose postings list has size n, say, $n = 10{,}000$. Compare the size of the γ-compressed gap-encoded postings list if the distribution of the term is uniform (i.e., all gaps have the same size) versus its size when the distribution is not uniform. Which compressed postings list is smaller?

Exercise 5.18

Work out the sum in Equation (5.7) and show it adds up to about 251 MB. Use the numbers in Table 4.2, but do not round Lc, c, and the number of vocabulary blocks.