Indexing

Compressing the information to be stored in a full-text database is only part of the solution to the information explosion. The techniques described in Chapter 2 may save a great deal of disk space, making it possible to store far more than might otherwise be handled, but compression does nothing to address the issue of how the information should be organized so that queries can be resolved efficiently and relevant portions of the data extracted quickly. For that, an index is necessary.

Most people are familiar with the use of an index in a book—there is one at the end of this book, for example, and if you look up the word *index* in it, it should refer you to this page. Using an index, it is possible to find information without resorting to a page-by-page search, and, provided that the index itself can be understood, it is possible to locate relevant pages in a book even if the book is written in another language. Indeed, if you wanted to obtain information from a book written in a foreign language, having an index would save an enormous amount of effort, since a translator could then be employed to "decode" just the pages actually required rather than the entire book. Although this scenario may sound far-fetched, it is in fact exactly the situation we are advocating in this book since the compressed documents that are stored in an information retrieval system might as well be stored in a foreign language, and some translation cost must certainly be paid to access them.

A book without an index can give rise to great frustration. Most people, at some time or another, have looked through a book for something they are sure is there but they simply cannot find. Tracking down the telephone number of a government department in a telephone directory is one such task that immediately springs to mind—you are never quite sure whether to look for "Taxation Department," or "Department of Taxation," or "Federal Office of Revenue," and so on. (In New Zealand, the correct answer is "Inland Revenue Department"; in Australia, it is the

"Australian Taxation Office"; in Canada, it is "Revenue Canada" or "Revenu Canada"; and in the United States, it is universally known as the "IRS.")

This difficulty of searching is the result of an inadequate index or no index at all. Of course, with a normal book (including this one) it is possible to skim-read every page, with a reasonable chance of being able to zero in, by various contextual clues, to the desired section. But with computers we are talking about gigabytes of data, millions of pages rather than hundreds, and with little structure and no contextual clues such as headings. Casual browsing of this much data by human means would be very costly, and even exhaustive searching by mechanical means is expensive. If no index is available, efforts to extract information are doomed to failure. For this reason, it is crucial to the success of an automated retrieval system that the stored information be accurately and comprehensively indexed; otherwise we might as well not have bothered accumulating the documents in the first place.

In this chapter we discuss a variety of indexing methods and show how the resulting indexes can themselves be compressed. For the most part it is supposed that a *document collection* or *document database* can be treated as a set of separate *documents*, each described by a set of *representative terms*, or simply *terms*, and that the index must be capable of identifying all documents that contain combinations of specified terms or are in some other way judged to be relevant to the set of query terms. A document will thus be the unit of text that is returned in response to queries.

For example, if the database consists of a collection of electronic office memoranda, and each memo is taken to be one document, then the representative terms might be the recipient's name, the sender's name, the date, and the subject line of the memo. It would then be possible to issue queries such as *find memos from Jane to John on the subject of taxation*. If a more detailed index is required, the entire text of the message might be regarded as its own set of representative terms, so that any words contained in the message could be used as query terms. If the documents are images, the terms to be indexed might be a few words describing each image, and a query might, for instance, ask that all images containing an *elephant* be retrieved. Note that in this latter case it is supposed that someone has examined the collection of images and decided in advance (by creating representative terms) which ones show elephants. The task of taking an arbitrary image and deciding mechanically what objects are portrayed is a major research area in its own right and is certainly not the subject of this book. Nevertheless, for certain restricted types of image, such as faxes and other scanned text, it is sometimes possible to infer a set of representative terms using OCR.

There will also be situations in which it is sensible to choose a document in the database to be one paragraph, or even just one sentence, of a source document. This would allow paragraphs that meet some requirement to be extracted, independent of the text in which they are embedded. In the previous example of office memos, it would be of dubious merit, but certainly possible, to define each field as one document—one document storing the sender, another the recipient, another the subject, and a fourth the actual message text. Similarly, it would be possible, but probably confusing, to store as a document a group of 10 memos on disparate

topics. As in this example, it is normally easy to decide, for a given collection, what the documents should be.

The database designer is also free to choose the *granularity* of the index—the resolution to which term locations are recorded within each document. Having decided for the office information system that a document will be a single memo, the system implementer may still require that the index be capable of ascertaining a more exact location within the document of each term, so documents in which the words *tax* and *avoidance* appear in the same sentence can be located using only the index, without recourse to extensive checking of every document in which they appear anywhere.

In the limit, if the granularity of the index is taken to be one word, then the index will record the exact location of every word in the collection, and so (with some considerable effort) the original text can be recovered from the index. In this case it is unlikely that the index can be stored in less space than the least amount that is possible for the main text using a normal text compression algorithm. If it could, the index compression method could be used as a better text compression algorithm, and, given the discussion in Chapter 2, that seems unlikely.

When the granularity of the index is coarser—to the sentence or document level—the input text can no longer be reproduced from the index, and a more economical representation becomes possible. Most of this chapter is devoted to *index compression*, the problem of representing the index efficiently. Each entry in a document-level index is a pointer to a particular document, and for a collection of a million documents, such a pointer would take 20 bits uncompressed. However, it is possible to reduce this to about 6 bits for typical document collections, a very worthwhile saving indeed.

The database designer is also free to decide how the representative terms for textual documents should be created. One simple possibility is to take each of the words that appears in the document and declare it verbatim to be a term. This tends to both enlarge the vocabulary of the collection—the number of distinct terms that appear—and increase the number of document identifiers that must be stored in the index. Having an overlarge vocabulary not only affects the storage space requirements of the system but can also make it harder to use since there are more potential query terms that must be considered when formulating requests of the system. For these reasons it is more usual for each word to be transformed in some way before being included in the index.

The first of these transformations is known as *case folding*—the conversion of all uppercase letters to their lowercase equivalents (or vice versa). For example, if all uppercase letters are folded to lowercase, *ACT*, *Act*, and *act* are all indexed as *act* and are regarded as equivalent at query time, irrespective of which original version appeared in the source document. This transformation is carried out so that those querying the database need not guess the exact case that has been used and can pose case-invariant queries. Certainly, we would not wish to distinguish between the two sentences *Data compression* . . . and *Compression of data* . . . when querying on *data* AND *compression*—both sentences (or rather, the documents containing them) should be retrieved. Of course, as with most generalizations, there are

counterexamples. In Australia, "ACT" stands for Australian Capital Territory, which is the seat of the federal government. This is quite different from the verb *to act* and only tenuously related to the noun *Act* (of parliament). And, given the authorship of this book, unification of *Bell* and *bell* might also introduce problems.

A second, less obvious, transformation is for words to be reduced to their morphological roots—that is, for all suffixes and other modifiers to be removed. For example, *compression*, *compressed*, and *compressor* all have the word *compress* as their common root. This process is known as *stemming* and is carried out so that queries retrieve relevant documents even if the exact form of the word is different. If the representative terms are created using stemming, and all query terms are also stemmed, the query *data* AND *compression* would retrieve documents containing phrases such as *compressed data is* and also documents containing the likes of *to compress the data*. It is difficult to deny the usefulness of this transformation, but the user needs to remember that it is taking place, as it can easily cause the retrieval of seemingly extraneous material.

A final transformation that is sometimes applied is the omission of *stop words*—words that are deemed to be sufficiently common or of such small information content that their use in a query would be unlikely to eliminate any documents since they are likely to be present in almost every document. Hence, nothing will be lost if they are simply excluded from the index. At the top of any stop word list for English is usually *the*, closely followed by *a*, *it*, and so on. Other terms might also be stopped in particular applications—in an online computer manual, appearances of the terms *options* and *usage* might not be indexed, and a financial archive might choose to stop words such as *dollar* and *stock* and perhaps even *Dow* and *Jones*. One automatic method that is sometimes used to derive a set of stop words is to determine, for each term, the extent to which it can be described by a random process, accepting as stop words those that appear in the collection as if they were randomly distributed.

All these transformations, and the effect they have upon index size, are considered in this chapter. A further possible transformation that we do not consider is that of *thesaural substitution*, where synonyms—*fast* and *rapid*, for example—are identified and indexed under a single representative term.

3.1 Sample document collections

To allow practical comparison of various algorithms and techniques, experiments have been performed on some real-life document collections. This section describes the four collections that have been used in preparing this book.[1] Some statistics for

1 Three of the four collections were modified slightly (to correct errors in the data) between 1993, when the results of the first edition of this book were prepared, and 1998, when the second edition was prepared. The stemming program used in 1998 is also different from that used in 1993. This is why most of the collection statistics listed in Table 3.1 are different from

Table 3.1 Statistics of document collections.

		Collection			
		Bible	*GNUbib*	*Comact*	*TREC*
Documents	N	31,101	64,343	261,829	741,856
Number of terms	F	884,994	2,570,906	22,805,920	333,338,738
Distinct terms	n	8,965	46,488	36,660	535,346
Index pointers	f	701,412	2,226,300	12,976,418	134,994,414
Total size (Mbytes)		4.33	14.05	131.86	2070.29

```
Genesis 1 1
In the beginning God created the heaven and the earth.

Genesis 1 2
And the earth was without form, and void; and darkness was
upon the face of the deep.  And the Spirit of God moved upon
the face of the waters.
```

Figure 3.1 Sample text from the *Bible* collection.

them are listed in Table 3.1. In Table 3.1, and throughout the remainder of this book, N is used to denote the number of documents in some collection; n is the number of distinct terms—that is, stemmed words—that appear; F is the total number of terms in the collection; and f indicates the number of *pointers* that appear in a document-level index. That is, f is the number of distinct "document, word" pairs to be stored—the size of the index.

Collection *Bible* is the King James version of the Bible, with each verse taken to be a document, including the book name, chapter number, and verse number. The first two documents in this collection, shown in Figure 3.1, are the well-known verses from Genesis.

The second collection, *GNUbib*, is a set of about 65,000 citations to papers that have appeared in the computing literature. These documents are again very short; for example, all of document number 8,425 is shown in Figure 3.2.

Collection *Comact* stores the Commonwealth Acts of Australia, from the 1901 constitution under which Australia became a federation through legislation passed in 1989. Each document in this collection corresponds to one physical page of an original printed version and contains 50 to 100 words. Two typical pages are

---

those listed in the corresponding table in the first edition. The various compression results in the remainder of this chapter are correct for the modified collections.

```
%A Ian H. Witten
%A Radford M. Neal
%A John G. Cleary
%T Arithmetic coding for data compression
%J Communications of the ACM
%K cacm
%V 30
%N 6
%D June 1987
%P 520--540
```

Figure 3.2 Sample text from *GNUbib*.

```
Page 92011
EVIDENCE AMENDMENT ACT 1978 No. 14 of 1978---SECT. 3.
''derived'' means derived, by the use of a computer or
otherwise, by calculation, comparison, selection, sorting or
consolidation or by accounting, statistical or logical
procedures;
''document'' includes---
(a) a book, plan, paper, parchment, film or other material
on which there is writing or printing, or on which there are
marks, symbols or perforations having a meaning for persons
qualified to interpret them;

Page 92012
EVIDENCE AMENDMENT ACT 1978 No. 14 of 1978---SECT. 3.
(b) a disc, tape, paper, film or other device from which
sounds or images are capable of being reproduced; and
(c) any other record of information;
''proceeding'' means a proceeding before the High Court or
any court (other than a court of a Territory) created by the
Parliament;
''qualified person,'' in relation to a statement made in the
course of, or for the purposes of, a business, means a
person who---
```

Figure 3.3 Sample text from *Comact*.

shown in Figure 3.3. As in all these examples, some liberties have been taken with formatting—line breaks have been altered and some white space has been removed.

The final collection that has been used is the first two disks of *TREC*, an acronym for Text REtrieval Conference. This is a very large document collection distributed to research groups worldwide for comparative information retrieval experiments. The documents in the first two disks of *TREC* are taken from five sources: the Asso-

ciated Press newswire; the U.S. Department of Energy; the U.S. Federal Register; the *Wall Street Journal*; and a selection of computer magazines and journals published by Ziff-Davis. In these five subcollections there is a total of more than 2 Gbytes of text and nearly 750,000 documents. These documents are much longer than those of the other three collections, averaging about 450 words. One document in the Federal Register subcollection is more than 2.5 Mbytes long. All the documents contain embedded standard generalized markup language (SGML) commands; one document (selected because it matched the query *managing* AND *gigabytes*) is shown in part in Figure 3.4. In dealing with *TREC*, all the SGML tags were stripped out before any indexing took place, and the values reported in Table 3.1 exclude the SGML markup.

It is also worth stating the definition of *word* that was used to obtain the statistics in Table 3.1. A practical rule of thumb for identifying words for indexing is

---

A word is a maximal sequence of alphanumeric characters, but limited to at most 256 characters in total and at most four numeric characters.

The latter restriction is to avoid sequences of page numbers becoming long runs of distinct words. Without this restriction, the size of the vocabulary might be unnecessarily inflated. For example, *Comact* contains 261,829 pages, beginning with page 1. Using this definition, a string such as *92011* is parsed as two distinct words, *9201* and *1*. Similarly, queries on *92011* are expanded to become *9201* AND *1*, introducing some small but acceptable likelihood of *false matches*—documents that satisfy the query according to the index but in fact are not answers—on queries involving large numbers. For example, a document containing the text *totalling 9201, of which 1* would be retrieved as a false match. (A more robust but more expensive strategy is to expand a query on *92011* into *9201* AND *2011*, supposing that a similar rule had been used during the creation of the index.) Years, such as *1901*, at four digits, were preserved as words. Of course, this whole strategy would have to be revised for a document such as the dictionary of real numbers mentioned in Chapter 1 (page 11) (Borwein and Borwein 1990).

All the words thus parsed are then stemmed to produce index terms, as described in Section 3.7.

3.2 Inverted file indexing

An index is a mechanism for locating a given term in a text. There are many ways to achieve this. In applications involving text, the single most suitable structure is an *inverted file*, sometimes known as a *postings file* and in normal English usage as a *concordance*. Other mechanisms—notably *signature files* and *bitmaps*—can also be used and may be appropriate in certain restricted applications. The emphasis in this section is on inverted file indexing; signature file and bitmap indexing are discussed

```
<DOC>
<DOCNO> ZF07-781-012 </DOCNO>
<DOCID> 07 781 012.  </DOCID>
<JOURNAL> Government Computer News Oct 16 1989 v8 n21 p39(2)
* Full Text COPYRIGHT Ziff-Davis Pub.  Co.  1989.
</JOURNAL>
<TITLE> Compressing data spurs growth of imaging.  </TITLE>
<AUTHOR> Hosinski, Joan M. </AUTHOR>
<DESCRIPTORS> Topic:  Data Compression, Data Communications,
Optical Disks, Imaging Technology.  Feature:  illustration
chart.  Caption:  Path taken by image file.  (chart)
</DESCRIPTORS>
<TEXT>
Compressing Data Spurs Growth of imaging

Data compression has spurred the growth of imaging
applications, many of which require users to send large
amounts of data between two locations, an Electronic Trend
Publications report said.
   Data compression is an ''essential enabling technology''
and the ''importance of the compression step is comparable
to the importance of the optical disk as a cost-effective
storage medium,'' the Saratoga, Calif., company said in the
report, Data Compression Impact on Document and Image
Processing Storage and Retrieval.
   Document images need to be viewed at a resolution of 100
to 300 dots per inch, and files quickly grow to the gigabyte
or terabyte range, the report said.  Data typically
compresses at a 10-to-1 ratio, but can go up to a 60--to-1
ratio.
   ''Use of document imaging has been slow to unfold,'' but
it is gaining acceptability beyond desktop publishing, where
document imaging already has been used, researchers said.
However, document imaging can be complex and can be
misapplied, they said.  Also, vendors have changed standards
or used only subsets of the standards in their products.
   The Defense Department's Computer-Aided Logistics Support
(CALS) program and use of image compression format standards
will help the government avoid problems with interchanging
data between systems, researchers said.

[Three paragraphs omitted]

</TEXT>
</DOC>
```

Figure 3.4 Sample text from *TREC*.

Table 3.2 Example text, each line is one document

Document	Text
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

in Section 3.5, and then Section 3.6 examines the factors that influence the choice of indexing method. However, as an initial rule of thumb:

> In most applications inverted files offer better performance than signature files and bitmaps, in terms of both size of index and speed of query handling.

Let us now define exactly what we mean by an inverted file index. An inverted file contains, for each term in the lexicon, an *inverted list* that stores a list of pointers to all occurrences of that term in the main text, where each pointer is, in effect, the number of a document in which that term appears. The inverted list is also sometimes known as a *postings list* and the pointers as *postings*. This is perhaps the most natural indexing method, corresponding closely to the index of a book and to the traditional use of concordances as an adjunct to the study of classical tracts such as the Bible and the Koran.

An inverted file index also requires a *lexicon*—a list of all terms that appear in the database. (The word "vocabulary" is also used to denote this list. We prefer "lexicon" when talking about the data structure that holds the list and "vocabulary" when referring to linguistic aspects of the text.) The lexicon supports a mapping from terms to their corresponding inverted lists and in its simplest form is a list of strings and disk addresses.

As an example of an inverted file index, consider the traditional children's nursery rhyme in Table 3.2, with each line taken to be a document for indexing purposes. The inverted file generated for this text is shown in Table 3.3, where the terms have been case-folded but with no stemming applied and no words stopped. Because of the unusual nature of the example, each word appears in exactly two of the lines. This would not normally be the case, and in general, the inverted lists for a collection are of widely differing lengths.

A query involving a single term is answered by scanning its inverted list and retrieving every document that it cites. For conjunctive Boolean queries of the form *term* AND *term* AND ... AND *term*, the intersection of the terms' inverted lists is formed. For disjunction, where the operator is OR, the union is taken; for negation

Table 3.3 Inverted file for text of Table 3.2

Number	Term	Documents
1	cold	⟨2; 1, 4⟩
2	days	⟨2; 3, 6⟩
3	hot	⟨2; 1, 4⟩
4	in	⟨2; 2, 5⟩
5	it	⟨2; 4, 5⟩
6	like	⟨2; 4, 5⟩
7	nine	⟨2; 3, 6⟩
8	old	⟨2; 3, 6⟩
9	pease	⟨2; 1, 2⟩
10	porridge	⟨2; 1, 2⟩
11	pot	⟨2; 2, 5⟩
12	some	⟨2; 4, 5⟩
13	the	⟨2; 2, 5⟩

using NOT, the complement is taken. The inverted lists are usually stored in order of increasing document number, so that these various merging operations can be performed in a time that is linear in the size of the lists. As an example, to locate lines containing *some* AND *hot* in the text of Table 3.2, the lists for the two words—⟨4, 5⟩ and ⟨1, 4⟩, respectively—are merged (or, strictly speaking, intersected), yielding the lines that they have in common, in this case the list ⟨4⟩. This line is then fetched, using whatever mechanism is being used to store the main text, and finally displayed.

The *granularity* of an index is the accuracy to which it identifies the location of a term. A coarse-grained index might identify only a block of text, where each block stores several documents; an index of moderate grain will store locations in terms of document numbers; while a fine-grained one will return a sentence or word number, perhaps even a byte number. Coarse indexes require less storage, but during retrieval, more of the plain text must be scanned to find terms. Also, with a coarse index, multiterm queries are more likely to give rise to *false matches*, where each of the desired terms appears somewhere in the block, but not all within the same document. At the other extreme, word-level indexing enables queries involving adjacency and proximity—for example, *text compression* as a phrase rather than as two individual words *text* AND *compression*—to be answered quickly because the desired relationship can be checked before the text is retrieved. However, adding precise locational information expands the index by at least a factor of two or three compared with a document-level index since not only are there more pointers in the index (as explained below), but each one requires more bits of storage because it indicates a more precise location. Unless a significant fraction of the queries are expected to be proximity-based, the usual granularity is to individual documents. Proximity-

Table 3.4 Word-level inverted file for text of Table 3.2

Number	Term	(Document; Words)
1	cold	⟨2; (1; 6), (4; 8)⟩
2	days	⟨2; (3; 2), (6; 2)⟩
3	hot	⟨2; (1; 3), (4; 4)⟩
4	in	⟨2; (2; 3), (5; 4)⟩
5	it	⟨2; (4; 3, 7), (5; 3)⟩
6	like	⟨2; (4; 2, 6), (5; 2)⟩
7	nine	⟨2; (3; 1), (6; 1)⟩
8	old	⟨2; (3; 3), (6; 3)⟩
9	pease	⟨2; (1; 1, 4), (2; 1)⟩
10	porridge	⟨2; (1; 2, 5), (2; 2)⟩
11	pot	⟨2; (2; 5), (5; 6)⟩
12	some	⟨2; (4; 1, 5), (5; 1)⟩
13	the	⟨2; (2; 4), (5; 5)⟩

and phrase-based queries can then be handled by the slightly slower method of a postretrieval scan.

Table 3.4 shows the text of Table 3.2 indexed by word number within document number, where the notation $(x; y_1, y_2, \ldots)$ indicates that the given word appears in document x as word number y_1, y_2, ... To find lines containing *hot* and *cold* less than two words apart, the two lists are again merged, but this time pairs of entries (one from each list) are only accepted when the same document number appears and the word number components differ by less than two. In this example there are no such entries, so nothing is read from the main text. The coarser inverted file of Table 3.3 gives two false matches, which require certain lines of the text to be checked and discarded.

Notice that the index has grown bigger. There are two reasons for this. First, there is more information to be stored for each pointer—a word number as well as a document number—and, given the discussion in Chapter 2, it is not surprising that more precise locational information requires a longer description. Second, several words appear more than once in a line. In the index of Table 3.3, duplicate appearances are represented with a single pointer, but in the word-level index of Table 3.4, both appearances require an entry. A word-level index must, of necessity, store one value for each word in the text (the value F in Table 3.1), while a document-level index benefits from multiple appearances of the same word within the document and stores fewer pointers (listed as f in Table 3.1).

More generally, an inverted file stores a hierarchical set of addresses—in an extreme case, a word number within a sentence number within a paragraph number within a chapter number within a volume number within a document number.

Each term location could be considered to be a vector (d, v, c, p, s, w) in coordinate form. However, within each coordinate the list of addresses can always be stored in the form illustrated in Table 3.4, and all the representations described in this chapter generalize readily to the multidimensional situation.

For this reason, throughout the following discussion it will be assumed that the index is a simple document-level one. In fact, given that a document can be defined to be a very small unit, such as a sentence or verse (as it is for the *Bible* database), in some ways word-level indexing is just an extreme case in which each word is defined as a document.

Uncompressed inverted files can consume considerable space and might occupy 50 to 100 percent of the space of the text itself. For example, in typical English prose the average word contains about five characters, and each word is normally followed by one or two bytes of white space or punctuation characters. Stored as 32-bit document numbers, and supposing that there is no duplication of words within documents, there might thus be four bytes of inverted list pointer information for every six bytes of text. If a two-byte "word number within a document" field is added to each pointer, the index consumes six bytes for roughly each six bytes of text.

For a text of N documents and an index containing f pointers, the total space required with a naive representation is $f \cdot \lceil \log N \rceil$ bits, provided that pointers are stored in a minimal number of bits.[2] Using 20-bit pointers to store the *TREC* document numbers gives a 324 Mbyte inverted file. This is already a form of compression compared to the more convenient 32-bit numbers usually used when programming, but even so, the index occupies a sizable fraction of the space taken to store the text. For the same collection, a word-level inverted file using 29-bit pointers requires approximately 1,200 Mbytes.

The use of a stop list (or rather, the omission of a set of stop words from the index) yields significant savings in an uncompressed inverted file since common terms usually account for a sizable fraction of total word occurrences. However, as will be demonstrated in the next section, there are more elegant ways to obtain the same space savings and still retain all terms as index words. Our favored approach is that all terms should be indexed—even if, to make query processing faster, they are simply ignored when present in queries.

3.3 Inverted file compression

The size of an inverted file can be reduced considerably by compressing it. This section describes models and coding methods to achieve this.

The key to compression is the observation that each inverted list can, without any loss of generality, be stored as an ascending sequence of integers. For example, sup-

2 The notation $\lceil x \rceil$ indicates the smallest integer greater than or equal to x; hence, $\lceil 3.3 \rceil = 4$. Similarly, $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x; $\lfloor 3.3 \rfloor = 3$.

pose that some term appears in eight documents of a collection—those numbered 3, 5, 20, 21, 23, 76, 77, 78. This term is described in the inverted file by a list:

$$\langle 8; 3, 5, 20, 21, 23, 76, 77, 78 \rangle,$$

the address of which is contained in the lexicon. More generally, the list for a term t stores the number of documents f_t in which the term appears and then a list of f_t document numbers:

$$\langle f_t; d_1, d_2, \ldots, d_{f_t} \rangle,$$

where $d_k < d_{k+1}$. Because the list of document numbers within each inverted list is in ascending order, and all processing is sequential from the beginning of the list, the list can be stored as an initial position followed by a list of *d-gaps*, the differences $d_{k+1} - d_k$. That is, the list for the term above could just as easily be stored as

$$\langle 8; 3, 2, 15, 1, 2, 53, 1, 1 \rangle.$$

No information has been lost, since the original document numbers can always be obtained by calculating cumulative sums of the d-gaps.

The two forms are equivalent, but it is not obvious that any saving has been achieved. The largest d-gap in the second representation is still potentially the same as the largest document number in the first, and so if there are N documents in the collection and a flat binary encoding is used to represent the gap sizes, both methods require $\lceil \log N \rceil$ bits per stored pointer. Nevertheless, considering each inverted list as a list of d-gaps, the sum of which is bounded by N, allows improved representation, and it is possible to code inverted lists using on average substantially fewer than $\lceil \log N \rceil$ bits per pointer.

Many specific models have been proposed for describing the probability distribution of d-gap sizes. The ones we will look at are listed in Table 3.5, along with references to papers where they are described. They are grouped into two broad classes: *global* methods, in which every inverted list is compressed using the same common model, and *local* methods, where the compression model for each term's list is adjusted according to some stored parameter, usually the frequency of the term. Local models tend to outperform global ones in terms of compression and are no less efficient in terms of the processing time required during decoding, though they tend to be somewhat more complex to implement. Global models themselves divide into *parameterized* and *nonparameterized*, the latter being fixed codes and the former involving some parameter that can be tailored to the actual distribution of gap sizes. Local methods are always parameterized—otherwise there would be no point in using them.

Global models are generally outperformed by local ones, and the following rule holds:

For the majority of practical purposes, the most suitable index compression technique is the local Bernoulli method, implemented using a technique called Golomb coding.

Table 3.5 Some methods for compressing inverted files.

Method	Reference
Global methods	
Nonparameterized	
Unary	
Binary	
γ	Elias (1975); Bentley and Yao (1976)
δ	Elias (1975); Bentley and Yao (1976)
Parameterized	
Bernoulli	Golomb (1966); Gallager and Van Voorhis (1975)
Observed frequency	
Local methods	
Bernoulli	Witten, Bell, and Nevill (1992); Bookstein, Klein, and Raita (1992)
Skewed Bernoulli	Teuhola (1978); Moffat and Zobel (1992)
Hyperbolic	Schuegraf (1976)
Observed frequency	
Batched frequency	Moffat and Zobel (1992)
Interpolative	Moffat and Stuiver (1996)

In the subsections that follow, we work our way through these coding methods. If you are in a hurry, you can skip to the sections on global Bernoulli models (page 119) and local Bernoulli models (page 121), but the material on the other schemes provides a fascinating account of the development of different coding methods. These apply not just to index compression: they can be used for different purposes and in other applications.

Nonparameterized models

The simplest global codes are fixed representations of the positive integers. For example, as has already been considered, if there are N documents in the collection, a flat binary encoding might be used, requiring $\lceil \log N \rceil$ bits for each pointer.

Shannon's relationship between ideal code length l_x and symbol probability $\Pr[x]$, namely, $l_x = -\log \Pr[x]$, allows the probability distribution implied by any particular encoding method to be determined. The implicit probability model associated with a flat binary encoding is that each d-gap size in each inverted list will be uniformly random in $1 \ldots N$, which is not a very accurate reflection of reality.

Thinking of a code in terms of the implied probability distribution is a good way to assess intuitively whether it is likely to do well, and when considered in this light, it seems unlikely that all gap sizes are equally probable. For example, common

Table 3.6 Example codes for integers

Gap x	Coding Method				
	Unary	γ	δ	Golomb	
				$b=3$	$b=6$
1	0	0	0	0 0	0 0 0
2	10	10 0	100 0	0 10	0 0 1
3	110	10 1	100 1	0 11	0 100
4	1110	110 00	101 00	10 0	0 101
5	11110	110 01	101 01	10 10	0 110
6	111110	110 10	101 10	10 11	0 111
7	1111110	110 11	101 11	110 0	10 00
8	11111110	1110 000	11000 000	110 10	10 01
9	111111110	1110 001	11000 001	110 11	10 100
10	1111111110	1110 010	11000 010	1110 0	10 101

words are likely to have small gaps between appearances—otherwise, they could not end up occurring frequently. Similarly, infrequent words are likely to have gaps that are very large, although if documents are stored in chronological or some other logical sequence, it may well be that appearances of rare terms tend to cluster and be nonuniform throughout the collection. Thus, variable-length representations should be considered in which small values are considered more likely, and coded more economically, than large ones.

One such code is the unary code. In this code an integer $x \geq 1$ is coded as $x-1$ one bits followed by a zero bit, so that the code for integer 3 is 110. The second column of Table 3.6 shows some unary codes. Although unary coding is certainly biased in favor of short gaps, the bias is usually far too extreme. An inverted list coded in unary will require d_{f_t} bits, since the code for a gap of x requires x bits, and in each inverted list the sum of the gap sizes is the document number d_{f_t} of the last appearance of the corresponding word. In total, an inverted file coded in unary might thus consume as many as $N \cdot n$ bits, and this quantity will generally be extremely large.

Looking at probabilities, it is apparent that the unary code is equivalent to assigning a probability of $\Pr[x] = 2^{-x}$ to gaps of length x, and this is far too small. However, unary coding does have its uses, and Chapters 4 and 5 describe some situations in which it is the method of choice.

There are many codes whose implied probability distributions lie somewhere between the uniform distribution assumed by a binary code and the binary exponential decay implied by the unary code. One is the γ code, which represents the number x as a unary code for $1 + \lfloor \log x \rfloor$ followed by a code of $\lfloor \log x \rfloor$ bits that represents the value of $x - 2^{\lfloor \log x \rfloor}$ in binary. The unary part specifies how many bits

are required to code x, and then the binary part actually codes x in that many bits. For example, consider $x = 9$. Then $\lfloor \log x \rfloor = 3$, and so $4 = 1 + 3$ is coded in unary (code 1110) followed by $1 = 9 - 8$ as a three-bit binary number (code 001), which combine to give a codeword of 1110001.

Other examples of γ codes are shown in the third column of Table 3.6. Although they are of differing lengths, the codewords can be unambiguously decoded. All the decoder has to do is first extract a unary code c_u, and then treat the next $c_u - 1$ bits as a binary code to get a second value c_b. The value x to be returned is then easily calculated as $2^{c_u - 1} + c_b$. For the code 1110001, $c_u = 4$, and $c_b = 1$ is the value of the next three bits, and so the value $x = 9 = 2^3 + 1$ is returned. Although it can be outperformed by some of the methods described below, the γ code is nevertheless much better for coding inverted file gaps than either a binary encoding or a unary encoding, and it is just as easy to encode and decode. It represents a gap x in $l_x \approx 1 + 2 \log x$ bits, so the implied probability of a gap of x is

$$\Pr[x] = 2^{-l_x} \approx 2^{-(1+2\log x)} = \frac{1}{2x^2}.$$

This gives an inverse square relationship between gap size and probability.

A more general way of looking at the γ code is to break it into two components: a unary code representing a value $k + 1$ relative to some vector $V = \langle v_i \rangle$ such that

$$\sum_{i=1}^{k} v_i < x \leq \sum_{i=1}^{k+1} v_i,$$

followed by a binary code of $\lceil \log v_k \rceil$ bits representing the residual value

$$r = x - \sum_{i=1}^{k} v_i - 1.$$

In this framework, the γ code uses the vector

$$V_\gamma = \langle 1, 2, 4, 8, 16, \ldots \rangle,$$

and $x = 9$ is coded with $k = 3$ and $r = 1$. Similarly, the unary code is relative (somewhat recursively) to the vector

$$V_U = \langle 1, 1, 1, 1, 1, \ldots \rangle.$$

Later in this section we will refer back to this general view of coding with respect to vectors.

A further development is the δ code, in which the prefix indicating the number of binary suffix bits is represented by the γ code rather than the unary code. Taking the same example of $x = 9$, the unary prefix of 1110 coding 4 is replaced by 11000, the γ code for 4. That is, the δ code for $x = 9$ is 11000001.

In general, the δ code for an arbitrary integer x requires

$$l_x = 1 + 2\lfloor \log(1 + \lfloor \log x \rfloor) \rfloor + \lfloor \log x \rfloor = 1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$$

bits. Inverting this, the distribution implied is approximated by

$$\Pr[x] \approx 2^{-(1+2\log\log x+\log x)} = \frac{1}{2x(\log x)^2}.$$

Table 3.6 gives examples of δ codes for various values x. Although for the small values of x shown the δ codes are longer than γ codes, in the limit, as x becomes large, the situation is reversed. For a value of x such as 1,000,000, the δ code is superior, requiring 28 bits compared with the 39 bits of γ.

Global Bernoulli model

One obvious way to parameterize the model and perhaps obtain better compression is to make use of the actual density of pointers in the inverted file. Suppose that the total number of pointers to be stored (the quantity f in Table 3.1) is known. Dividing this by the number of index terms, and then by the number of documents, gives a probability of $f/(N \cdot n)$ that any randomly selected document contains any randomly chosen term. The pointer occurrences can then be modeled as a Bernoulli process with this probability, by assuming that the f pointers of the inverted file are randomly selected from the $n \cdot N$ possible word-document pairs in the collection. For example, using the information contained in Table 3.1, the probability that any randomly selected word of *Bible* appears in any randomly chosen verse is calculated as $701{,}412/(31{,}101 \times 8{,}965) = 0.0025$, assuming that words are scattered completely uniformly across verses.

Making this assumption, the chance of a gap of size x is the probability of having $x - 1$ nonoccurrences of that particular word, each of probability $(1 - p)$, followed by one occurrence of probability p, which is $\Pr[x] = (1 - p)^{x-1}p$. This is called the *geometric* distribution and is equivalent to modeling each possible term-document pair as appearing independently with probability p. If arithmetic coding is to be used, the required cumulative probabilities can be calculated by summing this distribution:

$$low\_bound = \sum_{i=1}^{x-1}(1-p)^{i-1}p = 1 - (1-p)^{x-1}$$

$$high\_bound = \sum_{i=1}^{x}(1-p)^{i-1}p = 1 - (1-p)^{x}$$

When decoding, the cumulative probability formula $1 - (1 - p)^x$ must be inverted to determine x and inverted exactly in order for the decoder to proceed correctly. The inverse function $x = 1 + \lfloor(\log(1 - v))/(\log(1 - p))\rfloor$, where v is the current fractional value of the arithmetic coding target, yields the decoded value x.

The probabilities generated by the geometric distribution can also be represented by a surprisingly effective Huffman-style code, and this turns out to be a more useful alternative to arithmetic coding. The following method was first described in 1966 by Solomon Golomb of the University of Southern California and is referred to as the *Golomb code* (Golomb 1966). For some parameter b, any number $x > 0$ is coded