Indexing and Querying

Inverted lists and index compression

pdf page 2

Index construction

pdf page 17

Accessing the Lexicon and Boolean queries

pdf page 25

Querying and Ranking

pdf page 39

Indexing

three

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	$\langle 2; 1, 4\rangle$
2	days	$\langle 2; 3, 6\rangle$
3	hot	$\langle 2; 1, 4\rangle$
4	in	$\langle 2; 2, 5\rangle$
5	it	$\langle 2; 4, 5\rangle$
6	like	$\langle 2; 4, 5\rangle$
7	nine	$\langle 2; 3, 6\rangle$
8	old	$\langle 2; 3, 6\rangle$
9	pease	$\langle 2; 1, 2\rangle$
10	porridge	$\langle 2; 1, 2\rangle$
11	pot	$\langle 2; 2, 5\rangle$
12	some	$\langle 2; 4, 5\rangle$
13	the	$\langle 2; 2, 5\rangle$

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—$\langle 4, 5\rangle$ and $\langle 1, 4\rangle$, respectively—are merged (or, strictly speaking, intersected), yielding the lines that they have in common, in this case the list $\langle 4\rangle$. 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	$\langle 2; (1; 6), (4; 8)\rangle$
2	days	$\langle 2; (3; 2), (6; 2)\rangle$
3	hot	$\langle 2; (1; 3), (4; 4)\rangle$
4	in	$\langle 2; (2; 3), (5; 4)\rangle$
5	it	$\langle 2; (4; 3, 7), (5; 3)\rangle$
6	like	$\langle 2; (4; 2, 6), (5; 2)\rangle$
7	nine	$\langle 2; (3; 1), (6; 1)\rangle$
8	old	$\langle 2; (3; 3), (6; 3)\rangle$
9	pease	$\langle 2; (1; 1, 4), (2; 1)\rangle$
10	porridge	$\langle 2; (1; 2, 5), (2; 2)\rangle$
11	pot	$\langle 2; (2; 5), (5; 6)\rangle$
12	some	$\langle 2; (4; 1, 5), (5; 1)\rangle$
13	the	$\langle 2; (2; 4), (5; 5)\rangle$

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 $y_1, y_2, \ldots$. 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.

3.4 Performance of index compression methods

Table 3.8 shows the compression obtained on the four test collections by the various methods described above. Output sizes are expressed as bits per pointer. The total size of the index can be calculated by multiplying by the appropriate f value from Table 3.1. For example, use of the interpolative code yields a *TREC* index of 83.4 Mbytes, or just 4 percent of the text that it indexes. This is quite a remarkable achievement when it is remembered that a document number for *every* word and number in this 2 Gbyte collection is stored in the index. As a reference point, the second row of values shows the space that would be required, per pointer, by an ordinary binary encoding of the gap sizes.

The results of Table 3.8 include any necessary overheads, such as the γ-coded f_t values for the local Bernoulli model and the complete set of models and model selectors for the batched frequency model. The figures for unary-coded compression are less than what the average would be for a pure bitmap because in the unary-coded file γ is used to represent the local value f_t for each inverted list, followed by f_t unary codes rather than a complete bitvector. For example, a word with $f_t = 1$ that appears in the first document (number one) would be counted as taking two bits in the unary-coded implementation—one bit for the γ code for f_t and one bit for the unary code for 1.

Though it is the best of the global models, the global observed frequency model is outperformed by remarkably simple local models. The frequency of a term is a much better predictor of the distribution of its gap sizes than is the overall gap size distribution for all terms. Furthermore, given that the local Bernoulli or skewed Bernoulli models need just one parameter to be stored in memory during decoding, compared with the hundreds and possibly thousands of parameters required by the various observed frequency models, there is no contest—the local models are significantly better. Even the global γ and δ codes come surprisingly close to the compression attained by the global observed frequency model. These latter two codes also have the major advantage of requiring no parameters. This is useful when storing dynamic collections, which will be discussed in Chapter 5.

Of the results listed, only the hyperbolic model assumes the use of arithmetic coding. All of the other mechanisms rely upon prefix codes, so slight compression gains might result if the underlying probability distributions were used to drive an arithmetic coder instead. It is, however, unlikely that any gains would be sufficiently large to warrant the extra decoding time.

The interpolative code gives the best results on all four of the test files, followed by the batched frequency model and the skewed Bernoulli model, with the parameter b chosen as the median gap size in each inverted list. Furthermore, all of these codes require relatively modest computational resources during index compression and decompress as quickly during index access as do simple binary codes. The local Bernoulli model, coded using the Golomb code, is also a good choice, obtaining slightly less compression than the interpolative code but with a correspondingly simpler implementation. Compared to both of these mechanisms, the batched frequency model has the extra disadvantage that during inverted file decompression,

Table 3.8 Compression of inverted files in bits per pointer.

Method	Bits per pointer			
	Bible	*GNUbib*	*Comact*	*TREC*
Global methods				
Unary	262	909	487	1918
Binary	15.00	16.00	18.00	20.00
Bernoulli	9.86	11.06	10.90	12.30
γ	6.51	5.68	4.48	6.63
δ	6.23	5.08	4.35	6.38
Observed frequency	5.90	4.82	4.20	5.97
Local methods				
Bernoulli	6.09	6.16	5.40	5.84
Hyperbolic	5.75	5.16	4.65	5.89
Skewed Bernoulli	5.65	4.70	4.20	5.44
Batched frequency	5.58	4.64	4.02	5.41
Interpolative	5.24	3.98	3.87	5.18

either the appropriate model must be read off disk and stored in memory while the inverted list is being decoded, at the cost of an extra disk access, or the complete set of models must be held resident in memory. The latter choice can involve significant memory resources.

3.5 Signature files and bitmaps

Signature files and bitmaps are two further approaches to indexing. In certain combinations of circumstances both can offer faster query processing than inverted files but in those same situations are likely to require a great deal more storage space, which often outweighs that advantage. Signature files have been particularly popular in the past because they are, in a sense, implicitly compressed and so can consume less storage space than uncompressed inverted files.[3] The previous two sections have shown how much has been learned about inverted index representations in the last 10 years, and signature files no longer enjoy the relative advantage that they used to.

3 For example, the 1992 text *Information Retrieval: Data Structures and Algorithms* by Frakes and Baeza-Yates includes a chapter on signature files but no material on the compression of indexes.

Index Construction

Chapters 3 and 4 avoid the question of how the index is created: they simply suppose that it exists and can thus be compressed or queried. In reality, constructing the index is one of the most challenging tasks to be faced when a database is built. This chapter addresses the problem of creating the various index structures described in Chapters 3 and 4. The emphasis here, as in those two chapters, is on inverted file indexing since this is the most practical form of index for both Boolean and ranked queries.

The process of building an index is known as the *inversion* of the text. The *Concise Oxford Dictionary* defines "inversion" as "turning upside down, reversal of normal position, order, or relation," and this is exactly what must be done to create an index. "Inversion" is also used in a technical sense by meteorologists, musicians, electrical engineers, and mathematicians, to name but a few. In fact, to a mathematician the operation being performed here is more usually known as *transposition* (which is also a musical term); when a mathematician inverts a matrix, it is to calculate a multiplicative inverse. And, of course, the author of a technical book knows the required process only too well; it is the task of *indexing*. Indeed, the task of indexing was known and described well before the computer made it easy, and there are books on the subject that make interesting reading. In *Indexing Books: A Manual of Basic Principles*, Collison (1962, 16–17) instructs us:

> Whether handwriting or a typewriter is used, it is essential to have on hand a large number of slips or thin cards of the same size. . . . Indexing needs a lot of stationery, and it is best to order ten thousand slips at a time.

Given that we propose to deal with texts containing more than 1,000 individual books, it is clear that a very large number of "slips" indeed might be required.

To illustrate the magnitude of the inversion problem, we will first describe, through the use of an example, what is perhaps the most obvious method. Consider again the simple text that was used in Chapter 3, reproduced in Table 5.1.

Table 5.1 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.

Table 5.2 Frequency matrix for text of Table 5.1.

	Term												
	cold	days	hot	in	it	like	nine	old	pease	porridge	pot	some	the
1	1	—	1	—	—	—	—	—	2	2	—	—	—
2	—	—	—	1	—	—	—	—	1	1	1	—	1
3	—	1	—	—	—	—	1	1	—	—	—	—	—
4	1	—	1	—	2	2	—	—	—	—	—	2	—
5	—	—	—	1	1	1	—	—	—	—	1	1	1
6	—	1	—	—	—	—	1	1	—	—	—	—	—

Each line (document) of this text contains some index terms, and each index term appears in some of the lines. This relationship can be expressed with a *frequency matrix*, in which each column corresponds to one word, each row corresponds to one document, and the number stored at any row and column is the frequency, in that document, of the word indicated by that column. The frequency matrix for the text of Table 5.1 is shown in Table 5.2.

In effect, each document of the collection is summarized in one row of this frequency matrix. To create an index, the matrix must be transposed, forming a new version in which the rows are the term numbers. The inverted frequency matrix is shown in Table 5.3. From this it is easy to construct an inverted file of the form described in Chapter 3 or an augmented inverted file of the form described in Chapter 4.

One algorithm to create an inverted file is now clear: build in memory a transposed frequency matrix, reading the text in document order, one column of the matrix at a time; then write the matrix to disk row by row, in term order. Despite the attractive simplicity of this approach, in reality inversion is a much more difficult task. The problem is the size of the frequency matrix. Suppose that the text *Bible* is

Table 5.3 Transposed equivalent of frequency matrix of Table 5.2.

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

to be inverted. From Table 3.1, *Bible* contains 8,965 distinct terms and 31,101 documents. If a four-byte integer is allowed for each entry in the frequency matrix, the matrix will occupy 4 × 8,965 × 31,101 bytes of main memory. This is a little over 1 Gbyte, barely manageable even on a large machine. For the bigger *TREC* collection, the size of the matrix becomes even more daunting: 4 × 535,346 × 741,856 bytes, or 1.4 Tbytes (terabytes).

Supposing that one byte is sufficient to record each within-document frequency $f_{d,t}$ (for *TREC* it is not adequate) does not help either: the space requirements for the two collections are 250 Mbytes and 350 Gbytes, respectively, and the algorithm still is not viable. If only Boolean access is required, then a Boolean matrix is sufficient, and the frequencies can be dispensed with, reducing the sizes to 31 Mbytes and 46 Gbytes, respectively—still an unpleasantly large amount of memory. Of course, we could use a machine with a large virtual memory and let the operating system be responsible for paging the array into and out of memory as required. But the column-by-column access as the matrix is created means that there will probably be one page fault for each pointer in the eventual index, and about 700,000 page faults would be required to build the *Bible* index. At a rate of perhaps 50 page replacements per second, this corresponds to 14,000 seconds, about 4 hours. For *TREC*, use of virtual memory to build an explicit frequency matrix results in an inversion process that takes two nonstop calendar months, which is reminiscent in computer-based terms of the dedication needed for the manual indexing processes described in Chapter 1.

For these reasons, more economical methods for constructing and inverting a frequency matrix must be considered—the main theme of this chapter. The final method described has been used to create an augmented inverted index for the *TREC* collection (2 Gbytes of text) in under 2 hours on a personal computer, consuming just 30 Mbytes of main memory and less than 20 Mbytes of temporary disk space over and above the space required by the final inverted file. Needless to say, this final method bears little resemblance to the method sketched in these introductory paragraphs. The construction of signature files and bitmaps is discussed in this chapter too.

Chapters 3 and 4 also avoided mention of *dynamic* collections and concentrated exclusively on *static* ones. Such techniques are appropriate if an archive of material is to be distributed on some read-only medium such as CD-ROM. Then it is acceptable for a large amount of effort to go into preparing the files that will comprise the distribution, provided that access to the data is fast. However, in other situations the collection may be required to be dynamic, with new documents being added, existing ones being modified, and, sometimes, old ones being deleted. Keeping an index up-to-date requires file structures that can cope with these operations efficiently without consuming inordinate amounts of extra space. The final section of this chapter considers the problems posed by dynamic collections and explains how to compress and index large volumes of text when individual documents are subject to change and the collection itself is subject to extension.

Before we begin to look at the details of the various index construction methods, we introduce a benchmark on which their performance will be compared and preview the panoply of methods that will be presented.

Computational model

In order to assess the efficiency of index construction algorithms, it is useful to have as a reference point the cost of inverting some typical database. Table 5.4 describes a hypothetical collection of 5 Gbytes and five million documents. It also gives some nominal performance figures, which will be used to estimate the overall time required by each inversion method. Although not specifically derived from the execution speed of any particular machine, these provide a useful basis for comparing algorithms and capture the relative costs of the various operations involved. When the first edition of this book was written in 1993, the listed performance roughly corresponded to the $30,000 workstation used for our experiments with *TREC*-sized document collections; now, in 1999, they somewhat underestimate the performance of the $5,000 personal computer we use for the same experiments.

Preview of index construction methods

Table 5.5 gives an advance peek at the methods that we will develop for index construction, and their performance on the 5 Gbyte collection of five million documents in Table 5.4. For example, the standard linked-list inversion algorithm is summarized in the first entry: it is described in Section 5.1, and pseudocode appears in Figure 5.1. Although it could invert the sample document collection of Table 5.4

Table 5.4 Typical sizes and performance figures.

Parameter	Symbol	Assumed value
Total text size	B	5×10^9 bytes
Number of documents	N	5×10^6
Number of distinct words	n	1×10^6
Total number of words	F	800×10^6
Number of index pointers	f	400×10^6
Final size of compressed inverted file	I	400×10^6 bytes
Size of dynamic lexicon structure	L	30×10^6 bytes
Disk seek time	t_s	10×10^{-3} sec
Disk transfer time per byte	t_r	0.5×10^{-6} sec
Inverted file coding time per byte	t_d	5×10^{-6} sec
Time to compare and swap 10-byte records	t_c	10^{-6} sec
Time to parse, stem, and look up one term	t_p	20×10^{-6} sec
Amount of main memory available	M	40×10^6 bytes

Table 5.5 Predicted resource requirements to invert 5 Gbytes.

Method	Section	Figure	Memory (Mbytes)	Disk (Mbytes)	Time (hours)
Linked lists (memory)	5.1	5.1	4,000	0	6
Linked lists (disk)	5.1	5.1	30	4,000	1,100
Sort-based	5.2	5.3	40	8,000	20
Sort-based compressed	5.3	—	40	680	26
Sort-based multiway merge	5.3	—	40	540	11
Sort-based multiway in-place	5.3	5.8 and 5.9	40	150	11
In-memory compressed	5.4	5.12	420	1	12
Lexicon-based, no extra disk	5.4	—	40	0	79
Lexicon-based, extra disk	5.4	—	40	4,000	12
Text-based partition	5.4	—	40	35	15

in 6 hours, it would consume 4 Gbytes of main memory in doing so. No extra disk space is required beyond what is needed to store the inverted file. For algorithms in which a memory limit is enforced, it is assumed that 40 Mbytes of memory is available to be exploited. The final method is the one used for the 2-hour inversion

To produce an inverted index for a collection of documents,

1. /* Initialization */
   Create an empty dictionary structure S.
2. /* Phase one—collection of term appearances */
   For each document D_d in the collection, $1 \leq d \leq N$,
   (a) Read D_d, parsing it into index terms.
   (b) For each index term $t \in D_d$,
      i. Let $f_{d,t}$ be the frequency in D_d of term t.
      ii. Search S for t.
      iii. If t is not in S, insert it.
      iv. Append a node storing $\langle d, f_{d,t} \rangle$ to the list corresponding to term t.
3. /* Phase two—output of inverted file */
   For each term $1 \leq t \leq n$
   (a) Start a new inverted file entry.
   (b) For each $\langle d, f_{d,t} \rangle$ in the list corresponding to t,
      append $\langle d, f_{d,t} \rangle$ to this inverted file entry.
   (c) If required, compress the inverted file entry.
   (d) Append this inverted file entry to the inverted file.

Figure 5.1 Memory-based inversion algorithm.

of *TREC* mentioned above, except that the 2-hour inversion was done on a rather faster machine than that specified in Table 5.4.

All these algorithms will be explained in the sections that follow. Table 5.5 is just intended as a preview and summary.

5.1 Memory-based inversion

Implementing a cross-reference generator is a commonly assigned student project in "Data Structures and Algorithms" courses. In reality, a cross-reference is just another name for an inverted index, in which each term of some text (for example, identifiers in program source code) is listed in alphabetical order, together with a list of the line numbers in which it appears.

When set as a student exercise, the intended solution is usually that some kind of dynamic dictionary data structure such as a hash table or binary search tree be used to record the distinct terms in the collection, with a linked list of nodes storing line numbers associated with each dictionary entry. Once all documents have been processed, the dictionary structure is traversed, and the list of terms and corresponding

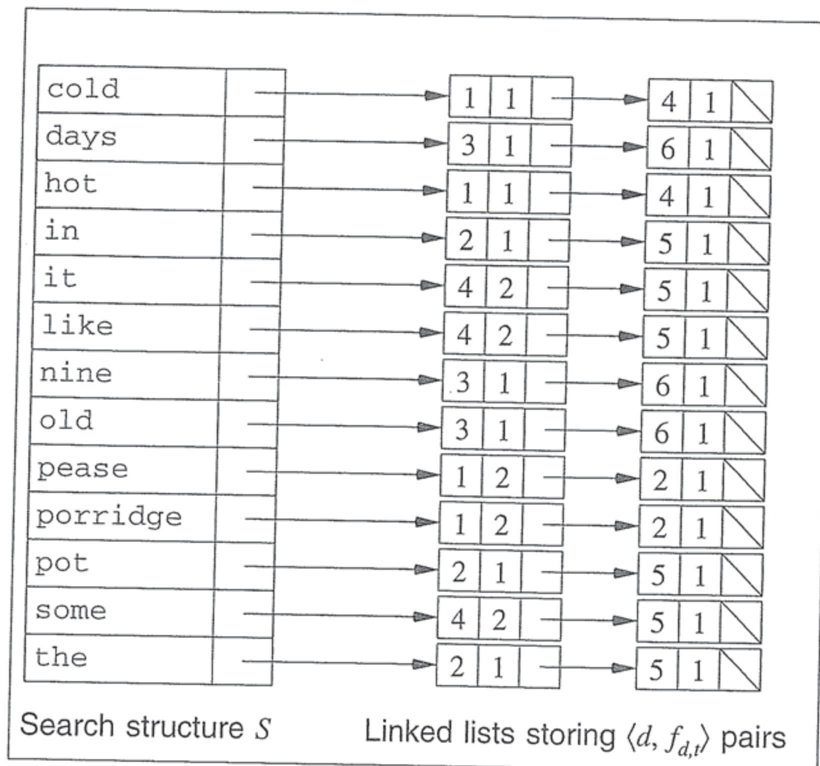

Figure 5.2 Data structure representing inverted file for text of Table 5.1.

line numbers is written. This process is detailed in Figure 5.1, and the state of the data structure at the completion of the first phase of processing on the sample text of Table 5.1 is shown in Figure 5.2. Here the terms are shown stored in sorted order in the dictionary, but this is not necessary. Any dynamic structure is suitable, but a hash table is generally the most economical choice in terms of both speed and space.

Now consider the cost of this inversion algorithm. At the assumed rate of 2 Mbytes per second, it takes about 40 minutes to read 5 Gbytes of text. Parsing and stemming to create index terms, and searching for these terms in the dictionary, takes much longer, more than 4 hours, at 20 microseconds per word. The second phase requires each list to be traversed so that the corresponding inverted list can be encoded and written. The encoding requires 2,000 seconds and the writing 200 seconds, a further 40 minutes.

More generally, the inversion time T required by this approach is

$$\begin{aligned} T = Bt_r + Ft_p + \quad & \text{(read and parse text)} \\ I(t_d + t_r) \quad & \text{(write compressed inverted file)} \end{aligned}$$

seconds, where the symbols used are defined in Table 5.4. This simple linked-list inversion method corresponds to the first line in Table 5.5.

The above analysis predicts an inversion time of about 6 hours for the database of Table 5.4. This might sound like a long time, but when the enormous volume of

data being processed is taken into account, it is very fast. For the *Bible* collection, for example, the same calculation gives an inversion time under 30 seconds.

There is, however, another resource to be considered: the memory space required by the algorithm. Each node in each list of document numbers typically requires at least 10 bytes: 4 for the document number d, 4 for the "next" pointer, and 2 or more for the frequency count $f_{d,t}$. For a small collection, this requirement is modest. But in the situation described in Table 5.4 there are 400 million such nodes, or 4 Gbytes of memory, and to require this much main memory is unrealistic, even given the advances in computing power noted above. Of course, as machine performance increases, so does the amount of memory supported, but it is also true that collection sizes are likely to grow at the same exponential rate and that memory will never overtake demand.

It is tempting to move the linked lists of document numbers from memory onto disk—or, equivalently, to run the program on a machine with a large virtual memory and a small physical memory—but this is not a viable alternative either. To see why, consider the steps of the algorithm again but assuming that the linked lists are stored on disk.

The sequence of disk accesses during the first phase is sequential, and generation of the threaded file containing the linked lists is largely unaffected. Each new node results in a record being appended to a file, so, in the example inversion, a file of 4 Gbytes is created in sequential fashion on disk, adding about 30 minutes to the 6 hours already allowed.

Now consider the cost of the second phase, when each list is traversed. The stored list nodes are interleaved in the same order on disk as they appeared in the text, and so each node access requires a random seek into the file on disk. At the assumed rate t_s of 10 milliseconds per seek, and with 10 bytes to be read per record, this corresponds to a phase two time of four million seconds, or 6 weeks. In terms of the parameters described in Table 5.4, the inversion time is now

$$
\begin{aligned}
T = {} & Bt_r + Ft_p + 10ft_r + && \text{(read and parse, write file)} \\
& ft_s + 10ft_r + && \text{(trace lists on disk)} \\
& I(t_d + t_r) && \text{(write compressed inverted file)}
\end{aligned}
$$

seconds, annoyingly slow for practical purposes. Indeed, the use of a similar disk-based random access inversion to invert an 806 Mbyte text—one-sixth the size considered in Table 5.4—has been reported to have required more than 13 days of processing, albeit on a minicomputer less powerful than the machine postulated here (Harman and Candela 1990). This disk-based linked-list inversion method corresponds to the second line in Table 5.5.

For the gigabyte collections that this book addresses, the linked-list approach is inadequate because it requires either too much memory or too much time. It is, however, an excellent project and the best method for small collections. For the *Bible*, an in-memory inversion takes half a minute and requires about 10 Mbytes of main memory, well within the reach of typical workstations.

Querying

four

Now that we have seen what an index contains and how to store it efficiently, we move on to consider how best to use an index to locate information in the text it describes. Two types of queries will be examined in this chapter. The first is a conventional *Boolean* query; this form was tacitly assumed during the discussion in Chapter 3. The second is a *ranked* query; we will discuss what this means shortly.

A Boolean query comprises a list of *terms*—words to be sought in the text—that are combined using the connectives AND, OR, and NOT. The *answers*, or *responses*, to the query are those documents that satisfy the stipulated condition. For example, an appropriate query to search for material relevant to this book is

text AND *compression* AND *retrieval*.

All three words (or variants considered equivalent by a stemming algorithm) must occur somewhere in every answer. They need not be adjacent nor appear in any particular order. Documents containing phrases like *the compression and retrieval of large amounts of text is an interesting problem* will be returned as answers. Also returned would be a document containing *this text describes the fractional distillation scavenging technique for retrieving argon from compressed air*—perhaps not quite what is sought, but nonetheless a correct answer to the query.

A problem with all retrieval systems is that answers are invariably returned that are not relevant, and these must be filtered out manually. A difficult choice must be made between casting a broad query to be sure of retrieving all relevant material, even if it is diluted with many irrelevant answers, and posing a narrow one, which ensures that most documents retrieved are of interest but risks eliminating others sight unseen because the query is too restrictive. A broad search that identifies virtually all the relevant documents is said to have *high recall*, while one in which virtually all the retrieved documents are relevant has *high precision*. An enduring theme in information retrieval is the tension between these two virtues. We

must choose in any particular application whether to prefer high precision or high recall and cast the query appropriately—just as a fisherman might choose to use a hand net to select a single prized catch or to trawl the ocean floor to make sure that nothing escapes, but in the process catching a great deal of junk as well.

Another problem with Boolean retrieval systems is that small variations in a query can generate very different results. The query *data* AND *compression* AND *retrieval* is likely to produce quite a different answer set to *text* AND *compression* AND *retrieval*, yet the person issuing these requests probably sees them as very similar. To be sure of catching all the required documents, users become adept at adding extra terms and learn to pose queries like

(*text* OR *data* OR *image*) AND
(*compression* OR *compaction* OR *decompression*) AND
(*archiving* OR *retrieval* OR *storage* OR *indexing*)

where the parentheses indicate operation order. This, perhaps, is one reason why librarians guard access to the large international databases. Formulating queries is an art, and librarians have the necessary insight and linguistic skills to guide a query toward an acceptable set of answers.

Despite these problems, Boolean retrieval systems were the primary mechanism used to access online information for more than three decades, in both commercial and scientific applications. They have satisfied countless users. Nevertheless, they are not the only way a database can be queried. Typical fishers of information might find it even better simply to list words that are of interest and have the retrieval mechanism supply the documents that seem most relevant, rather than seeking exact Boolean answers. For example, to locate material for this book, the query

text, data, image, compression, compaction, archiving, storage, retrieval, indexing, gigabyte, megabyte, document, database, searching, information

is, to a person at least, probably a clearer description of the topic than the Boolean query above.

Identifying documents relevant to a list of terms is not just a matter of converting the terms to a Boolean query. It would be fruitless to connect these particular terms with AND operations since vanishingly few documents are likely to match. (We cannot, of course, say that no documents will match. This page certainly does.) It would be just as pointless to use OR connectives since far too many documents will match and very few are likely to be useful answers.

One solution is to use a ranked query. This involves a heuristic that is applied to gauge the *similarity* of each document to the query. Based on this numeric indicator, the r most closely matching documents are returned as answers—r being perhaps 10 or 100. If the heuristic is good, or r is small, or (better still) both, there will be a predominance of relevant answers—high precision. If the heuristic is good and r is large, most of the documents in the collection that are relevant will fall within the top r—high recall. In practice, low precision invariably accompanies high recall

since many irrelevant documents will almost certainly come to light before the last of the relevant ones appears in the ranking. Conversely, when the precision is high, recall will probably be low, since precision will be high only near the beginning of the ranked list of documents, at which point only a few of the total set of relevant ones will have been encountered.

Great effort has been invested over the years in a quest for similarity measures and other ranking strategies that succeed in keeping both recall and precision reasonably high. Simple techniques just count the number of query terms that appear somewhere in the document: this is often called *coordinate matching*. A document that contains five of the query terms will be ranked higher than one containing only three, and documents that match just one query term will be ranked lower still. An obvious drawback is that long documents are favored over short ones since by virtue of size alone they are more likely to contain a wider selection of the query terms. Furthermore, common terms appearing in the query tend to discriminate unfairly against documents that do not happen to contain them, even ones that match on highly specific ones. For example, a query concerning *the electronic office* might rank a document containing *the office garbage can* ahead of one that discusses *an electronic workplace*. A word such as *the* in the query should probably not be given the same importance as *electronic*.

More sophisticated ranking techniques take into account the length of the documents and assign a numeric *weight* to each term. One such technique—the *cosine* measure—is examined in this chapter as an example of a *vector space* (also sometimes called *statistical*) method. Many other schemes have been proposed for ranking documents, some of which are also surveyed here.

Throughout this book our primary concern is with implementation issues and resource implications. Nevertheless, results are provided to show how useful the various ranking methods are. For example, on a set of 50 queries applied to the *TREC* collection, the cosine measure achieves an average precision exceeding 40 percent on the top 100 answers to each query. In other words, when a ranked query is posed to *TREC*—which contains nearly three-quarters of a million documents—40 of the top 100 documents returned for each query will be relevant. Such performance is difficult to achieve with a Boolean query, and, as will be illustrated below, the queries formulated from the *TREC* topics were anything but carefully phrased. Set against this improved retrieval is the extra expense of ranked queries. More complex index information is required than for Boolean queries, and processing costs are greater.

Before any of these issues can be tackled, a mechanism must be provided that locates the query terms in the collection's lexicon and identifies the addresses of the corresponding inverted lists. Although this is a simple task for small collections, it is not easy to do efficiently for large ones. The first section of this chapter deals with the problem of searching the lexicon to identify query terms.

There is also the possibility that query terms might include wildcard characters. For example, *lab*r* might be used to cover both *labor* (the American spelling) and *labour* (the British spelling); economy of typing might prompt the use of *superc*fra*exp*do*s* when looking for information about the movie *Mary Poppins*. The former query would also match *labrador*, an expansion that probably does

Table 4.1 Storage requirements for a million-term lexicon using various data structures.

Method	Storage
Fixed-length strings	28 Mbytes
Terminated strings	20 Mbytes
Four-entry blocking	18 Mbytes
Front coding	15.5 Mbytes
Minimal perfect hashing	13 Mbytes

not yield the desired effect, and so even at this level of query, there is a question of relevance. Methods for handling partially specified terms are considered in Section 4.2.

4.1 Accessing the lexicon

The lexicon for an inverted file index stores both the terms that can be used to search the collection and the auxiliary information needed to allow queries to be processed. The minimal information that must be stored for each term t in the lexicon is the address in the inverted file of I_t, the corresponding list of document numbers. To allow inverted lists to be retrieved in order of increasing term frequency, it is usual to store f_t, the number of documents containing the term, as well. The reason for this is discussed in Section 4.3. Other values, such as compression parameters, are normally stored as part of the inverted list since they are required only after the list has been retrieved.

In this section we develop ways to store lexicons. A poor choice of data structure can waste many megabytes of storage space during query processing, and since the intention is that query processing will be carried out on a relatively low-powered machine, this memory might be needed for other purposes. Table 4.1 summarizes the storage required by the data structures we develop, for a collection with one million terms. The first three techniques are very simple; the fourth is more involved, and the fifth, minimal perfect hashing, is distinctly subtle. In the end we will conclude that for querying purposes, the main memory requirement can be eliminated almost entirely simply by placing the lexicon on disk. However, there are situations, such as the index construction process described in Chapter 5, when this solution is inadequate, and minimal perfect hashing becomes the method of choice.

Access structures

A simple structure for the lexicon is to store an array of records, each comprising a string along with two integer fields. If the lexicon is sorted, a word can be located by a binary search of the strings, as was suggested in some of the calculations in

Term t	f_t	Disk address of I_t
jezebel	20	→
jezer	3	→
jezerit	1	→
jeziah	1	→
jeziel	1	→
jezliah	1	→
jezoar	1	→
jezrahiah	1	→
jezreel	39	→

Figure 4.1 Storing a lexicon as an array of records.

Chapter 1, and access is very fast. This structure is shown in Figure 4.1; the words used are part of the lexicon for the *Bible* collection.

Storing the strings in this way will consume a great deal of space. The lexicon for a large collection might contain one million terms. This is not extreme: in the 2 Gbyte *TREC* collection, for example, there are $n = 535{,}246$ distinct terms even after stemming (many of them are spelling mistakes, but that is unavoidable), and so one million terms could correspond to a collection of perhaps 5 Gbytes. Stored as 20-byte strings (and optimistically assuming that none is longer than 20 bytes), with a 4-byte inverted file address and a 4-byte f_t value, the lexicon requires more than 28 Mbytes.

The space for the strings is reduced if they are all concatenated into one long contiguous string and an array of 4-byte character pointers is used for access. Then each term consumes its exact number of characters, plus four for the pointer. This is likely to result in a net saving since the average length of terms in a large lexicon is typically about eight characters in English. Note that the average length of terms in a lexicon is considerably longer than the average length in the corresponding text, which is usually about four to five characters. For example, the average length in the 538,000-term *TREC* lexicon is 7.36 letters, whereas the average length in the 334,000,000-term *TREC* corpus is 3.86 letters. This difference in average length is because most of the very common terms are short; they are repeated many times in the text, but each appears only once in the lexicon.

The string and pointer structure is sketched in Figure 4.2. When every string is indexed, it is not necessary for a string length field or terminator character to be stored since the next pointer in the array indicates the end of the string. For the

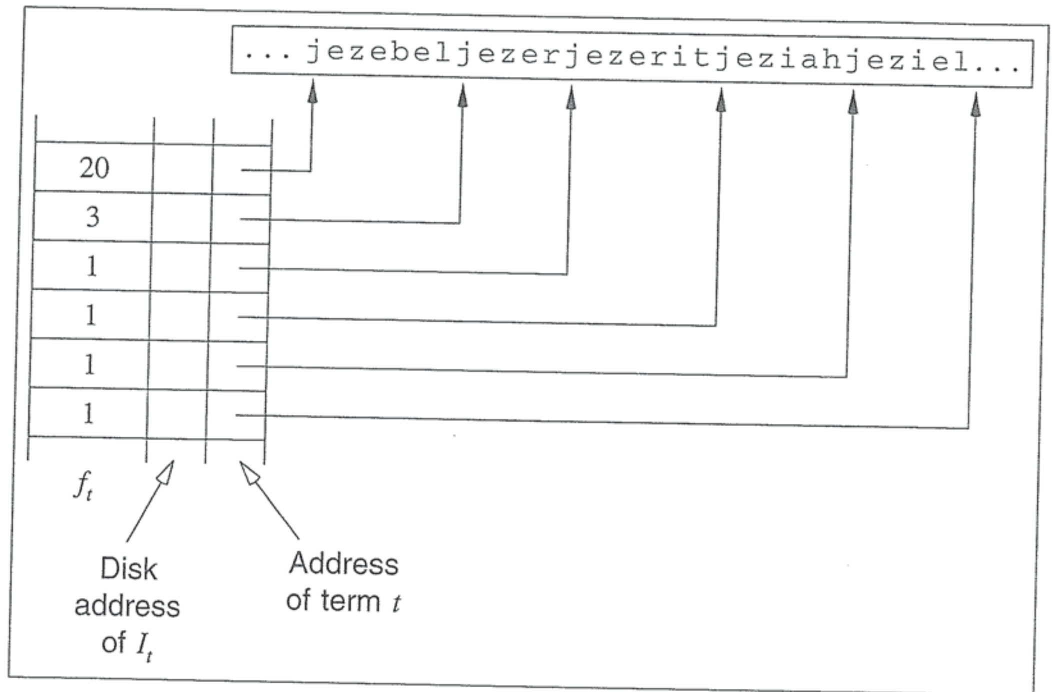

Figure 4.2 Storing a lexicon as an array of string pointers.

same hypothetical vocabulary of one million words, the memory space is reduced by 8 Mbytes to 20 Mbytes.

The memory required can be further reduced by eliminating many of the string pointers. Although it is clear that n inverted file addresses and f_t values must be stored, it is not necessary to use n pointers to index the array of words. Suppose that one word in four is indexed, and each stored word is prefixed by a 1-byte length field. The length field allows the start of the next string to be identified and the block of strings traversed. In each group of four words, a total of 12 bytes of pointers is saved, at the cost of including 4 bytes of length information. This structure is illustrated in Figure 4.3. For the one-million-word lexicon, the space required drops by another 2 Mbytes, and a total of 18 Mbytes suffices. For larger blocks, the savings continue to accrue but are less dramatic. For example, blocks of eight words rather than four save a further 0.5 Mbyte; blocks of 16 words, another 0.25 Mbyte; and so on.

Blocking makes the searching process somewhat more complex. To look up a term, the array of string pointers is first binary-searched to locate the correct block of words. This block is then scanned in a linear fashion to find the term, and its ordinal term number is inferred from the combination of block number and position within block. Once the ordinal term number has been calculated, the f_t array and the vector of inverted file addresses can be accessed in the usual way.

Best of all, using blocks of four words has only a very small effect on the cost of searching. This is because a linear search on the last three words within the block, should it be necessary, requires, on average, two string comparisons. This is only

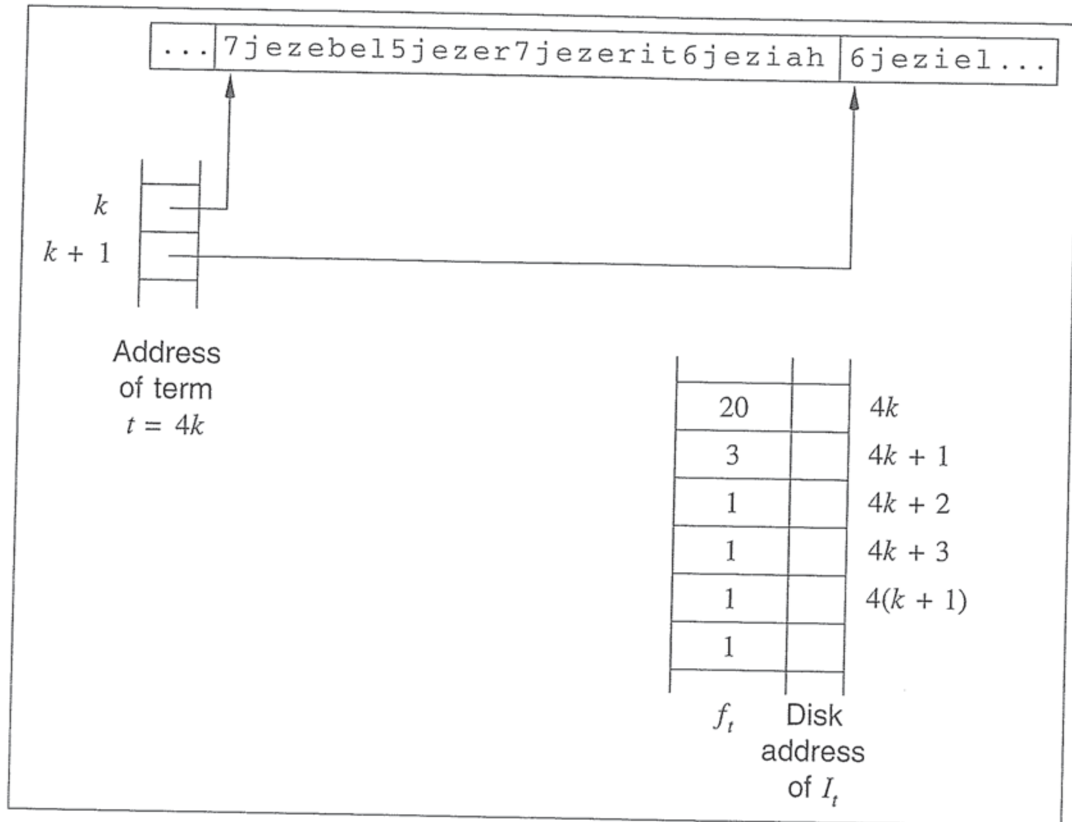

Figure 4.3 Storing a lexicon with one word in four indexed.

slightly more than the average of 1.7 comparisons spent if all strings are indexed and the binary search continued through to completion.

Front coding

Front coding is a further worthwhile improvement. It utilizes the fact that consecutive words in a sorted list are likely to share a common prefix. Two integers are stored with each word: one to indicate how many prefix characters are the same as the previous word and the other to record how many suffix characters remain when the prefix is removed. The two integers are followed by the stipulated number of suffix characters. Table 4.2 shows part of the lexicon of the *Bible* text. The first column gives the words in full, each one prefixed by its length in bytes. Allowing 1 byte for each length field, the storage required by this set of strings is 96 bytes.

The second column of Table 4.2 shows the front-coded version of Figure 4.1. The amount of saving depends on the lexicon, but typically an average of three to five prefix characters will match, at the cost of 1 extra byte to store the second integer. For the set of words shown, which were extracted from a relatively small lexicon, the average net saving is 2.5 bytes per word. The larger the lexicon, the more matching prefix characters there are, simply because more strings are forced into the same range of possible characters. To see this, recall that uppercase characters are folded

Table 4.2 Front coding (the word before *jezebel* was *jezaniah*).

Word	Complete front coding	Partial "3-in-4" front coding
7, jezebel	3, 4, ebel	, 7, jezebel
5, jezer	4, 1, r	4, 1, r
7, jezerit	5, 2, it	5, 2, it
6, jeziah	3, 3, iah	3, , iah
6, jeziel	4, 2, el	, 6, jeziel
7, jezliah	3, 4, liah	3, 4, liah
6, jezoar	3, 3, oar	3, 3, oar
9, jezrahiah	3, 6, rahiah	3, , rahiah
7, jezreel	4, 3, eel	, 7, jezreel
11, jezreelites	7, 4, ites	7, 4, ites
6, jibsam	1, 5, ibsam	1, 5, ibsam
7, jidlaph	2, 5, dlaph	2, , dlaph

to lowercase, leaving 10 digits and 26 letters, and imagine each alphanumeric string to be a radix-36 fractional number between zero and one. Character *0* is interpreted as digit 0, *1* as digit 1, and so on; characters *a* to *z* are interpreted as digits 10 through 35, respectively.

Further, assume a radix-36 point immediately to the left of the first digit. That is, instead of the ordinary radix-10 representation of numbers in which 10 digits are used, suppose that a number system with 36 digits is used. For example, the string 01 is represented by the decimal number $0.00077 = 0 \times 36^{-1} + 1 \times 36^{-2}$; the string *a* corresponds to the decimal number $0.27777 = 10 \times 36^{-1}$; the word *bed* corresponds to $0.31664 = 11 \times 36^{-1} + 14 \times 36^{-2} + 13 \times 36^{-3}$; and *zzzzz* . . . is, for all practical purposes, equivalent to 1.0. If a lexicon has n strings, the average gap between strings interpreted in this way must be less than $1/n$, and if one string in the sorted lexicon corresponds to the value x, the next one will, on average, correspond to the value $x + 1/n$. For any random value x in $[0, 1)$, the expected number of matching radix-k digits between x and $x+1/n$ is $\log_k n - 1$ for all but small values of k. Thus, for the text *Bible*, the average prefix match length must be at least $\log_{36} 9{,}020 - 1 = 1.5$, and for the hypothetical lexicon of $n = 1{,}000{,}000$ strings, it is at least $\log_{36} 1{,}000{,}000 - 1 = 2.9$.

Of course, the strings are not random. In any dictionary, some prefixes are used extensively, while others never appear. This means that the actual saving reaped from front coding considerably exceeds the amount just calculated. In *Bible*, the average prefix length is 3.6 characters, and the average string length is 6.1. This amounts to a net saving of 2.6 bytes out of each 7.1 bytes of string storage. On the

TREC lexicon, the corresponding values are 4.8 and 7.3, amounting to a net saving of 3.8 bytes per 8.3 bytes. As a general rule of thumb:

Front coding yields a net saving of about 40 percent of the space required for string storage in a typical lexicon for the English language.

The problem with the "Complete front coding" column of Table 4.2 is that binary search is no longer possible. Even if a string pointer leads directly to the entry *3, 4, ebel*, it is impossible to know what the word is. The third column shows a good strategy to use in practice. Here, every fourth word—the one indexed by the block pointer—is stored without front coding, so that binary search can proceed. Just one integer field—the length—need accompany these indexed words, as Table 4.2 shows. However, they might be stored with a prefix length of zero, introducing a small amount of redundancy for the sake of consistency. Similarly, the second suffix-length integer is omitted from the last string of each block since it can be inferred from the next string pointer.

On a large lexicon, a 3-in-4 front-coding strategy can be expected to save about 4 bytes on each of three words, at the cost of 2 extra bytes of prefix-length information—a net gain of 10 bytes per four-word block. This reduces the storage required for the hypothetical lexicon by a further 2.5 Mbytes, bringing the total to 15.5 Mbytes.

Each entry in the three arrays can be squeezed into less space. For example, $\lceil \log N \rceil$ bits is sufficient for each f_t value, where N is the number of documents in the collection; similarly, the inverted file pointer and string pointer can be stored as minimal binary codes in their respective ranges. The calculations above assumed four bytes for each f_t value and inverted file pointer, but coded in this way they will occupy perhaps 28 bits each, saving a further 1 Mbyte. The prefix- and suffix-length fields in the list of strings can be extracted and stored in their own parallel arrays, the former occupying perhaps three or four bits and the latter five or six bits. Placing a limit of 8 or 16 on the maximum prefix match length is not restrictive. If more characters match, a prefix length of too few characters can nevertheless be coded: this just means that a few extra bytes must be stored, an insignificant penalty compared to the saving of four or five bits for every word in the lexicon.

These efforts to reduce the size of the lexicon data structure are prompted by a desire to store the index entry address and f_t value in main memory. It seems that the only remaining mechanism to reduce the storage space would be to dispense with the strings entirely. Incredible as it may sound, this is exactly what we do next.

Minimal perfect hashing

A *hash function* h is a mechanism for mapping a set L of n keys x_j into a set of integer values $h(x_j)$ in the range $0 \leq h(x_j) \leq m-1$, with duplicates allowed. This is a standard method for implementing a lookup table and provides fast average-case behavior. When the data consists of n integer keys, for example, a common

hash function is to take $h(x) = x \bmod m$ for some value $m > n/\alpha$, where α is the *loading*, the ratio of records to available addresses, and m is usually chosen to be a prime number. Thus, if asked to provide a hash function for 1,000 integer keys, a programmer might suggest something like $h(x) = x \bmod 1{,}399$ to give a load factor of $\alpha = 0.7$ in a table declared to have 1,399 locations.

The smaller the value of α, the less likely it is that two of the keys *collide* at the same hash value. Nevertheless, collisions are almost impossible to avoid. This fact is somewhat surprising when first encountered and is demonstrated by the well-known *birthday paradox*, which asks, "Given that there are 365 days in the year, how many people must be collected together before the probability that two people share a birthday exceeds 0.5?" In other words, given 365 hash slots, how many keys can be randomly assigned before the probability of collision exceeds 0.5? The initial reaction is usually to say that lots of people are needed. In fact, the answer is just 23, and the chance that a hash function of realistic size is collision-free is insignificant.[1] For example, with 1,000 keys and 1,399 randomly selected slots, the probability of there being no collisions at all is 2.35×10^{-217}. (The derivation of this probability can be found in the next subsection: it is given by Equation 4.1 with $m = 1{,}399$ and $n = 1{,}000$.) The inevitability of collisions has led to a large body of literature on how best to handle them. Here, however, we seek instead those one-in-a-million hash functions that do manage to avoid all collisions.

If the hash function has the additional property that, for x_i and x_j in L, $h(x_i) = h(x_j)$ if and only if $i = j$, it is a *perfect* hash function. In this case, no collisions arise when hashing the set of keys L.

If a hash function h is both perfect and maps into the range $m = n$, each of the n keys hashes to a unique integer between 1 and n and the table loading is $\alpha = 1.0$. Then h is a *minimal perfect hash function*, or MPHF. An MPHF provides guaranteed one-probe access to a set of keys, and the table contains no unused slots.

Finally, if a hash function has the property that if $x_i < x_j$ then $h(x_i) < h(x_j)$, it is *order preserving*. Given an order-preserving minimal perfect hash function (abbreviated OPMPHF and pronounced "oomph!"), keys are located in constant time without any space overhead and can be processed in sorted order should that be necessary. An OPMPHF simply returns the sequence number of a key directly.

Of course, an MPHF or OPMPHF h for one set L will not be perfect for another set of keys, and so it is nothing more than a precalculated lookup function for a single set. Nevertheless, there are occasions when the precalculation is warranted, and the space saving can be great.

---

1 It is always interesting to try this experiment with groups of people. Having tried this experiment many times with students while teaching them about hash tables, there is one important tip that we would like to pass on: the participants should be asked to write down their birthday (or any other date) *before* the collation process is commenced, so that the temptation for mysterious negative feedback is eliminated. In our experience, unless this is done, it can sometimes take 366 students before the first collision. Perhaps there is a psychology paradox here too.

Table 4.3 Tables for a minimal perfect hash function: (a) terms and hash functions; (b) function g.

(a)

Term t	$h_1(t)$	$h_2(t)$	$h(t)$
jezebel	5	9	0
jezer	5	7	1
jezerit	10	12	2
jeziah	6	10	3
jeziel	13	7	4
jezliah	13	11	5
jezoar	4	2	6
jezrahiah	0	3	7
jezreel	6	3	8
jezreelites	8	4	9
jibsam	9	14	10
jidlaph	3	1	11

(b)

x	$g(x)$
0	0
1	4
2	0
3	7
4	6
5	0
6	1
7	1
8	3
9	0
10	2
11	2
12	0
13	3
14	10

As an example, Table 4.3 gives an OPMPHF for the same set of 12 keys that was used earlier. The methodology leading to this hash function is described in the next section. The construction presumes the existence of two normal hash functions $h_1(t)$ and $h_2(t)$ that map strings into integers in the range $0 \ldots m-1$ for some value $m \geq n$, with duplicates permitted. One way to define these is to take the numeric value for each character of a string radix 36, as before, and compute a weighted sum for some set of weights w_j,

$$h_j(t) = \left(\sum_{i=1}^{|t|} t[i] \times w_j[i] \right) \bmod m,$$

where $t[i]$ is the radix-36 value of the ith character of term t and $|t|$ is the length in characters of term t. Then two different sets of weights $w_1[i]$ and $w_2[i]$ for $1 \leq i \leq |t|$ yield two different functions $h_1(t)$ and $h_2(t)$. As well as these two functions, a rather special array g is needed that maps numbers $0 \ldots m-1$ into the range $0 \ldots n-1$; this is shown in Table 4.3b.

To evaluate the OPMPHF $h(t)$ for some string t, calculate

$$h(t) = g(h_1(t)) +_n g(h_2(t)),$$

4.3 Boolean query processing

The simplest type of query is the Boolean query, in which terms are combined with the connectives AND, OR, and NOT. It is relatively straightforward to process such a query using an inverted file index. The lexicon is searched for each term; each inverted list is retrieved and decoded; and the lists are merged, taking the intersection, union, or complement, as appropriate. Finally, the documents so indexed are retrieved and displayed to the user as the list of answers. For a typical query of 5 to 10 terms, a second or so is spent reading and decoding inverted lists; then—depending on the number of answers—accessing, decoding, and writing the documents takes anything from tenths of a second to hundreds of seconds.

Conjunctive queries

Let us examine this process in detail. In the next few pages we suppose that the query is a *conjunction*, consisting of terms connected with AND operations such as

text AND *compression* AND *retrieval*.

Although this is not the only form of Boolean query, it is sufficiently common to warrant special discussion.

Suppose a conjunctive query of r terms is being processed. First, each term is stemmed and then located in the lexicon. As discussed in Section 4.1, the lexicon might be resident in memory, if space is available, or on disk. In the latter case, one disk access per term is required. The next step is to sort the terms by increasing frequency; all subsequent processing is carried out in this order. This is one reason why we stipulated earlier that the lexicon should hold, for each term t, its frequency of appearance f_t. If this information were held with the inverted list instead, it would not be possible to process terms in increasing frequency order without fetching and buffering all the inverted lists. Processing the least frequent term first is not essential, but it makes retrieval significantly more efficient.

Next, the inverted list for the least frequent term is read into memory. This list establishes a set of *candidates*, documents that have not yet been eliminated and might be answers to the query. All remaining inverted lists are processed against this set of candidates, in increasing order of term frequency. This strategy is described in Figure 4.8, where C is the set of candidate document numbers and I_t the inverted list for term t.

In a conjunctive query, a candidate cannot be an answer unless it appears in all inverted lists; if it is omitted from any list, it can be discarded at once. This means that the size of the set of candidates is nonincreasing. To process a term, each document in the set is checked and removed if it does not appear in the term's inverted list. From this perspective, the dominant operation when processing conjunctive queries is not so much merging as "looking up" since the set of candidates is never larger than the inverted lists and shrinks as more terms are processed and the inverted lists grow longer.

To evaluate a conjunctive Boolean query,

1. For each query term t,
   (a) Stem t.
   (b) Search the lexicon.
   (c) Record f_t and the address of I_t, the inverted file entry for t.
2. Identify the query term t with the smallest f_t.
3. Read the corresponding inverted file entry I_t.
   Set $C \leftarrow I_t$. C is the list of *candidates*.
4. For each remaining term t,
   (a) Read the inverted file entry, I_t.
   (b) For each $d \in C$,
       if $d \notin I_t$ then
           set $C \leftarrow C - \{d\}$.
   (c) If $|C| = 0$,
       return, there are no answers.
5. For each $d \in C$,
   (a) Look up the address of document d.
   (b) Retrieve document d and present it to the user.

Figure 4.8 Evaluating conjunctive Boolean queries.

When all inverted lists have been processed, the remaining candidates, if there are any, are the desired answers.

Term processing order

There are two reasons to select the least frequent term to initialize the set of candidates. The first is to minimize the amount of temporary memory space required during query processing. Since the size of the set is nonincreasing, it is largest when it is initialized, so the memory required is minimized by selecting the shortest inverted list first. Processing the remaining terms in increasing frequency order does not affect the peak memory space required. It is still a good idea, though, because it may quickly reduce the number of candidates, perhaps even to zero. Obviously, if the candidate set becomes empty, no further terms need be considered at all. A query that returns no answers is not particularly informative; nevertheless, a surprising fraction of actual queries have just this result. Such queries are, of course, typically followed by broader queries formed by removing one or more query terms.

The second reason to process terms in order of frequency is that it is faster because each inverted list involves a sequence of lookup operations rather than a merge

operation. Merging takes time proportional to the sum of the sizes of the two sets. If there are $|C|$ candidates and f_t pointers in the inverted list, then a linear merge takes $|C| + f_t$ steps. This is unnecessarily expensive if $|C|$ is much smaller than f_t. For example, suppose that $|C| = 60$ and $f_t = 60{,}000$—not unreasonable figures with a large collection like *TREC*. Then merging will take 60,060 steps, or 1,000 steps per candidate.

Since the list of document numbers in any inverted list is sorted, instead of a linear merge the inverted list can be binary-searched for each of the candidates. This is not possible if the list is stored compressed, but we will continue the calculation just to see what might be achieved. Each binary search will take $\log 60{,}000 \approx 16$ computation steps, and the entire inverted list can be processed with just 1,000 steps.

Each inverted list must be read from disk, so the elapsed time to process term t remains proportional to f_t. Still, much less processor time is needed—if a binary search can be used. But a binary search is possible only when the document numbers are in sorted order within list I_t, and if they can be accessed randomly. Unfortunately, the use of compression destroys random access capabilities since it is impossible to index into the middle of a compressed inverted list and decode a document number. Thus, if the inverted file is compressed, not only must a linear merge be used irrespective of the length of the inverted list, but each inverted list must be fully decompressed in order to perform the merge. The cost is still linear, but the constant of proportionality is large. At face value, then, the use of compression saves a great deal of space in the inverted file but imposes a substantial time penalty during conjunctive query processing.

Random access and fast lookup

What is needed is some provision for random access into the inverted list to support faster searching. This is the problem of synchronization and was discussed in Section 2.7. Suppose that the compressed inverted list for some term t is partially indexed itself, and that every b_tth pointer is duplicated into an index array. For example, if $b_t = 4$, every fourth pointer in the inverted file is indexed, and both the bit address within the inverted file of that pointer and the document number it corresponds to are stored in some auxiliary structure. If a document number x is to be looked up, it is first sought in the auxiliary index. The result tells which block of the inverted file that document number appears in, if it does appear, and also gives the bit address in the inverted file at which decoding should commence to access it. This is much the same structure as was described in Section 4.1 for disk-based storage of the lexicon, except that here both components are read from disk when required, and the purpose of indexing is to save decompression time rather than memory space.

Several issues must be resolved. The first is the storage mechanism used for the index. The second is a suitable value for b_t, the blocking constant for term t. The third concerns the trade-off of time for space—how much time is saved and at what cost in space.

Querying and Ranking

Nonconjunctive queries

So far we have considered only conjunctive queries. Another common form is a conjunction of disjunctions, where several alternatives are specified for each component of what is basically a conjunctive query:

(*text* OR *data* OR *image*) AND
(*compression* OR *compaction*) AND
(*retrieval* OR *indexing* OR *archiving*).

In this case, the terms comprising each conjunct can be processed simultaneously, candidates remaining in the set if they appear in any member of the OR group. The set of candidates should be initialized to the union of the members of the smallest conjunct. As a pessimistic approximation, the size of each conjunct can be estimated by summing the f_t values for its constituent terms, ignoring the possibility of overlap in the OR component. This strategy should allow time savings similar to those described above to be achieved.

Even more general queries, such as

(*information* AND (*retrieval* OR *indexing*)) OR
((*text* OR *data*) AND (*compression* OR *compaction*)),

can be transformed into a conjunction of disjunctions by the query processing system, although this may cause terms to be duplicated. The above example could be recast as

(*information* OR *text* OR *data*) AND
(*retrieval* OR *indexing* OR *text* OR *data*) AND
(*information* OR *compression* OR *compaction*) AND
(*retrieval* OR *indexing* OR *compression* OR *compaction*).

When a Boolean query expression becomes as complex as this, it is time to consider changing tack entirely and using another information retrieval paradigm—*informal* or *ranked* queries.

4.4 Ranking and information retrieval

Boolean queries are not the only method of searching for information. If some exact subset of the document being sought is known, then they are certainly appropriate, which is why they have been so successful in areas such as commercial databases and bibliographic retrieval systems. Often, however, the information requirement is less precisely known. For this reason, it is sometimes useful to be able to specify a list of terms that give a good indication of which documents are *relevant*, though they will not necessarily all be present in the documents sought. The system should rank the entire collection with respect to the query, so that the top 100, say, ranked documents can be examined for relevance and those that constitute the answer set extracted. In this section we study how to assign a *similarity measure* to each document that indicates how closely it matches a query.

Table 4.6 A small document collection: six documents over 10 terms.

d	Document D_d
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	In the pot cold, in the pot hot,
5	Pease porridge, pease porridge,
6	Eat the lot.

Coordinate matching

One way to provide more flexibility than a simple binary yes-or-no answer is to count the number of query terms that appear in each document. The more terms that appear, the more likely it is that the document is relevant. This approach is called *coordinate matching*. The query becomes a hybrid, intermediate between a conjunctive AND query and a disjunctive OR query: a document that contains any of the terms is viewed as a potential answer, but preference is given to documents that contain all or most of them. All necessary information is in the inverted file, and it is relatively straightforward to implement this strategy.

Consider, for example, the six documents shown in Table 4.6—a revision of the doggerel already used as an example in Chapter 3. For the query *eat*, it is clear that document 6 is the best (and only) answer. But what about the query *hot porridge*? In a conjunctive Boolean sense, document 1 is the only answer. But three other documents might also be relevant, and coordinate matching yields a ranking $D_1 > D_2 = D_4 = D_5 > D_3 = D_6 = 0$. Documents containing only one of the terms are available as answers, should the user wish to inspect them.

Inner product similarity

This process can be formalized as an inner product of a query vector with a set of document vectors. Table 4.7a shows the same collection, with a set of binary document vectors represented by n components, n being the number of distinct terms in the collection. For convenience the terms are abbreviated to three letters, and to keep the example manageable it is assumed that the terms *in* and *the* are stopped. The two example queries can also be represented as n-dimensional vectors and are shown in Table 4.7b.

Using this formulation, the similarity measure of query Q with document D_d is expressed as

$$M(Q, D_d) = Q \cdot D_d$$

Table 4.7 Vectors for inner product calculation: (a) document vectors; (b) query vectors.

(a)	d	Document vectors $\langle w_{d,t} \rangle$									
		col	*day*	*eat*	*hot*	*lot*	*nin*	*old*	*pea*	*por*	*pot*
	1	1	0	0	1	0	0	0	1	1	0
	2	0	0	0	0	0	0	0	1	1	1
	3	0	1	0	0	0	1	1	0	0	0
	4	1	0	0	1	0	0	0	0	0	1
	5	0	0	0	0	0	0	0	1	1	0
	6	0	0	1	0	1	0	0	0	0	0
(b)	*eat*	0	0	1	0	0	0	0	0	0	0
	hot porridge	0	0	0	1	0	0	0	0	1	0

where the operation $\cdot$ is inner product multiplication. The *inner product* of two n-vectors $X = \langle x_i \rangle$ and $Y = \langle y_i \rangle$ is defined to be

$$X \cdot Y = \sum_{i=1}^{n} x_i y_i.$$

For example,

$$M(\textit{hot porridge}, D_1) = (0,0,0,1,0,0,0,0,1,0) \cdot (1,0,0,1,0,0,0,1,1,0) = 2.$$

Despite the additional power introduced by the notion of ranking, this simple coordinate-matching approach has three drawbacks. First, it takes no account of term frequency. In Table 4.6, *porridge* appears twice in document 1 and only once in document 2, yet on the query *porridge* the two documents are ranked equally. Second—and this may seem the same point, but it is not—the formula takes no account of term scarcity. Since *eat* appears in only one document, it is, at face value at least, a more important term than *porridge*, which appears in three of the documents. Third, long documents with many terms will automatically be favored by the ranking process because they are likely to contain more of any given list of query terms merely by virtue of the diversity of text present in a long document.

The first problem can be tackled by replacing the binary "present" or "not present" judgment with an integer indicating how many times the term appears in the document. This occurrence count is called the *within-document frequency* of the term and is denoted $f_{d,t}$. When the inner product is calculated, the $f_{d,t}$ values are then taken into account. For example, the similarity calculation for the sample query would become

$$M(\textit{hot porridge}, D_1) = (0,0,0,1,0,0,0,0,1,0) \cdot (1,0,0,1,0,0,0,2,2,0) = 3$$

since document D_1 contains *hot* once and *porridge* twice. More generally, term t in document d can be assigned a *document-term weight*, denoted $w_{d,t}$, and another weight $w_{q,t}$ in the query vector. The similarity measure is the inner product of these two—the sum of the products of the weights of the query terms and the corresponding document terms:

$$M(Q, D_d) = Q \cdot D_d = \sum_{t=1}^{n} w_{q,t} \cdot w_{d,t}.$$

It is normal to assign $w_{q,t} = 0$ if t does not appear in Q, so the measure can be stated as

$$M(Q, D_d) = \sum_{t \in Q} w_{q,t} \cdot w_{d,t}.$$

This suggests an evaluation mechanism based on inverted files. However, before discussing implementation options, let us explore the other two problems mentioned above.

The second problem is that no emphasis is given to scarce terms. Indeed, a document with enough appearances of a common term will always be ranked first if the query contains that term, irrespective of other words. The solution is for the term weights to be reduced for terms that appear in many documents, so that a single appearance of *the* counts far less than a single appearance of, say, *Jezebel*. This can be done by weighting terms according to their *inverse document frequency*, often abbreviated to IDF. This suggestion is consistent with the observations of George Zipf, who published a remarkable book about naturally occurring distributions called *Human Behavior and the Principle of Least Effort* (Zipf 1949). Zipf observed that the frequency of an item tends to be inversely proportional to its rank. That is, if rank can be regarded as a measure of importance, then the weight w_t of a term t might be calculated as

$$w_t = \frac{1}{f_t},$$

where, as before, f_t is the number of documents that contain term t.

The term weight can then be used in three different ways. First, and most obvious, it can be multiplied by a *relative term frequency* value, denoted $r_{d,t}$, to generate the document-term weight $w_{d,t}$, where $r_{d,t}$ itself can be calculated in several different ways and is discussed further below. Second, the term weight can be combined multiplicatively with $r_{q,t}$ to yield a query-term weight $w_{q,t}$. Third, it can be used in calculating both $w_{d,t}$ and $w_{q,t}$, that is, applied twice. Nor is the formulation above the only possibility for w_t, the IDF component. Others that have appeared in the

literature include

$$w_t = \log_e\left(1 + \frac{N}{f_t}\right),$$

$$w_t = \log_e\left(1 + \frac{f^m}{f_t}\right),$$

and

$$w_t = \log_e \frac{N - f_t}{f_t},$$

where N is the number of documents in the collection and f^m is the largest $f_{d,t}$ value in the collection. The first of these three is now regarded as being the "usual" mechanism, with the logarithm included to prevent a term for which $f_t = 1$ from being regarded as twice as important as a term for which $f_t = 2$.

Similarly, the relative term frequency component $r_{d,t}$ can be calculated in several different ways as a function of $f_{d,t}$, the within-document frequency:

$$r_{d,t} = 1,$$

$$r_{d,t} = f_{d,t},$$

$$r_{d,t} = 1 + \log_e f_{d,t},$$

$$r_{d,t} = \left(K + (1 - K)\frac{f_{d,t}}{\max_i f_{d,i}}\right),$$

and so on. The third formula uses a logarithm to give diminishing returns as term frequencies increase. No explicit upper bound is enforced, but a term must be very frequent indeed to have a term frequency contribution of greater than four. In the fourth formula, the first appearance of a term in a document contributes much more than the second and subsequent occurrences, with the constant $0 \leq K \leq 1$ controlling the balance between initial and later appearances. This is quite plausible, in that the first appearance of a term should contribute more of the available similarity than, say, the fifth. The factor $\max_i f_{d,i}$ is the maximum frequency of any term in document d and is introduced to keep the term frequency multiplier from becoming greater than one.

The document vectors are then calculated as either

$$w_{d,t} = r_{d,t}$$

or

$$w_{d,t} = r_{d,t} \cdot w_t \qquad (\text{TF} \times \text{IDF}).$$

The latter method for assigning document-term weights is called the TF×IDF rule: term frequency times inverse document frequency. Note that neither the TF nor the IDF components should be interpreted literally as being the functions that their names suggest. A similarity heuristic is called "TF×IDF" whenever it uses the term frequency $f_{d,t}$ in a monotonically increasing way, and the term's document frequency f_t in a monotonically decreasing way.

The query-term weights $w_{q,t}$ are calculated similarly. The within-query frequency $f_{q,t}$ may or may not be taken into account, and the term weight w_t may or may not be taken into account.

There is no particular magic in any of several hundred similarity formulas allowed by these various expressions, and no single combination of them outperforms any of the others over a range of different queries—Zobel and Moffat (1998) have evaluated a large number of them against the *TREC* data. Furthermore, the above lists are certainly not exhaustive—there are plenty of other formulations for both w_t and $r_{d,t}$ that have been proposed. What is worthy of note is that all of the suggestions comply with two straightforward monotonicity constraints:

A term that appears in many documents should not be regarded as being more important than a term that appears in a few, and a document with many occurrences of a term should not be regarded as being less important than a document that has just a few.

Beyond that, which is used in any particular situation tends to be a subjective choice rather than an objective one.

It is, however, helpful to assume a particular formulation in the development of the next section, and for the sake of concreteness, it will be supposed that the document and query vectors are described by

$$\begin{aligned} &w_t = \log_e(1 + N/f_t) \\ r_{d,t} &= 1 + \log_e f_{d,t} \qquad r_{q,t} = 1 \\ w_{d,t} &= r_{d,t} \qquad\qquad\quad w_{q,t} = r_{q,t} \cdot w_t \end{aligned} \tag{4.2}$$

Whatever the weighting rule, all inner product methods are vulnerable to the third effect described above: long documents are favored over short ones since they contain more terms and so the value of the inner product increases.

For this reason, it is also common to introduce a *normalization* factor to discount the contribution of long documents. Hence another variation of the inner product rule is to measure similarity by

$$M(Q, D_d) = \frac{\sum_{t \in Q} w_{q,t} \cdot w_{d,t}}{|D_d|}$$

where $|D_d| = \sum_i f_{d,i}$ is the *length* of document D_d, obtained by counting the number of indexed terms. Another proposal is to use the square root of this length.

Fortunately, there is a simple way to understand these various rules using vector space models.

Vector space models

Whatever term weights w_t and relative term and document frequencies $r_{d,t}$ and $r_{q,t}$ are assigned, and whatever document-term weights $w_{d,t}$ and query-term weights

$w_{q,t}$ arise from these assignments, the result is the same—each document is represented by a vector in n-dimensional space, and the query is also represented as an n-dimensional vector.

One obvious similarity measure for a pair of vectors is the familiar Euclidean distance:

$$M(Q, D_d) = \sqrt{\sum_{t=1}^{n} |w_{q,t} - w_{d,t}|^2}$$

This is actually a *dissimilarity* measure since a large numeric value indicates that the vectors are very different; to turn it into a similarity measure, the reciprocal is taken. This measure suffers from the opposite fault to the inner product—because the query is usually much shorter than the documents, it discriminates *against* long documents.

What is really of interest is the *direction* indicated by the two vectors, or, more precisely, the *difference* in direction, irrespective of length. Moreover, difference in direction is a well-understood concept in geometry—it is the angle between the two vectors.

Simple vector algebra yields an elegant method for determining similarity. If X and Y are two n-dimensional vectors $\langle x_i \rangle$ and $\langle y_i \rangle$, the angle θ between them satisfies

$$X \cdot Y = |X||Y| \cos\theta$$

where $X \cdot Y$ is the vector inner product defined above and

$$|X| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

is the Euclidean length of X. The angle θ can be calculated from

$$\cos\theta = \frac{X \cdot Y}{|X||Y|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

This has two implications. First, it justifies the normalization that was described at the end of the previous section. The normalization factor is the Euclidean length of the document—that is, the length in n-space of the set of document-term weights describing the document. Second, this formula provides a clear visualization of what the ranking rule accomplishes. Imagine the set of documents being points in the positive region of n-dimensional space, with short documents close to the origin and long ones farther away from it. A query can be imagined as a ray emanating from the origin, piercing this space in some desired direction. Within this framework, the task of the ranking method is to select those documents lying closest to this ray in an angular sense. Since $\cos\theta = 1$ when $\theta = 0$ and $\cos\theta = 0$ when the vectors are orthogonal, the similarity measure can be taken as the cosine of the an-

gle between the document and query vector—the larger this cosine, the greater the similarity.

These considerations lead to the *cosine rule* for ranking:

$$cosine(Q, D_d) = \frac{Q \cdot D_d}{|Q||D_d|} = \frac{1}{W_q W_d} \sum_{t=1}^{n} w_{q,t} \cdot w_{d,t}$$

where

$$W_d = \sqrt{\sum_{t=1}^{n} w_{d,t}^2}$$

is the Euclidean length—the *weight*—of document d and

$$W_q = \sqrt{\sum_{t=1}^{n} w_{q,t}^2}$$

is the weight of the query.

This rule can be used with any of the term-weighting methods described above. Suppose, for example, that the variant described in Equation 4.2 is used. The similarity calculation is then described by

$$cosine(Q, D_d) = \frac{1}{W_d W_q} \sum_{t \in Q \cap D_d} (1 + \log_e f_{d,t}) \cdot \log_e \left(1 + \frac{N}{f_t}\right). \qquad (4.3)$$

Indeed, there is no need to factor in W_q since it is constant for any given query, and while it affects the numeric similarity scores, the document ordering is unaffected.

Table 4.8 applies the cosine measure to the collection of $N = 6$ documents and $n = 10$ terms given in Table 4.6. Table 4.8a shows the corresponding document vectors, where the entry for row d and column t is $w_{d,t}$, the weight of term t in d. Also recorded are f_t, the number of documents containing t, and w_t, calculated using the inverse document frequency rule $w_t = \log_e(1 + N/f_t)$.

Table 4.8b shows four queries Q and the resulting values of $cosine(Q, D_d)$. For the single-term query *eat*, the ranking is simple since it appears in only one document. Not so straightforward is the second query, *porridge*—document 5 beats document 1 because it is shorter. The final query—*eat nine day old porridge*—is best matched by line 3, despite the fact that it does not contain the word *eat*, illustrating the power (and perhaps also a drawback) of ranked queries. Someone looking for this doggerel in a retrieval system would almost certainly find it with the last query.

Table 4.8 Application of the cosine measure: (a) term frequencies $f_{d,t}$ and document weights; (b) cosine similarities for queries.

(a)

d	Document vectors $\langle w_{d,t} \rangle$										W_d
	col	*day*	*eat*	*hot*	*lot*	*nin*	*old*	*pea*	*por*	*pot*	
1	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.7	1.7	0.0	2.78
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	1.73
3	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.73
4	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.7	2.21
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.7	1.7	0.0	2.39
6	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.41
f_t	2	1	1	2	1	1	1	3	3	2	
w_t	1.39	1.95	1.95	1.39	1.95	1.95	1.95	1.10	1.10	1.39	

(b)

d	Query			
	eat	*porridge*	*hot porridge*	*eat nine day old porridge*
	$W_q = 1.95$	$W_q = 1.10$	$W_q = 1.77$	$W_q = 3.55$
1	0.00	0.61	0.66	0.19
2	0.00	0.58	0.36	0.18
3	0.00	0.00	0.00	0.63
4	0.00	0.00	0.36	0.00
5	0.00	0.71	0.44	0.22
6	0.71	0.00	0.00	0.39
Top	6	5	1	3

4.5 Evaluating retrieval effectiveness

There are many variations on these ranking rules, some of which were described above. In order to compare them, we need some way to quantify their performance. A ranking rule's performance should be based on the total ranking it imposes on the collection with respect to a query. A number of methods have been suggested for this. None are entirely satisfactory, but this is a natural consequence of attempting to represent multidimensional behavior with a single representative value. First we define two important measures of effectiveness: recall and precision.

Recall and precision

The most common way to describe retrieval performance is to calculate how many of the relevant documents have been retrieved and how early in the ranking they were listed. This leads to the following definitions.