1 Introduction

1.1 What Is Information Retrieval?

Information retrieval (IR) is concerned with representing, searching, and manipulating large collections of electronic text and other human-language data. IR systems and services are now widespread, with millions of people depending on them daily to facilitate business, education, and entertainment. Web search engines — Google, Bing, and others — are by far the most popular and heavily used IR services, providing access to up-to-date technical information, locating people and organizations, summarizing news and events, and simplifying comparison shopping. Digital library systems help medical and academic researchers learn about new journal articles and conference presentations related to their areas of research. Consumers turn to local search services to find retailers providing desired products and services. Within large companies, enterprise search systems act as repositories for e-mail, memos, technical reports, and other business documents, providing corporate memory by preserving these documents and enabling access to the knowledge contained within them. Desktop search systems permit users to search their personal e-mail, documents, and files.

1.1.1 Web Search

Regular users of Web search engines casually expect to receive accurate and near-instantaneous answers to questions and requests merely by entering a short query — a few words — into a text box and clicking on a search button. Underlying this simple and intuitive interface are clusters of computers, comprising thousands of machines, working cooperatively to generate a ranked list of those Web pages that are likely to satisfy the information need embodied in the query. These machines identify a set of Web pages containing the terms in the query, compute a score for each page, eliminate duplicate and redundant pages, generate summaries of the remaining pages, and finally return the summaries and links back to the user for browsing.

In order to achieve the subsecond response times expected from Web search engines, they incorporate layers of caching and replication, taking advantage of commonly occurring queries and exploiting parallel processing, allowing them to scale as the number of Web pages and users increase. In order to produce accurate results, they store a "snapshot" of the Web. This snapshot must be gathered and refreshed constantly by a *Web crawler*, also running on a cluster of hundreds or thousands of machines, and downloading periodically — perhaps once a week — a fresh copy of each page. Pages that contain rapidly changing information of high quality, such as news services, may be refreshed daily or hourly.

Consider a simple example. If you have a computer connected to the Internet nearby, pause for a minute to launch a browser and try the query "information retrieval" on one of the major commercial Web search engines. It is likely that the search engine responded in well under a second. Take some time to review the top ten results. Each result lists the URL for a Web page and usually provides a title and a short snippet of text extracted from the body of the page. Overall, the results are drawn from a variety of different Web sites and include sites associated with leading textbooks, journals, conferences, and researchers. As is common for *informational* queries such as this one, the Wikipedia article[1] may be present. Do the top ten results contain anything inappropriate? Could their order be improved? Have a look through the next ten results and decide whether any one of them could better replace one of the top ten results.

Now, consider the millions of Web pages that contain the words "information" and "retrieval". This set of pages includes many that are relevant to the subject of information retrieval but are much less general in scope than those that appear in the top ten, such as student Web pages and individual research papers. In addition, the set includes many pages that just happen to contain these two words, without having any direct relationship to the subject. From these millions of possible pages, a search engine's ranking algorithm selects the top-ranked pages based on a variety of features, including the content and structure of the pages (e.g., their titles), their relationship to other pages (e.g., the hyperlinks between them), and the content and structure of the Web as a whole. For some queries, characteristics of the user such as her geographic location or past searching behavior may also play a role. Balancing these features against each other in order to rank pages by their expected relevance to a query is an example of *relevance ranking*. The efficient implementation and evaluation of relevance ranking algorithms under a variety of contexts and requirements represents a core problem in information retrieval, and forms the central topic of this book.

1.1.2 Other Search Applications

Desktop and file system search provides another example of a widely used IR application. A desktop search engine provides search and browsing facilities for files stored on a local hard disk and possibly on disks connected over a local network. In contrast to Web search engines, these systems require greater awareness of file formats and creation times. For example, a user may wish to search only within their e-mail or may know the general time frame in which a file was created or downloaded. Since files may change rapidly, these systems must interface directly with the file system layer of the operating system and must be engineered to handle a heavy update load.

[1] `en.wikipedia.org/wiki/Information_retrieval`

Lying between the desktop and the general Web, enterprise-level IR systems provide document management and search services across businesses and other organizations. The details of these systems vary widely. Some are essentially Web search engines applied to the corporate intranet, crawling Web pages visible only within the organization and providing a search interface similar to that of a standard Web search engine. Others provide more general document- and content-management services, with facilities for explicit update, versioning, and access control. In many industries, these systems help satisfy regulatory requirements regarding the retention of e-mail and other business communications.

Digital libraries and other specialized IR systems support access to collections of high-quality material, often of a proprietary nature. This material may consist of newspaper articles, medical journals, maps, or books that cannot be placed on a generally available Web site due to copyright restrictions. Given the editorial quality and limited scope of these collections, it is often possible to take advantage of structural features — authors, titles, dates, and other publication data — to narrow search requests and improve retrieval effectiveness. In addition, digital libraries may contain electronic text generated by *optical character recognition* (OCR) systems from printed material; character recognition errors associated with the OCR output create yet another complication for the retrieval process.

1.1.3 Other IR Applications

While search is the central task within the area of information retrieval, the field covers a wide variety of interrelated problems associated with the storage, manipulation, and retrieval of human-language data:

- Document *routing*, *filtering*, and *selective dissemination* reverse the typical IR process. Whereas a typical search application evaluates incoming queries against a given document collection, a routing, filtering, or dissemination system compares newly created or discovered documents to a fixed set of queries supplied in advance by users, identifying those that match a given query closely enough to be of possible interest to the users. A news aggregator, for example, might use a routing system to separate the day's news into sections such as "business," "politics," and "lifestyle," or to send headlines of interest to particular subscribers. An e-mail system might use a spam filter to block unwanted messages. As we shall see, these two problems are essentially the same, although differing in application-specific and implementation details.
- Text *clustering* and *categorization* systems group documents according to shared properties. The difference between clustering and categorization stems from the information provided to the system. Categorization systems are provided with *training data* illustrating the various classes. Examples of "business," "politics," and "lifestyle" articles might be provided to a categorization system, which would then sort unlabeled articles into the same categories. A clustering system, in contrast, is not provided with training examples. Instead, it sorts documents into groups based on patterns it discovers itself.

- *Summarization* systems reduce documents to a few key paragraphs, sentences, or phrases describing their content. The snippets of text displayed with Web search results represent one example.
- *Information extraction* systems identify named entities, such as places and dates, and combine this information into structured records that describe relationships between these entities — for example, creating lists of books and their authors from Web data.
- *Topic detection and tracking* systems identify events in streams of news articles and similar information sources, tracking these events as they evolve.
- *Expert search* systems identify members of organizations who are experts in a specified area.
- *Question answering* systems integrate information from multiple sources to provide concise answers to specific questions. They often incorporate and extend other IR technologies, including search, summarization, and information extraction.
- *Multimedia information retrieval* systems extend relevance ranking and other IR techniques to images, video, music, and speech.

Many IR problems overlap with the fields of library and information science, as well as with other major subject areas of computer science such as natural language processing, databases, and machine learning.

Of the topics listed above, techniques for categorization and filtering have the widest applicability, and we provide an introduction to these areas. The scope of this book does not allow us to devote substantial space to the other topics. However, all of them depend upon and extend the basic technologies we cover.

1.2 Information Retrieval Systems

Most IR systems share a basic architecture and organization that is adapted to the requirements of specific applications. Most of the concepts discussed in this book are presented in the context of this architecture. In addition, like any technical field, information retrieval has its own jargon. Words are sometimes used in a narrow technical sense that differs from their ordinary English meanings. In order to avoid confusion and to provide context for the remainder of the book, we briefly outline the fundamental terminology and technology of the subject.

1.2.1 Basic IR System Architecture

Figure 1.1 illustrates the major components in an IR system. Before conducting a search, a user has an *information need*, which underlies and drives the search process. We sometimes refer to this information need as a *topic*, particularly when it is presented in written form as part

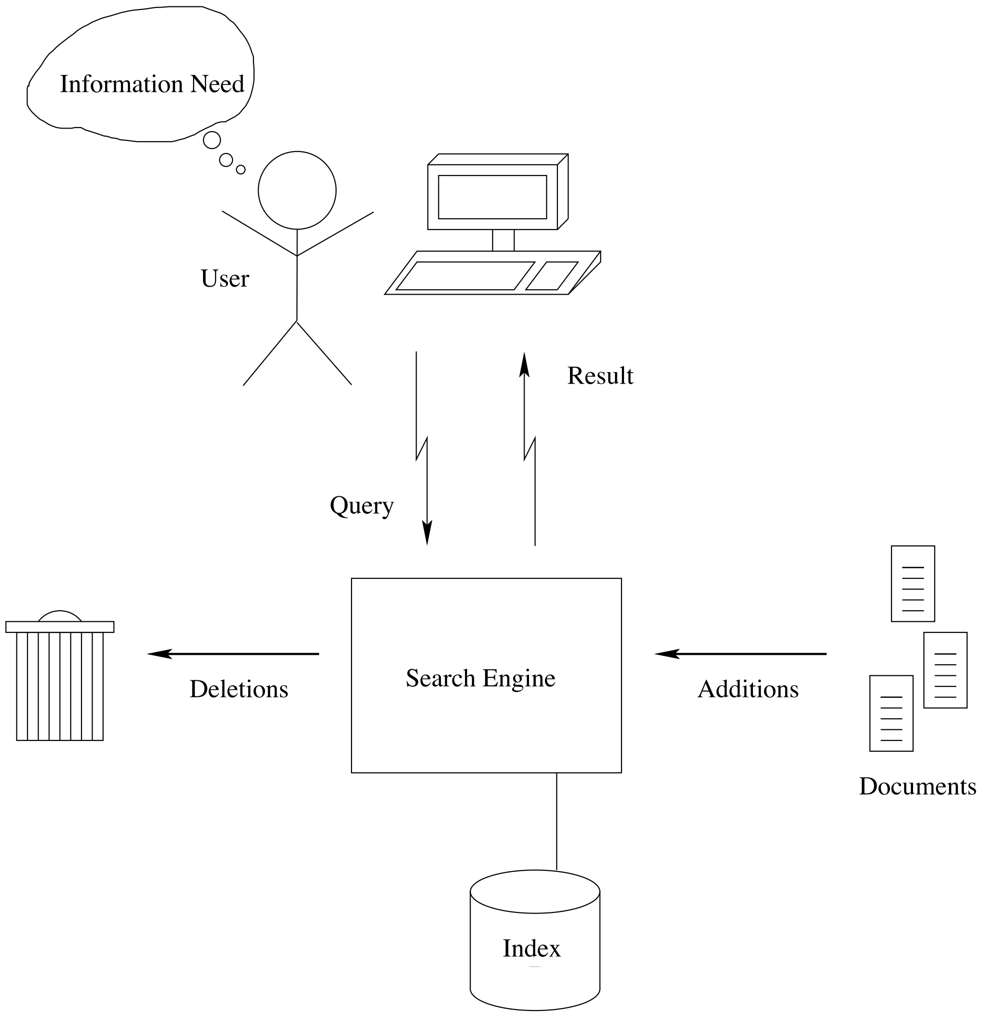


Figure 1.1 Components of an IR system.

of a test collection for IR evaluation. As a result of her information need, the user constructs and issues a *query* to the IR system. Typically, this query consists of a small number of *terms*, with two to three terms being typical for a Web search. We use "term" instead of "word", because a query term may in fact not be a word at all. Depending on the information need, a query term may be a date, a number, a musical note, or a phrase. Wildcard operators and other partial-match operators may also be permitted in query terms. For example, the term "inform*" might match any word starting with that prefix ("inform", "informs", "informal", "informant", "informative", etc.).

Although users typically issue simple keyword queries, IR systems often support a richer query syntax, frequently with complex Boolean and pattern matching operators (Chapter 5). These facilities may be used to limit a search to a particular Web site, to specify constraints on fields such as author and title, or to apply other *filters*, restricting the search to a subset of the collection. A user interface mediates between the user and the IR system, simplifying the query-creation process when these richer query facilities are required.

The user's query is processed by a *search engine*, which may be running on the user's local machine, on a large cluster of machines in a remote geographic location, or anywhere in between. A major task of a search engine is to maintain and manipulate an *inverted index* for a *document collection*. This index forms the principal data structure used by the engine for searching and relevance ranking. As its basic function, an inverted index provides a mapping between terms and the locations in the collection in which they occur. Because the size of an inverted list is on the same order of magnitude as the document collection itself, care must be taken that index access and update operations are performed efficiently.

To support relevance ranking algorithms, the search engine maintains collection statistics associated with the index, such as the number of documents containing each term and the length of each document. In addition, the search engine usually has access to the original content of the documents, in order to report meaningful results back to the user.

Using the inverted index, collection statistics, and other data, the search engine accepts queries from its users, processes these queries, and returns ranked lists of results. To perform relevance ranking, the search engine computes a *score*, sometimes called a *retrieval status value* (RSV), for each document. After sorting documents according to their scores, the result list may be subjected to further processing, such as the removal of duplicate or redundant results. For example, a Web search engine might report only one or two results from a single host or domain, eliminating the others in favor of pages from different sources. The problem of scoring documents with respect to a user's query is one of the most fundamental in the field.

1.2.2 Documents and Update

Throughout this book we use "document" as a generic term to refer to any self-contained unit that can be returned to the user as a search result. In practice, a particular document might be an e-mail message, a Web page, a news article, or even a video. When predefined components of a larger object may be returned as individual search results, such as pages or paragraphs from a book, we refer to these components as *elements*. When arbitrary text passages, video segments, or similar material may be returned from larger objects, we refer to them as *snippets*.

For most applications, the update model is relatively simple. Documents may be added or deleted in their entirety. Once a document has been added to the search engine, its contents are not modified. This update model is sufficient for most IR applications. For example, when a Web crawler discovers that a page has been modified, the update may be viewed as a deletion of the old page and an addition of the new page. Even in the context of file system search, in which individual blocks of a file may be arbitrarily modified, most word processors and other applications dealing with textual data rewrite entire files when a user saves changes. One important exception is e-mail applications, which often append new messages to the ends of mail folders. Because mail folders can be large in size and can grow quickly, it may be important to support append operations.

1.2.3 Performance Evaluation

There are two principal aspects to measuring IR system performance: *efficiency* and *effectiveness*. Efficiency may be measured in terms of time (e.g., seconds per query) and space (e.g., bytes per document). The most visible aspect of efficiency is the *response time* (also known as *latency*) experienced by a user between issuing a query and receiving the results. When many simultaneous users must be supported, the query *throughput*, measured in queries per second, becomes an important factor in system performance. For a general-purpose Web search engine, the required throughput may range well beyond tens of thousands of queries per second. Efficiency may also be considered in terms of storage space, measured by the bytes of disk and memory required to store the index and other data structures. In addition, when thousands of machines are working cooperatively to generate a search result, their power consumption and carbon footprint become important considerations.

Effectiveness is more difficult to measure than efficiency, since it depends entirely on human judgment. The key idea behind measuring effectiveness is the notion of *relevance*: A document is considered relevant to a given query if its contents (completely or partially) satisfy the information need represented by the query. To determine relevance, a human *assessor* reviews a document/topic pair and assigns a relevance value. The relevance value may be *binary* ("relevant" or "not relevant") or *graded* (e.g., "perfect", "excellent", "good", "fair", "acceptable", "not relevant", "harmful").

The fundamental goal of relevance ranking is frequently expressed in terms of the *Probability Ranking Principle* (PRP), which we phrase as follows:

> *If an IR system's response to each query is a ranking of the documents in the collection in order of decreasing probability of relevance, then the overall effectiveness of the system to its users will be maximized.*

This assumption is well established in the field of IR and forms the basis of the standard IR evaluation methodology. Nonetheless, it overlooks important aspects of relevance that must be considered in practice. In particular, the basic notion of relevance may be extended to consider the size and scope of the documents returned. The *specificity* of a document reflects the degree to which its contents are focused on the information need. A highly specific document consists primarily of material related to the information need. In a marginally specific document, most of the material is not related to the topic. The *exhaustivity* of a document reflects the degree to which it covers the information related to the need. A highly exhaustive document provides full coverage; a marginally exhaustive document may cover only limited aspects. Specificity and exhaustivity are independent dimensions. A large document may provide full coverage but contain enough extraneous material that it is only marginally specific.

When relevance is viewed in the context of a complete ranked document list, the notion of *novelty* comes to light. Once the user examines the top-ranked document and learns its relevant content, her information need may shift. If the second document contains little or no novel information, it may not be relevant with respect to this revised information need.

1.3 Working with Electronic Text

Human-language data in the form of electronic text represents the raw material of information retrieval. Building an IR system requires an understanding of both electronic text formats and the characteristics of the text they encode.

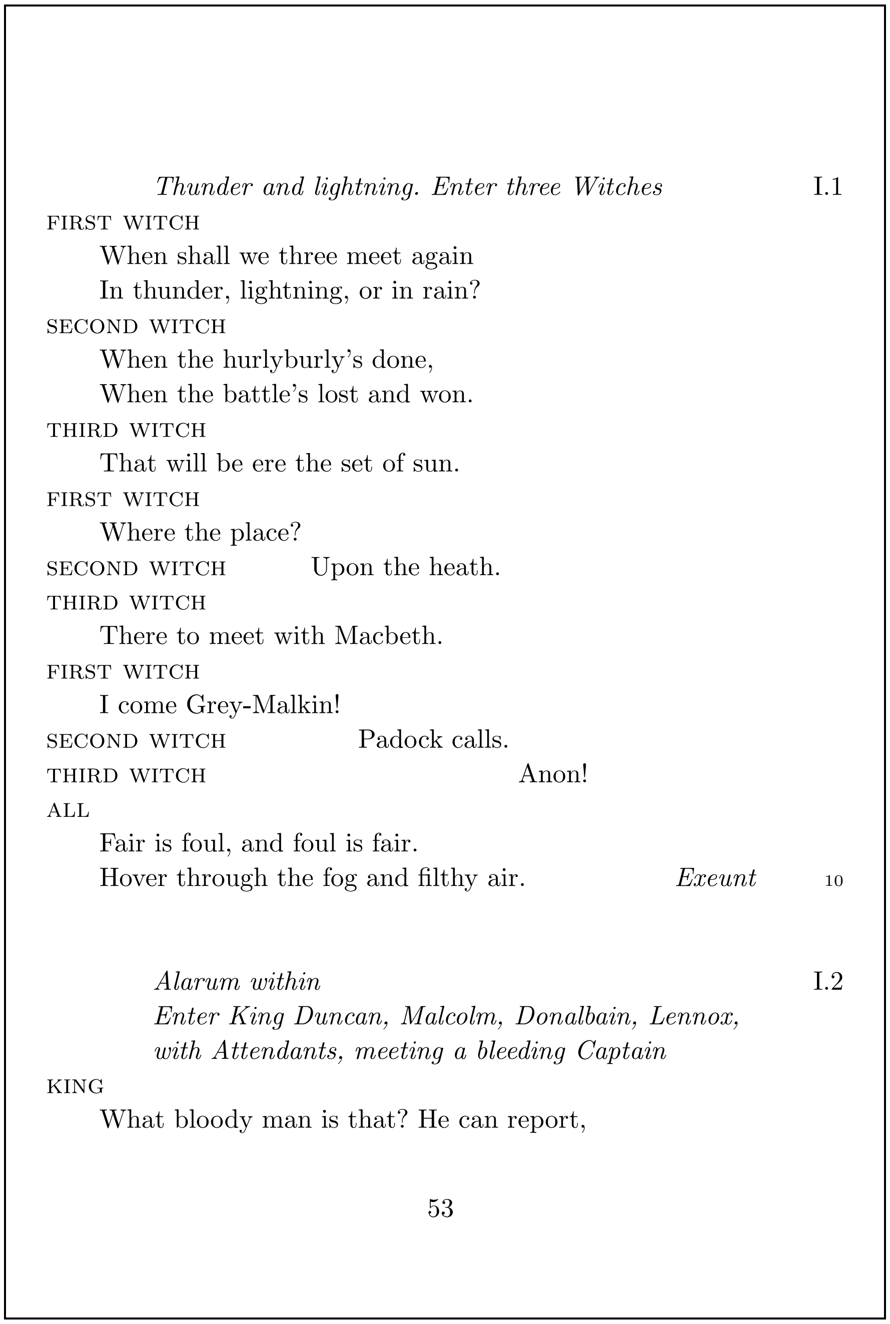
1.3.1 Text Formats

The works of William Shakespeare provide a ready example of a large body of English-language text with many electronic versions freely available on the Web. Shakespeare's canonical works include 37 plays and more than a hundred sonnets and poems. Figure 1.2 shows the start of the first act of one play, *Macbeth*.

This figure presents the play as it might appear on a printed page. From the perspective of an IR system, there are two aspects of this page that must be considered when it is represented in electronic form, and ultimately when it is indexed by the system. The first aspect, the *content* of the page, is the sequence of words in the order they might normally be read: "Thunder and lightning. Enter three Witches First Witch When shall we..." The second aspect is the *structure* of the page: the breaks between lines and pages, the labeling of speeches with speakers, the stage directions, the act and scene numbers, and even the page number.

The content and structure of electronic text may be encoded in myriad document formats supported by various word processing programs and desktop publishing systems. These formats include Microsoft Word, HTML, XML, XHTML, $\LaTeX$, MIF, RTF, PDF, PostScript, SGML, and others. In some environments, such as file system search, e-mail formats and even program source code formats would be added to this list. Although a detailed description of these formats is beyond our scope, a basic understanding of their impact on indexing and retrieval is important.

Two formats are of special interest to us. The first, HTML (HyperText Markup Language), is the fundamental format for Web pages. Of particular note is its inherent support for hyperlinks, which explicitly represent relationships between Web pages and permit these relationships to be exploited by Web search systems. Anchor text often accompanies a hyperlink, partially describing the content of the linked page.



Thunder and lightning. Enter three Witches I.1

FIRST WITCH
When shall we three meet again
In thunder, lightning, or in rain?

SECOND WITCH
When the hurlyburly's done,
When the battle's lost and won.

THIRD WITCH
That will be ere the set of sun.

FIRST WITCH
Where the place?

SECOND WITCH Upon the heath.

THIRD WITCH
There to meet with Macbeth.

FIRST WITCH
I come Grey-Malkin!

SECOND WITCH Padock calls.

THIRD WITCH Anon!

ALL
Fair is foul, and foul is fair.
Hover through the fog and filthy air. *Exeunt*

Alarum within I.2
Enter King Duncan, Malcolm, Donalbain, Lennox,
with Attendants, meeting a bleeding Captain

KING
What bloody man is that? He can report,

53

Figure 1.2 The beginning of the first act of Shakespeare's *Macbeth*.

The second format, XML (eXtensible Markup Language), is not strictly a document format but rather a *metalanguage* for defining document formats. Although we leave a detailed discussion of XML to later in the book (Chapter 16), we can begin using it immediately. XML possesses the convenient attribute that it is possible to construct human-readable encodings that are reasonably self-explanatory. As a result, it serves as a format for examples throughout the remainder of the book, particularly when aspects of document structure are discussed. HTML and XML share a common ancestry in the Standard Generalized Markup Language (SGML) developed in the 1980s and resemble each other in their approach to tagging document structure.

Figure 1.3 presents an XML encoding of the start of *Macbeth* taken from a version of Shakespeare's plays. The encoding was constructed by Jon Bosak, one of the principal creators of the XML standard. Tags of the form `<`*name*`>` represent the start of a structural element and tags of the form `</`*name*`>` represent the end of a structural element. Tags may take other forms, and they may include attributes defining properties of the enclosed text, but these details are left for Chapter 16. For the bulk of the book, we stick to the simple tags illustrated by the figure.

In keeping with the traditional philosophy of XML, this encoding represents only the *logical structure* of the document — speakers, speeches, acts, scenes, and lines. Determination of the *physical structure* — fonts, bolding, layout, and page breaks — is deferred until the page is actually rendered for display, where the details of the target display medium will be known.

Unfortunately, many document formats do not make the same consistent distinction between logical and physical structure. Moreover, some formats impede our ability to determine a document's content or to return anything less than an entire document as the result of a retrieval request. Many of these formats are *binary formats*, so called because they contain internal pointers and other complex organizational structures, and cannot be treated as a stream of characters.

For example, the content of a PostScript document is encoded in a version of the programming language Forth. A PostScript document is essentially a program that is executed to render the document when it is printed or displayed. Although using a programming language to encode a document provides for maximum flexibility, it may also make the content of the document difficult to extract for indexing purposes. The PDF format, PostScript's younger sibling, does not incorporate a complete programming language but otherwise retains much of the flexibility and complexity of the older format. PostScript and PDF were originally developed by Adobe Systems, but both are now open standards and many third-party tools are available to create and manipulate documents in these formats, including open-source tools.

Various conversion utilities are available to extract content from PostScript and PDF documents, and they may be pressed into service for indexing purposes. Each of these utilities implements its own heuristics to analyze the document and guess at its content. Although they often produce excellent output for documents generated by standard tools, they may fail when faced with a document from a more unusual source. At an extreme, a utility might render a document into an internal buffer and apply a pattern-matching algorithm to recognize characters and words.

```
<STAGEDIR>Thunder and lightning. Enter three Witches</STAGEDIR>

<SPEECH>
<SPEAKER>First Witch</SPEAKER>
<LINE>When shall we three meet again</LINE>
<LINE>In thunder, lightning, or in rain?</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Second Witch</SPEAKER>
<LINE>When the hurlyburly's done,</LINE>
<LINE>When the battle's lost and won.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Third Witch</SPEAKER>
<LINE>That will be ere the set of sun.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>First Witch</SPEAKER>
<LINE>Where the place?</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>Second Witch</SPEAKER>
<LINE>Upon the heath.</LINE>
</SPEECH>

<SPEECH>
  <SPEAKER>Third Witch</SPEAKER>
  <LINE>There to meet with Macbeth.</LINE>
</SPEECH>
```

Figure 1.3 An XML encoding of Shakespeare's *Macbeth*.

Moreover, identifying even the simplest logical structure in PostScript or PDF poses a significant challenge. Even a document's title may have to be identified entirely from its font, size, location in the document, and other physical characteristics. In PostScript, extracting individual pages can also cause problems because the code executed to render one page may have an impact on later pages. This aspect of PostScript may limit the ability of an IR system to return a range of pages from a large document, for example, to return a single section from a long technical manual.

Other document formats are proprietary, meaning they are associated with the products of a single software manufacturer. These proprietary formats include Microsoft's "doc" format. Until recently, due to the market dominance of Microsoft Office, this format was widely used for document exchange and collaboration. Although the technical specifications for such proprietary formats are often available, they can be complex and may be modified substantially from version to version, entirely at the manufacturer's discretion. Microsoft and other manufacturers have now shifted toward XML-based formats (such as the OpenDocument format or Microsoft's OOXML), which may ameliorate the complications of indexing.

In practice, HTML may share many of the problems of binary formats. Many HTML pages include scripts in the JavaScript or Flash programming languages. These scripts may rewrite the Web page in its entirety and display arbitrary content on the screen. On pages in which the content is generated by scripts, it may be a practical impossibility for Web crawlers and search engines to extract and index meaningful content.

1.3.2 A Simple Tokenization of English Text

Regardless of a document's format, the construction of an inverted index that can be used to process search queries requires each document to be converted into a sequence of *tokens*. For English-language documents, a token usually corresponds to a sequence of alphanumeric characters (A to Z and 0 to 9), but it may also encode structural information, such as XML tags, or other characteristics of the text. Tokenization is a critical step in the indexing process because it effectively limits the class of queries that may be processed by the system.

As a preliminary step before tokenization, documents in binary formats must be converted to *raw text* — a stream of characters. The process of converting a document to raw text generally discards font information and other lower-level physical formatting but may retain higher-level logical formatting, perhaps by re-inserting appropriate tags into the raw text. This higher-level formatting might include titles, paragraph boundaries, and similar elements. This preliminary step is not required for documents in XML or HTML because these already contain the tokens in the order required for indexing, thus simplifying the processing cost substantially. Essentially, these formats are in situ raw text.

For English-language documents, characters in the raw text may be encoded as seven-bit ASCII values. However, ASCII is not sufficient for documents in other languages. For these languages, other coding schemes must be used, and it may not be possible to encode each character as a single byte. The UTF-8 representation of Unicode provides one popular method for encoding these characters (see Section 3.2). UTF-8 provides a one-to-four byte encoding for the characters in most living languages, as well as for those in many extinct languages, such as Phoenician and Sumerian cuneiform. UTF-8 is backwards compatible with ASCII, so that ASCII text is automatically UTF-8 text.

To tokenize the XML in Figure 1.3, we treat each XML tag and each sequence of consecutive alphanumeric characters as a token. We convert uppercase letters outside tags to lowercase in order to simplify the matching process, meaning that "FIRST", "first" and "First" are treated

...

745396	745397	745398	745399	745400
`<STAGEDIR>`	`thunder`	`and`	`lightning`	`enter`
745401	745402	745403	745404	745405
`three`	`witches`	`</STAGEDIR>`	`<SPEECH>`	`<SPEAKER>`
745406	745407	745408	745409	745410
`first`	`witch`	`</SPEAKER>`	`<LINE>`	`when`
745411	745412	745413	745414	745415
`shall`	`we`	`three`	`meet`	`again`
745416	745417	745418	745419	745420
`</LINE>`	`<LINE>`	`in`	`thunder`	`lightning`
745421	745422	745423	745424	745425
`or`	`in`	`rain`	`</LINE>`	`</SPEECH>`
745426	745427	745428	745429	745430
`<SPEECH>`	`<SPEAKER>`	`second`	`witch`	`</SPEAKER>`

...

Figure 1.4 A tokenization of Shakespeare's *Macbeth*.

as equivalent. The result of our tokenization is shown in Figure 1.4. Each token is accompanied by an integer that indicates its position in a collection of Shakespeare's 37 plays, starting with position 1 at the beginning of *Antony and Cleopatra* and finishing with position 1,271,504 at the end of *The Winter's Tale*. This simple approach to tokenization is sufficient for our purposes through the remainder of Chapters 1 and 2, and it is assumed where necessary. We will reexamine tokenization for English and other languages in Chapter 3.

The set of distinct tokens, or *symbols*, in a text collection is called the *vocabulary*, denoted as $\mathcal{V}$. Our collection of Shakespeare's plays has $|\mathcal{V}| = 22{,}987$ symbols in its vocabulary.

$$\mathcal{V} \;=\; \{\texttt{a}, \texttt{aaron}, \texttt{abaissiez}, ..., \texttt{zounds}, \texttt{zwaggered}, ..., \texttt{<PLAY>}, ..., \texttt{<SPEAKER>}, ..., \texttt{</PLAY>}, ...\}$$

Table 1.1 The twenty most frequent terms in Bosak's XML version of Shakespeare.

Rank	Frequency	Token	Rank	Frequency	Token
1	107,833	`<LINE>`	11	17,523	`of`
2	107,833	`</LINE>`	12	14,914	`a`
3	31,081	`<SPEAKER>`	13	14,088	`you`
4	31,081	`</SPEAKER>`	14	12,287	`my`
5	31,028	`<SPEECH>`	15	11,192	`that`
6	31,028	`</SPEECH>`	16	11,106	`in`
7	28,317	`the`	17	9,344	`is`
8	26,022	`and`	18	8,506	`not`
9	22,639	`i`	19	7,799	`it`
10	19,898	`to`	20	7,753	`me`

For Shakespeare, the vocabulary includes 22,943 words and 44 tags, in which we consider any string of alphanumeric characters to be a word. In this book, we usually refer to symbols in the vocabulary as "terms" because they form the basis for matching against the terms in a query. In addition, we often refer to a token as an "occurrence" of a term. Although this usage helps reinforce the link between the tokens in a document and the terms in a query, it may obscure the crucial difference between a symbol and a token. A symbol is an abstraction; a token is an instance of that abstraction. In philosophy, this difference is called the "type-token distinction." In object-oriented programming, it is the difference between a class and an object.

1.3.3 Term Distributions

Table 1.1 lists the twenty most frequent terms in the XML collection of Shakespeare's plays. Of these terms, the top six are tags for lines, speakers, and speeches. As is normally true for English text, "the" is the most frequent word, followed by various pronouns, prepositions, and other function words. More than one-third (8,336) of the terms, such as "abaissiez" and "zwaggered", appear only once.

The frequency of the tags is determined by the structural constraints of the collection. Each start tag <*name*> has a corresponding end tag </*name*>. Each play has exactly one title. Each speech has at least one speaker, but a few speeches have more than one speaker, when a group of characters speak in unison. On average, a new line starts every eight or nine words.

While the type and relative frequency of tags will be different in other collections, the relative frequency of words in English text usually follows a consistent pattern. Figure 1.5 plots the frequency of the terms in Shakespeare in rank order, with tags omitted. Logarithmic scales are used for both the x and the y axes. The points fall roughly along a line with slope -1, although both the most frequent and the least frequent terms fall below the line. On this plot, the point

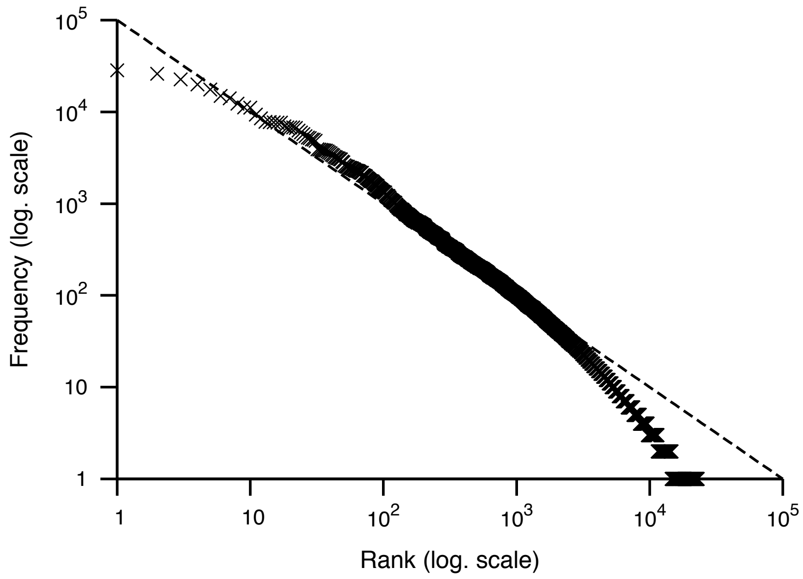


Figure 1.5 Frequency of words in the Shakespeare collection, by rank order. The dashed line corresponds to Zipf's law with $\alpha = 1$.

corresponding to "the" appears in the upper left. The point corresponding to "zwaggered" appears in the lower right, together with the other words that appear exactly once.

The relationship between frequency and rank represented by this line is known as *Zipf's law*, after the linguist George Zipf, who developed it in the 1930s and used it to model the relative frequencies of data across a number of areas in the social sciences (Zipf, 1949). Mathematically, the relationship may be expressed as

$$\log(\text{frequency}) \;=\; C - \alpha \cdot \log(\text{rank})\,, \tag{1.1}$$

or equivalently as

$$\mathcal{F}_i \sim \frac{1}{i^\alpha}\,, \tag{1.2}$$

where $\mathcal{F}_i$ is the frequency of the ith most frequent term. For English text, the value of α may vary, but it is usually close to 1. Zipf's law is known to apply to the relative word frequencies in other natural languages, as well as to other types of data. In Chapter 4, it motivates the use of a certain data structure for indexing natural language text. In Chapter 15, we apply Zipf's law to model the relative frequency of search engine queries.

1.3.4 Language Modeling

There are 912,052 tokens in Bosak's XML version of Shakespeare's plays, excluding tags (but including some front matter that does not appear in the original versions). If we pick a token uniformly at random from these plays, the probability of picking "the" is $28,317/912,052 \approx 3.1\%$, whereas the probability of picking "zwaggered" is only $1/912,052 \approx 0.00011\%$. Now, imagine that a previously unknown Shakespearean play is discovered. Can we predict anything about the content of this play from what we know of the existing plays? In making these predictions, we redefine the vocabulary $\mathcal{V}$ to exclude tags. An unknown Shakespearean play is unlikely to be encoded in XML (at least when it is first discovered).

Predictions concerning the content of unseen text may be made by way of a special kind of probability distribution known as a *language model*. The simplest language model is a fixed probability distribution $\mathcal{M}(\sigma)$ over the symbols in the vocabulary:

$$\sum_{\sigma \in \mathcal{V}} \mathcal{M}(\sigma) \;=\; 1. \tag{1.3}$$

A language model is often based on an existing text. For example, we might define

$$\mathcal{M}(\sigma) \;=\; \frac{\text{frequency}(\sigma)}{\sum_{\sigma' \in \mathcal{V}} \text{frequency}(\sigma')} \tag{1.4}$$

where frequency(σ) represents the number of occurrences of the term σ in Shakespeare's plays. Thus, we have $\mathcal{M}(\text{"the"}) \approx 3.1\%$ and $\mathcal{M}(\text{"zwaggered"}) \approx 0.00011\%$.

If we pick a token uniformly at random from Shakespeare's known plays, the probably of picking the term σ is $\mathcal{M}(\sigma)$. Based on this knowledge of Shakespeare, if we pick a token uniformly at random from the previously unseen play, we might assume that the probably of picking σ is also $\mathcal{M}(\sigma)$. If we start reading the unseen play, we can use the language model to make a prediction about the next term in the text, with $\mathcal{M}(\sigma)$ giving the probability that the next term is σ. For this simple language model, we consider each term in isolation. The language model makes the same independent prediction concerning the first term, and the next, and the next, and so on. Based on this language model, the probability that the next six terms are "to be or not to be" is:

$$2.18\% \times 0.76\% \times 0.27\% \times 0.93\% \times 2.18\% \times 0.76\% \;=\; 0.000000000069\%.$$

Equation 1.4 is called the *maximum likelihood estimate* (MLE) for this simple type of language model. In general, maximum likelihood is the standard method for estimating unknown parameters of a probability distribution, given a set of data. Here, we have a parameter corresponding to each term — the probability that the term appears next in the unseen text. Roughly speaking, the maximum likelihood estimation method chooses values for the parameters that make the data set most likely. In this case, we are treating the plays of Shakespeare as providing the necessary data set. Equation 1.4 is the assignment of probabilities that is most likely to produce

Shakespeare. If we assume that the unseen text is similar to the existing text, the maximum likelihood model provides a good starting point for predicting its content.

Language models can be used to quantify how close a new text fragment is to an existing corpus. Suppose we have a language model representing the works of Shakespeare and another representing the works of the English playwright John Webster. A new, previously unknown, play is found; experts debate about who might be its author. Assuming that our two language models capture salient characteristics of the two writers, such as their preferred vocabulary, we may apply the language models to compute the probability of the new text according to each. The language model that assigns the higher probability to the new text may indicate its author.

However, a language model need not use maximum likelihood estimates. Any probability distribution over a vocabulary of terms may be treated as a language model. For example, consider the probability distribution

$$\mathcal{M}(\text{“to”}) = 0.40 \quad \mathcal{M}(\text{“be”}) = 0.30 \quad \mathcal{M}(\text{“or”}) = 0.20 \quad \mathcal{M}(\text{“not”}) = 0.10.$$

Based on this distribution, the probability that the next six words are “to be or not to be” is

$$0.40 \times 0.30 \times 0.20 \times 0.10 \times 0.40 \times 0.30 = 0.029\%.$$

Of course, based on this model, the probability that the next six words are “the lady doth protest too much” is zero.

In practice, unseen text might include unseen terms. To accommodate these unseen terms, the vocabulary might be extended by adding an UNKNOWN symbol to represent these “out-of-vocabulary” terms.

$$\mathcal{V}' \;=\; \mathcal{V} \,\cup\, \{\text{UNKNOWN}\} \tag{1.5}$$

The corresponding extended language model $\mathcal{M}'(\sigma)$ would then assign a positive probability to this UNKNOWN term

$$\mathcal{M}'(\text{UNKNOWN}) = \beta, \tag{1.6}$$

where $0 \le \beta \le 1$. The value β represents the probability that the next term does not appear in the existing collection from which the model $\mathcal{M}$ was estimated. For other terms, we might then define

$$\mathcal{M}'(\sigma) \;=\; \mathcal{M}(\sigma) \cdot (1 - \beta)\,, \tag{1.7}$$

where $\mathcal{M}(\sigma)$ is the maximum likelihood language model. The choice of a value for β might be based on characteristics of the existing text. For example, we might guess that β should be roughly half of the probability of a unique term in the existing text:

$$\beta \;=\; 0.5 \cdot \frac{1}{\sum_{\sigma' \in \mathcal{V}} \text{frequency}(\sigma')}\,. \tag{1.8}$$

Fortunately, out-of-vocabulary terms are not usually a problem in IR systems because the complete vocabulary of the collection may be determined during the indexing process.

Index and text compression (Chapter 6) represents another important area for the application of language models. When used in compression algorithms, the terms in the vocabulary are usually individual characters or bits rather than entire words. Language modeling approaches that were invented in the context of compression are usually called *compression models*, and this terminology is common in the literature. However, compression models are just specialized language models. In Chapter 10, compression models are applied to the problem of detecting e-mail spam and other filtering problems.

Language models may be used to generate text, as well as to predict unseen text. For example, we may produce "random Shakespeare" by randomly generating a sequence of terms based on the probability distribution $\mathcal{M}(\sigma)$:

> *strong die hat circumstance in one eyes odious love to our the wrong wailful would all sir you to babies a in in of er immediate slew let on see worthy all timon nourish both my how antonio silius my live words our my ford scape*

Higher-order models

The random text shown above does not read much like Shakespeare, or even like English text written by lesser authors. Because each term is generated in isolation, the probability of generating the word "the" immediately after generating "our" remains at 3.1%. In real English text, the possessive adjective "our" is almost always followed by a common noun. Even though the phrase "our the" consists of two frequently occurring words, it rarely occurs in English, and never in Shakespeare.

Higher-order language models allow us to take this context into account. A *first-order* language model consists of conditional probabilities that depend on the previous symbol. For example:

$$\mathcal{M}_1(\sigma_2 \,|\, \sigma_1) \;=\; \frac{\text{frequency}(\sigma_1\sigma_2)}{\sum_{\sigma' \in \mathcal{V}} \text{frequency}(\sigma_1\sigma')}\,. \tag{1.9}$$

A first-order language model for terms is equivalent to the zero-order model for term *bigrams* estimated using the same technique (e.g., MLE):

$$\mathcal{M}_1(\sigma_2 \,|\, \sigma_1) \;=\; \frac{\mathcal{M}_0(\sigma_1\sigma_2)}{\sum_{\sigma' \in \mathcal{V}} \mathcal{M}_0(\sigma_1\sigma')}\,. \tag{1.10}$$

More generally, every nth-order language model may be expressed in terms of a zero-order $(n+1)$-gram model:

$$\mathcal{M}_n(\sigma_{n+1} \,|\, \sigma_1 \ldots \sigma_n) \;=\; \frac{\mathcal{M}_0(\sigma_1 \ldots \sigma_{n+1})}{\sum_{\sigma' \in \mathcal{V}} \mathcal{M}_0(\sigma_1 \ldots \sigma_n\, \sigma')}\,. \tag{1.11}$$

As an example, consider the phrase "first witch" in Shakespeare's plays. This phrase appears a total of 23 times, whereas the term "first" appears 1,349 times. The maximum likelihood bigram model thus assigns the following probability:

$$\mathcal{M}_0(\text{“first witch”}) \;=\; \frac{23}{912{,}051} \;\approx\; 0.0025\%$$

(note that the denominator is 912,051, not 912,052, because the total number of bigrams is one less than the total number of tokens). The corresponding probability in the first-order model is

$$\mathcal{M}_1(\text{“witch”} \,|\, \text{“first”}) \;=\; \frac{23}{1349} \;\approx\; 1.7\%.$$

Using Equations 1.9 and 1.10, and ignoring the difference between the number of tokens and the number of bigrams, the maximum likelihood estimate for “our the” is

$$\mathcal{M}_0(\text{“our the”}) \;=\; \mathcal{M}_0(\text{“our”}) \cdot \mathcal{M}_1(\text{“the”} \,|\, \text{“our”}) \;=\; 0\%,$$

which is what we would expect, because the phrase never appears in the text. Unfortunately, the model also assigns a zero probability to more reasonable bigrams that do not appear in Shakespeare, such as “fourth witch”. Because “fourth” appears 55 times and “witch” appears 92 times, we can easily imagine that an unknown play might contain this bigram. Moreover, we should perhaps assign a small positive probability to bigrams such as “our the” to accommodate unusual usage, including archaic spellings and accented speech. For example, *The Merry Wives of Windsor* contains the apparently meaningless bigram “a the” in a speech by the French physician Doctor Caius: “If dere be one or two, I shall make-a the turd.” Once more context is seen, the meaning becomes clearer.

Smoothing

One solution to this problem is to *smooth* the first-order model $\mathcal{M}_1$ with the corresponding zero-order model $\mathcal{M}_0$. Our smoothed model $\mathcal{M}_1'$ is then a linear combination of $\mathcal{M}_0$ and $\mathcal{M}_1$:

$$\mathcal{M}_1'(\sigma_2 \,|\, \sigma_1) \;=\; \gamma \cdot \mathcal{M}_1(\sigma_2 \,|\, \sigma_1) + (1-\gamma) \cdot \mathcal{M}_0(\sigma_2) \tag{1.12}$$

and equivalently

$$\mathcal{M}_0'(\sigma_1 \sigma_2) \;=\; \gamma \cdot \mathcal{M}_0(\sigma_1 \sigma_2) + (1-\gamma) \cdot \mathcal{M}_0(\sigma_1) \cdot \mathcal{M}_0(\sigma_2), \tag{1.13}$$

where γ in both cases is a smoothing parameter ($0 \leq \gamma \leq 1$). For example, using maximum likelihood estimates and setting $\gamma = 0.5$, we have

$$\begin{aligned} \mathcal{M}_1'(\text{“first witch”}) \;&=\; \gamma \cdot \mathcal{M}_1(\text{“first witch”}) + (1-\gamma) \cdot \mathcal{M}_0(\text{“first”}) \cdot \mathcal{M}_0(\text{“witch”}) \\ &=\; 0.5 \cdot \frac{23}{912,051} + 0.5 \cdot \frac{1,349}{912,052} \cdot \frac{92}{912,052} \;\approx\; 0.0013\% \end{aligned}$$

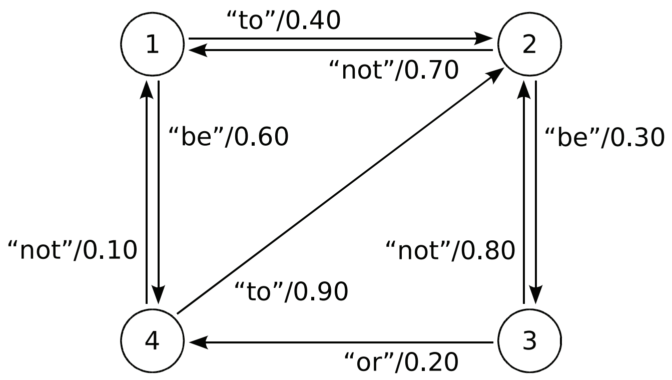


Figure 1.6 A Markov model.

and

$$\begin{aligned}\mathcal{M}'_1(\text{“fourth witch”}) &= \gamma \cdot \mathcal{M}_1(\text{“fourth witch”}) + (1-\gamma) \cdot \mathcal{M}_0(\text{“fourth”}) \cdot \mathcal{M}_0(\text{“witch”}) \\ &= 0.5 \cdot \frac{0}{912{,}051} + 0.5 \cdot \frac{55}{912{,}052} \cdot \frac{92}{912{,}052} \approx 0.00000030\%.\end{aligned}$$

First-order models can be smoothed using zero-order models; second-order models can be smoothed using first-order models; and so forth. Obviously, for zero-order models, this approach does not work. However, we can follow the same approach that we used to address out-of-vocabulary terms (Equation 1.5). Alternatively (and more commonly), the zero-order model $\mathcal{M}_{S,0}$ for a small collection S can be smoothed using another zero-order model, built from a larger collection L:

$$\mathcal{M}'_{S,0} = \gamma \cdot \mathcal{M}_{S,0} + (1-\gamma) \cdot \mathcal{M}_{L,0}\,, \tag{1.14}$$

where L could be an arbitrary (but large) corpus of English text.

Markov models

Figure 1.6 illustrates a *Markov model*, another important method for representing term distributions. Markov models are essentially finite-state automata augmented with transition probabilities. When used to express a language model, each transition is labeled with a term, in addition to the probability. Following a transition corresponds to predicting or generating that term. Starting in state 1, we may generate the string “to be or not to be” by following the state sequence $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 3$, with the associated probability

$$0.40 \times 0.30 \times 0.20 \times 0.10 \times 0.40 \times 0.30 = 0.029\%.$$

Missing transitions (e.g., between state 1 and state 4) are equivalent to transitions with zero probability. Because these transitions will never be taken, it is pointless to associate a term with them. For simplicity, we also assume that there is at most one transition between any two states. We do not sacrifice anything by making this simplification. For any Markov model with multiple transitions between the same pair of states, there is a corresponding model without them (see Exercise 1.7).

The probability predicted by a Markov model depends on the starting state. Starting in state 3, the probability of generating "to be or not to be" is

$$0.90 \times 0.30 \times 0.20 \times 0.10 \times 0.40 \times 0.30 \;=\; 0.065\%.$$

Markov models form the basis for a text compression model introduced for filtering in Chapter 10.

We may represent a Markov model with n states as an $n \times n$ *transition matrix* M, where $M[i][j]$ gives the probability of a transition from state i to state j. The transition matrix corresponding to the Markov model in Figure 1.6 is

$$M \;=\; \begin{pmatrix} 0.00 & 0.40 & 0.00 & 0.60 \\ 0.70 & 0.00 & 0.30 & 0.00 \\ 0.00 & 0.80 & 0.00 & 0.20 \\ 0.10 & 0.90 & 0.00 & 0.00 \end{pmatrix}. \tag{1.15}$$

Note that all of the values in the matrix fall in the range $[0, 1]$ and that each row of M sums to 1. An $n \times n$ matrix with these properties is known as a *stochastic matrix*.

Given a transition matrix, we can compute the outcome of a transition by multiplying the transition matrix by a *state vector* representing the current state. For example, we can represent an initial start state of 1 by the vector (1 0 0 0). After one step, we have

$$\begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0.00 & 0.40 & 0.00 & 0.60 \\ 0.70 & 0.00 & 0.30 & 0.00 \\ 0.00 & 0.80 & 0.00 & 0.20 \\ 0.10 & 0.90 & 0.00 & 0.00 \end{pmatrix} \;=\; \begin{pmatrix} 0.00 & 0.40 & 0.00 & 0.60 \end{pmatrix}.$$

That is, after one step, the probability of being in state 2 is 0.40 and the probability of being in state 4 is 0.60, as expected. Multiplying again, we have

$$\begin{pmatrix} 0.00 & 0.40 & 0.00 & 0.60 \end{pmatrix} \begin{pmatrix} 0.00 & 0.40 & 0.00 & 0.60 \\ 0.70 & 0.00 & 0.30 & 0.00 \\ 0.00 & 0.80 & 0.00 & 0.20 \\ 0.10 & 0.90 & 0.00 & 0.00 \end{pmatrix} \;=\; \begin{pmatrix} 0.34 & 0.54 & 0.12 & 0.00 \end{pmatrix}.$$

This result tells us, for example, that the probability of being in state 2 after two steps is 0.54. In general, the state vector may be any n-dimensional vector whose elements sum to 1. Multiplying the state vector by the transition matrix k times gives the probability of being in each state after k steps.

A stochastic matrix together with an initial state vector is known as a *Markov chain*. Markov chains are used in the presentation of Web link analysis algorithms in Chapter 15. Markov chains — and by extension Markov models — are named after the Russian statistician Andrey Markov (1856–1922), who stated and proved many of their properties.

1.4 Test Collections

Although *Macbeth* and Shakespeare's other plays provide an excellent source of examples for simple IR concepts, researchers have developed more substantial test collections for evaluation purposes. Many of these collections have been created as part of TREC[2] (Text REtrieval Conference), a series of experimental evaluation efforts conducted annually since 1991 by the U.S. National Institute of Standards and Technology (NIST). TREC provides a forum for researchers to test their IR systems on a broad range of problems. For example, more than 100 groups from universities, industry, and government participated in TREC 2007.

In a typical year, TREC experiments are structured into six or seven *tracks*, each devoted to a different area of information retrieval. In recent years, TREC has included tracks devoted to enterprise search, genomic information retrieval, legal discovery, e-mail spam filtering, and blog search. Each track is divided into several tasks that test different aspects of that area. For example, at TREC 2007, the enterprise search track included an e-mail discussion search task and a task to identify experts on given topics. A track typically operates for three or more years before being retired.

TREC provides at least two important benefits to the IR community. First, it focuses researchers on common problems, using common data, thus providing a forum for them to present and discuss their work and facilitating direct inter-system comparisons. What works and what does not work can be determined quickly. As a result, considerable progress and substantial performance improvements are often seen immediately following the introduction of a new track into TREC. As a second benefit, TREC aims to create *reusable* test collections that can be used by participating groups to validate further improvements and by non-participating groups to evaluate their own work. In addition, since its inception, TREC has formed the inspiration for a number of similar experimental efforts throughout the world. These include the European INEX effort for XML retrieval, the CLEF effort for multi-lingual information retrieval, the Japanese NTCIR effort for Asian language information retrieval, and the Indian FIRE effort.

[2] `trec.nist.gov`

```
<DOC>
<DOCNO> LA051990-0141 </DOCNO>

<HEADLINE> COUNCIL VOTES TO EDUCATE DOG OWNERS </HEADLINE>

<P>
The City Council stepped carefully around enforcement of a dog-curbing
ordinance this week, vetoing the use of police to enforce the law.
</P>

...

</DOC>
```

Figure 1.7 Example TREC document (LA051990-0141) from disk 5 of the TREC CDs.

1.4.1 TREC Tasks

Basic search tasks — in which systems return a ranked list from a static set of documents using previously unseen topics — are referred to as "adhoc" tasks in TREC jargon (often written as one word). Along with a set of documents, a test collection for an adhoc task includes sets of topics, from which queries may be created, and sets of relevance judgments (known as "qrel files" or just "qrels"), indicating documents that are relevant or not relevant to each topic. Over the history of TREC, adhoc tasks have been a part of tracks with a number of different research themes, such as Web retrieval or genomic IR. Despite differences in themes, the organization and operation of an adhoc task is basically the same across the tracks and is essentially unchanged since the earliest days of TREC.

Document sets for older TREC adhoc tasks (before 2000) were often taken from a set of 1.6 million documents distributed to TREC participants on five CDs. These disks contain selections of newspaper and newswire articles from publications such as the *Wall Street Journal* and the *LA Times*, and documents published by the U.S. federal government, such as the *Federal Register* and the *Congressional Record*. Most of these documents are written and edited by professionals reporting factual information or describing events.

Figure 1.7 shows a short excerpt from a document on disk 5 of the TREC CDs. The document appeared as a news article in the *LA Times* on May 19, 1990. For the purposes of TREC experiments, it is marked up in the style of XML. Although the details of the tagging schemes vary across the TREC collections, all TREC documents adhere to the same tagging convention for identifying document boundaries and document identifiers. Every TREC document is surrounded by `<DOC>`...`</DOC>` tags; `<DOCNO>`...`</DOCNO>` tags indicate its unique identifier. This identifier is used in qrels files when recording judgments for the document. This convention simplifies the indexing process and allows collections to be combined easily. Many research IR systems provide out-of-the-box facilities for working with documents that follow this convention.

```
<top>

<num> Number: 426
<title> law enforcement, dogs

<desc> Description:
Provide information on the use of dogs worldwide for
law enforcement purposes.

<narr> Narrative:
Relevant items include specific information on the
use of dogs during an operation. Training of dogs
and their handlers are also relevant.

</top>
```

Figure 1.8 TREC topic 426.

Document sets for newer TREC adhoc tasks are often taken from the Web. Until 2009, the largest of these was the 426GB GOV2 collection, which contains 25 million Web pages crawled from sites in the U.S. government's `gov` domain in early 2004. This crawl attempted to reach as many pages as possible within the `gov` domain, and it may be viewed as a reasonable snapshot of that domain within that time period. GOV2 contains documents in a wide variety of formats and lengths, ranging from lengthy technical reports in PDF to pages of nothing but links in HTML. GOV2 formed the document set for the Terabyte Track from TREC 2004 until the track was discontinued at the end of 2006. It also formed the collection for the Million Query Track at TREC 2007 and 2008.

Although the GOV2 collection is substantially larger than any previous TREC collection, it is still orders of magnitude smaller than the collections managed by commercial Web search engines. TREC 2009 saw the introduction of a billion-page Web collection, known as the ClueWeb09 collection, providing an opportunity for IR researchers to work on a scale comparable to commercial Web search.[3]

For each year that a track operates an adhoc task, NIST typically creates 50 new topics. Participants are required to freeze development of their systems before downloading these topics. After downloading the topics, participants create queries from them, run these queries against the document set, and return ranked lists to NIST for evaluation.

A typical TREC adhoc topic, created for TREC 1999, is shown in Figure 1.8. Like most TREC topics, it is structured into three parts, describing the underlying information need in several forms. The *title* field is designed to be treated as a keyword query, similar to a query that might be entered into a search engine. The *description* field provides a longer statement of the topic requirements, in the form of a complete sentence or question. It, too, may be used

[3] `boston.lti.cs.cmu.edu/Data/clueweb09`

Table 1.2 Summary of the test collections used for many of the experiments described in this book.

Document Set	Number of Docs	Size (GB)	Year	Topics
TREC45	0.5 million	2	1998	351–400
			1999	401–450
GOV2	25.2 million	426	2004	701–750
			2005	751–800

as a query, particularly by research systems that apply natural language processing techniques as part of retrieval. The *narrative*, which may be a full paragraph in length, supplements the other two fields and provides additional information required to specify the nature of a relevant document. The narrative field is primarily used by human assessors, to help determine if a retrieved document is relevant or not.

Most retrieval experiments in this book report results over four TREC test collections based on two document sets, a small one and a larger one. The small collection consists of the documents from disks 4 and 5 of the TREC CDs described above, excluding the documents from the *Congressional Record*. It includes documents from the *Financial Times*, the U.S. *Federal Register*, the U.S. Foreign Broadcast Information Service, and the *LA Times*. This document set, which we refer to as *TREC45*, was used for the main adhoc task at TREC 1998 and 1999.

In both 1998 and 1999, NIST created 50 topics with associated relevance judgments over this document set. The 1998 topics are numbered 350–400; the 1999 topics are numbered 401–450. Thus, we have two test collections over the TREC45 document set, which we refer to as *TREC45 1998* and *TREC45 1999*. Although there are minor differences between our experimental procedure and that used in the corresponding TREC experiments (which we will ignore), our experimental results reported over these collections may reasonably be compared with the published results at TREC 1998 and 1999.

The larger one of the two document sets used in our experiments is the GOV2 corpus mentioned previously. We take this set together with topics and judgments from the TREC Terabyte track in 2004 (topics 701–750) and 2005 (751–800) to form the GOV2 2004 and GOV2 2005 collections. Experimental results reported over these collections may reasonably be compared with the published results for the Terabyte track of TREC 2004 and 2005.

Table 1.2 summarizes our four test collections. The TREC45 collection may be obtained from the NIST Standard Reference Data Products Web page as Special Databases 22 and 23.[4] The GOV2 collection is distributed by the University of Glasgow.[5] Topics and qrels for these collections may be obtained from the TREC data archive.[6]

[4] `www.nist.gov/srd`

[5] `ir.dcs.gla.ac.uk/test_collections`

[6] `trec.nist.gov`

1.5 Open-Source IR Systems

There exists a wide variety of open-source information retrieval systems that you may use for exercises in this book and to start conducting your own information retrieval experiments. As always, a (non-exhaustive) list of open-source IR systems can be found in Wikipedia.[7]

Since this list of available systems is so long, we do not even try to cover it in detail. Instead, we restrict ourselves to a very brief overview of three particular systems that were chosen because of their popularity, their influence on IR research, or their intimate relationship with the contents of this book. All three systems are available for download from the Web and may be used free of charge, according to their respective licenses.

1.5.1 Lucene

Lucene is an indexing and search system implemented in Java, with ports to other programming languages. The project was started by Doug Cutting in 1997. Since then, it has grown from a single-developer effort to a global project involving hundreds of developers in various countries. It is currently hosted by the Apache Foundation.[8] Lucene is by far the most successful open-source search engine. Its largest installation is quite likely Wikipedia: All queries entered into Wikipedia's search form are handled by Lucene. A list of other projects relying on its indexing and search capabilities can be found on Lucene's "PoweredBy" page.[9]

Known for its modularity and extensibility, Lucene allows developers to define their own indexing and retrieval rules and formulae. Under the hood, Lucene's retrieval framework is based on the concept of *fields*: Every document is a collection of fields, such as its title, body, URL, and so forth. This makes it easy to specify structured search requests and to give different weights to different parts of a document.

Due to its great popularity, there is a wide variety of books and tutorials that help you get started with Lucene quickly. Try the query "lucene tutorial" in your favorite Web search engine.

1.5.2 Indri

Indri[10] is an academic information retrieval system written in C++. It is developed by researchers at the University of Massachusetts and is part of the Lemur project,[11] a joint effort of the University of Massachusetts and Carnegie Mellon University.

[7] `en.wikipedia.org/wiki/List_of_search_engines`
[8] `lucene.apache.org`
[9] `wiki.apache.org/lucene-java/PoweredBy`
[10] `www.lemurproject.org/indri/`
[11] `www.lemurproject.org`

Indri is well known for its high retrieval effectiveness and is frequently found among the top-scoring search engines at TREC. Its retrieval model is a combination of the language modeling approaches discussed in Chapter 9. Like Lucene, Indri can handle multiple fields per document, such as title, body, and anchor text, which is important in the context of Web search (Chapter 15). It supports automatic query expansion by means of pseudo-relevance feedback, a technique that adds related terms to an initial search query, based on the contents of an initial set of search results (see Section 8.6). It also supports query-independent document scoring that may, for instance, be used to prefer more recent documents over less recent ones when ranking the search results (see Sections 9.1 and 15.3).

1.5.3 Wumpus

Wumpus[12] is an academic search engine written in C++ and developed at the University of Waterloo. Unlike most other search engines, Wumpus has no built-in notion of "documents" and does not know about the beginning and the end of each document when it builds the index. Instead, every part of the text collection may represent a potential unit for retrieval, depending on the structural search constraints specified in the query. This makes the system particularly attractive for search tasks in which the ideal search result may not always be a whole document, but may be a section, a paragraph, or a sequence of paragraphs within a document.

Wumpus supports a variety of different retrieval methods, including the proximity ranking function from Chapter 2, the BM25 algorithm from Chapter 8, and the language modeling and divergence from randomness approaches discussed in Chapter 9. In addition, it is able to carry out real-time index updates (i.e., adding/removing files to/from the index) and provides support for multi-user security restrictions that are useful if the system has more than one user, and each user is allowed to search only parts of the index.

Unless explicitly stated otherwise, all performance figures presented in this book were obtained using Wumpus.

1.6 Further Reading

This is not the first book on the topic of information retrieval. Of the older books providing a general introduction to IR, several should be mentioned. The classic books by Salton (1968) and van Rijsbergen (1979) continue to provide insights into the foundations of the field. The treatment of core topics given by Grossman and Frieder (2004) remains relevant. Witten et al. (1999) provide background information on many related topics that we do not cover in this book, including text and image compression.

[12] `www.wumpus-search.org`

Several good introductory texts have appeared in recent years. The textbook by Croft et al. (2010) is intended to give an undergraduate-level introduction to the area. Baeza-Yates and Ribeiro-Neto (2010) provide a broad survey of the field, with experts contributing individual chapters on their areas of expertise. Manning et al. (2008) provide another readable survey.

Survey articles on specific topics appear regularly as part of the journal series *Foundations and Trends in Information Retrieval.* The *Encyclopedia of Database Systems* (Özsu and Liu, 2009) contains many introductory articles on topics related to information retrieval. Hearst (2009) provides an introduction to user interface design for information retrieval applications. The field of natural language processing, particularly the sub-field of statistical natural language processing, is closely related to the field of information retrieval. Manning and Schütze (1999) provide a thorough introduction to that area.

The premier research conference in the area is the *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (SIGIR), now well into its fourth decade. Other leading research conferences and workshops include the *ACM Conference on Information and Knowledge Management* (CIKM), the *Joint Conference on Digital Libraries* (JCDL), the *European Conference on Information Retrieval* (ECIR), the *ACM International Conference on Web Search and Data Mining* (WSDM), the conference on *String Processing and Information Retrieval* (SPIRE), and the *Text REtrieval Conference* (TREC). Important IR research also regularly appears in the premier venues of related fields, such as the *World Wide Web Conference* (WWW), the *Annual Conference on Neural Information Processing Systems* (NIPS), the *Conference on Artificial Intelligence* (AAAI), and the *Knowledge Discovery and Data Mining Conference* (KDD). The premier IR journal is *ACM Transactions on Information Systems.* Other leading journals include *Information Retrieval* and the venerable *Information Processing & Management.*

Many books and Web sites devoted to learning and using XML are available. One important resource is the XML home page of the World Wide Web Consortium (W3C),[13] the organization responsible for defining and maintaining the XML technical specification. Along with extensive reference materials, the site includes pointers to introductory tutorials and guides. Jon Bosak's personal Web page[14] contains many articles and other information related to the early development and usage of XML, including Shakespeare's plays.

[13] `www.w3.org/XML/`

[14] `www.ibiblio.org/bosak`

1.7 Exercises

Exercise 1.1 Record the next ten queries you issue to a Web search engine (or check your Web history if your search engine allows it). Note how many are *refinements* of previous queries, adding or removing terms to narrow or broaden the focus of the query. Choose three commercial search engines, including the engine you normally use, and reissue the queries on all three. For each query, examine the top five results from each engine. For each result, rate on it a scale from -10 to +10, where +10 represents a perfect or ideal result and -10 represents a misleading or harmful result (spam). Compute an average score for each engine over all ten queries and results. Did the engine you normally use receive the highest score? Do you believe that the results of this exercise accurately reflect the relative quality of the search engines? Suggest three possible improvements to this experiment.

Exercise 1.2 Obtain and install an open-source IR system, such as one of those listed in Section 1.5. Create a small document collection from your e-mail, or from another source. A few dozen documents should be enough. Index your collection. Try a few queries.

Exercise 1.3 Starting in state 3 of the Markov model in Figure 1.6, what is the probability of generating "not not be to be"?

Exercise 1.4 Starting in an unknown state, the Markov model in Figure 1.6 generates "to be". What state or states could be the current state of the model after generating this text?

Exercise 1.5 Is there a finite n, such that the current state of the Markov model in Figure 1.6 will always be known after it generates a string of length n or greater, regardless of the starting state?

Exercise 1.6 For a given Markov model, assume there is a finite n so that the current state of the model will always be known after it generates a string of length n or greater. Describe a procedure for converting such a Markov model into an nth-order finite-context model.

Exercise 1.7 Assume that we extend Markov models to allow multiple transitions between pairs of states, where each transition between a given pair of states is labeled with a different term. For any such model, show that there is an equivalent Markov model in which there is no more than one transition between any pair of states. (*Hint*: Split target states.)

Exercise 1.8 Outline a procedure for converting an nth-order finite-context language model into a Markov model. How many states might be required?

Exercise 1.9 (project exercise) This exercise develops a test corpus, based on Wikipedia, that will be used in a number of exercises throughout Part I.

To start, download a current copy of the English-language Wikipedia. At the time of writing, its downloadable version consisted of a single large file. Wikipedia itself contains documentation describing the format of this download.[15]

Along with the text of the articles themselves, the download includes *redirection records* that provide alternative titles for articles. Pre-process the download to remove these records and any other extraneous information, leaving a set of individual articles. Wikipedia-style formatting should also be removed, or replaced with XML-style tags of your choosing. Assign a unique identifier to each article. Add <DOC> and <DOCNO> tags. The result should be consistent with TREC conventions, as described in Section 1.4 and illustrated in Figure 1.7.

Exercise 1.10 (project exercise) Following the style of Figure 1.8, create three to four topics suitable for testing retrieval performance over the English-language Wikipedia. Try to avoid topics expressing an information need that can be completely satisfied by a single "best" article. Instead, try to create topics that require multiple articles in order to cover all relevant information. (*Note*: This exercise is suitable as the foundation for a class project to create a Wikipedia test collection, with each student contributing enough topics to make a combined set of 50 topics or more. (See Exercise 2.13 for further details.)

Exercise 1.11 (project exercise) Obtain and install an open-source IR system (see Exercise 1.2). Using this IR system, index the collection you created in Exercise 1.9. Submit the titles of the topics you created in Exercise 1.10 as queries to the system. For each topic, judge the top five documents returned as either relevant or not. Does the IR system work equally well on all topics?

Exercise 1.12 (project exercise) Tokenize the collection you created in Exercise 1.9, following the procedure of Section 1.3.2. For this exercise, discard the tags and keep only the words (i.e., strings of alphanumeric characters). Wikipedia text is encoded in UTF-8 Unicode, but you may treat it as ASCII text for the purpose of this exercise. Generate a log-log plot of frequency vs. rank order, equivalent to that shown in Figure 1.5. Does the data follow Zipf's law? If so, what is an approximate value for α?

Exercise 1.13 (project exercise) Create a trigram language model based on the tokenization of Exercise 1.12. Using your language model, implement a random Wikipedia text generator. How could you extend your text generator to generate capitalization and punctuation, making your text look more like English? How could you extend your text generator to generate random Wikipedia articles, including tagging and links?

[15] `en.wikipedia.org/wiki/Wikipedia:Database_download`

1.8 Bibliography

Baeza-Yates, R. A., and Ribeiro-Neto, B. (2010). *Modern Information Retrieval* (2nd ed.). Reading, Massachusetts: Addison-Wesley.

Croft, W. B., Metzler, D., and Strohman, T. (2010). *Search Engines: Information Retrieval in Practice*. London, England: Pearson.

Grossman, D. A., and Frieder, O. (2004). *Information Retrieval: Algorithms and Heuristics* (2nd ed.). Berlin, Germany: Springer.

Hearst, M. A. (2009). *Search User Interfaces*. Cambridge, England: Cambridge University Press.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge, England: Cambridge University Press.

Manning, C. D., and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. Cambridge, Massachusetts: MIT Press.

Özsu, M. T., and Liu, L., editors (2009). *Encyclopedia of Database Systems*. Berlin, Germany: Springer.

Salton, G. (1968). *Automatic Information Organziation and Retrieval*. New York: McGraw-Hill.

van Rijsbergen, C. J. (1979). *Information Retrieval* (2nd ed.). London, England: Butterworths.

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

Zipf, G. K. (1949). *Human Behavior and the Principle of Least-Effort*. Cambridge, Massachusetts: Addison-Wesley.

2 Basic Techniques

As a foundation for the remainder of the book, this chapter takes a tour through the elements of information retrieval outlined in Chapter 1, covering the basics of indexing, retrieval and evaluation. The material on indexing and retrieval, constituting the first two major sections, is closely linked, presenting a unified view of these topics. The third major section, on evaluation, examines both the efficiency and the effectiveness of the algorithms introduced in the first two sections.

2.1 Inverted Indices

The inverted index (sometimes called *inverted file*) is the central data structure in virtually every information retrieval system. At its simplest, an inverted index provides a mapping between terms and their locations of occurrence in a text collection $\mathcal{C}$. The fundamental components of an inverted index are illustrated in Figure 2.1, which presents an index for the text of Shakespeare's plays (Figures 1.2 and 1.3). The *dictionary* lists the terms contained in the vocabulary $\mathcal{V}$ of the collection. Each term has associated with it a *postings list* of the positions in which it appears, consistent with the positional numbering in Figure 1.4 (page 14).

If you have encountered inverted indices before, you might be surprised that the index shown in Figure 2.1 contains not document identifiers but "flat" word positions of the individual term occurrences. This type of index is called a *schema-independent* index because it makes no assumptions about the structure (usually referred to as *schema* in the database community) of the underlying text. We chose the schema-independent variant for most of the examples in this chapter because it is the simplest. An overview of alternative index types appears in Section 2.1.3.

Regardless of the specific type of index that is used, its components — the dictionary and the postings lists — may be stored in memory, on disk, or a combination of both. For now, we keep the precise data structures deliberately vague. We define an inverted index as an abstract data type (ADT) with four methods:

- **first**(t) returns the first position at which the term t occurs in the collection
- **last**(t) returns the last position at which t occurs in the collection
- **next**(t, *current*) returns the position of t's first occurrence after the *current* position
- **prev**(t, *current*) returns the position of t's last occurrence before the *current* position.

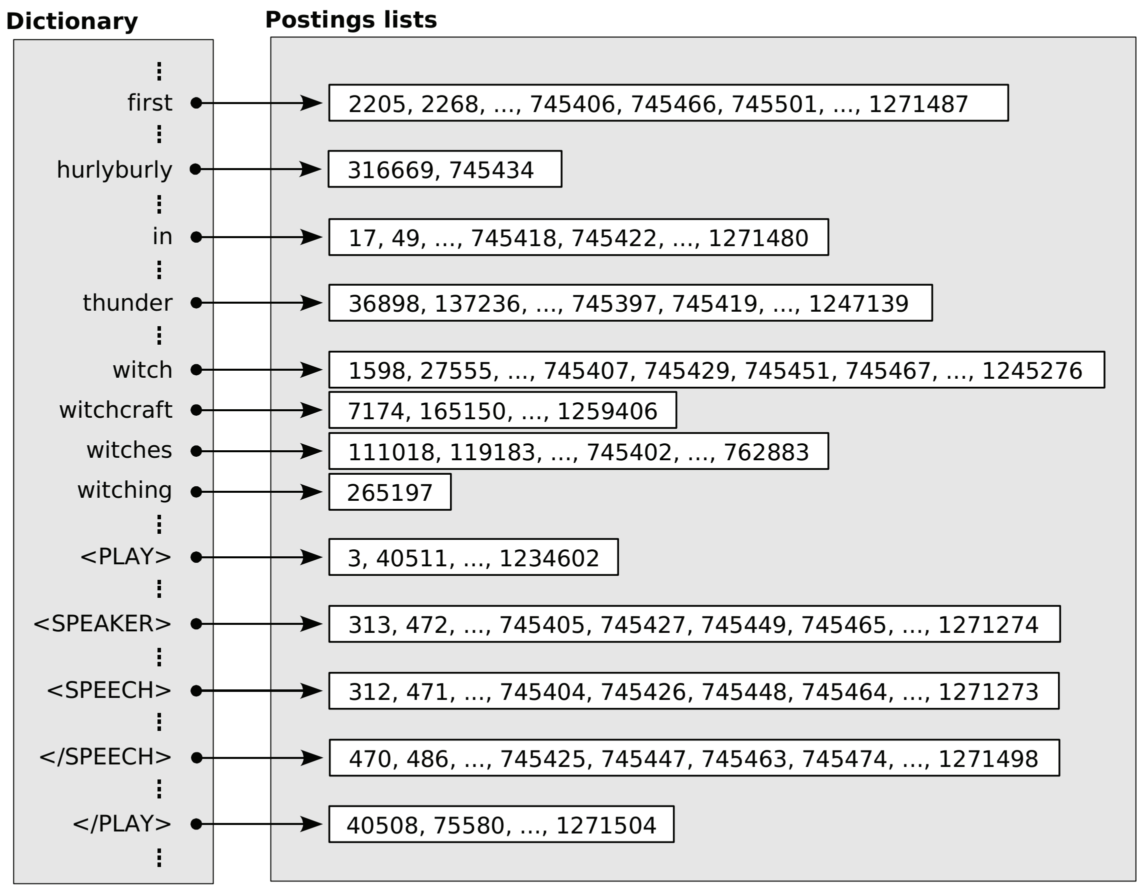


Figure 2.1 A schema-independent inverted index for Shakespeare's plays. The dictionary provides a mapping from terms to their positions of occurrence.

In addition, we define l_t to represent the total number of times the term t appears in the collection (i.e., the length of its postings list). We define $l_{\mathcal{C}}$ to be the length of the collection, so that $\sum_{t \in \mathcal{V}} l_t = l_{\mathcal{C}}$ (where $\mathcal{V}$ is the collection's vocabulary).

For the inverted index in Figure 2.1, we have:

$$\begin{array}{ll} \textbf{first}(\text{“hurlyburly”}) = 316669 & \textbf{last}(\text{“thunder”}) = 1247139 \\ \textbf{first}(\text{“witching”}) = 265197 & \textbf{last}(\text{“witching”}) = 265197 \end{array}$$

$$\begin{array}{ll} \textbf{next}(\text{"witch"},\ 745429) = 745451 & \textbf{prev}(\text{"witch"},\ 745451) = 745429 \\ \textbf{next}(\text{"hurlyburly"},\ 345678) = 745434 & \textbf{prev}(\text{"hurlyburly"},\ 456789) = 316669 \\ \textbf{next}(\text{"witch"},\ 1245276) = \infty & \textbf{prev}(\text{"witch"},\ 1598) = -\infty \\ l_{\text{<PLAY>}} = 37 & l_{\mathcal{C}} = 1271504 \\ l_{\text{witching}} = 1 & \end{array}$$

The symbols ∞ and $-\infty$ act as beginning-of-file and end-of-file markers, representing positions beyond the beginning and the end of the term sequence. As a practical convention we define:

$$\begin{array}{ll} \textbf{next}(t,\ -\infty) = \textbf{first}(t) & \textbf{next}(t,\ \infty) = \infty \\ \textbf{prev}(t,\ \infty) = \textbf{last}(t) & \textbf{prev}(t,\ -\infty) = -\infty \end{array}$$

The methods of our inverted index permit both sequential and random access into postings lists, with a sequential scan of a postings list being a simple loop:

```
$current \leftarrow -\infty$
while $current < \infty$ do
    $current \leftarrow \textbf{next}(t,\ current)$
    do something with the $current$ value
```

However, many algorithms require random access into postings lists, including the phrase search algorithm we present next. Often, these algorithms take the result of a method call for one term and apply it as an argument to a method call for another term, skipping through the postings lists nonsequentially.

2.1.1 Extended Example: Phrase Search

Most commercial Web search engines, as well as many other IR systems, treat a list of terms enclosed in double quotes (`"..."`) as a phrase. To process a query that contains a phrase, the IR system must identify the occurrences of the phrase in the collection. This information may then be used during the retrieval process for filtering and ranking — perhaps excluding documents that do not contain an exact phrase match.

Phrase search provides an excellent example of how algorithms over inverted indices operate. Suppose we wish to locate all occurrences of the phrase “first witch” in our collection of Shakespeare’s plays. Perhaps we wish to identify all speeches by this character. By visually scanning the postings lists in Figure 2.1, we can locate one occurrence starting at 745406 and ending at 745407. We might locate other occurrences in a similar fashion by scanning the postings lists for an occurrence of “first” immediately followed by an occurrence of “witch”. In this section we

```
    nextPhrase ($t_1 t_2 ... t_n$, position) $\equiv$
1       $v \leftarrow$ position
2       for $i \leftarrow 1$ to $n$ do
3           $v \leftarrow$ next($t_i$, $v$)
4       if $v = \infty$ then
5           return [$\infty$, $\infty$]
6       $u \leftarrow v$
7       for $i \leftarrow n - 1$ down to 1 do
8           $u \leftarrow$ prev($t_i$, $u$)
9       if $v - u = n - 1$ then
10          return [$u$, $v$]
11      else
12          return nextPhrase($t_1 t_2 ... t_n$, $u$)
```

Figure 2.2 Function to locate the first occurrence of a phrase after a given position. The function calls the **next** and **prev** methods of the inverted index ADT and returns an interval in the text collection as a result.

present an algorithm that formalizes this process, efficiently locating all occurrences of a given phrase with the aid of our inverted index ADT.

We specify the location of a phrase by an interval [u,v], where u indicates the start of the phrase and v indicates the end of the phrase. In addition to the occurrence at [745406, 745407], the phrase "first witch" may be found at [745466, 745467], at [745501, 745502], and elsewhere. The goal of our phrase searching algorithm is to determine values of u and v for all occurrences of the phrase in the collection.

We use the above interval notation to specify retrieval results throughout the book. In some contexts it is also convenient to think of an interval as a stand-in for the text at that location. For example, the interval [914823, 914829] might represent the text

O Romeo, Romeo! wherefore art thou Romeo?

Given the phrase "$t_1 t_2 ... t_n$", consisting of a sequence of n terms, our algorithm works through the postings lists for the terms from left to right, making a call to the **next** method for each term, and then from right to left, making a call to the **prev** method for each term. After each pass from left to right and back, it has computed an interval in which the terms appear in the correct order and as close together as possible. It then checks whether the terms are in fact adjacent. If they are, an occurrence of the phrase has been found; if not, the algorithm moves on.

Figure 2.2 presents the core of the algorithm as a function **nextPhrase** that locates the next occurrence of a phrase after a given position. The loop over lines 2–3 calls the methods of the inverted index to locate the terms in order. At the end of the loop, if the phrase occurs in the interval [*position*,v], it ends at v. The loop over lines 7–8 then shrinks the interval to the smallest size possible while still including all terms in order. Finally, lines 9–12 verify that the terms are adjacent, forming a phrase. If they are not adjacent, the function makes a tail-recursive call. On line 12, note that u (and not v) is passed as the second argument to the recursive call. If the terms in the phrase are all different, then v could be passed. Passing u correctly handles the case in which two terms t_i and t_j are equal ($1 \leq i < j \leq n$).

As an example, suppose we want to find the first occurrences of the phrase "first witch": **nextPhrase**("first witch", $-\infty$). The algorithm starts by identifying the first occurrence of "first":

next("first", $-\infty$) = **first**("first") = 2205.

If this occurrence of "first" is part of the phrase, then the next occurrence of "witch" should immediately follow it. However,

next("witch", 2205) = 27555,

that is, it does not immediately follow it. We now know that the first occurrence of the phrase cannot end before position 27555, and we compute

prev("first", 27555) = 26267.

In jumping from 2205 to 26267 in the postings list for "first", we were able to skip 15 occurrences of "first". Because interval [26267, 27555] has length 1288, and not the required length 2, we move on to consider the next occurrence of "first" at

next("first", 26267) = 27673.

Note that the calls to the **prev** method in line 8 of the algorithm are not strictly necessary (see Exercise 2.2), but they help us to analyze the complexity of the algorithm.

If we want to generate all occurrences of the phrase instead of just a single occurrence, an additional loop is required, calling **nextPhrase** once for each occurrence of the phrase:

```
$u \leftarrow -\infty$
while $u < \infty$ do
    $[u,v] \leftarrow$ nextPhrase("$t_1 t_2 ... t_n$", $u$)
    if $u \neq \infty$ then
        report the interval $[u,v]$
```

The loop reports each interval as it is generated. Depending on the application, reporting $[u,v]$ might involve returning the document containing the phrase to the user, or it might involve storing the interval in an array or other data structure for further processing. Similar to the code in Figure 2.2, u (and not v) is passed as the second argument to **nextPhrase**. As a result, the function can correctly locate all six occurrences of the phrase "spam spam spam" in the follow passage from the well-known Monty Python song:

Spam spam spam spam
Spam spam spam spam

To determine the time complexity of the algorithm, first observe that each call to **nextPhrase** makes $O(n)$ calls to the **next** and **prev** methods of the inverted index (n calls to **next**, followed by $n-1$ calls to **prev**). After line 8, the interval $[u,v]$ contains all terms in the phrase in order, and there is no smaller interval contained within it that also contains all the terms in order. Next, observe that each occurrence of a term t_i in the collection can be included in no more than one of the intervals computed by lines 1–8. Even if the phrase contains two identical terms, t_i and t_j, a matching token in the collection can be included in only one such interval as a match to t_i, although it might be included in another interval as a match to t_j. The time complexity is therefore determined by the length of the shortest postings list for the terms in the phrase:

$$l = \min_{1 \le i \le n} l_{t_i}. \tag{2.1}$$

Combining these observations, in the worst case the algorithm requires $O(n \cdot l)$ calls to methods of our ADT to locate all occurrences of the phrase. If the phrase includes both common and uncommon terms ("Rosencrantz and Guildenstern are dead"), the number of calls is determined by the least frequent term ("Guildenstern") and not the most frequent one ("and").

We emphasize that $O(n \cdot l)$ represents the number of method calls, not the number of steps taken by the algorithm, and that the time for each method call depends on the details of how it is implemented. For the access patterns generated by the algorithm, there is a surprisingly simple and efficient implementation that gives good performance for phrases containing any mixture of frequent and infrequent terms. We present the details in the next section.

Although the algorithm requires $O(n \cdot l)$ method calls in the worst case, the actual number of calls depends on the relative location of the terms in the collection. For example, suppose we are searching for the phrase "hello world" and the text in the collection is arranged:

hello ... hello ... hello ... hello world ... world ... world ... world

with all occurrences of "hello" before all occurrences of "world". Then the algorithm makes only four method calls to locate the single occurrence of the phrase, regardless of the size of the text or the number of occurrences of each term. Although this example is extreme and artificial, it illustrates the *adaptive* nature of the algorithm — its actual execution time is determined by characteristics of the data. Other IR problems may be solved with adaptive algorithms, and we exploit this approach whenever possible to improve efficiency.

To make the adaptive nature of the algorithm more explicit, we introduce a measure of the characteristics of the data that determines the actual number of method calls. Consider the interval $[u,v]$ just before the test at line 9 of Figure 2.2. The interval contains all the terms in the phrase in order, but does not contain any smaller interval containing all the terms in order. We call an interval with this property a *candidate phrase* for the terms. If we define κ to be the number of candidate phrases in a given document collection, then the number of method calls required to locate all occurrences is $O(n \cdot \kappa)$.

2.1.2 Implementing Inverted Indices

It moves across the blackness that lies between stars, and its mechanical legs move slowly. Each step that it takes, however, crossing from nothing to nothing, carries it twice the distance of the previous step. Each stride also takes the same amount of time as the prior one. Suns flash by, fall behind, wink out. It runs through solid matter, passes through infernos, pierces nebulae, faster and faster moving through the starfall blizzard in the forest of the night. Given a sufficient warm-up run, it is said that it could circumnavigate the universe in a single stride. What would happen if it kept running after that, no one knows.

— Roger Zelazny, *Creatures of Light and Darkness*

When a collection will never change and when it is small enough to be maintained entirely in memory, an inverted index may be implemented with very simple data structures. The dictionary may be stored in a hash table or similar structure, and the postings list for each term t may be stored in a fixed array $P_t[]$ with length l_t. For the term "witch" in the Shakespeare collection, this array may be represented as follows:

1	2		31	32	33	34		92
1598	27555	$\cdots$	745407	745429	745451	745467	$\cdots$	1245276

The **first** and **last** methods of our inverted index ADT may be implemented in constant time by returning $P_t[1]$ and $P_t[l_t]$, respectively. The **next** and **prev** methods may be implemented by a binary search with time complexity $O(\log(l_t))$. Details for the **next** method are provided in Figure 2.3; the details for the **prev** method are similar.

Recall that the phrase searching algorithm of Section 2.1.1 requires $O(n \cdot l)$ calls to the **next** and **prev** methods in the worst case. If we define

$$L = \max_{1 \le i \le n} l_{t_i}\,, \tag{2.2}$$

then the time complexity of the algorithms becomes $O(n \cdot l \cdot \log(L))$ because each call to a method may require up to $O(\log(L))$ time. When expressed in terms of κ, the number of candidate phrases, the time complexity becomes $O(n \cdot \kappa \cdot \log(L))$.

When a phrase contains both frequent and infrequent terms, this implementation can provide excellent performance. For example, the term "tempest" appears only 49 times in the works of Shakespeare. As we saw in Section 1.3.3, the term "the" appears 28,317 times. However, when

```
   next ($t$, $current$) ≡
1     if $l_t = 0$ or $P_t[l_t] \leq current$ then
2        return $\infty$
3     if $P_t[1] > current$ then
4        return $P_t[1]$
5     return $P_t$[binarySearch ($t$, 1, $l_t$, $current$)]

   binarySearch ($t$, $low$, $high$, $current$) ≡
6     while $high - low > 1$ do
7        $mid \leftarrow \lfloor (low + high)/2 \rfloor$
8        if $P_t[mid] \leq current$ then
9           $low \leftarrow mid$
10       else
11          $high \leftarrow mid$
12    return $high$
```

Figure 2.3 Implementation of the **next** method through a binary search that is implemented by a separate function. The array $P_t[]$ (of length l_t) contains the postings list for term t. The **binarySearch** function assumes that $P_t[low] \leq current$ and $P_t[high] > current$. Lines 1–4 establish this precondition, and the loop at lines 6–11 maintains it as an invariant.

searching for the phrase "the tempest", we access the postings list array for "the" less than two thousand times while conducting at most $2 \cdot 49 = 98$ binary searches.

On the other hand, when a phrase contains terms with similar frequencies, the repeated binary searches may be wasteful. The terms in the phrase "two gentlemen" both appear a few hundred times in Shakespeare (702 and 225 times, to be exact). Identifying all occurrences of this phrase requires more than two thousand accesses to the postings list array for "two". In this case, it would be more efficient if we could scan sequentially through both arrays at the same time, comparing values as we go. By changing the definition of the **next** and **prev** methods, the phrase search algorithm can be adapted to do just that.

To start with, we note that as the phrase search algorithm makes successive calls to the **next** method for a given term t_i, the values passed as the second argument strictly increase across calls to **nextPhrase**, including the recursive calls. During the process of finding all occurrences of a given phrase, the algorithm may make up to l calls to **next** for that term (where l, as before, is the length of the shortest postings list):

$$\textbf{next}(t_i,\, v_1),\ \textbf{next}(t_i,\, v_2),\ ...,\ \textbf{next}(t_i,\, v_l)$$

with

$$v_1 < v_2 < ... < v_l\,.$$

Moreover, the results of these calls also strictly increase:

$$\textbf{next}(t_i,\, v_1) < \textbf{next}(t_i,\, v_2) < ... < \textbf{next}(t_i,\, v_l)\,.$$

```
   next ($t$, $current$) ≡
1      if $l_t = 0$ or $P_t[l_t] \leq current$ then
2          return $\infty$
3      if $P_t[1] > current$ then
4          $c_t \leftarrow 1$
5          return $P_t[c_t]$
6      if $c_t > 1$ and $P_t[c_t - 1] > current$ then
7          $c_t \leftarrow 1$
8      while $P_t[c_t] \leq current$ do
9          $c_t \leftarrow c_t + 1$
10     return $P_t[c_t]$
```

Figure 2.4 Implementation of the **next** method through a linear scan. This implementation updates a cached index offset c_t for each term t, where $P_t[c_t]$ represents the last noninfinite result returned from a call to **next** for this term. If possible, the implementation starts its scan from this cached offset. If not, the cached offset is reset at lines 6–7.

For example, when searching for "first witch" in Shakespeare, the sequence of calls for "first" begins:

$$\textbf{next}(\text{"first"}, -\infty),\ \textbf{next}(\text{"first"}, 26267),\ \textbf{next}(\text{"first"}, 30608),\ ...$$

returning the values

$$2205 < 27673 < 32995 < ...$$

Of course, the exact values for v_1, v_2, ... and the actual number of calls to **next** depend on the locations of the other terms in the phrase. Nonetheless, we know that these values will increase and that there may be l of them in the worst case.

To implement our sequential scan, we remember (or cache) the value returned by a call to **next** for a given term. When the function is called again (for the same term), we continue our scan at this cached location. Figure 2.4 provides the details. The variable c_t caches the array offset of the value returned by the previous call, with a separate value cached for each term in the phrase (e.g., c_{first} and c_{witch}). Because the method may be used in algorithms that do not process postings lists in a strictly increasing order, we are careful to reset c_t if this assumption is violated (lines 6–7).

If we take a similar approach to implementing **prev** and maintain corresponding cached values, the phrase search algorithm scans the postings lists for the terms in the phrase, accessing each element of the postings list arrays a bounded number of times ($O(1)$). Because the algorithm may fully scan the longest postings list (of size L), and all postings lists may be of this length, the overall time complexity of the algorithm is $O(n \cdot L)$. In this case the adaptive nature of the algorithm provides no benefit.

We now have two possible implementations for **next** and **prev** that in effect produce two implementations of **nextPhrase**. The first implementation, with overall time complexity $O(n \cdot l \cdot \log(L))$, is particularly appropriate when the shortest postings list is considerably

```
next ($t$, $current$) ≡
    if $l_t = 0$ or $P_t[l_t] \leq current$ then
        return $\infty$
    if $P_t[1] > current$ then
        $c_t \leftarrow 1$
        return $P_t[c_t]$
    if $c_t > 1$ and $P_t[c_t - 1] \leq current$ then
        $low \leftarrow c_t - 1$
    else
        $low \leftarrow 1$
    $jump \leftarrow 1$
    $high \leftarrow low + jump$
    while $high < l_t$ and $P_t[high] \leq current$ do
        $low \leftarrow high$
        $jump \leftarrow 2 \cdot jump$
        $high \leftarrow low + jump$
    if $high > l_t$ then
        $high \leftarrow l_t$
    $c_t \leftarrow$ binarySearch ($t$, $low$, $high$, $current$)
    return $P_t[c_t]$
```

Figure 2.5 Implementation of the **next** method through a galloping search. Lines 6–9 determine an initial value for *low* such that $P_t[low] \leq current$, using the cached value if possible. Lines 12–17 gallop ahead in exponentially increasing steps until they determine a value for *high* such that $P_t[high] > current$. The final result is determined by a binary search (from Figure 2.3).

shorter than the longest postings list ($l \ll L$). The second implementation, with time complexity $O(n \cdot L)$, is appropriate when all postings lists have approximately the same length ($l \approx L$).

Given this dichotomy, we might imagine choosing between the algorithms at run-time by comparing l with L. However, it is possible to define a third implementation of the methods that combines features of both algorithms, with a time complexity that explicitly depends on the relative sizes of the longest and shortest lists (L/l). This third algorithm is based on a *galloping search*. The idea is to scan forward from a cached position in exponentially increasing steps ("galloping") until the answer is passed. At this point, a binary search is applied to the range formed by the last two steps of the gallop to locate the exact offset of the answer. Figure 2.5 provides the details.

Figure 2.6 illustrates and compares the three approaches for a call to **prev**("witch", 745429) over the Shakespeare collection. Using a binary search (part a), the method would access the array seven times, first at positions 1 and 92 to establish the invariant required by the binary search (not shown), and then at positions 46, 23, 34, 28, and 31 during the binary search itself. Using a sequential scan (part b) starting from an initial cached offset of 1, the method would access the array 34 times, including the accesses required to check boundary conditions (not shown). A galloping search (part c) would access positions 1, 2, 4, 8, 16, and 32 before

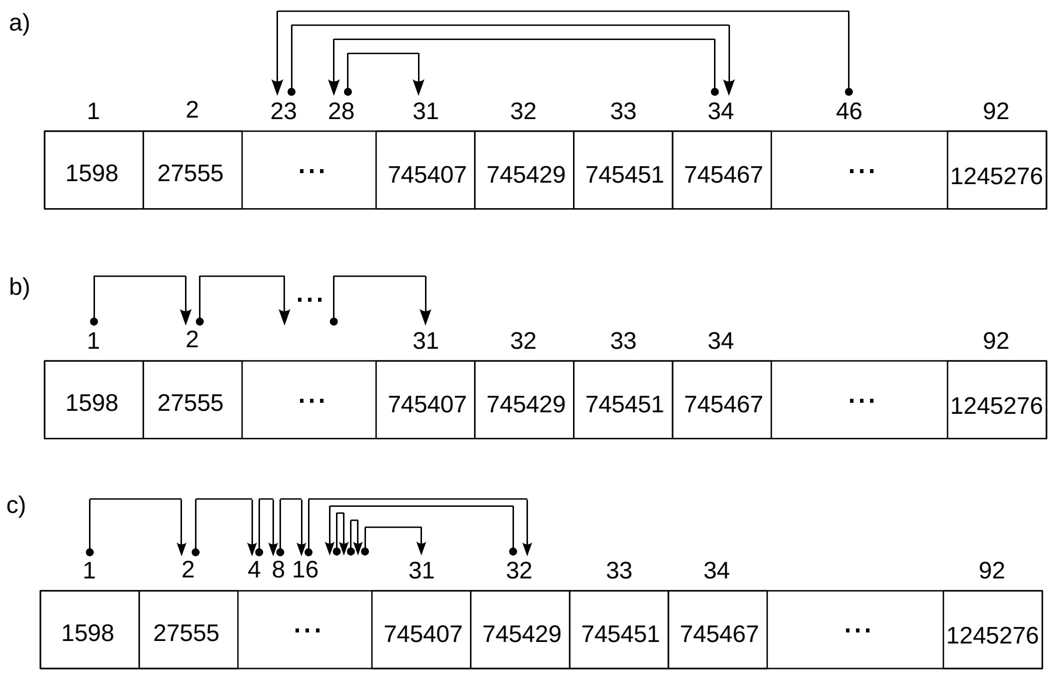


Figure 2.6 Access patterns for three approaches to solving **prev**(“witch”, 745429) = 745407: (a) binary search, (b) sequential scan, and (c) galloping. For (b) and (c), the algorithms start at an initial cached position of 1.

establishing the conditions for a binary search, which would then access positions 24, 28, 30, and 31, for a total of twelve accesses to the postings list array (including checking the boundary conditions). At the end of both the scanning and the galloping methods, the cached array offset would be updated to 31.

To determine the time complexity of galloping search, we return to consider the sequence of calls to **next** that originally motivated the sequential scanning algorithm. Let c_t^j be the cached value after the jth call to **next** for term t during the processing of a given phrase search.

$$\begin{aligned} P_t[c_t^1] &= \textbf{next}\ (t,\, v_1) \\ P_t[c_t^2] &= \textbf{next}\ (t,\, v_2) \\ &\ldots \\ P_t[c_t^l] &= \textbf{next}\ (t,\, v_l) \end{aligned}$$

For a galloping search the amount of work done by a particular call to **next** depends on the change in the cached value from call to call. If the cached value changes by Δc, then the amount of work done by a call is $O(\log(\Delta c))$. Thus, if we define

$$\begin{aligned}\Delta c_1 &= c_t^1\\ \Delta c_2 &= c_t^2 - c_t^1\\ &\cdots\\ \Delta c_l &= c_t^l - c_t^{l-1},\end{aligned}$$

then the total work done by calls to **next** for term t is

$$\sum_{j=1}^{l} O(\log(\Delta c_j)) \;=\; O\left(\log\left(\prod_{j=1}^{l} \Delta c_j\right)\right). \tag{2.3}$$

We know that the arithmetic mean of a list of nonnegative numbers is always greater than its geometric mean

$$\frac{\sum_{j=1}^{l} \Delta c_j}{l} \;\geq\; \sqrt[l]{\prod_{j=1}^{l} \Delta c_j}\,, \tag{2.4}$$

and since $\sum_{j=1}^{l} \Delta c_j \leq L$, we have

$$\prod_{j=1}^{l} \Delta c_j \;\leq\; (L/l)^l. \tag{2.5}$$

Therefore, the total work done by calls to **next** (or **prev**) for the term t is

$$\begin{aligned} O\left(\log\left(\prod_{j=1}^{l} \Delta c_j\right)\right) &\subseteq O\left(\log\left((L/l)^l\right)\right) \qquad (2.6)\\ &= O\left(l \cdot \log\left(L/l\right)\right). \qquad (2.7)\end{aligned}$$

The overall time complexity for a phrase with n terms is $O\left(n \cdot l \cdot \log\left(L/l\right)\right)$. When $l \ll L$, this performance is similar to that of binary search; when $l \approx L$, it is similar to scanning. Taking the adaptive nature of the algorithm into account, a similar line of reasoning gives a time complexity of $O(n \cdot \kappa \cdot \log(L/\kappa))$.

Although we have focused on phrase searching in our discussion of the implementation of inverted indices, we shall see that galloping search is an appropriate technique for other problems, too. Part II of the book extends these ideas to data structures stored on disk.

2.1.3 Documents and Other Elements

Most IR systems and algorithms operate over a standard unit of retrieval: the document. As was discussed in Chapter 1, requirements of the specific application environment determine exactly what constitutes a document. Depending on these requirements, a document might be an e-mail message, a Web page, a newspaper article, or similar element.

In many application environments, the definition of a document is fairly natural. However, in a few environments, such as a collection of books, the natural unit (an entire book) may sometimes be too large to return as a reasonable result, particularly when the relevant material is limited to a small part of the text. Instead, it may be desirable to return a chapter, a section, a subsection, or even a range of pages.

In the case of our collection of Shakespeare's plays, the most natural course is probably to treat each play as a document, but acts, scenes, speeches, and lines might all be appropriate units of retrieval in some circumstances. For the purposes of a simple example, assume we are interested in speeches and wish to locate those spoken by the "first witch".

The phrase "first witch" first occurs at $[745406, 745407]$. Computing the speech that contains this phrase is reasonably straightforward. Using the methods of our inverted index ADT, we determine that the start of a speech immediately preceding this phrase is located at

prev("<SPEECH>", 745406) = 745404.

The end of this speech is located at

next("</SPEECH>", 754404) = 745425.

Once we confirm that the interval $[745406, 745407]$ is contained in the interval $[745404, 745425]$, we know we have located a speech that contains the phrase. This check to confirm the containment is necessary because the phrase may not always occur as part of a speech. If we wish to locate all speeches by the "first witch", we can repeat this process with the next occurrence of the phrase.

A minor problem remains. Although we know that the phrase occurs in the speech, we do not know that the "first witch" is the speaker. The phrase may actually appear in the lines spoken. Fortunately, confirming that the witch is the speaker requires only two additional calls to methods of the inverted index (Exercise 2.4). In fact, simple calls to these methods are sufficient to compute a broad range of structural relationships, such as the following:

1. Lines spoken by any witch.
2. The speaker who says, "To be or not to be".
3. Titles of plays mentioning witches and thunder.

In a broader context, this flexible approach to specifying retrieval units and filtering them with simple containment criteria has many applications. In Web search systems, simple filtering can restrict retrieval results to a single domain. In enterprise search, applying constraints to the sender field allows us to select messages sent by a particular person. In file system search, structural constraints may be used to determine if file permissions and security restrictions allow a user to search a directory.

Because a requirement for "lightweight" structure occurs frequently in IR applications, we adopt a simple and uniform approach to supporting this structure by incorporating it directly into the inverted index, making it part of the basic facilities an inverted index provides. The examples above illustrate our approach. Complete details will be presented in Chapter 5, in which the approach forms the basis for implementing the advanced search operators, which are widely used in domain-specific IR applications, such as legal search. The approach may be used to implement the differential field weighting described in Section 8.7, which recognizes that the presence of a query term in a document's title may be a stronger indicator of relevance than its presence in the body. The approach may also be used to provide a foundation for the implementation of the more complex index structures required to fully support XML retrieval (see Chapter 16).

Notwithstanding those circumstances in which lightweight structure is required, most IR research assumes that the text collection naturally divides into documents, which are considered to be atomic units for retrieval. In a system for searching e-mail, messages form this basic retrieval unit. In a file system, files do; on the Web, Web pages. In addition to providing a natural unit of retrieval, documents also provide natural divisions in the text, allowing a collection to be partitioned into multiple subcollections for parallel retrieval and allowing documents to be reordered to improve efficiency, perhaps by grouping all documents from a single source or Web site.

Document-Oriented Indices

Because document retrieval represents such an important special case, indices are usually optimized around it. To accommodate this optimization, the numbering of positions in a document collection may be split into two components: a document number and an offset within the document.

We use the notation $n{:}m$ to indicate positions within a document-oriented index in which n is a document identifier (or *docid*) and m is an *offset*. Figure 2.7 presents an inverted index for Shakespeare's plays that treats plays as documents. The methods of our inverted index ADT continue to operate as before, but they accept *docid*:*offset* pairs as arguments and return them

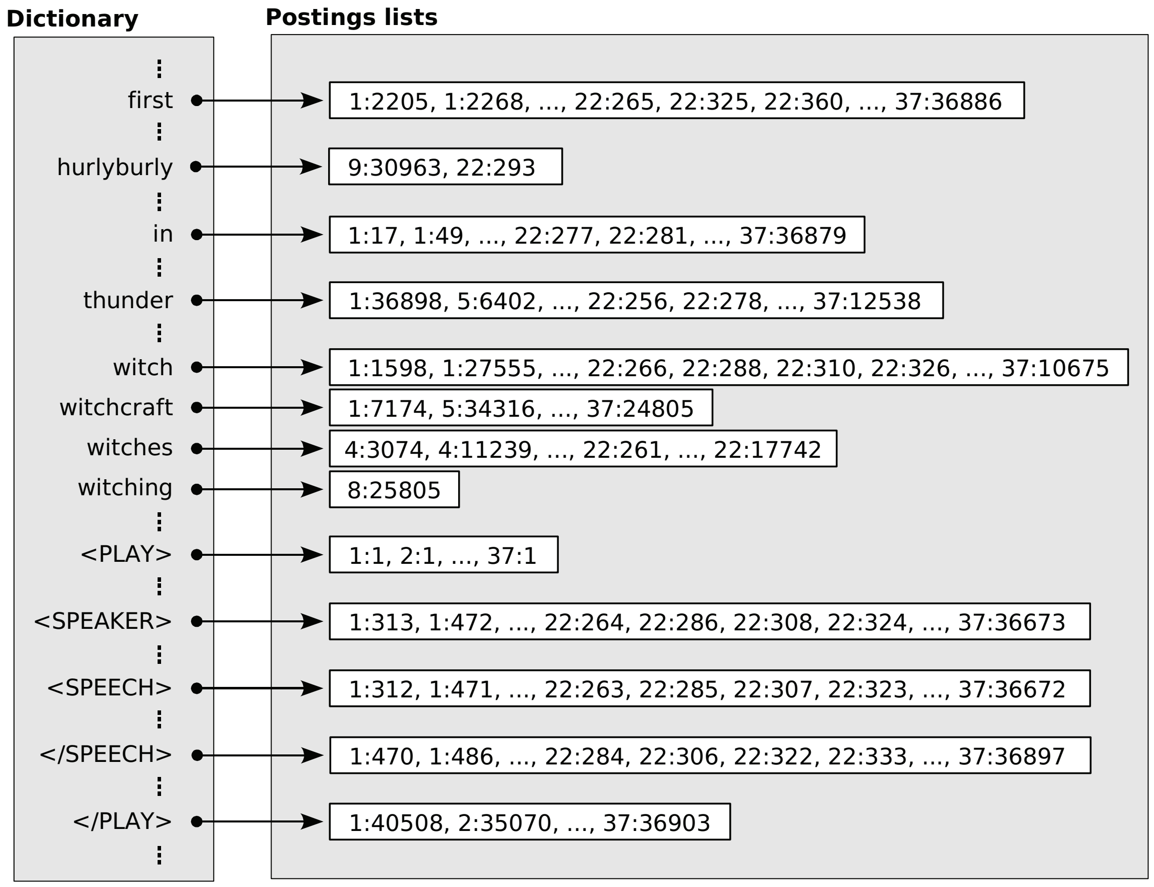


Figure 2.7 A document-centric index for Shakespeare's plays equivalent to the one shown in Figure 2.1 (page 34). Each posting is of the form *docid*:*within-document-position*.

as results. For example,

$$\begin{aligned}
&\textbf{first}(\text{“hurlyburly”}) = 9{:}30963 && \textbf{last}(\text{“thunder”}) = 37{:}12538 \\
&\textbf{first}(\text{“witching”}) = 8{:}25805 && \textbf{last}(\text{“witching”}) = 8{:}25805 \\
&\textbf{next}(\text{“witch”}, 22{:}288) = 22{:}310 && \textbf{prev}(\text{“witch”}, 22{:}310) = 22{:}288 \\
&\textbf{next}(\text{“hurlyburly”}, 9{:}30963) = 22{:}293 && \textbf{prev}(\text{“hurlyburly”}, 22{:}293) = 9{:}30963 \\
&\textbf{next}(\text{“witch”}, 37{:}10675) = \infty && \textbf{prev}(\text{“witch”}, 1{:}1598) = -\infty
\end{aligned}$$

Offsets within a document start at 1 and range up to the length of the document. We continue to use $-\infty$ and ∞ as beginning-of-file and end-of-file markers despite the slight notational inconsistency, with $-\infty$ read as $-\infty$:$-\infty$ and ∞ read as ∞:∞. When term positions are expressed in this form, their values are compared by treating the document number as the primary key:

$$n{:}m < n'{:}m' \text{ if and only if } (n < n' \text{ or } (n = n' \text{ and } m < m')).$$

We refer to an index whose structure has been optimized to support document-oriented retrieval as a *schema-dependent* inverted index, because the division of the text into retrieval units (its schema) is determined at the time its inverted index is constructed.

An index without these optimizations is a *schema-independent* inverted index. A schema-independent index allows the definition of a document to be specified at query time, with a possible cost in execution time in comparison with its schema-dependent equivalent.

To support ranking algorithms, a schema-dependent inverted index usually maintains various document-oriented statistics. We adopt the following notation for these statistics:

N_t	*document frequency*	the number of documents in the collection containing the term t
$f_{t,d}$	*term frequency*	the number of times term t appears in document d
l_d	*document length*	measured in tokens
l_{avg}	*average length*	average document length across the collection
N	*document count*	total number of documents in the collection

Note that $\sum_{d \in \mathcal{C}} l_d = \sum_{t \in \mathcal{V}} l_t = l_{\mathcal{C}}$, and $l_{avg} = l_{\mathcal{C}}/N$.

Over the collection of Shakespeare's plays, $l_{avg} = 34363$. If t = "witch" and $d = 22$ (i.e., d = *Macbeth*), then $N_t = 18$, $f_{t,d} = 52$, and $l_d = 26805$. In a schema-dependent index, these statistics are usually maintained as an integral part of the index data structures. With a schema-independent index, they have to be computed at query time with the methods of our inverted index ADT (see Exercise 2.5).

To help us write algorithms that operate at the document level, we define additional methods of our inverted index ADT. The first two break down positions into separate docids and offsets:

docid(*position*)	returns the docid associated with a *position*
offset(*position*)	returns the within-document offset associated with a *position*

When a posting takes the form $[u{:}v]$, these methods simply return u and v, respectively. They may also be implemented on top of a schema-independent index, albeit slightly less efficiently.

We also define document-oriented versions of our basic inverted index methods that will prove useful in later parts of this chapter:

firstDoc(t)	returns the docid of the first document containing the term t
lastDoc(t)	returns the docid of the last document containing the term t
nextDoc(t, *current*)	returns the docid of the first document after *current* that contains the term t
prevDoc(t, *current*)	returns the docid of the last document before *current* that contains the term t

In a schema-dependent index, many postings may now share a common prefix in their docids. We can separate out these docids to produce postings of the form

$$(d, f_{t,d}, \langle p_0, ..., p_{f_{t,d}} \rangle)$$

where $\langle p_0, ..., p_{f_{t,d}} \rangle$ is the list of the offsets of all $f_{t,d}$ occurrences of the term t within document d. Besides eliminating unnecessary repetition, this notation better reflects how the postings are actually represented in an implementation of a schema-dependent index. Using this notation, we would write the postings list for the term "witch" as

$$(1, 3, \langle 1598, 27555, 31463 \rangle),\ ...,\ (22, 52, \langle 266, 288, ... \rangle),\ ...,\ (37, 1, \langle 10675 \rangle)$$

In specialized circumstances it may not be necessary for positional offsets to be maintained by an inverted index. Basic keyword search is sufficient for some applications, and effective ranking can be supported with document-level statistics. For the simplest ranking and filtering techniques, these document-level statistics are not even required.

Based on the type of information found in each postings list, we can distinguish four types of inverted indices, the first three of which are schema-dependent:

- A *docid index* is the simplest index type. For each term, it contains the document identifiers of all documents in which the term appears. Despite its simplicity, this index type is sufficient to support filtering with basic Boolean queries (Section 2.2.3) and a simple form of relevance ranking known as coordination level ranking (Exercise 2.7).
- In a *frequency index*, each entry in an inverted list consists of two components: a document ID and a term frequency value. Each posting in a frequency index is of the form $(d, f_{t,d})$. Frequency indices are sufficient to support many effective ranking methods (Section 2.2.1), but are insufficient for phrase searching and advanced filtering.
- A *positional index* consists of postings of the form $(d, f_{t,d}, \langle p_1, ..., p_{f_{t,d}} \rangle)$. Positional indices support all search operations supported by a frequency index. In addition, they can be used to realize phrase queries, proximity ranking (Section 2.2.2), and other query types that take the exact position of a query term within a document into account, including all types of structural queries.
- A *schema-independent index* does not include the document-oriented optimizations found in a positional index, but otherwise the two may be used interchangeably.

Table 2.1 Text fragment from Shakespeare's *Romeo and Juliet*, act I, scene 1.

Document ID	**Document Content**
1	Do you quarrel, sir?
2	Quarrel sir! no, sir!
3	If you do, sir, I am for you: I serve as good a man as you.
4	No better.
5	Well, sir.

Table 2.2 Postings lists for the terms shown in Table 2.1. In each case the length of the list is appended to the start of the actual list.

Term	**Docid list**	**Positional list**	**Schema-independent**
a	1; 3	1; $(3, 1, \langle 13 \rangle)$	1; 21
am	1; 3	1; $(3, 1, \langle 6 \rangle)$	1; 14
as	1; 3	1; $(3, 2, \langle 11, 15 \rangle)$	2; 19, 23
better	1; 4	1; $(4, 1, \langle 2 \rangle)$	1; 26
do	2; 1, 3	2; $(1, 1, \langle 1 \rangle), (3, 1, \langle 3 \rangle)$	2; 1, 11
for	1; 3	1; $(3, 1, \langle 7 \rangle)$	1; 15
good	1; 3	1; $(3, 1, \langle 12 \rangle)$	1; 20
i	1; 3	1; $(3, 2, \langle 5, 9 \rangle)$	2; 13, 17
if	1; 3	1; $(3, 1, \langle 1 \rangle)$	1; 9
man	1; 3	1; $(3, 1, \langle 14 \rangle)$	1; 22
no	2; 2, 4	2; $(2, 1, \langle 3 \rangle), (4, 1, \langle 1 \rangle)$	2; 7, 25
quarrel	2; 1, 2	2; $(1, 1, \langle 3 \rangle), (2, 1, \langle 1 \rangle)$	2; 3, 5
serve	1; 3	1; $(3, 1, \langle 10 \rangle)$	1; 18
sir	4; 1, 2, 3, 5	4; $(1, 1, \langle 4 \rangle), (2, 2, \langle 2, 4 \rangle), (3, 1, \langle 4 \rangle), (5, 1, \langle 2 \rangle)$	5; 4, 6, 8, 12, 28
well	1; 5	1; $(5, 1, \langle 1 \rangle)$	1; 27
you	2; 1, 3	2; $(1, 1, \langle 2 \rangle), (3, 3, \langle 2, 8, 16 \rangle)$	4; 2, 10, 16, 24

Table 2.1 shows an excerpt from Shakespeare's *Romeo and Juliet.* Here, each line is treated as a document — we have omitted the tags to help shorten the example to a reasonable length. Table 2.2 shows the corresponding postings lists for all terms that appear in the excerpt, giving examples of docid lists, positional postings lists, and schema-independent postings lists.

Of the four different index types, the docid index is always the smallest one because it contains the least information. The positional and the schema-independent indices consume the greatest space, between two times and five times as much space as a frequency index, and between three times and seven times as much as a docid index, for typical text collections. The exact ratio depends on the lengths of the documents in the collection, the skewedness of the term distribution, and the impact of compression. Index sizes for the four different index types and

Table 2.3 Index sizes for various index types and three test collections, with and without applying index compression techniques. In each case the first number refers to an index in which each component is stored as a simple 32-bit integer, and the second number refers to an index in which each entry is compressed using a byte-aligned encoding method.

	Shakespeare	**TREC**	**GOV2**
Docid index	n/a	578 MB/200 MB	37751 MB/12412 MB
Frequency index	n/a	1110 MB/333 MB	73593 MB/21406 MB
Positional index	n/a	2255 MB/739 MB	245538 MB/78819 MB
Schema-independent index	5.7 MB/2.7 MB	1190 MB/533 MB	173854 MB/65960 MB

our three example collections are shown in Table 2.3. Index compression has a substantial effect on index size, and it is discussed in detail in Chapter 6.

With the introduction of document-oriented indices, we have greatly expanded the notation associated with inverted indices from the four basic methods introduced at the beginning of the chapter. Table 2.4 provides a summary of this notation for easy reference throughout the remainder of the book.

2.2 Retrieval and Ranking

Building on the data structures of the previous section, this section presents three simple retrieval methods. The first two methods produce ranked results, ordering the documents in the collection according to their expected relevance to the query. Our third retrieval method allows Boolean filters to be applied to the collection, identifying those documents that match a predicate.

Queries for ranked retrieval are often expressed as *term vectors*. When you type a query into an IR system, you express the components of this vector by entering the terms with white space between them. For example, the query

william shakespeare marriage

might be entered into a commercial Web search engine with the intention of retrieving a ranked list of Web pages concerning Shakespeare's marriage to Anne Hathaway. To make the nature of these queries more obvious, we write term vectors explicitly using the notation $\langle t_1, t_2, ..., t_n \rangle$. The query above would then be written

$\langle$"william", "shakespeare", "marriage"$\rangle$.

You may wonder why we represent these queries as vectors rather than sets. Representation as a vector (or, rather, a *list*, since we do not assume a fixed-dimensional vector space) is useful when terms are repeated in a query and when the ordering of terms is significant. In ranking formulae, we use the notation q_t to indicate the number of times term t appears in the query.

Table 2.4 Summary of notation for inverted indices.

Basic inverted index methods	
first(*term*)	returns the first position at which the *term* occurs
last(*term*)	returns the last position at which the *term* occurs
next(*term*, *current*)	returns the next position at which the *term* occurs after the *current* position
prev(*term*, *current*)	returns the previous position at which the *term* occurs before the *current* position
Document-oriented equivalents of the basic methods	
firstDoc(*term*), **lastDoc**(*term*), **nextDoc**(*term*, *current*), **lastDoc**(*term*, *current*)	
Schema-dependent index positions	
$n{:}m$	n = *docid* and m = *offset*
docid(*position*)	returns the docid associated with a *position*
offset(*position*)	returns the within-document offset associated with a *position*
Symbols for document and term statistics	
l_t	the length of t's postings list
N_t	the number of documents containing t
$f_{t,d}$	the number of occurrences of t within the document d
l_d	length of the document d, in tokens
l_{avg}	the average document length in the collection
N	the total number of documents in the collection
The structure of postings lists	
docid index	$d_1, d_2, \ldots, d_{N_t}$
frequency index	$(d_1, f_{t,d_1}), (d_2, f_{t,d_2}), \ldots$
positional index	$(d_1, f_{t,d_1}, \langle p_1, \ldots, p_{f_{t,d_1}} \rangle), \ldots$
schema-independent	$p_1, p_2, \ldots, p_{l_t}$

Boolean predicates are composed with the standard Boolean operators (AND, OR, NOT). The result of a Boolean query is a set of documents matching the predicate. For example, the Boolean query

"william" AND "shakespeare" AND NOT ("marlowe" OR "bacon")

specifies those documents containing the terms "william" and "shakespeare" but not containing either "marlowe" or "bacon". In later chapters we will extend this standard set of Boolean operators, which will allow us to specify additional constraints on the result set.

There is a key difference in the conventional interpretations of term vectors for ranked retrieval and predicates for Boolean retrieval. Boolean predicates are usually interpreted as strict filters — if a document does not match the predicate, it is not returned in the result set. Term vectors, on the other hand, are often interpreted as summarizing an information need. Not all the terms in the vector need to appear in a document for it to be returned. For example, if we are interested in the life and works of Shakespeare, we might attempt to create an exhaustive (and exhausting) query to describe our information need by listing as many related terms as we can:

> william shakespeare stratford avon london plays sonnets poems tragedy comedy poet playwright players actor anne hathaway susanna hamnet judith folio othello hamlet macbeth king lear tempest romeo juliet julius caesar twelfth night antony cleopatra venus adonis willie hughe wriothesley henry ...

Although many relevant Web pages will contain some of these terms, few will contain all of them. It is the role of a ranked retrieval method to determine the impact that any missing terms will have on the final document ordering.

Boolean retrieval combines naturally with ranked retrieval into a two-step retrieval process. A Boolean predicate is first applied to restrict retrieval to a subset of the document collection. The documents contained in the resulting subcollection are then ranked with respect to a given topic. Commercial Web search engines follow this two-step retrieval process. Until recently, most of these systems would interpret the query

> william shakespeare marriage

as both a Boolean conjunction of the terms — "william" AND "shakespeare" AND "marriage" — and as a term vector for ranking — ⟨"william", "shakespeare", "marriage"⟩. For a page to be returned as a result, each of the terms was required to appear in the page itself or in the anchor text linking to the page.

Filtering out relevant pages that are missing one or more terms may have the paradoxical effect of harming performance when extra terms are added to a query. In principle, adding extra terms should improve performance by serving to better define the information need. Although some commercial Web search engines now apply less restrictive filters, allowing additional documents to appear in the ranked results, this two-step retrieval process still takes place. These systems may handle longer queries poorly, returning few or no results in some cases.

In determining an appropriate document ranking, basic ranked retrieval methods compare simple features of the documents. One of the most important of these features is term frequency, $f_{t,d}$, the number of times query term t appears in document d. Given two documents d_1 and d_2, if a query term appears many more times in d_1 than in d_2, this may suggest that d_1 should be ranked higher than d_2, other factors being equal. For the query ⟨"william", "shakespeare", "marriage"⟩, the repeated occurrence of the term "marriage" throughout a document may suggest that it should be ranked above one containing the term only once.

Another important feature is term proximity. If query terms appear closer together in document d_1 than in document d_2, this may suggest that d_1 should be ranked higher than d_2, other factors being equal. In some cases, terms form a phrase ("william shakespeare") or other collocation, but the importance of proximity is not merely a matter of phrase matching. The co-occurrence of "william", "shakespeare", and "marriage" together in a fragment such as

> ... while no direct evidence of the **marriage** of Anne Hathaway and **William Shakespeare** exists, the wedding is believed to have taken place in November of 1582, while she was pregnant with his child ...

suggests a relationship between the terms that might not exist if they appeared farther apart.

Other features help us make trade-offs between competing factors. For example, should a thousand-word document containing four occurrences of "william", five of "shakespeare", and two of "marriage" be ranked before or after a five-hundred-word document containing three occurrences of "william", two of "shakespeare", and seven of "marriage"? These features include the lengths of the documents (l_d) relative to the average document length (l_{avg}), as well as the number of documents in which a term appears (N_t) relative to the total number of documents in the collection (N).

Although the basic features listed above form the core of many retrieval models and ranking methods, including those discussed in this chapter, additional features may contribute as well. In some application areas, such as Web search, the exploitation of these additional features is critical to the success of a search engine.

One important feature is document structure. For example, a query term may be treated differently if it appears in the title of a document rather than in its body. Often the relationship between documents is important, such as the links between Web documents. In the context of Web search, the analysis of the links between Web pages may allow us to assign them a query-independent ordering or *static rank*, which can then be a factor in retrieval. Finally, when a large group of people make regular use of an IR system within an enterprise or on the Web, their behavior can be monitored to improve performance. For example, if results from one Web site are clicked more than results from another, this behavior may indicate a user preference for one site over the other — other factors being equal — that can be exploited to improve ranking. In later chapters these and other additional features will be covered in detail.

2.2.1 The Vector Space Model

The *vector space model* is one of the oldest and best known of the information retrieval models we examine in this book. Starting in the 1960s and continuing into 1990s, the method was developed and promulgated by Gerald Salton, who was perhaps the most influential of the early IR researchers. As a result, the vector space model is intimately associated with the field as a whole and has been adapted to many IR problems beyond ranked retrieval, including document clustering and classification, in which it continues to play an important role. In recent years, the

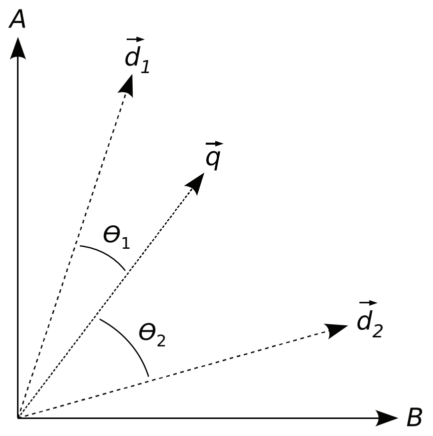


Figure 2.8 Document similarity under the vector space model. Angles are computed between a query vector $\vec{q}$ and two document vectors $\vec{d_1}$ and $\vec{d_2}$. Because $\theta_1 < \theta_2$, d_1 should be ranked higher than d_2.

vector space model has been largely overshadowed by probabilistic models, language models, and machine learning approaches (see Part III). Nonetheless, the simple intuition underlying it, as well as its long history, makes the vector space model an ideal vehicle for introducing ranked retrieval.

The basic idea is simple. Queries as well as documents are represented as vectors in a high-dimensional space in which each vector component corresponds to a term in the vocabulary of the collection. This query vector representation stands in contrast to the term vector representation of the previous section, which included only the terms appearing in the query. Given a query vector and a set of document vectors, one for each document in the collection, we rank the documents by computing a similarity measure between the query vector and each document vector, comparing the angle between them. The smaller the angle, the more similar the vectors. Figure 2.8 illustrates the basic idea, using vectors with only two components (A and B).

Linear algebra provides us with a handy formula to determine the angle θ between two vectors. Given two $|\mathcal{V}|$-dimensional vectors $\vec{x} = \langle x_1, x_2, ..., x_{|\mathcal{V}|} \rangle$ and $\vec{y} = \langle y_1, y_2, ..., y_{|\mathcal{V}|} \rangle$, we have

$$\vec{x} \cdot \vec{y} \;=\; |\vec{x}| \cdot |\vec{y}| \cdot \cos(\theta). \tag{2.8}$$

where $\vec{x} \cdot \vec{y}$ represents the *dot product* (also called the *inner product* or *scalar product*) between the vectors; $|\vec{x}|$ and $|\vec{y}|$ represent the lengths of the vectors. The dot product is defined as

$$\vec{x} \cdot \vec{y} \;=\; \sum_{i=1}^{|\mathcal{V}|} x_i \cdot y_i \tag{2.9}$$

and the length of a vector may be computed from the Euclidean distance formula

$$|\vec{x}| \;=\; \sqrt{\sum_{i=1}^{|\mathcal{V}|} {x_i}^2}\,. \tag{2.10}$$

Substituting and rearranging these equations gives

$$\cos(\theta) \;=\; \frac{\vec{x}}{|\vec{x}|} \cdot \frac{\vec{y}}{|\vec{y}|} \;=\; \frac{\sum_{i=1}^{|\mathcal{V}|} x_i\, y_i}{\left(\sqrt{\sum_{i=1}^{|\mathcal{V}|} {x_i}^2}\right)\left(\sqrt{\sum_{i=1}^{|\mathcal{V}|} {y_i}^2}\right)}\,. \tag{2.11}$$

We could now apply arccos to determine θ, but because the cosine is monotonically decreasing with respect to the angle θ, it is convenient to stop at this point and retain the cosine itself as our measure of similarity. If $\theta = 0°$, then the vectors are colinear, as similar as possible, with $\cos(\theta) = 1$. If $\theta = 90°$, then the vectors are orthogonal, as dissimilar as possible, with $\cos(\theta) = 0$.

To summarize, given a document vector $\vec{d}$ and a query vector $\vec{q}$, the cosine similarity $sim(\vec{d}, \vec{q})$ is computed as

$$sim(\vec{d}, \vec{q}) \;=\; \frac{\vec{d}}{|\vec{d}|} \cdot \frac{\vec{q}}{|\vec{q}|}\,, \tag{2.12}$$

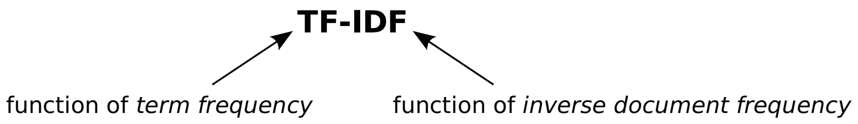
the dot product of the document and query vectors normalized to unit length. Provided all components of the vectors are nonnegative, the value of this *cosine similarity measure* ranges from 0 to 1, with its value increasing with increasing similarity.

Naturally, for a collection of even modest size, this vector space model produces vectors with millions of dimensions. This high-dimensionality might appear inefficient at first glance, but in many circumstances the query vector is sparse, with all but a few components being zero. For example, the vector corresponding to the query ⟨“william”, “shakespeare”, “marriage”⟩ has only three nonzero components. To compute the length of this vector, or its dot product with a document vector, we need only consider the components corresponding to these three terms. On the other hand, a document vector typically has a nonzero component for each unique term contained in the document, which may consist of thousands of terms. However, the length of a document vector is independent of the query. It may be precomputed and stored in a frequency or positional index along with other document-specific information, or it may be applied to normalize the document vector in advance, with the components of the normalized vector taking the place of term frequencies in the postings lists.

Although queries are usually short, the symmetry between how documents and queries are treated in the vector space model allows entire documents to be used as queries. Equation 2.12 may then be viewed as a formula determining the similarity between two documents. Treating a document as a query is one possible method for implementing the “Similar pages” and “More like this” features seen in some commercial search engines.

As a ranking method the cosine similarity measure has intuitive appeal and natural simplicity. If we can appropriately represent queries and documents as vectors, cosine similarity may be used to rank the documents with respect to the queries. In representing a document or query as a vector, a *weight* must be assigned to each term that represents the value of the corresponding component of the vector. Throughout the long history of the vector space model, many formulae for assigning these weights have been proposed and evaluated. With few exceptions, these formulae may be characterized as belonging to a general family known as *TF-IDF weights*.

When assigning a weight in a document vector, the TF-IDF weights are computed by taking the product of a function of term frequency ($f_{t,d}$) and a function of the inverse of document frequency ($1/N_t$). When assigning a weight to a query vector, the within-query term frequency (q_t) may be substituted for $f_{t,d}$, in essence treating the query as a tiny document. It is also possible (and not at all unusual) to use different TF and IDF functions to determine weights for document vectors and query vectors.


TF-IDF
function of *term frequency*
function of *inverse document frequency*

We emphasize that a TF-IDF weight is a product of *functions* of term frequency and inverse document frequency. A common error is to use the raw $f_{t,d}$ value for the term frequency component, which may lead to poor performance.

Over the years a number of variants for both the TF and the IDF functions have been proposed and evaluated. The IDF functions typically relate the document frequency to the total number of documents in the collection (N). The basic intuition behind the IDF functions is that a term appearing in many documents should be assigned a lower weight than a term appearing in few documents. Of the two functions, IDF comes closer to having a "standard form",

$$\text{IDF} = \log(N/N_t), \tag{2.13}$$

with most IDF variants structured as a logarithm of a fraction involving N_t and N.

The basic intuition behind the various TF functions is that a term appearing many times in a document should be assigned a higher weight for that document than for a document in which it appears fewer times. Another important consideration behind the definition of a TF function is that its value should not necessarily increase linearly with $f_{t,d}$. Although two occurrences of a term should be given more weight than one occurrence, they shouldn't necessarily be given twice the weight. The following function meets these requirements and appears in much of Salton's later work:

$$\text{TF} = \begin{cases} \log(f_{t,d}) + 1 & \text{if } f_{t,d} > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.14}$$

When this equation is used with a query vector, $f_{t,d}$ is replaced by q_t the query term frequency of t in q. We use this equation, along with Equation 2.13, to compute both document and query weights in the example that follows.

Consider the *Romeo and Juliet* document collection in Table 2.1 (page 50) and the corresponding postings lists given in Table 2.2. Because there are five documents in the collection and "sir" appears in four of them, the IDF value for "sir" is

$$\log(N/f_{\text{sir}}) \;=\; \log(5/4) \;\approx\; 0.32.$$

In this formula and in other TF-IDF formulae involving logarithms, the base of the logarithm is usually unimportant. As necessary, for purposes of this example and others throughout the book, we assume a base of 2.

Because "sir" appears twice in document 2, the TF-IDF value for the corresponding component of its vector is

$$(\log(f_{\text{sir},2})+1)\cdot(\log(N/f_{\text{sir}})) \;=\; (\log(2)+1)\cdot(\log(5/4)) \;\approx\; 0.64.$$

Computing TF-IDF values for the remaining components and the remaining documents, gives the following set of vectors:

$$\begin{aligned}
\vec{d_1} &\approx \langle 0.00, 0.00, 0.00, 0.00, 1.32, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 1.32, 0.00, 0.32, 0.00, 1.32\rangle\\
\vec{d_2} &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 1.32, 1.32, 0.00, 0.64, 0.00, 0.00\rangle\\
\vec{d_3} &\approx \langle 2.32, 2.32, 4.64, 0.00, 1.32, 2.32, 2.32, 4.64, 2.32, 2.32, 0.00, 0.00, 2.32, 0.32, 0.00, 3.42\rangle\\
\vec{d_4} &\approx \langle 0.00, 0.00, 0.00, 2.32, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 1.32, 0.00, 0.00, 0.00, 0.00, 0.00\rangle\\
\vec{d_5} &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.32, 2.32, 0.00\rangle
\end{aligned}$$

where the components are sorted alphabetically according to their corresponding terms. Normalizing these vectors, dividing by their lengths, produces:

$$\begin{aligned}
\vec{d_1}/|\vec{d_1}| &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.57, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.57, 0.00, 0.14, 0.00, 0.57\rangle\\
\vec{d_2}/|\vec{d_2}| &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.67, 0.67, 0.00, 0.33, 0.00, 0.00\rangle\\
\vec{d_3}/|\vec{d_3}| &\approx \langle 0.24, 0.24, 0.48, 0.00, 0.14, 0.24, 0.24, 0.48, 0.24, 0.24, 0.00, 0.00, 0.24, 0.03, 0.00, 0.35\rangle\\
\vec{d_4}/|\vec{d_4}| &\approx \langle 0.00, 0.00, 0.00, 0.87, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.49, 0.00, 0.00, 0.00, 0.00, 0.00\rangle\\
\vec{d_5}/|\vec{d_5}| &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.14, 0.99, 0.00\rangle
\end{aligned}$$

If we wish to rank these five documents with respect to the query ⟨"quarrel", "sir"⟩, we first construct the query vector, normalized by length:

$$\vec{q}/|\vec{q}| \;\approx\; \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.97, 0.00, 0.24, 0.00, 0.00\rangle$$

```
rankCosine ($\langle t_1, ..., t_n \rangle$, k) $\equiv$
    $j \leftarrow 1$
    $d \leftarrow \min_{1 \le i \le n}$ nextDoc ($t_i$, $-\infty$)
    while $d < \infty$ do
        $Result[j].docid \leftarrow d$
        $Result[j].score \leftarrow \frac{\vec{d}}{|\vec{d}|} \cdot \frac{\vec{q}}{|\vec{q}|}$
        $j \leftarrow j + 1$
        $d \leftarrow \min_{1 \le i \le n}$ nextDoc ($t_i$, $d$)
    sort Result by score
    return Result[1..k]
```

Figure 2.9 Query processing for ranked retrieval under the vector space model. Given the term vector $\langle t_1, ..., t_n \rangle$ (with corresponding query vector $\vec{q}$), the function identifies the top k documents.

Computing the dot product between this vector and each document vector gives the following cosine similarity values:

Document ID	1	2	3	4	5
Similarity	0.59	0.73	0.01	0.00	0.03

The final document ranking is 2, 1, 5, 3, 4.

Query processing for the vector space model is straightforward (Figure 2.9), essentially performing a merge of the postings lists for the query terms. Docids and corresponding scores are accumulated in an array of records as the scores are computed. The function operates on one document at a time. During each iteration of the **while** loop, the algorithm computes the score for document d (with corresponding document vector $\vec{d}$), stores the docid and score in the array of records *Result*, and determines the next docid for processing. The algorithm does not explicitly compute a score for documents that do not contain any of the query terms, which are implicitly assigned a score of zero. At the end of the function, *Result* is sorted by *score* and the top k documents are returned.

For many retrieval applications, the entire ranked list of documents is not required. Instead we return at most k documents, where the value of k is determined by the needs of the application environment. For example, a Web search engine might return only the first $k = 10$ or 20 results on its first page. It then may seem inefficient to compute the score for every document containing any of the terms, even a single term with low weight, when only the top k documents are required. This apparent inefficiency has led to proposals for improved query processing methods that are applicable to other IR models as well as to the vector space model. These query processing methods will be discussed in Chapter 5.

Of the document features listed at the start of this section — term frequency, term proximity, document frequency, and document length — the vector space model makes explicit use of only term frequency and document frequency. Document length is handled implicitly when the vectors are normalized to unit length. If one document is twice as long as another but contains the same terms in the same proportions, their normalized vectors are identical. Term proximity is not considered by the model at all. This property has led to the colorful description of the vector space model (and other IR models with the same property) as a "bag of words" model.

We based the version of the vector space model presented in this section on the introductory descriptions given in Salton's later works. In practice, implementations of the vector space model often eliminate both length normalization and the IDF factor in document vectors, in part to improve efficiency. Moreover, the Euclidean length normalization inherent in the cosine similarity measure has proved inadequate to handle collections containing mixtures of long and short documents, and substantial adjustments are required to support these collections. These efficiency and effectiveness issues are examined in Section 2.3.

The vector space model may be criticized for its entirely heuristic nature. Beyond intuition, its simple mathematics, and the experiments of Section 2.3, we do not provide further justification for it. IR models introduced in later chapters (Chapters 8 and 9) are more solidly grounded in theoretical frameworks. Perhaps as a result, these models are more adaptable, and are more readily extended to accommodate additional document features.

2.2.2 Proximity Ranking

The vector space ranking method from the previous section explicitly depends only on TF and IDF. In contrast, the method detailed in this section explicitly depends only on term proximity. Term frequency is handled implicitly; document frequency, document length, and other features play no role at all.

When the components of a term vector $\langle t_1, t_2, \ldots, t_n \rangle$ appear in close proximity within a document, it suggests that the document is more likely to be relevant than one in which the terms appear farther apart. Given a term vector $\langle t_1, t_2, ..., t_n \rangle$, we define a *cover* for the vector as an interval in the collection $[u, v]$ that contains a match to all the terms without containing a smaller interval $[u', v']$, $u \leq u' \leq v' \leq v$, that also contains a match to all the terms. The candidate phrases defined on page 39 are a special case of a cover in which all the terms appear in order.

In the collection of Table 2.1 (page 50), the intervals [1:2, 1:4], [3:2, 3:4], and [3:4, 3:8] are covers for the term vector ⟨"you", "sir"⟩. The interval [3:4, 3:16] is not a cover, even though both terms are contained within it, because it contains the cover [3:4, 3:8]. Similarly, there are two covers for the term vector ⟨"quarrel", "sir"⟩: [1:3, 1:4] and [2:1, 2:2].

Note that covers may overlap. However, a token matching a term t_i appears in at most $n \cdot l$ covers, where l is the length of the shortest postings list for the terms in the vector. To see that there may be as many as $n \cdot l$ covers for the term vector $\langle t_1, t_2, ..., t_n \rangle$, consider a collection in which all the terms occur in the same order the same number of times:

$$... \, t_1 \, ... \, t_2 \, ... \, t_3 \, ... \, t_n \, ... \, t_1 \, ... \, t_2 \, ... \, t_3 \, ... \, t_n \, ... \, t_1 \, ...$$

```
nextCover (⟨$t_1, ..., t_n$⟩, position) ≡
1    $v \leftarrow \max_{1 \le i \le n}(\textbf{next}(t_i, \textit{position}))$
2    if $v = \infty$ then
3        return $[\infty, \infty]$
4    $u \leftarrow \min_{1 \le i \le n}(\textbf{prev}(t_i, v + 1))$
5    if docid($u$) = docid($v$) then
6        return $[u, v]$
7    else
8        return nextCover(⟨$t_1, ..., t_n$⟩, $u$)
```

Figure 2.10 Function to locate the next occurrence of a cover for the term vector $\langle t_1, ..., t_n \rangle$ after a given position.

We leave the demonstration that there may be no more than $n \cdot l$ covers to Exercise 2.8. A new cover starts at each occurrence of a term from the vector. Thus, the total number of covers for a term vector is constrained by $n \cdot l$ and does not depend on the length of the longest postings list L. With respect to proximity ranking, we define κ to be the number of covers for a term vector occurring in a document collection where $\kappa \le n \cdot l$.

Perhaps not surprisingly, our algorithm to compute covers is a close cousin of the phrase searching algorithm in Figure 2.2 (page 36). The function in Figure 2.10 locates the next occurrence of a cover after a given position. On line 1, the algorithm determines the smallest position v such that the interval $[\textit{position}, v]$ contains all the terms in the vector. A cover starting after u cannot end before this position. On line 4, the algorithm shrinks the interval ending at v, adjusting u so that no smaller interval ending at v contains all the terms. The check at line 5 determines whether u and v are contained in the same document. If not, **nextCover** is called recursively.

This last check is required only because the cover will ultimately contribute to a document's score for ranking. Technically, the interval [1:4, 2:1] is a perfectly acceptable cover for the term vector ⟨"quarrel","sir"⟩. However, in a schema-dependent index, a cover that crosses document boundaries is unlikely to be meaningful.

Ranking is based on two assumptions: (1) the shorter the cover, the more likely that the text containing the cover is relevant, and (2) the more covers contained in a document, the more likely that the document is relevant. These assumptions are consistent with intuition. The first assumption suggests that a score for an individual cover may be based on its length. The second assumption suggests that a document may be assigned a score by summing the individual scores of the covers contained within it. Combining these ideas, we score a document d containing covers $[u_1,v_1]$, $[u_2,v_2]$, $[u_3,v_3]$, ... using the formula

$$\textit{score}(d) \;=\; \sum \left(\frac{1}{v_i - u_i + 1} \right). \tag{2.15}$$

```
rankProximity ($\langle t_1, ..., t_n\rangle$, k) $\equiv$
1   $[u, v] \leftarrow$ nextCover($\langle t_0, t_1, ..., t_n\rangle$, $-\infty$)
2   $d \leftarrow$ docid($u$)
3   $score \leftarrow 0$
4   $j \leftarrow 0$
5   while $u < \infty$ do
6       if $d <$ docid($u$) then
7           $j \leftarrow j + 1$
8           $Result[j].docid \leftarrow d$
9           $Result[j].score \leftarrow score$
10          $d \leftarrow$ docid($u$)
11          $score \leftarrow 0$
12      $score \leftarrow score + 1/(v - u + 1)$
13      $[u, v] \leftarrow$ nextCover($\langle t_1, \ldots, t_n\rangle$, $u$)
14  if $d < \infty$ then
15      $j \leftarrow j + 1$
16      $Result[j].docid \leftarrow d$
17      $Result[j].score \leftarrow score$
18  sort $Result[1..j]$ by $score$
19  return $Result[1..k]$
```

Figure 2.11 Query processing for proximity ranking. The **nextCover** function from Figure 2.10 is called to generate each cover.

Query processing for proximity ranking is presented in Figure 2.11. Covers are generated by calls to the **nextCover** function and processed one by one in the while loop of lines 5–13. The number of covers κ in the collection is exactly equal to the number of calls to **nextCover** at line 13. When a document boundary is crossed (line 6), the score and docid are stored in an array of records *Result* (lines 8–9). After all covers are processed, information on the last document is recorded in the array (lines 14–17), the array is sorted by score (line 18), and the top k documents are returned (line 19).

As the **rankProximity** function makes calls to the **nextCover** function, the position passed as its second argument strictly increases. In turn, as the **nextCover** function makes successive calls to the **next** and **prev** methods, the values of their second arguments also strictly increase. As we did for the phrase searching algorithm of Section 2.1.1, we may exploit this property by implementing **next** and **prev** using galloping search. Following a similar argument, when galloping search is used, the overall time complexity of the **rankProximity** algorithm is $O\left(n^2 l \cdot \log\left(L/l\right)\right)$.

Note that the time complexity is quadratic in n, the size of the term vector, because there may be $O(n \cdot l)$ covers in the worst case. Fortunately, the adaptive nature of the algorithm comes to our assistance again, giving a time complexity of $O\left(n \cdot \kappa \cdot \log\left(L/\kappa\right)\right)$.

For a document to receive a nonzero score, all terms must be present in it. In this respect, proximity ranking shares the behavior exhibited until recently by many commercial search engines. When applied to the document collection of Table 2.1 to rank the collection with respect to the query ⟨“you”, “sir”⟩, proximity ranking assigns a score of 0.33 to document 1, a score of 0.53 to document 3, and a score of 0 to the remaining documents.

When applied to rank the same collection with respect to the query ⟨“quarrel”,“sir”⟩, the method assigns scores of 0.50 to documents 1 and 2, and a score of 0.00 to documents 3 to 5. Unlike cosine similarity, the second occurrence of “sir” in document 2 does not contribute to the document’s score. The frequency of individual terms is not a factor in proximity ranking; rather, the frequency and proximity of their co-occurrence are factors. It is conceivable that a document could include many matches to all the terms but contain only a single cover, with the query terms clustered into discrete groups.

2.2.3 Boolean Retrieval

Apart from the implicit Boolean filters applied by Web search engines, explicit support for Boolean queries is important in specific application areas such as digital libraries and the legal domain. In contrast to ranked retrieval, Boolean retrieval returns sets of documents rather than ranked lists. Under the Boolean retrieval model, a term t is considered to specify the set of documents containing it. The standard Boolean operators (AND, OR, and NOT) are used to construct Boolean queries, which are interpreted as operations over these sets, as follows:

A AND B	intersection of A and B ($A \cap B$)
A OR B	union of A and B ($A \cup B$)
NOT A	complement of A with respect to the document collection ($\bar{A}$)

where A and B are terms or other Boolean queries. For example, over the collection in Table 2.1, the query

(“quarrel” OR “sir”) AND “you”

specifies the set $\{1, 3\}$, whereas the query

(“quarrel” OR “sir”) AND NOT “you”

specifies the set $\{2, 5\}$.

Our algorithm for solving Boolean queries is another variant of the phrase searching algorithm of Figure 2.2 and the cover finding algorithm of Figure 2.10. The algorithm locates *candidate solutions* to a Boolean query where each candidate solution represents a *range* of documents that together satisfy the Boolean query, such that no smaller range of documents contained within it also satisfies the query. When the range represented by a candidate solution has a length of 1, this single document satisfies the query and should be included in the result set.

The same overall method of operation appears in both of the previous algorithms. In the phrase search algorithm, lines 1–6 identify a range containing all the terms in order, such that no smaller range contained within it also contains all the terms in order. In the cover finding algorithm, lines 1–4 similarly locate all the terms as close together as possible. In both algorithms an additional constraint is then applied.

To simplify our definition of our Boolean search algorithm, we define two functions that operate over Boolean queries, extending the **nextDoc** and **prevDoc** methods of schema-dependent inverted indices.

$\mathbf{docRight}(Q, u)$ —	end point of the first candidate solution to Q starting after document u
$\mathbf{docLeft}(Q, v)$ —	start point of the last candidate solution to Q ending before document v

For terms we define:

$$\begin{aligned}\mathbf{docRight}(t, u) &\equiv \mathbf{nextDoc}(t, u)\\ \mathbf{docLeft}(t, v) &\equiv \mathbf{prevDoc}(t, v)\end{aligned}$$

and for the AND and OR operators we define:

$$\begin{aligned}\mathbf{docRight}(A \textbf{ AND } B, u) &\equiv \max(\mathbf{docRight}(A, u), \mathbf{docRight}(B, u))\\ \mathbf{docLeft}(A \textbf{ AND } B, v) &\equiv \min(\mathbf{docLeft}(A, v), \mathbf{docLeft}(B, v))\\ \mathbf{docRight}(A \textbf{ OR } B, u) &\equiv \min(\mathbf{docRight}(A, u), \mathbf{docRight}(B, u))\\ \mathbf{docLeft}(A \textbf{ OR } B, v) &\equiv \max(\mathbf{docLeft}(A, v), \mathbf{docLeft}(B, v))\end{aligned}$$

To determine the result for a given query, these definitions are applied recursively. For example:

$$\begin{aligned}&\mathbf{docRight}((\text{“quarrel” OR “sir”}) \text{ AND “you”}, 1)\\ &\quad\equiv \max(\mathbf{docRight}(\text{“quarrel” OR “sir”}, 1), \mathbf{docRight}(\text{“you”}, 1))\\ &\quad\equiv \max(\min(\mathbf{docRight}(\text{“quarrel”}, 1), \mathbf{docRight}(\text{“sir”}, 1)), \mathbf{nextDoc}(\text{“you”}, 1))\\ &\quad\equiv \max(\min(\mathbf{nextDoc}(\text{“quarrel”}, 1), \mathbf{nextDoc}(\text{“sir”}, 1)), 3)\\ &\quad\equiv \max(\min(2, 2), 3)\\ &\quad\equiv 3\end{aligned}$$

$$\begin{aligned}&\mathbf{docLeft}((\text{“quarrel” OR “sir”}) \text{ AND “you”}, 4)\\ &\quad\equiv \min(\mathbf{docLeft}(\text{“quarrel” OR “sir”}, 4), \mathbf{docLeft}(\text{“you”}, 4))\\ &\quad\equiv \min(\max(\mathbf{docLeft}(\text{“quarrel”}, 4), \mathbf{docLeft}(\text{“sir”}, 4)), \mathbf{prevDoc}(\text{“you”}, 4))\\ &\quad\equiv \min(\max(\mathbf{prevDoc}(\text{“quarrel”}, 4), \mathbf{prevDoc}(\text{“sir”}, 4)), 3)\\ &\quad\equiv \min(\max(2, 3), 3)\\ &\quad\equiv 3\end{aligned}$$

```
nextSolution ($Q$, $position$) ≡
    $v \leftarrow$ docRight($Q, position$)
    if $v = \infty$ then
        return $\infty$
    $u \leftarrow$ docLeft($Q, v + 1$)
    if $u = v$ then
        return $u$
    else
        return nextSolution ($Q$, $v$)
```

Figure 2.12 Function to locate the next solution to the Boolean query Q after a given position. The function **nextSolution** calls **docRight** and **docLeft** to generate a candidate solution. These functions make recursive calls that depend on the structure of the query.

Definitions for the NOT operator are more problematic, and we ignore the operator until after we present the main algorithm.

Figure 2.12 presents the **nextSolution** function, which locates the next solution to a Boolean query after a given position. The function calls **docRight** and **docLeft** to generate a candidate solution. Just after line 4, the interval $[u,v]$ contains this candidate solution. If the candidate solution consists of a single document, it is returned. Otherwise, the function makes a recursive call. Given this function, all solutions to Boolean query Q may be generated by the following:

```
$u \leftarrow -\infty$
while $u < \infty$ do
    $u \leftarrow$ nextSolution(Q, $u$)
    if $u < \infty$ then
        report docid($u$)
```

Using a galloping search implementation of nextDoc and prevDoc, the time complexity of this algorithm is $O(n \cdot l \cdot \log(L/l))$, where n is the number of terms in the query. If a docid or frequency index is used, and positional information is not recorded in the index, l and L represent the lengths of the shortest and longest postings lists of the terms in the query as measured by the number of documents. The reasoning required to demonstrate this time complexity is similar to that of our phrase search algorithm and proximity ranking algorithm. When considered in terms of the number of candidate solutions κ, which reflects the adaptive nature of the algorithm, the time complexity becomes $O(n \cdot \kappa \cdot \log(L/\kappa))$. Note that the call to the **docLeft** method in line 4 of the algorithm can be avoided (see Exercise 2.9), but it helps us to analyze the complexity of the algorithm, by providing a clear definition of a candidate solution.

We ignored the NOT operator in our definitions of **docRight** and **docLeft**. Indeed, it is not necessary to implement general versions of these functions in order to implement the NOT operator. Instead, De Morgan's laws may be used to transform a query, moving any NOT operators inward until they are directly associated with query terms:

$$\begin{aligned} \textbf{NOT}\ (A\ \ \textbf{AND}\ \ B) &\equiv \textbf{NOT}\ A\ \ \textbf{OR}\ \ \textbf{NOT}\ B \\ \textbf{NOT}\ (A\ \ \textbf{OR}\ \ B) &\equiv \textbf{NOT}\ A\ \ \textbf{AND}\ \ \textbf{NOT}\ B \end{aligned}$$

For example, the query

“william” AND “shakespeare” AND NOT (“marlowe” OR “bacon”)

would be transformed into

“william” AND “shakespeare” AND (NOT “marlowe” AND NOT “bacon”) .

This transformation does not change the number of AND and OR operators appearing in the query, and hence does not change the number of terms appearing in the query (n). After appropriate application of De Morgan’s laws, we are left with a query containing expressions of the form **NOT** t, where t is a term. In order to process queries containing expressions of this form, we require corresponding definitions of **docRight** and **docLeft**. It is possible to write these definitions in terms of **nextDoc** and **prevDoc**.

```
docRight(NOT $t, u$) $\equiv$
    $u' \leftarrow$ nextDoc($t, u$)
    while $u' = u + 1$ do
        $u \leftarrow u'$
        $u' \leftarrow$ nextDoc($t, u$)
    return $u + 1$
```

Unfortunately, this approach introduces potential inefficiencies. Although this definition will exhibit acceptable performance when few documents contain the term t, it may exhibit unacceptable performance when most documents contain t, essentially reverting to the linear scan of the postings list that we avoided by introducing galloping search. Moreover, the equivalent implementation of **docLeft**(**NOT** t, v) requires a scan *backward* through the postings list, violating the requirement necessary to realize the benefits of galloping search.

Instead, we may implement the NOT operator directly over the data structures described in Section 2.1.2, extending the methods supported by our inverted index with explicit methods for **nextDoc**(**NOT** t, u) and **prevDoc**(**NOT** t, v). We leave the details for Exercise 2.5.

2.3 Evaluation

Our presentation of both cosine similarity and proximity ranking relies heavily on intuition. We appeal to intuition to justify the representation of documents and queries as vectors, to justify the determination of similarity by comparing angles, and to justify the assignment of higher weights when terms appear more frequently or closer together. This reliance on intuition can be accepted only when the methods are effective in practice. Moreover, an implementation of a retrieval method must be efficient enough to compute the results of a typical query in adequate time to satisfy the user, and possible trade-offs between efficiency and effectiveness must be considered. A user may not wish to wait for a longer period of time — additional seconds or even minutes — in order to receive a result that is only slightly better than a result she could have received immediately.

2.3.1 Recall and Precision

Measuring the effectiveness of a retrieval method depends on human assessments of relevance. In some cases, it might be possible to infer these assessments implicitly from user behavior. For example, if a user clicks on a result and then quickly backtracks to the result page, we might infer a negative assessment. Nonetheless, most published information retrieval experiments are based on manual assessments created explicitly for experimental purposes, such as the assessments for the TREC experiments described in Chapter 1. These assessments are often binary — an assessor reads the document and judges it *relevant* or *not relevant* with respect to a topic. TREC experiments generally use these binary judgments, with a document being judged relevant if any part of it is relevant.

For example, given the information need described by TREC topic 426 (Figure 1.8 on page 25), a user might formulate the Boolean query

(("law" AND "enforcement") OR "police") AND ("dog" OR "dogs").

Running this query over the TREC45 collection produces a set of 881 documents, representing 0.17% of the half-million documents in the collection.

In order to determine the effectiveness of a Boolean query such as this one, we compare two sets: (1) the set of documents returned by the query, *Res*, and (2) the set of relevant documents for the topic contained in the collection, *Rel*. From these two sets we may then compute two standard effectiveness measures: *recall* and *precision*.

$$\text{recall} = \frac{|Rel \cap Res|}{|Rel|} \tag{2.16}$$

$$\text{precision} = \frac{|Rel \cap Res|}{|Res|} \tag{2.17}$$

In a nutshell, recall indicates the fraction of relevant documents that appears in the result set, whereas precision indicates the fraction of the result set that is relevant.

According to official NIST judgments, there are 202 relevant documents in the TREC45 test collection for topic 426. Our query returns 167 of these documents, giving a precision of 0.190 and a recall of 0.827. A user may find this result acceptable. Just 35 relevant documents are outside the result set. However, in order to find a relevant document, the user must read an average of 4.28 documents that are not relevant.

You will sometimes see recall and precision combined into a single value known as an *F-measure*. The simplest F-measure, F_1, is the harmonic mean of recall and precision:

$$F_1 \;=\; \frac{2}{\frac{1}{R} + \frac{1}{P}} \;=\; \frac{2 \cdot R \cdot P}{R + P}\,, \tag{2.18}$$

where R represents recall and P represents precision. In comparison with the arithmetic mean $((R+P)/2)$, the harmonic mean enforces a balance between recall and precision. For example, if we return the entire collection as the result of a query, recall will be 1 but precision will almost always be close to 0. The arithmetic mean gives a value greater than 0.5 for such a result. In contrast, the harmonic mean gives a value of $2P/(1+P)$, which will be close to 0 if P is close to 0.

This formula may be generalized through a *weighted harmonic mean* to allow greater emphasis to be placed on either precision or recall,

$$\frac{1}{\alpha \frac{1}{R} + (1-\alpha)\frac{1}{P}}\,. \tag{2.19}$$

where $0 \leq \alpha \leq 1$. For $\alpha = 0$, the measure is equivalent to precision. For $\alpha = 1$, it is equivalent to recall. For $\alpha = 0.5$ it is equivalent to Equation 2.18. Following standard practice (van Rijsbergen, 1979, Chapter 7), we set $\alpha = 1/(\beta^2+1)$ and define the F-measure as

$$F_\beta \;=\; \frac{(\beta^2+1) \cdot R \cdot P}{\beta^2 \cdot R + P}\,, \tag{2.20}$$

where β may be any real number. Thus, F_0 is recall and F_∞ is precision. Values of $|\beta| < 1$ place emphasis on recall; values of $|\beta| > 1$ place emphasis on precision.

2.3.2 Effectiveness Measures for Ranked Retrieval

If the user is interested in reading only one or two relevant documents, ranked retrieval may provide a more useful result than Boolean retrieval. To extend our notions of recall and precision to the ordered lists returned by ranked retrieval algorithms, we consider the top k documents returned by a query, $Res[1..k]$, and define:

$$\text{recall@}k \;=\; \frac{|Rel \cap Res[1..k]|}{|Rel|} \tag{2.21}$$

$$\text{precision@}k \;=\; \frac{|Rel \cap Res[1..k]|}{|Res[1..k]|}\,, \tag{2.22}$$

where precision@k is often written as P@k. If we treat the title of topic 426 as a term vector for ranked retrieval, ⟨"law", "enforcement", "dogs"⟩, proximity ranking gives P@10 = 0.400 and recall@10 = 0.0198. If the user starts reading from the top of the list, she will find four relevant documents in the top ten.

By definition, recall@k increases monotonically with respect to k. Conversely, if a ranked retrieval method adheres to the Probability Ranking Principle defined in Chapter 1 (i.e., ranking documents in order of decreasing probability of relevance), then P@k will tend to decrease as k increases. For topic 426, proximity ranking gives

k	10	20	50	100	200	1000
P@k	0.400	0.450	0.380	0.230	0.115	0.023
recall@k	0.020	0.045	0.094	0.114	0.114	0.114

Since only 82 documents contain all of the terms in the query, proximity ranking cannot return 200 or 1,000 documents with scores greater than 0. In order to allow comparison with other ranking methods, we compute precision and recall at these values by assuming that the empty lower ranks contain documents that are not relevant. All things considered, a user may be happier with the results of the Boolean query. But of course the comparison is not really fair. The Boolean query contains terms not found in the term vector, and perhaps would require more effort to craft. Using the same term vector, cosine similarity gives

k	10	20	50	100	200	1000
P@k	0.000	0.000	0.000	0.060	0.070	0.051
recall@k	0.000	0.000	0.000	0.030	0.069	0.253

Cosine ranking performs surprisingly poorly on this query, and it is unlikely that a user would be happy with these results.

By varying the value of k, we may trade precision for recall, accepting a lower precision in order to identify more relevant documents and vice versa. An IR experiment might consider k over a range of values, with lower values corresponding to a situation in which the user is interested in a quick answer and is willing to examine only a few results. Higher values correspond to a situation in which the user is interested in discovering as much information as possible about a topic and is willing to explore deep into the result list. The first situation is common in Web search, where the user may examine only the top one or two results before trying something else. The second situation may occur in legal domains, where a case may turn on a single precedent or piece of evidence and a thorough search is required.

As we can see from the example above, even at $k = 1000$ recall may be considerably lower than 100%. We may have to go very deep into the results to increase recall substantially beyond this point. Moreover, because many relevant documents may not contain any of the query terms at

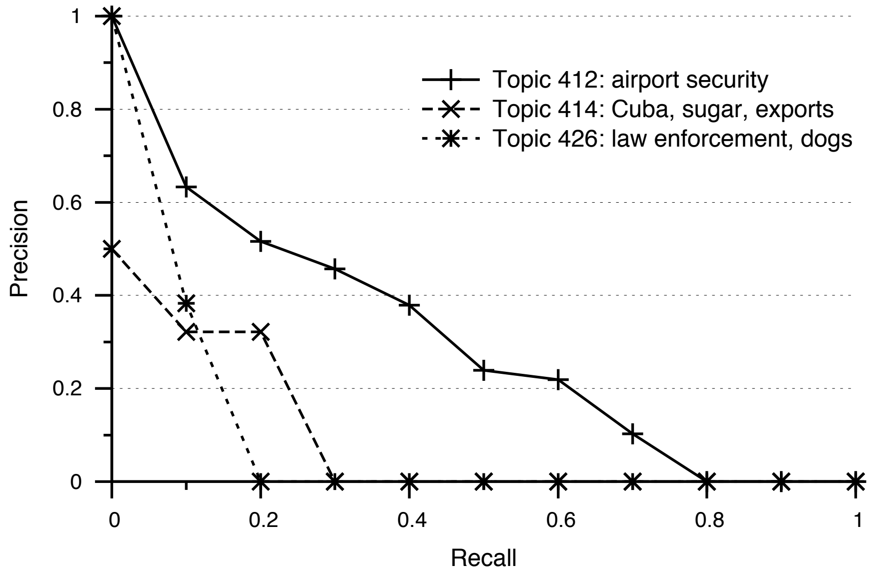


Figure 2.13 Eleven-point interpolated recall-precision curves for three TREC topics over the TREC45 collection. Results were generated with proximity ranking.

all, it is usually not possible to achieve 100% recall without including documents with 0 scores. For simplicity and consistency, information retrieval experiments generally consider only a fixed number of documents, often the top $k = 1000$. At higher levels we simply treat precision as being equal to 0. When conducting an experiment, we pass this value for k as a parameter to the retrieval function, as shown in Figures 2.9 and 2.11.

In order to examine the trade-off between recall and precision, we may plot a *recall-precision curve*. Figure 2.13 shows three examples for proximity ranking. The figure plots curves for topic 426 and two other topics taken from the 1998 TREC adhoc task: topic 412 ("airport security") and topic 414 ("Cuba, sugar, exports"). For 11 recall points, from 0% to 100% by 10% increments, the curve plots the maximum precision achieved at that recall level or higher. The value plotted at 0% recall represents the highest precision achieved at any recall level. Thus, the highest precision achieved for topic 412 is 80%, for topic 414 is 50%, and for topic 426 is 100%. At 20% or higher recall, proximity ranking achieves a precision of up to 57% for topic 412, 32% for topic 414, but 0% for topic 426. This technique of taking the maximum precision achieved at or above a given recall level is called *interpolated precision*. Interpolation has the pleasant property of producing monotonically decreasing curves, which may better illustrate the trade-off between recall and precision.

As an indication of effectiveness across the full range of recall values, we may compute an *average precision* value, which we define as follows:

$$\frac{1}{|Rel|} \cdot \sum_{i=1}^{k} \text{relevant}(i) \times \text{P@}i \tag{2.23}$$

where relevant$(i) = 1$ if the document at rank i is relevant (i.e., if $Res[i] \in Rel$) and 0 if it is not. Average precision represents an approximation of the area under a (noninterpolated) recall-precision curve. Over the top one thousand documents returned for topic 426, proximity ranking achieves an average precision of 0.058; cosine similarity achieves an average precision of 0.016.

So far, we have considered effectiveness measures for a single topic only. Naturally, performance on a single topic tells us very little, and a typical IR experiment will involve fifty or more topics. The standard procedure for computing effectiveness measures over a set of topics is to compute the measure on individual topics and then take the arithmetic mean of these values. In the IR research literature, values stated for P@k, recall@k, and other measures, as well as recall-precision curves, generally represent averages over a set of topics. You will rarely see values or plots for individual topics unless authors wish to discuss specific characteristics of these topics. In the case of average precision, its arithmetic mean over a set of topics is explicitly referred to as *mean average precision* or MAP, thus avoiding possible confusion with averaged P@k values.

Partly because it encapsulates system performance over the full range of recall values, and partly because of its prevalence at TREC and other evaluation forums, the reporting of MAP values for retrieval experiments was nearly ubiquitous in the IR literature until a few years ago. Recently, various limitations of MAP have become apparent and other measures have become more widespread, with MAP gradually assuming a secondary role.

Unfortunately, because it is an average of averages, it is difficult to interpret MAP in a way that provides any clear intuition regarding the actual performance that might be experienced by the user. Although a measure such as P@10 provides less information on the overall performance of a system, it does provide a more understandable number. As a result, we report both P@10 and MAP in experiments throughout the book. In Part III, we explore alternative effectiveness measures, comparing them with precision, recall, and MAP.

For simplicity, and to help guarantee consistency with published results, we suggest you do not write your own code to compute effectiveness measures. NIST provides a program, `trec_eval`[1] that computes a vast array of standard measures, including P@k and MAP. The program is the standard tool for computing results reported at TREC. Chris Buckley, the creator and maintainer of `trec_eval`, updates the program regularly, often including new measures as they appear in the literature.

[1] `trec.nist.gov/trec_eval`

Table 2.5 Effectiveness measures for selected retrieval methods discussed in this book.

	TREC45				**GOV2**			
	1998		**1999**		**2004**		**2005**	
Method	**P@10**	**MAP**	**P@10**	**MAP**	**P@10**	**MAP**	**P@10**	**MAP**
Cosine (2.2.1)	0.264	0.126	0.252	0.135	0.120	0.060	0.194	0.092
Proximity (2.2.2)	0.396	0.124	0.370	0.146	0.425	0.173	0.562	0.230
Cosine (raw TF)	0.266	0.106	0.240	0.120	0.298	0.093	0.282	0.097
Cosine (TF docs)	0.342	0.132	0.328	0.154	0.400	0.144	0.466	0.151
BM25 (Ch. 8)	0.424	0.178	0.440	0.205	0.471	0.243	0.534	0.277
LMD (Ch. 9)	0.450	0.193	0.428	0.226	0.484	0.244	0.580	0.293
DFR (Ch. 9)	0.426	0.183	0.446	0.216	0.465	0.248	0.550	0.269

Effectiveness results

Table 2.5 presents MAP and P@10 values for various retrieval methods over our four test collections. The first row provides values for the cosine similarity ranking described in Section 2.2.1. The second row provides values for the proximity ranking method described in Section 2.2.2.

As we indicated in Section 2.2.1, a large number of variants of cosine similarity have been explored over the years. The next two lines of the table provide values for two of them. The first of these replaces Equation 2.14 (page 57) with raw TF values, $f_{t,d}$, the number of occurrences of each term. In the case of the TREC45 collection, this change harms performance but substantially improves performance on the GOV2 collection. For the second variant (the fourth row) we omitted both document length normalization and document IDF values (but kept IDF in the query vector). Under this variant we compute a score for a document simply by taking the inner product of this unnormalized document vector and the query vector:

$$\mathit{score}(q,d) \;=\; \sum_{t \in (q \cap d)} q_t \cdot \log\left(\frac{N}{N_t}\right) \cdot (\log(f_{t,d}) + 1)\,. \tag{2.24}$$

Perhaps surprisingly, this change substantially improves performance to roughly the level of proximity ranking.

How can we explain this improvement? The vector space model was introduced and developed at a time when documents were of similar length, generally being short abstracts of books or scientific articles. The idea of representing a document as a vector represents the fundamental inspiration underlying the model. Once we think of a document as a vector, it is not difficult to take the next step and imagine applying standard mathematical operations to these vectors, including addition, normalization, inner product, and cosine similarity. Unfortunately, when it is applied to collections containing documents of different lengths, vector normalization does

Table 2.6 Selected ranking formulae discussed in later chapters. In these formulae the value q_t represents query term frequency, the number of times term t appears in the query. b and k_1, for BM25, and μ, for LMD, are free parameters set to $b = 0.75$, $k_1 = 1.2$, and $\mu = 1000$ in our experiments.

Method	**Formula**
BM25 (Ch. 8)	$\sum_{t \in q} q_t \cdot (f_{t,d} \cdot (k_1 + 1))/(k_1 \cdot ((1 - b) + b \cdot (l_d/l_{avg})) + f_{t,d}) \cdot \log(N/N_t)$
LMD (Ch. 9)	$\sum_{t \in q} q_t \cdot (\log(f_{t,d} + \mu \cdot l_t/\mathcal{C}) - \log(l_d + \mu))$
DFR (Ch. 9)	$\sum_{t \in q} q_t \cdot (\log(1 + l_t/N) + f'_{t,d} \cdot \log(1 + N/l_t))/(f'_{t,d} + 1)$, where $f'_{t,d} = f_{t,d} \cdot \log(1 + l_{avg}/l_d)$

not successfully adjust for these varying lengths. For this reason, by the early 1990s this last variant of the vector space model had become standard.

The inclusion of the vector space model in this chapter is due more to history than anything else. Given its long-standing influence, no textbook on information retrieval would be complete without it. In later chapters we will present ranked retrieval methods with different theoretical underpinnings. The final three rows of Table 2.5 provide results for three of these methods: a method based on probabilistic models (BM25), a method based on language modeling (LMD), and a method based on divergence from randomness (DFR). The formulae for these methods are given in Table 2.6. As can be seen from the table, all three of these methods depend only on the simple features discussed at the start of Section 2.2. All outperform both cosine similarity and proximity ranking.

Figure 2.14 plots 11-point interpolated recall-precision curves for the 1998 TREC adhoc task, corresponding to the fourth and fifth columns of Table 2.5. The LM and DFR methods appear to slightly outperform BM25, which in turn outperforms proximity ranking and the TF docs version of cosine similarity.

2.3.3 Building a Test Collection

Measuring the effectiveness of a retrieval method depends on human assessments of relevance. Given a topic and a set of documents, an assessor reads each document and makes a decision: Is it relevant or not? Once a topic is well understood, assessment can proceed surprisingly quickly. Once up to speed, an assessor might be able to make a judgment in 10 seconds or less. Unfortunately, even at a rate of one judgment every ten seconds, an assessor working eight-hour days with breaks and holidays would spend nearly a year judging a single topic over the half-million documents in TREC45, and her entire career judging a topic over the 25 million documents in GOV2.

Given the difficulty of judging a topic over the entire collection, TREC and other retrieval efforts depend on a technique known as *pooling* to limit the number of judgments required. The standard TREC experimental procedure is to accept one or more experimental *runs* from each group participating in an adhoc task. Each of these runs consists of the top 1,000 or 10,000

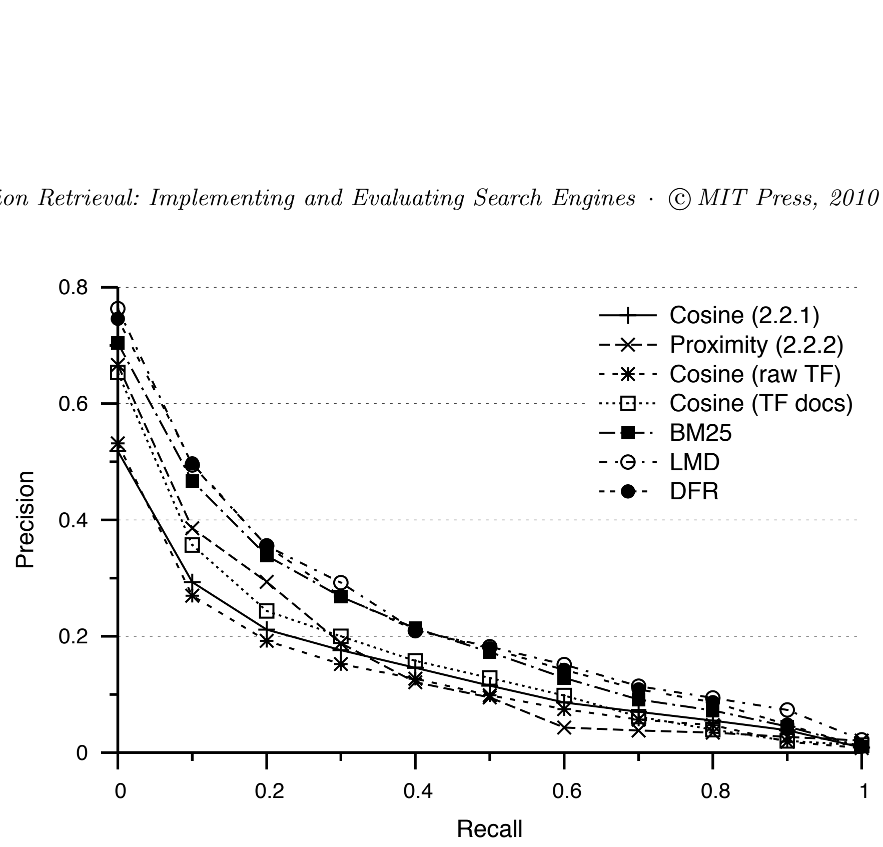


Figure 2.14 11-point interpolated recall-precision curves for various basic retrieval methods on the 1999 TREC adhoc task (topics 401–450, TREC45 document collection).

documents for each topic. Under the pooling method, the top 100 or so documents from each run are then *pooled* into a single set and presented to assessors in random order for judging. Even with dozens of participants the size of this pool may be less than a thousand documents per topic, because the same document may be returned by multiple systems. Even if assessment is done very carefully, with each judgment taking several minutes on average, it is possible for an assessor to work through this pool in less than a week. Often, less time is required.

The goal of the pooling method is to reduce the amount of assessment effort while retaining the ability to make reasonable estimates of precision and recall. For runs contributing to a pool, the values for P@k and recall@k are accurate at least down to the pool depth. Beyond this depth many documents will still be judged, because other runs will have ranked these documents above the pool depth. MAP is still calculated over the entire 1,000 or 10,000 documents in each run, with *unjudged documents treated as not relevant*. Because the top documents have the greatest impact on MAP, estimates made in this way are still acceptably accurate.

Armed with the pooling method, the creation of a test collection at TREC and similar evaluation efforts generally proceed as follows:

1. Obtain an appropriate document set either from public sources or through negotiation with its owners.
2. Develop at least 50 topics, the minimum generally considered acceptable for a meaningful evaluation.

3. Release the topics to the track participants and receive their experimental runs.
4. Create the pools, judge the topics, and return the results to the participants.

When developing a topic, a preliminary test of the topic on a standard IR system should reveal a reasonable mixture of relevant and non-relevant documents in the top ranks. With many relevant documents it may be too easy for systems to achieve a MAP value close to 1.0, making it difficult to distinguish one system from another. With few relevant documents, many systems will fail to achieve a MAP value above 0.0, again making it difficult to distinguish between them. Borrowing our criteria from Goldilocks in *The Story of the Three Bears*, we want not too many, not too few, but just right.

Ideally, a test collection would be reusable. A researcher developing a new retrieval method could use the documents, topics, and judgments to determine how the new method compares with previous methods. Although the pooling method allows us to create a test collection in less than a lifetime, you may have some legitimate concern regarding its ability to create such reusable collections. A truly novel retrieval method might surface many relevant documents that did not form part of the original pool and are not judged. If unjudged documents are treated as nonrelevant for the purposes of computing evaluation measures, this novel method might receive an unduly harsh score. Tragically, it might even be discarded by the researcher and lost to science forever. Alternative effectiveness measures to address this and related issues are covered in Chapter 12.

2.3.4 Efficiency Measures

In addition to evaluating a system's retrieval effectiveness, it is often necessary to evaluate its efficiency because it affects the cost of running the system and the happiness of its users. From a user's perspective the only efficiency measure of interest is *response time*, the time between entering a query and receiving the results. *Throughput*, the average number of queries processed in a given period of time, is primarily of interest to search engine operators, particularly when the search engine is shared by many users and must cope with thousands of queries per second. Adequate resources must be available to cope with the query load generated by these users without compromising the response time experienced by each. Unacceptably long response times may lead to unhappy users and, as a consequence, fewer users.

A realistic measurement of throughput and its trade-off against response time requires a detailed simulation of query load. For the purposes of making simple efficiency measurements, we may focus on response time and consider the performance seen by a single user issuing one query at a time on an otherwise unburdened search engine. Chapter 13 provides a more detailed discussion of throughput along with a broader discussion of efficiency.

A simple but reasonable procedure for measuring response time is to execute a full query set, capturing the start time from the operating system at a well-defined point just before the first query is issued, and the end time just after the last result is generated. The time to execute the full set is then divided by the number of queries to give an *average response time* per query.

Table 2.7 Average time per query for the Wumpus implementation of Okapi BM25 (Chapter 8), using two different index types and four different query sets.

	TREC45		GOV2	
Index type	**1998**	**1999**	**2004**	**2005**
Schema-independent index	61 ms	57 ms	1686 ms	4763 ms
Frequency index	41 ms	41 ms	204 ms	202 ms

Results may be discarded as they are generated, rather than stored to disk or sent over a network, because the overhead of these activities can dominate the response times, particularly with small collections.

Before executing a query set, the IR system should be restarted or reset to clear any information precomputed and stored in memory from previous queries, and the operating system's I/O cache must be flushed. To increase the accuracy of the measurements, the query set may be executed multiple times, with the system reset each time and an average of the measured execution times used to compute the average response time.

An an example, Table 2.7 compares the average response time of a schema-independent index versus a frequency index, using the Wumpus implementation of the Okapi BM25 ranking function (shown in Table 2.6). We use Okapi BM25 for this example, because the Wumpus implementation of this ranking function has been explicitly tuned for efficiency.

The efficiency benefits of using the frequency index are obvious, particularly on the larger GOV2 collection. The use of a schema-independent index requires the computation at run-time of document and term statistics that are precomputed in the frequency index. To a user, a 202 ms response time would seem instantaneous, whereas a 4.7 sec response time would be a noticeable lag. However, with a frequency index it is not possible to perform phrase searches or to apply ranking functions to elements other than documents.

The efficiency measurements shown in the table, as well as others throughout the book, were made on a rack-mounted server based on an AMD Opteron processor (2.8 GHz) with 2 GB of RAM. A detailed performance overview of the computer system is in Appendix A.

2.4 Summary

This chapter has covered a broad range of topics, often in considerable detail. The key points include the following:

- We view an inverted index as an abstract data type that may be accessed through the methods and definitions summarized in Table 2.4 (page 52). There are four important variants — docid indices, frequency indices, positional indices, and schema-independent indices — that differ in the type and format of the information they store.

- Many retrieval algorithms — such as the phrase searching, proximity ranking, and Boolean query processing algorithms presented in this chapter — may be efficiently implemented using galloping search. These algorithms are adaptive in the sense that their time complexity depends on characteristics of the data, such as the number of candidate phrases.
- Both ranked retrieval and Boolean filters play important roles in current IR systems. Reasonable methods for ranked retrieval may be based on simple document and term statistics, such as term frequency (TF), inverse document frequency (IDF), and term proximity. The well-known cosine similarity measure represents documents and queries as vectors and ranks documents according to the cosine of the angle between them and the query vector.
- Recall and precision are two widely used effectiveness measures. A trade-off frequently exists between them, such that increasing recall leads to a corresponding decrease in precision. Mean average precision (MAP) represents a standard method for summarizing the effectiveness of an IR system over a broad range of recall levels. MAP and P@10 are the principal effectiveness measures reported throughout the book.
- Response time represents the efficiency of an IR system as experienced by a user. We may make reasonable estimates of minimum response time by processing queries sequentially, reading them one at a time and reporting the results of one query before starting the next query.

2.5 Further Reading

Inverted indices have long been the standard data structure underlying the implementation of IR systems (Faloutsos, 1985; Knuth, 1973, pages 552–554). Other data structures have been proposed, generally with the intention of providing more efficient support for specific retrieval operations. However, none of these data structures provide the flexibility and generality of inverted indices, and they have mostly fallen out of use.

Signature files were long viewed as an important competitor to inverted indices, particularly when disk and memory were at a premium (Faloutsos, 1985; Faloutsos and Christodoulakis, 1984; Zobel et al., 1998). Signature files are intended to provide efficient support for Boolean queries by quickly eliminating documents that do not match the query. They provide one method for implementing the filtering step of the two-step retrieval process described on page 53. Unfortunately, false matches can be reported by signature files, and they cannot be easily extended to support phrase queries and ranked retrieval.

A suffix tree (Weiner, 1973) is a search tree in which every path from the root to a leaf corresponds to a unique suffix in a text collection. A suffix array (Gonnet, 1987; Manber and Myers, 1990) is an array of pointers to unique suffixes in the collection that is sorted in lexicographical order. Both suffix trees and suffix arrays are intended to support efficient phrase searching, as well as operations such as lexicographical range searching and structural retrieval (Gonnet et al., 1992). Unfortunately, both data structures provide poor support for ranked retrieval.

Galloping search was first described by Bentley and Yao (1976), who called it "unbounded search". The algorithms presented in this chapter for phrase searching, proximity ranking, and Boolean retrieval may all be viewed as variants of algorithms for computing set operations over sorted lists. The term "galloping" was coined by Demaine et al. (2000), who present and analyze adaptive algorithms for set union, intersection and difference. Other algorithms for computing the intersection of sorted lists are described by Barbay and Kenyon (2002) and by Baeza-Yates (2004). Our abstraction of an inverted index into an ADT accessed through a small number of simple methods is based on Clarke et al. (2000). The proximity ranking algorithm presented in this chapter is a simplified version of the algorithm presented in that paper.

The vector space model was developed into the version presented in this chapter through a long process that can be traced back at least to the work of Luhn in the 1950s (Luhn, 1957, 1958). The long-standing success and influence of the model are largely due to the efforts of Salton and his research group at Cornell, who ultimately made the model a key element of their SMART IR system, the first version of which became operational in the fall of 1964 (Salton, 1968, page 422). Until the early 1990s SMART was one of a small number of platforms available for IR research, and during that period it was widely adopted by researchers outside Salton's group as a foundation for their own work.

By the time of the first TREC experiments in the early 1990s, the vector space model had evolved into a form very close to that presented in this chapter (Buckley et al., 1994). A further development, introduced at the fourth TREC in 1995, was the addition of an explicit adjustment for document length, known as *pivoted document length normalization* (Singhal et al., 1996). This last development has been accepted as an essential part of the vector space model for ranked retrieval but not for other applications, such as clustering or classification. After Salton's death in 1995, development of the SMART system continued under the supervision of Buckley, and it remained a competitive research system more than ten years later (Buckley, 2005).

Latent Semantic Analysis (LSA) is an important and well-known extension of the vector space model (Deerwester et al., 1990) that we do not cover in this book. LSA applies singular value decomposition, from linear algebra, to reduce the dimensionality of the term-vector space. The goal is to reduce the negative impact of synonymy — multiple terms with the same meaning — by merging related words into common dimensions. A related technique, Probabilistic Latent Semantic Analysis (PLSA), approaches the same goal by way of a probabilistic model (Hofmann, 1999). Unfortunately, the widespread application of LSA and PLSA to IR systems has been limited by the difficulty of efficient implementation.

The Spam song (page 38) has had perhaps the greatest influence of any song on computer terminology. The song was introduced in the twelfth episode of the second season of *Monty Python's Flying Circus*, which first aired on December 15, 1970, on BBC Television.

2.6 Exercises

Exercise 2.1 Simplify **next**(“the”, **prev**(“the”, **next** (“the”, $-\infty$))).

Exercise 2.2 Consider the following version of the phrase searching algorithm shown in Figure 2.2 (page 36).

```
nextPhrase2 ($t_1 t_2 ... t_n$, $position$) $\equiv$
    $u \leftarrow$ next($t_1$, $position$)
    $v \leftarrow u$
    for $i \leftarrow 2$ to $n$ do
        $v \leftarrow$ next($t_i$, $v$)
    if $v = \infty$ then
        return $[\infty, \infty]$
    if $v - u = n - 1$ then
        return $[u, v]$
    else
        return nextPhrase2($t_1 t_2 ... t_n$, $v - n$)
```

It makes the same number of calls to **next** as the version shown in the figure but does not make any calls to **prev**. Explain how the algorithm operates. How does it find the next occurrence of a phrase after a given position? When combined with galloping search to find all occurrences of a phrase, would you expect improved efficiency compared to the original version? Explain.

Exercise 2.3 Using the methods of the inverted index ADT, write an algorithm that locates all intervals corresponding to speeches (<SPEECH>...</SPEECH>). Assume the schema-independent indexing shown in Figure 2.1, as illustrated by Figure 1.3.

Exercise 2.4 Using the methods of the inverted index ADT, write an algorithm that locates all speeches by a witch in Shakespeare’s plays. Assume the schema-independent indexing shown in Figure 2.1, as illustrated by Figure 1.3.

Exercise 2.5 Assume we have a schema-independent inverted index for Shakespeare’s plays. We decide to treat each PLAY as a separate document.

(a) Write algorithms to compute the following statistics, using only the **first**, **last**, **next**, and **prev** methods: (i) N_t, the number of documents in the collection containing the term t; (ii) $f_{t,d}$, the number of times term t appears in document d; (iii) l_d, the length of document d; (iv) l_{avg}, average document length; and (v) N, total number of documents.

(b) Write algorithms to implement the **docid**, **offset**, **firstDoc**, **lastDoc**, **nextDoc**, and **prevDoc** methods, using only the **first**, **last**, **next**, and **prev** methods.

For simplicity, you may treat the position of each ⟨PLAY⟩ tag as the document ID for the document starting at that position; there's no requirement for document IDs to be assigned sequentially. For example, *Macbeth* would be assigned docid 745142 (with an initial occurrence of "first witch" at [745142:265, 745142:266]).

Exercise 2.6 When postings lists are stored on disk rather than entirely in memory, the time required by an inverted index method call can depend heavily on the overhead required for accessing the disk. When conducting a search for a phrase such as "Winnie the Pooh", which contains a mixture of frequent and infrequent terms, the first method call for "winnie" (**next**("winnie", $-\infty$)) might read the entire postings list from disk into an array. Further calls would return results from this array using galloping search. Similarly, the postings list for "pooh" could be read into memory during its first method call. On the other hand, the postings list for "the" may not fit into memory in its entirety, and only a very small number of these postings may form part of the target phrase.

Ideally, all of the postings for "the" would not be read from disk. Instead, the methods might read a small part of the postings list at a time, generating multiple disk accesses as the list is traversed and skipping portions of the list as appropriate.

Because each method call for "the" may generate a disk access, it may be preferable to search for the phrase "Winnie the Pooh" by immediately checking for an occurrence of "pooh" (two positions away) after locating each occurrence of "winnie". If "pooh" does not occur in the expected location, we may abandon this occurrence of "winnie" and continue to another occurrence without checking for an occurrence of "the" between the two. Only when we have located an occurrence of "winnie" and "pooh" correctly spaced ("winnie ___ pooh") do we need to make a method call for "the".

To generalize, given a phrase "$t_1 t_2 ... t_n$", a search algorithm might work from the least frequent term to the most frequent term, abandoning a potential phrase once a term is found to be missing and thus minimizing the number of calls for the most frequent terms. Using the basic definitions and methods of our inverted index ADT, complete the details of this algorithm. What is the time complexity of the algorithm in terms of calls to the **next** and **prev** methods? What is the overall time complexity if galloping search is used to implement these methods?

Can the algorithm be considered adaptive? (*Hint:* Consider a search in the "hello world" collection on page 38. How many method calls must be made?)

Exercise 2.7 Coordination level is the number of terms from a vector $\langle t_1, ..., t_n \rangle$ that appear in a document. Coordination level may range from 1 to n. Using the methods of our inverted index ADT, write an algorithm that ranks documents according to their coordination level.

Exercise 2.8 Demonstrate that the term vector $\langle t_1, ..., t_n \rangle$ may have at most $n \cdot l$ covers (see Section 2.2.2), where l is the length of the shortest postings list for the terms in the vector.

Exercise 2.9 Taking a hint from Exercise 2.2, design a version of the algorithm shown in Figure 2.10 (page 61) that does not make a call to the **docLeft** method.

Exercise 2.10 If $\alpha = 1/(\beta^2 + 1)$, show that Equations 2.19 and 2.20 are equivalent.

Exercise 2.11 (project exercise) Implement the in-memory array-based version of inverted indices presented in Section 2.1.2, including all three versions of the **next** method: binary search, sequential scan and galloping search. Using your implementation of inverted indices, implement the phrase searching algorithm of Section 2.1.1.

To test your phrase searching implementation, we suggest a corpus 256MB or larger that fits comfortably in the main memory of your computer. The corpus created in Exercise 1.9 (or a subset) would be suitable. Select at least 10,000 phrases of various lengths from your corpus and verify that your implementation successfully locates these phrases. Your selection should include short phrases of length 2–3, longer phrases of length 4–10, and very long phrases with length 100 or greater. Include phrases containing both frequent and infrequent terms. At least half of the phrases should contain a least one very common term, such as an article or a preposition.

Following the guidelines of Section 2.3.4, compare the efficiency of phrase searching, using your three versions of the **next** method. Compute average response times. Plot response time against phrase length for all three versions. For clarity, you may need to plot the results for linear scan separately from those of the other two methods.

Select another set of phrases, all with length 2. Include phrases containing combinations of frequent and infrequent terms. Plot response time against L (the length of the longest postings list) for all three versions. Plot response time against l (the length of the shortest postings list) for all three versions.

Optional extension: Implement the phrase searching algorithm described in Exercise 2.6. Repeat the performance measurements using this new implementation.

Exercise 2.12 (project exercise) Implement cosine similarity ranking, as described in Section 2.2.1. Test your implementation using the test collection developed in Exercise 2.13 or with any other available collection, such as a TREC collection.

Exercise 2.13 (project exercise) As a class project, undertake a TREC-style adhoc retrieval experiment, developing a test collection based on Wikipedia. Continuing from Exercises 1.9 and 1.10, each student should contribute enough topics to make a combined set of 50 topics or more. Each student should implement their own retrieval system, perhaps by starting with an open-source IR system and extending it with techniques from later chapters, or with techniques of their own invention. Each student should then run the titles of the topics as queries to their system. Pool and judge the results, with each student judging the topics they created. The design and implementation of an interactive judging interface might be undertaken as a sub-project by a group of interested students. Use `trec_eval` to compare runs and techniques.

2.7 Bibliography

Baeza-Yates, R. (2004). A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching*, pages 400–408. Istanbul, Turkey.

Barbay, J., and Kenyon, C. (2002). Adaptive intersection and t-threshold problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 390–399. San Francisco, California.

Bentley, J. L., and Yao, A. C. C. (1976). An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87.

Buckley, C. (2005). The SMART project at TREC. In Voorhees, E. M., and Harman, D. K., editors, *TREC — Experiment and Evaluation in Information Retrieval*, chapter 13, pages 301–320. Cambridge, Massachusetts: MIT Press.

Buckley, C., Salton, G., Allan, J., and Singhal, A. (1994). Automatic query expansion using SMART: TREC 3. In *Proceedings of the 3rd Text REtrieval Conference*. Gaithersburg, Maryland.

Clarke, C. L. A., Cormack, G. V., and Tudhope, E. A. (2000). Relevance ranking for one to three term queries. *Information Processing & Management*, 36(2):291–311.

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407.

Demaine, E. D., López-Ortiz, A., and Munro, J. I. (2000). Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 743–752. San Francisco, California.

Faloutsos, C. (1985). Access methods for text. *ACM Computing Surveys*, 17(1):49–74.

Faloutsos, C., and Christodoulakis, S. (1984). Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288.

Gonnet, G. H. (1987). PAT *3.1 — An Efficient Text Searching System — User's Manual*. University of Waterloo, Canada.

Gonnet, G. H., Baeza-Yates, R. A., and Snider, T. (1992). New indices for text — PAT trees and PAT arrays. In Frakes, W. B., and Baeza-Yates, R., editors, *Information Retrieval — Data Structures and Algorithms*, chapter 5, pages 66–82. Englewood Cliffs, New Jersey: Prentice Hall.

Hofmann, T. (1999). Probabilistic latent semantic indexing. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 50–57. Berkeley, California.

Knuth, D. E. (1973). *The Art of Computer Programming*, volume 3. Reading, Massachusetts: Addison-Wesley.

Luhn, H. P. (1957). A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of Research and Development*, 1(4):309–317.

Luhn, H. P. (1958). The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2(2):159–165.

Manber, U., and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327. San Francisco, California.

Salton, G. (1968). *Automatic Information Organziation and Retrieval*. New York: McGraw-Hill.

Singhal, A., Salton, G., Mitra, M., and Buckley, C. (1996). Document length normalization. *Information Processing & Management*, 32(5):619–633.

van Rijsbergen, C. J. (1979). *Information Retrieval* (2nd ed.). London, England: Butterworths.

Weiner, P. (1973). Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11. Iowa City, Iowa.

Zobel, J., Moffat, A., and Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490.

4 Static Inverted Indices

In this chapter we describe a set of index structures that are suitable for supporting search queries of the type outlined in Chapter 2. We restrict ourselves to the case of static text collections. That is, we assume that we are building an index for a collection that never changes. Index update strategies for dynamic text collections, in which documents can be added to and removed from the collection, are the topic of Chapter 7.

For performance reasons, it may be desirable to keep the index for a text collection completely in main memory. However, in many applications this is not feasible. In file system search, for example, a full-text index for all data stored in the file system can easily consume several gigabytes. Since users will rarely be willing to dedicate the most part of their available memory resources to the search system, it is not possible to keep the entire index in RAM. And even for dedicated index servers used in Web search engines it might be economically sensible to store large portions of the index on disk instead of in RAM, simply because disk space is so much cheaper than RAM. For example, while we are writing this, a gigabyte of RAM costs around \$40 (U.S.), whereas a gigabyte of hard drive space costs only about \$0.20 (U.S.). This factor-200 price difference, however, is not reflected by the relative performance of the two storage media. For typical index operations, an in-memory index is usually between 10 and 20 times faster than an on-disk index. Hence, when building two equally priced retrieval systems, one storing its index data on disk, the other storing them in main memory, the disk-based system may actually be faster than the RAM-based one (see Bender et al. (2007) for a more in-depth discussion of this and related issues).

The general assumption that will guide us throughout this chapter is that main memory is a scarce resource, either because the search engine has to share it with other processes running on the same system or because it is more economical to store data on disk than in RAM. In our discussion of data structures for inverted indices, we focus on hybrid organizations, in which some parts of the index are kept in main memory, while the majority of the data is stored on disk. We examine a variety of data structures for different parts of the search engine and evaluate their performance through a number of experiments. A performance summary of the computer system used to conduct these experiments can be found in the appendix.

4.1 Index Components and Index Life Cycle

When we discuss the various aspects of inverted indices in this chapter, we look at them from two perspectives: the structural perspective, in which we divide the system into its components and examine aspects of an individual component of the index (e.g., an individual postings list); and the operational perspective, in which we look at different phases in the life cycle of an inverted index and discuss the essential index operations that are carried out in each phase (e.g., processing a search query).

As already mentioned in Chapter 2, the two principal components of an inverted index are the *dictionary* and the *postings lists*. For each term in the text collection, there is a postings list that contains information about the term's occurrences in the collection. The information found in these postings lists is used by the system to process search queries. The dictionary serves as a lookup data structure on top of the postings lists. For every query term in an incoming search query, the search engine first needs to locate the term's postings list before it can start processing the query. It is the job of the dictionary to provide this mapping from terms to the location of their postings lists in the index.

In addition to dictionary and postings lists, search engines often employ various other data structures. Many engines, for instance, maintain a *document map* that, for each document in the index, contains document-specific information, such as the document's URL, its length, PageRank (see Section 15.3.1), and other data. The implementation of these data structures, however, is mostly straightforward and does not require any special attention.

The life cycle of a static inverted index, built for a never-changing text collection, consists of two distinct phases (for a dynamic index the two phases coincide):

1. *Index construction*: The text collection is processed sequentially, one token at a time, and a postings list is built for each term in the collection in an incremental fashion.
2. *Query processing*: The information stored in the index that was built in phase 1 is used to process search queries.

Phase 1 is generally referred to as *indexing time*, while phase 2 is referred to as *query time*. In many respects these two phases are complementary; by performing additional work at indexing time (e.g., precomputing score contributions — see Section 5.1.3), less work needs to be done at query time. In general, however, the two phases are quite different from one another and usually require different sets of algorithms and data structures. Even for subcomponents of the index that are shared by the two phases, such as the search engine's dictionary data structure, it is not uncommon that the specific implementation utilized during index construction is different from the one used at query time.

The flow of this chapter is mainly defined by our bifocal perspective on inverted indices. In the first part of the chapter (Sections 4.2–4.4), we are primarily concerned with the query-time aspects of dictionary and postings lists, looking for data structures that are most suitable for supporting efficient index access and query processing. In the second part (Section 4.5) we focus on aspects of the index construction process and discuss how we can efficiently build the data structures outlined in the first part. We also discuss how the organization of the dictionary and the postings lists needs to be different from the one suggested in the first part of the chapter, if we want to maximize their performance at indexing time.

For the sake of simplicity, we assume throughout this chapter that we are dealing exclusively with *schema-independent* indices. Other types of inverted indices, however, are similar to the schema-independent variant, and the methods discussed in this chapter apply to all of them (see Section 2.1.3 for a list of different types of inverted indices).

4.2 The Dictionary

The *dictionary* is the central data structure that is used to manage the set of terms found in a text collection. It provides a mapping from the set of index terms to the locations of their postings lists. At query time, locating the query terms' postings lists in the index is one of the first operations performed when processing an incoming keyword query. At indexing time, the dictionary's lookup capability allows the search engine to quickly obtain the memory address of the inverted list for each incoming term and to append a new posting at the end of that list.

Dictionary implementations found in search engines usually support the following set of operations:

1. Insert a new entry for term T.
2. Find and return the entry for term T (if present).
3. Find and return the entries for all terms that start with a given prefix P.

When building an index for a text collection, the search engine performs operations of types 1 and 2 to look up incoming terms in the dictionary and to add postings for these terms to the index. After the index has been built, the search engine can process search queries, performing operations of types 2 and 3 to locate the postings lists for all query terms. Although dictionary operations of type 3 are not strictly necessary, they are a useful feature because they allow the search engine to support *prefix queries* of the form "inform∗", matching all documents containing a term that begins with "inform".

Table 4.1 Index sizes for various index types and three example collections, with and without applying index compression techniques. In each case the first number refers to an index in which each component is stored as a simple 32-bit integer, while the second number refers to an index in which each entry is compressed using a byte-aligned encoding method.

	Shakespeare	**TREC45**	**GOV2**
Number of tokens	1.3×10^6	3.0×10^8	4.4×10^{10}
Number of terms	2.3×10^4	1.2×10^6	4.9×10^7
Dictionary (uncompr.)	0.4 MB	24 MB	1046 MB
Docid index	n/a	578 MB/200 MB	37751 MB/12412 MB
Frequency index	n/a	1110 MB/333 MB	73593 MB/21406 MB
Positional index	n/a	2255 MB/739 MB	245538 MB/78819 MB
Schema-ind. index	5.7 MB/2.7 MB	1190 MB/532 MB	173854 MB/63670 MB

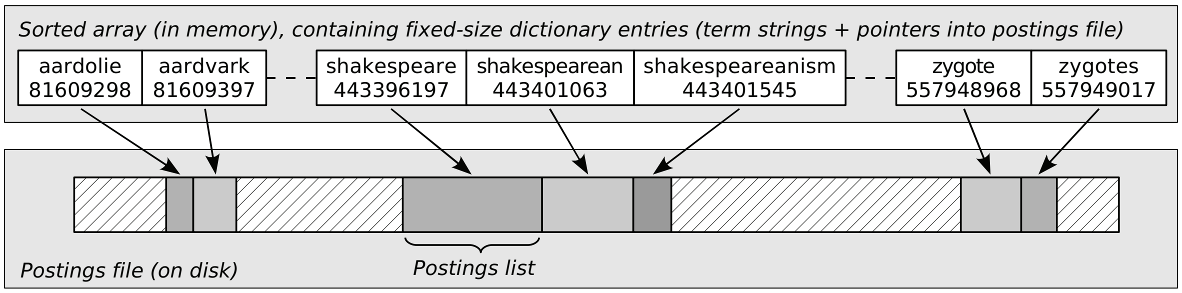


Figure 4.1 Dictionary data structure based on a sorted array (data extracted from a schema-independent index for TREC45). The array contains fixed-size dictionary entries, composed of a zero-terminated string and a pointer into the postings file that indicates the position of the term's postings list.

For a typical natural-language text collection, the dictionary is relatively small compared to the total size of the index. Table 4.1 shows this for the three example collections used in this book. The size of the uncompressed dictionary is only between 0.6% (GOV2) and 7% (Shakespeare) of the size of an uncompressed schema-independent index for the respective collection (the fact that the relative size of the dictionary is smaller for large collections than for small ones follows directly from Zipf's law — see Equation 1.2 on page 16). We therefore assume, at least for now, that the dictionary is small enough to fit completely into main memory.

The two most common ways to realize an in-memory dictionary are:

- A *sort-based* dictionary, in which all terms that appear in the text collection are arranged in a sorted array or in a search tree, in lexicographical (i.e., alphabetical) order, as shown in Figure 4.1. Lookup operations are realized through tree traversal (when using a search tree) or binary search (when using a sorted list).
- A *hash-based* dictionary, in which each index term has a corresponding entry in a hash table. Collisions in the hash table (i.e., two terms are assigned the same hash value) are resolved by means of *chaining* — terms with the same hash value are arranged in a linked list, as shown in Figure 4.2.

Storing the dictionary terms

When implementing the dictionary as a sorted array, it is important that all array entries are of the same size. Otherwise, performing a binary search may be difficult. Unfortunately, this causes some problems. For example, the longest sequence of alphanumeric characters in GOV2 (i.e., the longest term in the collection) is 74,147 bytes long. Obviously, it is not feasible to allocate 74 KB of memory for each term in the dictionary. But even if we ignore such extreme outliers and truncate each term after, say, 20 bytes, we are still wasting precious memory resources. Following the simple tokenization procedure from Section 1.3.2, the average length of a term

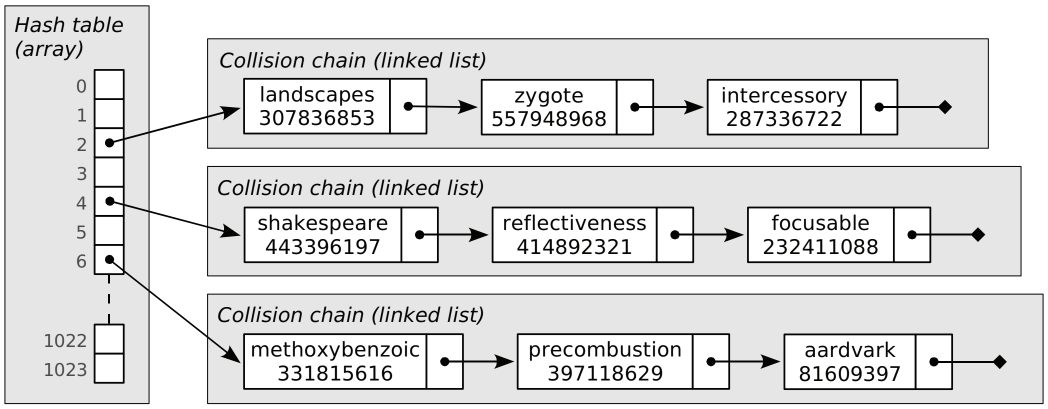


Figure 4.2 Dictionary data structure based on a hash table with $2^{10} = 1024$ entries (data extracted from schema-independent index for TREC45). Terms with the same hash value are arranged in a linked list (*chaining*). Each term descriptor contains the term itself, the position of the term's postings list, and a pointer to the next entry in the linked list.

in GOV2 is 9.2 bytes. Storing each term in a fixed-size memory region of 20 bytes wastes 10.8 bytes per term on average (*internal fragmentation*).

One way to eliminate the internal fragmentation is to not store the index terms themselves in the array, but only pointers to them. For example, the search engine could maintain a *primary* dictionary array, containing 32-bit pointers into a *secondary* array. The secondary array then contains the actual dictionary entries, consisting of the terms themselves and the corresponding pointers into the postings file. This way of organizing the search engine's dictionary data is shown in Figure 4.3. It is sometimes referred to as the *dictionary-as-a-string approach*, because there are no explicit delimiters between two consecutive dictionary entries; the secondary array can be thought of as a long, uninterrupted string.

For the GOV2 collection, the dictionary-as-a-string approach, compared to the dictionary layout shown in Figure 4.1, reduces the dictionary's storage requirements by $10.8 - 4 = 6.8$ bytes per entry. Here the term 4 stems from the pointer overhead in the primary array; the term 10.8 corresponds to the complete elimination of any internal fragmentation.

It is worth pointing out that the term strings stored in the secondary array do not require an explicit termination symbol (e.g., the "\0" character), because the length of each term in the dictionary is implicitly given by the pointers in the primary array. For example, by looking at the pointers for "shakespeare" and "shakespearean" in Figure 4.3, we know that the dictionary entry for "shakespeare" requires $16629970 - 16629951 = 19$ bytes in total: 11 bytes for the term plus 8 bytes for the 64-bit file pointer into the postings file.

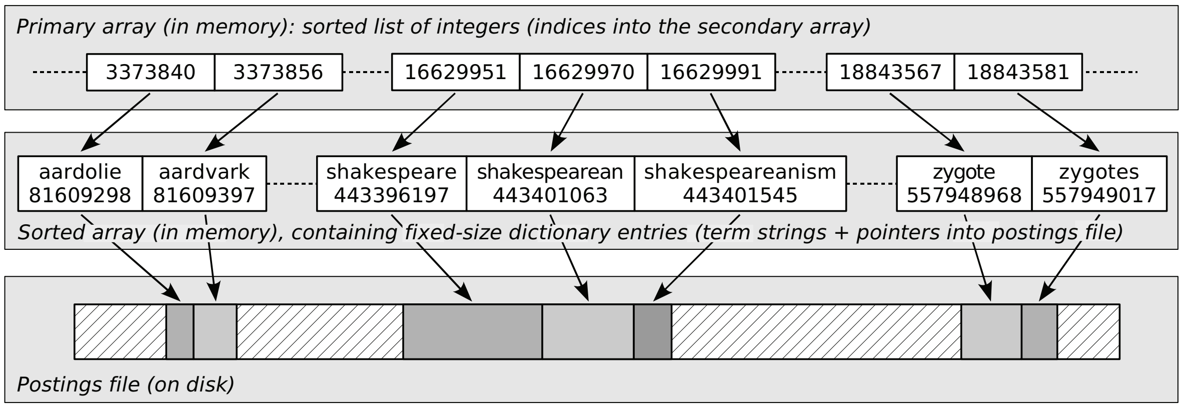

Primary array (in memory): sorted list of integers (indices into the secondary array)
3373840
3373856
16629951
16629970
16629991
18843567
18843581
aardolie 81609298
aardvark 81609397
shakespeare 443396197
shakespearean 443401063
shakespeareanism 443401545
zygote 557948968
zygotes 557949017
Sorted array (in memory), containing fixed-size dictionary entries (term strings + pointers into postings file)
Postings file (on disk)

Figure 4.3 Sort-based dictionary data structure with an additional level of indirection (the so-called *dictionary-as-a-string* approach).

Sort-based versus hash-based dictionary

For most applications a hash-based dictionary will be faster than a sort-based implementation, as it does not require a costly binary search or the traversal of a path in a search tree to find the dictionary entry for a given term. The precise speed advantage of a hash-based dictionary over a sort-based dictionary depends on the size of the hash table. If the table is too small, then there will be many collisions, potentially reducing the dictionary's performance substantially. As a rule of thumb, in order to keep the lengths of the collision chains in the hash table small, the size of the table should grow linearly with the number of terms in the dictionary.

Table 4.2 Lookup performance at query time. Average latency of a single-term lookup for a sort-based (shown in Figure 4.3) and a hash-based (shown in Figure 4.2) dictionary implementation. For the hash-based implementation, the size of the hash table (number of array entries) is varied between 2^{18} (≈ 262,000) and 2^{24} (≈ 16.8 million).

	Sorted	**Hashed (2^{18})**	**Hashed (2^{20})**	**Hashed (2^{22})**	**Hashed (2^{24})**
Shakespeare	0.32 μs	0.11 μs	0.13 μs	0.14 μs	0.16 μs
TREC45	1.20 μs	0.53 μs	0.34 μs	0.27 μs	0.25 μs
GOV2	2.79 μs	19.8 μs	5.80 μs	2.23 μs	0.84 μs

Table 4.2 shows the average time needed to find the dictionary entry for a random index term in each of our three example collections. A larger table usually results in a shorter lookup time, except for the *Shakespeare* collection, which is so small (23,000 different terms) that the effect of decreasing the term collisions is outweighed by the less efficient CPU cache utilization. A bigger hash table in this case results in an increased lookup time. Nonetheless, if the table size is chosen properly, a hash-based dictionary is usually at least twice as fast as a sort-based one.

Unfortunately, this speed advantage for single-term dictionary lookups has a drawback: A sort-based dictionary offers efficient support for prefix queries (e.g., "inform∗"). If the dictionary is based on a sorted array, for example, then a prefix query can be realized through two binary search operations, to find the first term T_j and the last term T_k matching the given prefix query, followed by a linear scan of all $k - j + 1$ dictionary entries between T_j and T_k. The total time complexity of this procedure is

$$\Theta(\log(|\mathcal{V}|)) + \Theta(m), \tag{4.1}$$

where $m = k - j + 1$ is the number of terms matching the prefix query and $\mathcal{V}$ is the vocabulary of the search engine.

If prefix queries are to be supported by a hash-based dictionary implementation, then this can be realized only through a linear scan of all terms in the hash table, requiring $\Theta(|\mathcal{V}|)$ string comparisons. It is therefore not unusual that a search engine employs two different dictionary data structures: a hash-based dictionary, used during the index construction process and providing efficient support of operations 1 (insert) and 2 (single-term lookup), and a sort-based dictionary that is created after the index has been built and that provides efficient support of operations 2 (single-term lookup) and 3 (prefix lookup).

The distinction between an indexing-time and a query-time dictionary is further motivated by the fact that support for high-performance single-term dictionary lookups is more important during index construction than during query processing. At query time the overhead associated with finding the dictionary entries for all query terms is negligible (a few microseconds) and is very likely to be outweighed by the other computations that have to be performed while processing a query. At indexing time, however, a dictionary lookup needs to be performed for every token in the text collection — 44 billion lookup operations in the case of GOV2. Thus, the dictionary represents a major bottleneck in the index construction process, and lookups should be as fast as possible.

4.3 Postings Lists

The actual index data, used during query processing and accessed through the search engine's dictionary, is stored in the index's postings lists. Each term's postings list contains information about the term's occurrences in the collection. Depending on the type of the index (docid, frequency, positional, or schema-independent — see Section 2.1.3), a term's postings list contains more or less detailed, and more or less storage-intensive, information. Regardless of the actual type of the index, however, the postings data always constitute the vast majority of all the data in the index. In their entirety, they are therefore usually too large to be stored in main memory and have to be kept on disk. Only during query processing are the query terms' postings lists (or small parts thereof) loaded into memory, on a by-need basis, as required by the query processing routines.

To make the transfer of postings from disk into main memory as efficient as possible, each term's postings list should be stored in a contiguous region of the hard drive. That way, when accessing the list, the number of disk seek operations is minimized. The hard drives of the computer system used in our experiments (summarized in the appendix) can read about half a megabyte of data in the time it takes to perform a single disk seek, so discontiguous postings lists can reduce the system's query performance dramatically.

Random list access: The per-term index

The search engine's list access pattern at query time depends on the type of query being processed. For some queries, postings are accessed in an almost strictly sequential fashion. For other queries it is important that the search engine can carry out efficient random access operations on the postings lists. An example of the latter type is phrase search or — equivalently — conjunctive Boolean search (processing a phrase query on a schema-independent index is essentially the same as resolving a Boolean AND on a docid index).

Recall from Chapter 2 the two main access methods provided by an inverted index: **next** and **prev**, returning the first (or last) occurrence of the given term after (or before) a given index address. Suppose we want to find all occurrences of the phrase "iterative binary search" in GOV2. After we have found out that there is exactly one occurrence of "iterative binary" in the collection, at position [33,399,564,886, 33,399,564,887], a single call to

$$\textbf{next}(\text{“search”},\ 33{,}399{,}564{,}887)$$

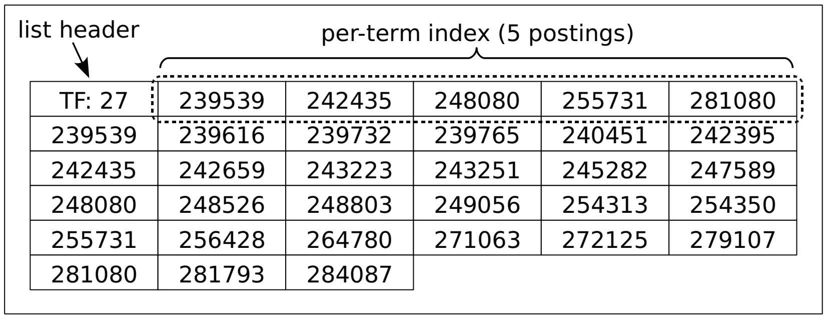
will tell us whether the phrase "iterative binary search" appears in the corpus. If the method returns 33,399,564,888, then the answer is yes. Otherwise, the answer is no.

If postings lists are stored in memory, as arrays of integers, then this operation can be performed very efficiently by conducting a binary search (or galloping search — see Section 2.1.2) on the postings array for "search". Since the term "search" appears about 50 million times in GOV2, the binary search requires a total of

$$\lceil \log_2(5 \times 10^7) \rceil = 26$$

random list accesses. For an on-disk postings list, the operation could theoretically be carried out in the same way. However, since a hard disk is not a true random access device, and a disk seek is a very costly operation, such an approach would be prohibitively expensive. With its 26 random disk accesses, a binary search on the on-disk postings list can easily take more than 200 milliseconds, due to seek overhead and rotational latency of the disk platter.

As an alternative, one might consider loading the entire postings list into memory in a single sequential read operation, thereby avoiding the expensive disk seeks. However, this is not a good solution, either. Assuming that each posting requires 8 bytes of disk space, it would take our computer more than 4 seconds to read the term's 50 million postings from disk.



TF: 27	239539	242435	248080	255731	281080
239539	239616	239732	239765	240451	242395
242435	242659	243223	243251	245282	247589
248080	248526	248803	249056	254313	254350
255731	256428	264780	271063	272125	279107
281080	281793	284087			

Figure 4.4 Schema-independent postings list for “denmark” (extracted from the Shakespeare collection) with per-term index: one synchronization point for every six postings. The number of synchronization points is implicit from the length of the list: $\lceil 27/6 \rceil = 5$.

In order to provide efficient random access into any given on-disk postings list, each list has to be equipped with an auxiliary data structure, which we refer to as the *per-term index*. This data structure is stored on disk, at the beginning of the respective postings list. It contains a copy of a subset of the postings in the list, for instance, a copy of every 5,000th posting. When accessing the on-disk postings list for a given term T, before performing any actual index operations on the list, the search engine loads T's per-term index into memory. Random-access operations of the type required by the **next** method can then be carried out by performing a binary search on the in-memory array representing T's per-term index (identifying a candidate range of 5,000 postings in the on-disk postings list), followed by loading up to 5,000 postings from the candidate range into memory and then performing a random access operation on those postings.

This approach to random access list operations is sometimes referred to as *self-indexing* (Moffat and Zobel, 1996). The entries in the per-term index are called *synchronization points*. Figure 4.4 shows the postings list for the term “denmark”, extracted from the Shakespeare collection, with a per-term index of granularity 6 (i.e., one synchronization point for every six postings in the list). In the figure, a call to **next**(250,000) would first identify the postings block starting with 248,080 as potentially containing the candidate posting. It would then load this block into memory, carry out a binary search on the block, and return 254,313 as the answer. Similarly, in our example for the phrase query “iterative binary search”, the random access operation into the list for the term “search” would be realized using only 2 disk seeks and loading a total of about 15,000 postings into memory (10,000 postings for the per-term index and 5,000 postings from the candidate range) — translating into a total execution time of approximately 30 ms.

Choosing the granularity of the per-term index, that is, the number of postings between two synchronization points, represents a trade-off. A greater granularity increases the amount of data between two synchronization points that need to be loaded into memory for every random access operation; a smaller granularity, conversely, increases the size of the per-term index and thus the amount of data read from disk when initializing the postings list (Exercise 4.1 asks you to calculate the optimal granularity for a given list).

In theory it is conceivable that, for a very long postings list containing billions of entries, the optimal per-term index (with a granularity that minimizes the total amount of disk activity) becomes so large that it is no longer feasible to load it completely into memory. In such a situation it is possible to build an index for the per-term index, or even to apply the whole procedure recursively. In the end this leads to a multi-level static B-tree that provides efficient random access into the postings list. In practice, however, such a complicated data structure is rarely necessary. A simple two-level structure, with a single per-term index for each on-disk postings list, is sufficient. The term "the", for instance, the most frequent term in the GOV2 collection, appears roughly 1 billion times in the collection. When stored uncompressed, its postings list in a schema-independent index consumes about 8 billion bytes (8 bytes per posting). Suppose the per-term index for "the" contains one synchronization point for every 20,000 postings. Loading the per-term index with its 50,000 entries into memory requires a single disk seek, followed by a sequential transfer of 400,000 bytes ($\approx$ 4.4 ms). Each random access operation into the term's list requires an additional disk seek, followed by loading 160,000 bytes ($\approx$ 1.7 ms) into RAM. In total, therefore, a single random access operation into the term's postings list requires about 30 ms (two random disk accesses, each taking about 12 ms, plus reading 560,000 bytes from disk). In comparison, adding an additional level of indexing, by building an index for the per-term index, would increase the number of disk seeks required for a single random access to at least three, and would therefore most likely decrease the index's random access performance.

Compared to an implementation that performs a binary search directly on the on-disk postings list, the introduction of the per-term index improves the performance of random access operations quite substantially. The true power of the method, however, lies in the fact that it allows us to store postings of variable length, for example postings of the form (*docid, tf,* $\langle$*positions*$\rangle$), and in particular compressed postings. If postings are stored not as fixed-size (e.g., 8-byte) integers, but in compressed form, then a simple binary search is no longer possible. However, by compressing postings in small chunks, where the beginning of each chunk corresponds to a synchronization point in the per-term index, the search engine can provide efficient random access even for compressed postings lists. This application also explains the choice of the term "synchronization point": a synchronization point helps the decoder establish synchrony with the encoder, thus allowing it to start decompressing data at an (almost) arbitrary point within the compressed postings sequence. See Chapter 6 for details on compressed inverted indices.

Prefix queries

If the search engine has to support prefix queries, such as "inform$*$", then it is imperative that postings lists be stored in lexicographical order of their respective terms. Consider the GOV2 collection; 4,365 different terms with a total of 67 million occurrences match the prefix query "inform$*$". By storing lists in lexicographical order, we ensure that the inverted lists for these 4,365 terms are close to each other in the inverted file and thus close to each other on disk. This decreases the seek distance between the individual lists and leads to better query performance. If lists were stored on disk in some random order, then disk seeks and rotational latency alone would account for almost a minute (4,365 $\times$ 12 ms), not taking into account any of the other operations that need to be carried out when processing the query. By arranging the inverted lists in lexicographical order of their respective terms, a query asking for all documents matching "inform$*$" can be processed in less than 2 seconds when using a frequency index; with a schema-independent index, the same query takes about 6 seconds. Storing the lists in the inverted file in some predefined order (e.g., lexicographical) is also important for efficient index updates, as discussed in Chapter 7.

A separate positional index

If the search engine is based on a document-centric positional index (containing a docid, a frequency value, and a list of within-document positions for each document that a given term appears in), it is not uncommon to divide the index data into two separate inverted files: one file containing the docid and frequency component of each posting, the other file containing the exact within-document positions. The rationale behind this division is that for many queries — and many scoring functions — access to the positional information is not necessary. By excluding it from the main index, query processing performance can be increased.

4.4 Interleaving Dictionary and Postings Lists

For many text collections the dictionary is small enough to fit into the main memory of a single machine. For large collections, however, containing many millions of different terms, even the collective size of all dictionary entries might be too large to be conveniently stored in RAM. The GOV2 collection, for instance, contains about 49 million distinct terms. The total size of the concatenation of these 49 million terms (if stored as zero-terminated strings) is 482 MB. Now suppose the dictionary data structure used in the search engine is based on a sorted array, as shown in Figure 4.3. Maintaining for each term in the dictionary an additional 32-bit pointer in the primary sorted array and a 64-bit file pointer in the secondary array increases the overall memory consumption by another $12 \times 49 = 588$ million bytes (approximately), leading to a total memory requirement of 1046 MB. Therefore, although the GOV2 collection is small enough to be managed by a single machine, the dictionary may be too large to fit into the machine's main memory.

To some extent, this problem can be addressed by employing dictionary compression techniques (discussed in Section 6.4). However, dictionary compression can get us only so far. There exist situations in which the number of distinct terms in the text collection is so enormous that it becomes impossible to store the entire dictionary in main memory, even after compression. Consider, for example, an index in which each postings list represents not an individual term but a term bigram, such as "information retrieval". Such an index is very useful for processing

Table 4.3 Number of unique terms, term bigrams, and trigrams for our three text collections. The number of unique bigrams is much larger than the number of unique terms, by about one order of magnitude.

	Tokens	**Unique Words**	**Unique Bigrams**	**Unique Trigrams**
Shakespeare	1.3×10^6	2.3×10^4	2.9×10^5	6.5×10^5
TREC45	3.0×10^8	1.2×10^6	2.5×10^7	9.4×10^7
GOV2	4.4×10^{10}	4.9×10^7	5.2×10^8	2.3×10^9

phrase queries. Unfortunately, the number of unique bigrams in a text collection is substantially larger than the number of unique terms. Table 4.3 shows that GOV2 contains only about 49 million distinct terms, but 520 million distinct term bigrams. Not surprisingly, if trigrams are to be indexed instead of bigrams, the situation becomes even worse — with 2.3 billion different trigrams in GOV2, it is certainly not feasible to keep the entire dictionary in main memory anymore.

Storing the entire dictionary on disk would satisfy the space requirements but would slow down query processing. Without any further modifications an on-disk dictionary would add at least one extra disk seek per query term, as the search engine would first need to fetch each term's dictionary entry from disk before it could start processing the given query. Thus, a pure on-disk approach is not satisfactory, either.

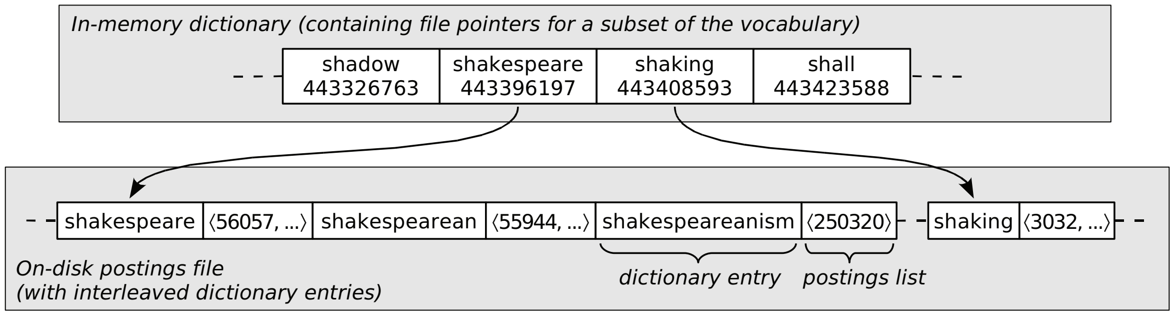


Figure 4.5 Interleaving dictionary and postings lists: Each on-disk inverted list is immediately preceded by the dictionary entry for the respective term. The in-memory dictionary contains entries for only *some* of the terms. In order to find the postings list for "shakespeareanism", a sequential scan of the on-disk data between "shakespeare" and "shaking" is necessary.

A possible solution to this problem is called *dictionary interleaving*, shown in Figure 4.5. In an interleaved dictionary all entries are stored on disk, each entry right before the respective postings list, to allow the search engine to fetch dictionary entry and postings list in one sequential read operation. In addition to the on-disk data, however, copies of *some* dictionary entries

Table 4.4 The impact of dictionary interleaving on a schema-independent index for GOV2 (49.5 million distinct terms). By choosing an index block size $B = 16{,}384$ bytes, the number of in-memory dictionary entries can be reduced by over 99%, at the cost of a minor query slowdown: 1 ms per query term.

Index Block Size (in bytes)	1,024	4,096	16,384	65,536	262,144
No. of in-memory dict. entries ($\times 10^6$)	3.01	0.91	0.29	0.10	0.04
Avg. index access latency (in ms)	11.4	11.6	12.3	13.6	14.9

(but not all of them) are kept in memory. When the search engine needs to determine the location of a term's postings list, it first performs a binary search on the sorted list of in-memory dictionary entries, followed by a sequential scan of the data found between two such entries. For the example shown in the figure, a search for "shakespeareanism" would first determine that the term's postings list (if it appears in the index) must be between the lists for "shakespeare" and "shaking". It would then load this index range into memory and scan it in a linear fashion to find the dictionary entry (and thus the postings list) for the term "shakespeareanism".

Dictionary interleaving is very similar to the self-indexing technique from Section 4.3, in the sense that random access disk operations are avoided by reading a little bit of extra data in a sequential manner. Because sequential disk operations are so much faster than random access, this trade-off is usually worthwhile, as long as the additional amount of data transferred from disk into main memory is small. In order to make sure that this is the case, we need to define an upper limit for the amount of data found between each on-disk dictionary entry and the closest preceding in-memory dictionary entry. We call this upper limit the *index block size*. For instance, if it is guaranteed for every term T in the index that the search engine does not need to read more than 1,024 bytes of on-disk data before it reaches T's on-disk dictionary entry, then we say that the index has a block size of 1,024 bytes.

Table 4.4 quantifies the impact that dictionary interleaving has on the memory consumption and the list access performance of the search engine (using GOV2 as a test collection). Without interleaving, the search engine needs to maintain about 49.5 million in-memory dictionary entries and can access the first posting in a random postings list in 11.3 ms on average (random disk seek + rotational latency). Choosing a block size of $B = 1{,}024$ bytes, the number of in-memory dictionary entries can be reduced to 3 million. At the same time, the search engine's list access latency (accessing the first posting in a randomly chosen list) increases by only 0.1 ms — a negligible overhead. As we increase the block size, the number of in-memory dictionary entries goes down and the index access latency goes up. But even for a relatively large block size of $B = 256$ KB, the additional cost — compared with a complete in-memory dictionary — is only a few milliseconds per query term.

Note that the memory consumption of an interleaved dictionary with block size B is quite different from maintaining an in-memory dictionary entry for every B bytes of index data. For example, the total size of the (compressed) schema-independent index for GOV2 is 62 GB. Choosing an index block size of $B = 64$ KB, however, does not lead to $62\,\text{GB} / 64\,\text{KB} \approx 1$ million

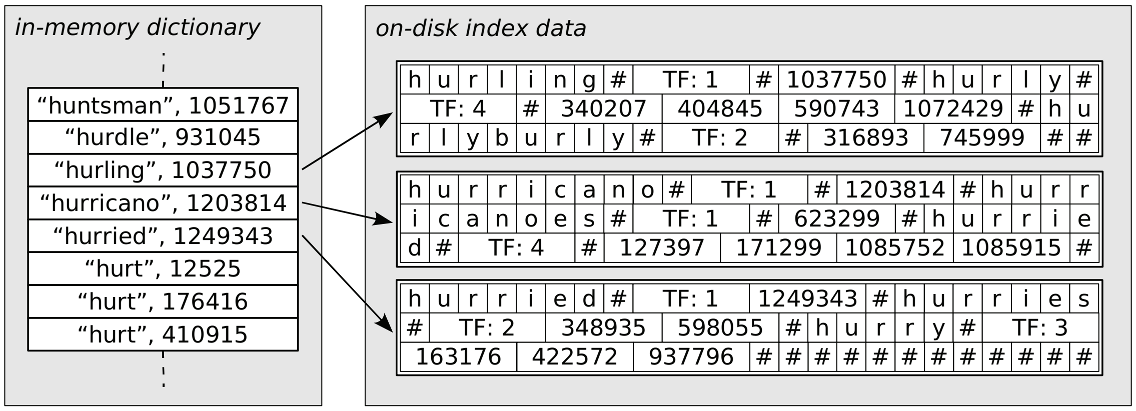

in-memory dictionary
"huntsman", 1051767
"hurdle", 931045
"hurling", 1037750
"hurricano", 1203814
"hurried", 1249343
"hurt", 12525
"hurt", 176416
"hurt", 410915
on-disk index data
h u r l i n g # TF: 1 # 1037750 # h u r l y # TF: 4 # 340207 404845 590743 1072429 # h u r l y b u r l y # TF: 2 # 316893 745999 # #
h u r r i c a n o # TF: 1 # 1203814 # h u r r i c a n o e s # TF: 1 # 623299 # h u r r i e d # TF: 4 # 127397 171299 1085752 1085915 #
h u r r i e d # TF: 1 1249343 # h u r r i e s # TF: 2 348935 598055 # h u r r y # TF: 3 163176 422572 937796 # # # # # # # # # # # #

Figure 4.6 Combining dictionary and postings lists. The index is split into blocks of 72 bytes. Each entry of the in-memory dictionary is of the form (*term*, *posting*), indicating the first term and first posting in a given index block. The "#" symbols in the index data are record delimiters that have been inserted for better readability.

dictionary entries, but 10 times less. The reason is that frequent terms, such as "the" and "of", require only a single in-memory dictionary entry each, even though their postings lists each consume far more disk space than 64 KB (the compressed list for "the" consumes about 1 GB).

In practice, a block size between 4 KB and 16 KB is usually sufficient to shrink the in-memory dictionary to an acceptable size, especially if dictionary compression (Section 6.4) is used to decrease the space requirements of the few remaining in-memory dictionary entries. The disk transfer overhead for this range of block sizes is less than 1 ms per query term and is rather unlikely to cause any performance problems.

Dropping the distinction between terms and postings

We may take the dictionary interleaving approach one step further by dropping the distinction between terms and postings altogether, and by thinking of the index data as a sequence of pairs of the form (*term*, *posting*). The on-disk index is then divided into fixed-size index blocks, with each block perhaps containing 64 KB of data. All postings are stored on disk, in alphabetical order of their respective terms. Postings for the same term are stored in increasing order, as before. Each term's dictionary entry is stored on disk, potentially multiple times, so that there is a dictionary entry in every index block that contains at least one posting for the term. The in-memory data structure used to access data in the on-disk index then is a simple array, containing for each index block a pair of the form (*term*, *posting*), where *term* is the first term in the given block, and *posting* is the first posting for *term* in that block.

An example of this new index layout is shown in Figure 4.6 (data taken from a schema-independent index for the Shakespeare collection). In the example, a call to

$$\textbf{next}(\text{“hurried”},\ 1{,}000{,}000)$$

would load the second block shown (starting with “hurricano”) from disk, would search the block for a posting matching the query, and would return the first matching posting (1,085,752). A call to

$$\textbf{next}(\text{“hurricano”},\ 1{,}000{,}000)$$

would load the first block shown (starting with “hurling”), would not find a matching posting in that block, and would then access the second block, returning the posting 1,203,814.

The combined representation of dictionary and postings lists unifies the interleaving method explained above and the self-indexing technique described in Section 4.3 in an elegant way. With this index layout, a random access into an arbitrary term's postings list requires only a single disk seek (we have eliminated the initialization step in which the term's per-term index is loaded into memory). On the downside, however, the total memory consumption of the index is higher than if we employ self-indexing and dictionary interleaving as two independent techniques. A 62-GB index with block size B = 64 KB now in fact requires approximately 1 million in-memory entries.

4.5 Index Construction

In the previous sections of this chapter, you have learned about various components of an inverted index. In conjunction, they can be used to realize efficient access to the contents of the index, even if all postings lists are stored on disk, and even if we don't have enough memory resources to hold a complete dictionary for the index in RAM. We now discuss how to efficiently construct an inverted index for a given text collection.

From an abstract point of view, a text collection can be thought of as a sequence of term occurrences — tuples of the form (*term*, *position*), where *position* is the word count from the beginning of the collection, and *term* is the term that occurs at that position. Figure 4.7 shows a fragment of the Shakespeare collection under this interpretation. Naturally, when a text collection is read in a sequential manner, these tuples are ordered by their second component, the position in the collection. The task of constructing an index is to change the ordering of the tuples so that they are sorted by their first component, the term they refer to (ties are broken by the second component). Once the new order is established, creating a proper index, with all its auxiliary data structures, is a comparatively easy task.

In general, index construction methods can be divided into two categories: in-memory index construction and disk-based index construction. In-memory indexing methods build an index for a given text collection entirely in main memory and can therefore only be used if the collection

Text fragment:

⟨SPEECH⟩
⟨SPEAKER⟩ JULIET ⟨/SPEAKER⟩
⟨LINE⟩ O Romeo, Romeo! wherefore art thou Romeo? ⟨/LINE⟩
⟨LINE⟩ ...

Original tuple ordering (collection order):

..., ("⟨speech⟩", 915487), ("⟨speaker⟩", 915488), ("juliet", 915489), ("⟨/speaker⟩", 915490), ("⟨line⟩", 915491), ("o", 915492), ("romeo", 915493), ("romeo", 915494), ("wherefore", 915495), ("art", 915496), ("thou", 915497), ("romeo", 915498), ("⟨/line⟩", 915499), ("⟨line⟩", 915500), ...

New tuple ordering (index order):

..., ("⟨line⟩", 915491), ("⟨line⟩", 915500), ...,
("romeo", 915411), ("romeo", 915493), ("romeo", 915494), ("romeo", 915498),
..., ("wherefore", 913310), ("wherefore", 915495), ("wherefore", 915849), ...

Figure 4.7 The index construction process can be thought of as reordering the tuple sequence that constitutes the text collection. Tuples are rearranged from their original *collection order* (sorted by position) to their new *index order* (sorted by term).

is small relative to the amount of available RAM. They do, however, form the basis for more sophisticated, disk-based index construction methods. Disk-based methods can be used to build indices for very large collections, much larger than the available amount of main memory.

As before, we limit ourselves to schema-independent indices because this allows us to ignore some details that need to be taken care of when discussing more complicated index types, such as document-level positional indices, and lets us focus on the essential aspects of the algorithms. The techniques presented, however, can easily be applied to other index types.

4.5.1 In-Memory Index Construction

Let us consider the simplest form of index construction, in which the text collection is small enough for the index to fit entirely into main memory. In order to create an index for such a collection, the indexing process needs to maintain the following data structures:

- a dictionary that allows efficient single-term lookup and insertion operations;
- an extensible (i.e., dynamic) list data structure that is used to store the postings for each term.

Assuming these two data structures are readily available, the index construction procedure is straightforward, as shown in Figure 4.8. If the right data structures are used for dictionary and extensible postings lists, then this method is very efficient, allowing the search engine to build

```
buildIndex (inputTokenizer) ≡
    $position \leftarrow 0$
    while $inputTokenizer.hasNext()$ do
        $T \leftarrow inputTokenizer.getNext()$
        obtain dictionary entry for $T$; create new entry if necessary
        append new posting $position$ to $T$'s postings list
        $position \leftarrow position + 1$
    sort all dictionary entries in lexicographical order
    for each term $T$ in the dictionary do
        write $T$'s postings list to disk
    write the dictionary to disk
    return
```

Figure 4.8 In-memory index construction algorithm, making use of two abstract data types: in-memory dictionary and extensible postings lists.

an index for the Shakespeare collection in less than one second. Thus, there are two questions that need to be discussed: 1. What data structure should be used for the dictionary? 2. What data structure should be used for the extensible postings lists?

Indexing-time dictionary

The dictionary implementation used during index construction needs to provide efficient support for single-term lookups and term insertions. Data structures that support these kinds of operations are very common and are therefore part of many publicly available programming libraries. For example, the C++ Standard Template Library (STL) published by SGI[1] provides a `map` data structure (binary search tree), and a `hash_map` data structure (variable-size hash table) that can carry out the lookup and insert operations performed during index construction. If you are implementing your own indexing algorithm, you might be tempted to use one of these existing implementations. However, it is not always advisable to do so.

We measured the performance of the STL data structures on a subset of GOV2, indexing the first 10,000 documents in the collection. The results are shown in Table 4.5. At first sight, it seems that the lookup performance of STL's `map` and `hash_map` implementations is sufficient for the purposes of index construction. On average, `map` needs about 630 ns per lookup; `hash_map` is a little faster, requiring 240 ns per lookup operation.

Now suppose we want to index the entire GOV2 collection instead of just a 10,000-document subset. Assuming the lookup performance stays at roughly the same level (an optimistic assumption, since the number of terms in the dictionary will keep increasing), the total time spent on

[1] `www.sgi.com/tech/stl/`

`hash_map` lookup operations, one for each of the 44 billion tokens in GOV, is:

$$44 \times 10^9 \times 240 \text{ ns} \;=\; 10{,}560 \text{ sec} \;\approx\; 3 \text{ hrs.}$$

Given that the fastest publicly available retrieval systems can build an index for the same collection in about 4 hours, including reading the input files, parsing the documents, and writing the (compressed) index to disk, there seems to be room for improvement. This calls for a custom dictionary implementation.

When designing our own dictionary implementation, we should try to optimize its performance for the specific type of data that we are dealing with. We know from Chapter 1 that term occurrences in natural-language text data roughly follow a Zipfian distribution. A key property of such a distribution is that the vast majority of all tokens in the collection correspond to a surprisingly small number of terms. For example, although there are about 50 million different terms in the GOV2 corpus, more than 90% of all term occurrences correspond to one of the 10,000 most frequent terms. Therefore, the bulk of the dictionary's lookup load may be expected to stem from a rather small set of very frequent terms. If the dictionary is based on a hash table, and collisions are resolved by means of chaining (as in Figure 4.2), then it is crucial that the dictionary entries for those frequent terms are kept near the beginning of each chain. This can be realized in two different ways:

1. **The insert-at-back heuristic**
   If a term is relatively frequent, then it is likely to occur very early in the stream of incoming tokens. Conversely, if the first occurrence of a term is encountered rather late, then chances are that the term is not very frequent. Therefore, when adding a new term to an existing chain in the hash table, it should be inserted at the end of the chain. This way, frequent terms, added early on, stay near the front, whereas infrequent terms tend to be found at the end.
2. **The move-to-front heuristic**
   If a term is frequent in the text collection, then it should be at the beginning of its chain in the hash table. Like insert-at-back, the move-to-front heuristic inserts new terms at the end of their respective chain. In addition, however, whenever a term lookup occurs and the term descriptor is not found at the beginning of the chain, it is relocated and moved to the front. This way, when the next lookup for the term takes place, the term's dictionary entry is still at (or near) the beginning of its chain in the hash table.

We evaluated the lookup performance of these two alternatives (using a handcrafted hash table implementation with a fixed number of hash slots in both cases) and compared it against a third alternative that inserts new terms at the beginning of the respective chain (referred to as the *insert-at-front* heuristic). The outcome of the performance measurements is shown in Table 4.5.

The insert-at-back and move-to-front heuristics achieve approximately the performance level (90 ns per lookup for a hash table with 2^{14} slots). Only for a small table size does move-to-front have a slight edge over insert-at-back (20% faster for a table with 2^{10} slots). The insert-at-front

Table 4.5 Indexing the first 10,000 documents of GOV2 (≈ 14 million tokens; 181,334 distinct terms). Average dictionary lookup time per token in microseconds. The rows labeled "Hash table" represent a handcrafted dictionary implementation based on a fixed-size hash table with chaining.

Dictionary Implementation	Lookup Time	String Comparisons
Binary search tree (STL `map`)	0.63 μs per token	18.1 per token
Variable-size hash table (STL `hash_map`)	0.24 μs per token	2.2 per token
Hash table (2^{10} entries, insert-at-front)	6.11 μs per token	140 per token
Hash table (2^{10} entries, insert-at-back)	0.37 μs per token	8.2 per token
Hash table (2^{10} entries, move-to-front)	0.31 μs per token	4.8 per token
Hash table (2^{14} entries, insert-at-front)	0.32 μs per token	10.1 per token
Hash table (2^{14} entries, insert-at-back)	0.09 μs per token	1.5 per token
Hash table (2^{14} entries, move-to-front)	0.09 μs per token	1.3 per token

heuristic, however, exhibits a very poor lookup performance and is between 3 and 20 times slower than move-to-front. To understand the origin of this extreme performance difference, look at the table column labeled "String Comparisons". When storing the 181,344 terms from the 10,000-document subcollection of GOV2 in a hash table with 2^{14} = 16,384 slots, we expect about 11 terms per chain on average. With the insert-at-front heuristic, the dictionary — on average — performs 10.1 string comparisons before it finds the term it is looking for. Because the frequent terms tend to appear early on in the collection, insert-at-front places them at the end of the respective chain. This is the cause of the method's dismal lookup performance. Incidentally, STL's `hash_map` implementation also inserts new hash table entries at the beginning of the respective chain, which is one reason why it does not perform so well in this benchmark.

A hash-based dictionary implementation employing the move-to-front heuristic is largely insensitive to the size of the hash table. In fact, even when indexing text collections containing many millions of different terms, a relatively small hash table, containing perhaps 2^{16} slots, will be sufficient to realize efficient dictionary lookups.

Extensible in-memory postings lists

The second interesting aspect of the simple in-memory index construction method, besides the dictionary implementation, is the implementation of the extensible in-memory postings lists. As suggested earlier, realizing each postings list as a singly linked list allows incoming postings to be appended to the existing lists very efficiently. The disadvantage of this approach is its rather high memory consumption: For every 32-bit (or 64-bit) posting, the indexing process needs to store an additional 32-bit (or 64-bit) pointer, thus increasing the total space requirements by 50–200%.

Various methods to decrease the storage overhead of the extensible postings lists have been proposed. For example, one could use a fixed-size array instead of a linked list. This avoids the

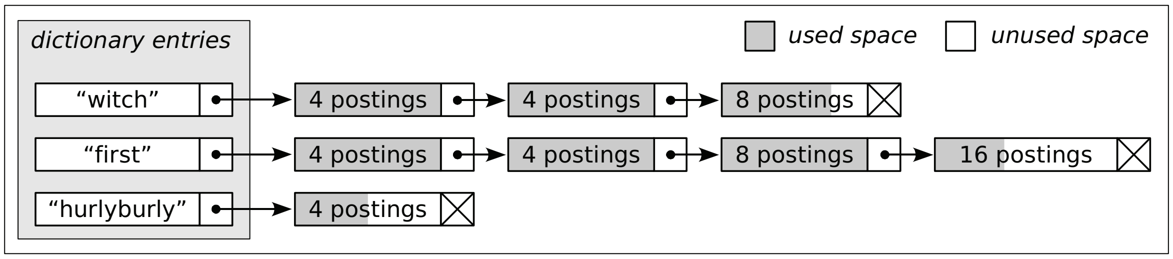


Figure 4.9 Realizing per-term extensible postings list by linking between groups of postings (*unrolled linked list*). Proportional pre-allocation (here with pre-allocation factor $k = 2$) provides an attractive trade-off between space overhead due to `next` pointers and space overhead due to internal fragmentation.

storage overhead associated with the `next` pointers in the linked list. Unfortunately, the length of each term's postings list is not known ahead of time, so this method requires an additional pass over the input data: In the first pass, term statistics are collected and space is allocated for each term's postings list; in the second pass, the pre-allocated arrays are filled with postings data. Of course, reading the input data twice harms indexing performance.

Another solution is to pre-allocate a postings array for every term in the dictionary, but to allow the indexing process to change the size of the array, for example, by performing a call to `realloc` (assuming that the programming language and run-time environment support this kind of operation). When a novel term is encountered, *init* bytes are allocated for the term's postings (e.g., $init = 16$). Whenever the capacity of the existing array is exhausted, a `realloc` operation is performed in order to increase the size of the array. By employing a proportional pre-allocation strategy, that is, by allocating a total of

$$s_{\text{new}} = \max\{s_{\text{old}} + init, k \times s_{\text{old}}\} \tag{4.2}$$

bytes in a reallocation operation, where s_{old} is the old size of the array and k is a constant, called the *pre-allocation factor*, the number of calls to `realloc` can be kept small (logarithmic in the size of each postings list). Nonetheless, there are some problems with this approach. If the pre-allocation factor is too small, then a large number of list relocations is triggered during index construction, possibly affecting the search engine's indexing performance. If it is too big, then a large amount of space is wasted due to allocated but unused memory (internal fragmentation). For instance, if $k = 2$, then 25% of the allocated memory will be unused on average.

A third way of realizing extensible in-memory postings lists with low storage overhead is to employ a linked list data structure, but to have multiple postings share a `next` pointer by linking to *groups* of postings instead of individual postings. When a new term is inserted into the dictionary, a small amount of space, say 16 bytes, is allocated for its postings. Whenever the space reserved for a term's postings list is exhausted, space for a new group of postings is

Table 4.6 Indexing the TREC45 collection. Index construction performance for various memory allocation strategies used to manage the extensible in-memory postings lists (32-bit postings; 32-bit pointers). Arranging postings for the same term in small groups and linking between these groups is faster than any other strategy. Pre-allocation factor used for both `realloc` and *grouping*: $k = 1.2$.

Allocation Strategy	Memory Consumption	Time (Total)	Time (CPU)
Linked list (simple)	2,312 MB	88 sec	77 sec
Two-pass indexing	1,168 MB	123 sec	104 sec
`realloc`	1,282 MB	82 sec	71 sec
Linked list (grouping)	1,208 MB	71 sec	61 sec

allocated. The amount of space allotted to the new group is

$$s_{\text{new}} = \min\{limit, \max\{16, (k-1) \times s_{\text{total}}\}\}, \tag{4.3}$$

where *limit* is an upper bound on the size of a postings group, say, 256 postings, s_{total} is the amount of memory allocated for postings so far, and k is the pre-allocation factor, just as in the case of `realloc`.

This way of combining postings into groups and having several postings share a single `next` pointer is shown in Figure 4.9. A linked list data structure that keeps more than one value in each list node is sometimes called an *unrolled linked list*, in analogy to loop unrolling techniques used in compiler optimization. In the context of index construction, we refer to this method as *grouping*.

Imposing an upper limit on the amount of space pre-allocated by the grouping technique allows to control internal fragmentation. At the same time, it does not constitute a performance problem; unlike in the case of `realloc`, allocating more space for an existing list is a very lightweight operation and does not require the indexing process to relocate any postings data.

A comparative performance evaluation of all four list allocation strategies — simple linked list, two-pass, `realloc`, and linked list with grouping — is given by Table 4.6. The two-pass method exhibits the lowest memory consumption but takes almost twice as long as the grouping approach. Using a pre-allocation factor $k = 1.2$, the `realloc` method has a memory consumption that is about 10% above that of the space-optimal two-pass strategy. This is consistent with the assumption that about half of the pre-allocated memory is never used. The grouping method, on the other hand, exhibits a memory consumption that is only 3% above the optimum. In addition, grouping is about 16% faster than `realloc` (61 sec vs. 71 sec CPU time).

Perhaps surprisingly, linking between groups of postings is also faster than linking between individual postings (61 sec vs. 77 sec CPU time). The reason for this is that the unrolled linked list data structure employed by the grouping technique not only decreases internal fragmentation, but also improves CPU cache efficiency, by keeping postings for the same term close

```
buildIndex_sortBased (inputTokenizer) ≡
    position ← 0
    while inputTokenizer.hasNext() do
        T ← inputTokenizer.getNext()
        obtain dictionary entry for T; create new entry if necessary
        termID ← unique term ID of T
        write record $R_{position} \equiv (termID, position)$ to disk
        position ← position + 1
    tokenCount ← position
    sort $R_0 \ldots R_{tokenCount-1}$ by first component; break ties by second component
    perform a sequential scan of $R_0 \ldots R_{tokenCount-1}$, creating the final index
    return
```

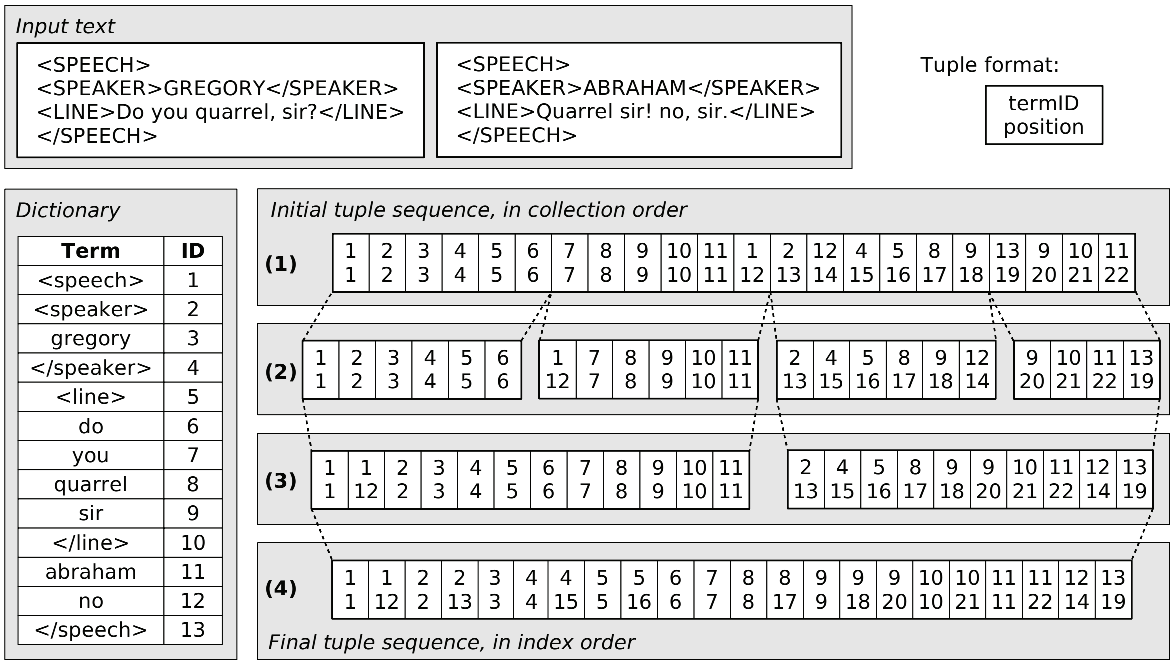
Figure 4.10 Sort-based index construction algorithm, creating a schema-independent index for a text collection. The main difficulty lies in efficiently sorting the on-disk records $R_0 \ldots R_{tokenCount-1}$.

together. In the simple linked-list implementation, postings for a given term are randomly scattered across the used portion of the computer's main memory, resulting in a large number of CPU cache misses when collecting each term's postings before writing them to disk.

4.5.2 Sort-Based Index Construction

Hash-based in-memory index construction algorithms can be extremely efficient, as demonstrated in the previous section. However, if we want to index text collections that are substantially larger than the available amount of RAM, we need to move away from the idea that we can keep the entire index in main memory, and need to look toward disk-based approaches instead. The sort-based indexing method presented in this section is the first of two disk-based index construction methods covered in this chapter. It can be used to index collections that are much larger than the available amount of main memory.

Recall from the beginning of this section that building an inverted index can be thought of as the process of reordering the sequence of term-position tuples that represents the text collection — transforming it from collection order into index order. This is a sorting process, and sort-based index construction realizes the transformation in the simplest way possible. Consuming tokens from the input files, a sort-based indexing process emits records of the form (*termID*, *position*) and writes them to disk immediately. The result is a sequence of records, sorted by their second component (*position*). When it is done processing the input data, the indexer sorts all tuples written to disk by their first component, using the second component to break ties. The result is a new sequence of records, sorted by their first component (*termID*). Transforming this new sequence into a proper inverted file, using the information found in the in-memory dictionary, is straightforward.


Input text
Tuple format:
termID
position
Dictionary
Initial tuple sequence, in collection order
(1)
(2)
(3)
(4)
Final tuple sequence, in index order

```
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>Do you quarrel, sir?</LINE>
</SPEECH>
```

```
<SPEECH>
<SPEAKER>ABRAHAM</SPEAKER>
<LINE>Quarrel sir! no, sir.</LINE>
</SPEECH>
```

Term	ID
<speech>	1
<speaker>	2
gregory	3
</speaker>	4
<line>	5
do	6
you	7
quarrel	8
sir	9
</line>	10
abraham	11
no	12
</speech>	13

(1)

1	2	3	4	5	6	7	8	9	10	11	1	2	12	4	5	8	9	13	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

(2)

1	2	3	4	5	6
1	2	3	4	5	6

1	7	8	9	10	11
12	7	8	9	10	11

2	4	5	8	9	12
13	15	16	17	18	14

9	10	11	13
20	21	22	19

(3)

1	1	2	3	4	5	6	7	8	9	10	11
1	12	2	3	4	5	6	7	8	9	10	11

2	4	5	8	9	9	10	11	12	13
13	15	16	17	18	20	21	22	14	19

(4)

1	1	2	2	3	4	4	5	5	6	7	8	8	9	9	9	10	10	11	11	12	13
1	12	2	13	3	4	15	5	16	6	7	8	17	9	18	20	10	21	11	22	14	19

Figure 4.11 Sort-based index construction with global term IDs. Main memory is large enough to hold 6 (*termID*, *position*) tuples at a time. (1)→(2): sorting blocks of size ≤ 6 in memory, one at a time. (2)→(3) and (3)→(4): merging sorted blocks into bigger blocks.

The method, shown as pseudo-code in Figure 4.10, is very easy to implement and can be used to create an index that is substantially larger than the available amount of main memory. Its main limitation is the available amount of disk space.

Sorting the on-disk records can be a bit tricky. In many implementations, it is performed by loading a certain number of records, say n, into main memory at a time, where n is defined by the size of a record and the amount of available main memory. These n records are then sorted in memory and written back to disk. The process is repeated until we have $\lceil \frac{tokenCount}{n} \rceil$ blocks of sorted records. These blocks can then be combined, in a *multiway merge operation* (processing records from all blocks at the same time) or in a *cascaded merge operation* (merging, for example, two blocks at a time), resulting in a sorted sequence of *tokenCount* records. Figure 4.11 shows a sort-based indexing process that creates the final tuple sequence by means of a cascaded merge operation, merging two tuple sequences at a time. With some additional data structures, and with the *termID* component removed from each posting, this final sequence can be thought of as the index for the collection.

Despite its ability to index text collections much larger than the available amount of main memory, sort-based indexing has two essential limitations:

- It requires a substantial amount of disk space, usually at least 8 bytes per input token (4 bytes for the term ID + 4 bytes for the position), or even 12 bytes (4 + 8) for larger text collections. When indexing GOV2, for instance, the disk space required to store the temporary files is at least $12 \times 44 \times 10^9$ bytes (= 492 GB) — more than the uncompressed size of the collection itself (426 GB).
- To be able to properly sort the (*termID*, *docID*) pairs emitted in the first phase, the indexing process needs to maintain globally unique term IDs, which can be realized only through a complete in-memory dictionary. As discussed earlier, a complete dictionary for GOV2 might consume more than 1 GB of RAM, making it difficult to index the collection on a low-end machine.

There are various ways to address these issues. However, in the end they all have in common that they transform the sort-based indexing method into something more similar to what is known as *merge-based index construction*.

4.5.3 Merge-Based Index Construction

In contrast to sort-based index construction methods, the merge-base method presented in this section does not need to maintain any global data structures. In particular, there is no need for globally unique term IDs. The size of the text collection to be indexed is therefore limited only by the amount of disk space available to store the temporary data and the final index, but not by the amount of main memory available to the indexing process.

Merge-based indexing is a generalization of the in-memory index construction method discussed in Section 4.5.1, building inverted lists by means of hash table lookup. In fact, if the collection for which an index is being created is small enough for the index to fit completely into main memory, then merge-based indexing behaves exactly like in-memory indexing. If the text collection is too large to be indexed completely in main memory, then the indexing process performs a *dynamic partitioning* of the collection. That is, it starts building an in-memory index. As soon as it runs out of memory (or when a predefined memory utilization threshold is reached), it builds an on-disk inverted file by transferring the in-memory index data to disk, deletes the in-memory index, and continues indexing. This procedure is repeated until the whole collection has been indexed. The algorithm is shown in Figure 4.12.

The result of the process outlined above is a set of inverted files, each representing a certain part of the whole collection. Each such subindex is referred to as an *index partition*. In a final step, the index partitions are merged into the final index, representing the entire text collection. The postings lists in the index partitions (and in the final index) are usually stored in compressed form (see Chapter 6), in order to keep the disk I/O overhead low.

The index partitions written to disk as intermediate output of the indexing process are completely independent of each other. For example, there is no need for globally unique term IDs; there is not even a need for numerical term IDs. Each term is its own ID; the postings lists in each partition are stored in lexicographical order of their terms, and access to a term's list can

```
buildIndex_mergeBased (inputTokenizer, memoryLimit) ≡
    $n \leftarrow 0$   // initialize the number of index partitions
    $position \leftarrow 0$
    $memoryConsumption \leftarrow 0$
    while inputTokenizer.hasNext() do
        $T \leftarrow inputTokenizer.getNext()$
        obtain dictionary entry for $T$; create new entry if necessary
        append new posting position to $T$'s postings list
        $position \leftarrow position + 1$
        $memoryConsumption \leftarrow memoryConsumption + 1$
        if $memoryConsumption \geq memoryLimit$ then
            createIndexPartition()
    if $memoryConsumption > 0$ then
        createIndexPartition()
    merge index partitions $I_0 \ldots I_{n-1}$, resulting in the final on-disk index $I_{\text{final}}$
    return

createIndexPartition () ≡
    create empty on-disk inverted file $I_n$
    sort in-memory dictionary entries in lexicographical order
    for each term $T$ in the dictionary do
        add $T$'s postings list to $I_n$
    delete all in-memory postings lists
    reset the in-memory dictionary
    $memoryConsumption \leftarrow 0$
    $n \leftarrow n + 1$
    return
```

Figure 4.12 Merge-based indexing algorithm, creating a set of independent sub-indices (*index partitions*). The final index is generated by combining the sub-indices via a multi-way merge operation.

be realized by using the data structures described in Sections 4.3 and 4.4. Because of the lexicographical ordering and the absence of term IDs, merging the individual partitions into the final index is straightforward. Pseudo-code for a very simple implementation, performing repeated linear probing of all subindices, is given in Figure 4.13. If the number of index partitions is large (more than 10), then the algorithm can be improved by arranging the index partitions in a priority queue (e.g., a heap), ordered according to the next term in the respective partition. This eliminates the need for the linear scan in lines 7–10.

Overall performance numbers for merge-based index construction, including all components, are shown in Table 4.7. The total time necessary to build a schema-independent index for GOV2 is around 4 hours. The time required to perform the final merge operation, combining the n index partitions into one final index, is about 30% of the time it takes to generate the partitions.

```
mergeIndexPartitions ($\langle I_0, \ldots, I_{n-1} \rangle$) ≡
    create empty inverted file $I_{\text{final}}$
    for $k \leftarrow 0$ to $n-1$ do
        open index partition $I_k$ for sequential processing
    $currentIndex \leftarrow 0$
    while $currentIndex \neq nil$ do
        $currentIndex \leftarrow nil$
        for $k \leftarrow 0$ to $n-1$ do
            if $I_k$ still has terms left then
                if $(currentIndex = nil) \lor (I_k.currentTerm < currentTerm)$ then
                    $currentIndex \leftarrow I_k$
                    $currentTerm \leftarrow I_k.currentTerm$
        if $currentIndex \neq nil$ then
            $I_{\text{final}}.addPostings(currentTerm, currentIndex.getPostings(currentTerm))$
            $currentIndex.advanceToNextTerm()$
    delete $I_0 \ldots I_{n-1}$
    return
```

Figure 4.13 Merging a set of n index partitions $I_0 \ldots I_{n-1}$ into an index I_{final}. This is the final step in merge-based index construction.

The algorithm is very scalable: On our computer, indexing the whole GOV2 collection (426 GB of text) takes only 11 times as long as indexing a 10% subcollection (43 GB of text).

There are, however, some limits to the scalability of the method. When merging the index partitions at the end of the indexing process, it is important to have at least a moderately sized read-ahead buffer, a few hundred kilobytes, for each partition. This helps keep the number of disk seeks (jumping back and forth between the different partitions) small. Naturally, the size of the read-ahead buffer for each partition is bounded from above by M/n, where M is the amount of available memory and n is the number of partitions. Thus, if n becomes too large, merging becomes slow.

Reducing the amount of memory available to the indexing process therefore has two effects. First, it decreases the total amount of memory available for the read-ahead buffers. Second, it increases the number of index partitions. Thus, reducing main memory by 50% decreases the size of each index partition's read-ahead buffer by 75%. Setting the memory limit to $M = 128$ MB, for example, results in 3,032 partitions that need to be merged, leaving each partition with a read-ahead buffer of only 43 KB. The general trend of this effect is depicted in Figure 4.14. The figure shows that the performance of the final merge operation is highly dependent on the amount of main memory available to the indexing process. With 128 MB of available main memory, the final merge takes 6 times longer than with 1,024 MB.

There are two possible countermeasures that could be taken to overcome this limitation. The first is to replace the simple multiway merge by a cascaded merge operation. For instance, if 1,024 index partitions need to be merged, then we could first perform 32 merge operations

Table 4.7 Building a schema-independent index for various text collections, using merge-based index construction with 512 MB of RAM for the in-memory index. The indexing-time dictionary is realized by a hash table with 2^{16} entries and move-to-front heuristic. The extensible in-memory postings lists are unrolled linked lists, linking between groups of postings, with a pre-allocation factor $k = 1.2$.

	Reading, Parsing & Indexing	**Merging**	**Total Time**
Shakespeare	1 sec	0 sec	1 sec
TREC45	71 sec	11 sec	82 sec
GOV2 (10%)	20 min	4 min	24 min
GOV2 (25%)	51 min	11 min	62 min
GOV2 (50%)	102 min	25 min	127 min
GOV2 (100%)	205 min	58 min	263 min

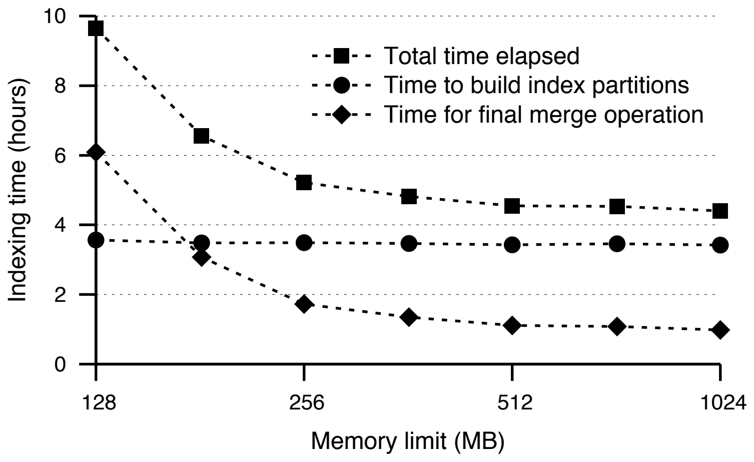


Figure 4.14 The impact of available RAM on the performance of merge-based indexing (data set: GOV2). The performance of phase 1 (building index partitions) is largely independent of the amount of available main memory. The performance of phase 2 (merging partitions) suffers severely if little main memory is available.

involving 32 partitions each, and then merge the resulting 32 new partitions into the final index. This is a generalization of the process shown in Figure 4.11, which depicts a cascaded merge operation that works on 2 partitions at a time. The second countermeasure is to decrease the space consumed by the postings lists, by means of compressed in-memory inversion. Compressing postings on-the-fly, as they enter the in-memory index, allows the indexing process to accumulate more postings before it runs out of memory, thus decreasing the number of on-disk index partitions created.

In conclusion, despite some problems with the final merge operation, merge-based index construction lets us build an inverted file for very large text collections, even on a single PC. Its advantage over the sort-based method is that it does not require globally unique term IDs. It is therefore especially attractive if the number of dictionary terms is very large. Another great advantage of this index construction algorithm over the sort-based method is that it produces an in-memory index that is immediately queriable. This feature is essential when the search engine needs to deal with dynamic text collections (see Chapter 7).

4.6 Other Types of Indices

In our discussion of index data structures, we have limited ourselves to the case of inverted indices. However, an inverted index is just one possible type of index that can be used as part of a search engine.

A *forward index* (or *direct index*) is a mapping from document IDs to the list of terms appearing in each document. Forward indices complement inverted indices. They are usually not used for the actual search process, but to obtain information about per-document term distributions at query time, which is required by query expansion techniques such as pseudo-relevance feedback (see Chapter 8), and to produce result *snippets*. Compared to extracting this information from the raw text files, a forward index has the advantage that the text has already been parsed, and the relevant data can be extracted more efficiently.

Signature files (Faloutsos and Christodoulakis, 1984) are an alternative to docid indices. Similar to a Bloom filter (Bloom, 1970), a signature file can be used to obtain a list of documents that *may* contain the given term. In order to find out whether the term actually appears in a document, the document itself (or a forward index) needs to be consulted. By changing the parameters of the signature file, it is possible to trade time for speed: A smaller index results in a greater probability of false positives, and vice versa.

Suffix trees (Weiner, 1973) and *suffix arrays* (Manber and Myers, 1990) can be used to efficiently find all occurrences of a given n-gram sequence in a given text collection. They can be used either to index character n-grams (without tokenizing the input text) or to index word n-grams (after tokenization). Suffix trees are attractive data structures for phrase search or regular expression search, but are usually larger than inverted indices and provide less efficient search operations when stored on disk instead of in RAM.

4.7 Summary

In this chapter we have covered the essential algorithms and data structures necessary to build and access inverted indices. The main points of the chapter are:

- Inverted indices are usually too large to be loaded completely into memory. It is therefore common to keep only the dictionary, which is relatively small compared to the size of the entire index, in memory, while storing the postings lists on disk (Section 4.2).
- For a large text collection even the dictionary might be too large to fit into RAM. The memory requirements of the dictionary can be substantially decreased by keeping an incomplete dictionary in memory, interleaving dictionary entries with on-disk postings lists, and exploiting the fact that all lists in the index are sorted in lexicographical order (Section 4.4).
- A sort-based dictionary implementation should be used, and postings lists should be stored on disk in lexicographical order, if prefix queries are to be supported by the search engine (Sections 4.2 and 4.3).
- For each on-disk postings list, a per-term index, containing a subset of the term's postings, can be used to realize efficient quasi-random access into the list (Section 4.3).
- Highly efficient in-memory index construction can be realized by employing a hash-based in-memory dictionary with move-to-front heuristic and by using grouped linked lists to implement extensible in-memory postings lists (Section 4.5.1).
- If the amount of main memory available to the indexing process is too small to allow the index to be built completely in RAM, then the in-memory index construction method can be extended to a merge-based method, in which the text collection is divided into subcollections, dynamically and based on the available amount of main memory. An index for each subcollection is built using the in-memory indexing method. After the entire collection has been indexed, the indices for the individual subcollections are merged in a single multiway merge operation or a cascaded merge process (Section 4.5.3).
- The performance of merge-based index construction is essentially linear in the size of the collection. However, the final merge operation may suffer severely if too little main memory is available for the subindices' read buffers (Section 4.5.3).

4.8 Further Reading

Good entry points for an in-depth understanding of the architecture and performance of inverted indices are provided by Witten et al. (1999, chapters 3 and 5) and Zobel and Moffat (2006). A high-level overview of the index data structures employed by Google around 1998 is given by Brin and Page (1998).

Moffat and Zobel (1996) discuss query-time efficiency issues of inverted files, including the per-term index data structure ("self-indexing") used for random list access that is outlined in Section 4.3. Rao and Ross (1999, 2000) demonstrate that random access issues arise not only in the context of on-disk indices but also for in-memory inverted files. They show that binary search is *not* the best way to realize random access for in-memory postings lists.

Heinz and Zobel (2003) discuss single-pass merge-based index construction and its advantages over the sort-based method. They also examine the efficiency of various in-memory dictionary implementations, including the move-to-front heuristic described in Section 4.5.1 (Zobel et al., 2001), and propose a new dictionary data structure, the *burst trie* (Heinz et al., 2002), which achieves a single-term lookup performance close to a hash table but — unlike a hash table — can also be used to resolve prefix queries.

Memory management strategies for extensible in-memory postings lists (e.g., unrolled linked lists) are examined by Büttcher and Clarke (2005) and, more recently, by Luk and Lam (2007).

With a naïve implementation of the final merge procedure in merge-based index construction (and variants of sort-based indexing), the total storage requirement is twice as large as the size of the final index. Moffat and Bell (1995) describe a clever technique that can be used to realize the merge in situ, re-using the disk space occupied by already processed parts of the input partitions to store the final index.

Faloutsos and Christodoulakis (1984) give a nice overview of signature files, including some theoretical properties. Zobel et al. (1998) discuss the relative performance of inverted files and signature files in text search. They come to the conclusion that, for many applications, inverted files are the better choice. Carterette and Can (2005) argue that, under certain circumstances, signature files can be almost as fast as inverted files.

Suffix trees made their first appearance under the name *position trees* in a paper by Weiner (1973). Ukkonen (1995) presents a linear-time construction method for suffix trees. Clark and Munro (1996) discuss a variant of suffix trees that allows efficient search operations when stored on disk instead of main memory.

4.9 Exercises

Exercise 4.1 In Section 4.3 we introduced the concept of the *per-term index* as a means to improve the index's random access performance. Suppose the postings list for some term consists of 64 million postings, each of which consumes 4 bytes. In order to carry out a single random access into the term's postings list, the search engine needs to perform two disk read operations:

1. Loading the per-term index (list of synchronization points) into RAM.
2. Loading a block B of postings into RAM, where B is identified by means of binary search on the list of synchronization points.

Let us call the number of postings per synchronization point the *granularity* of the per-term index. For the above access pattern, what is the optimal granularity (i.e., the one that minimizes disk I/O)? What is the total number of bytes read from disk?

Exercise 4.2 The introduction of the per-term index in Section 4.3 was motivated by the performance characteristics of typical hard disk drives (in particular, the high cost of disk seeks). However, the same method can also be used to improve the random access performance of in-memory indices. To confirm this claim, you have to implement two different data structures for in-memory postings lists and equip them with a **next** access method (see Chapter 2). The first data structure stores postings in a simple array of 32-bit integers. Its **next** method operates by performing a binary search on that array. In the second data structure, an auxiliary array is used to store a copy of every 64th posting in the postings list. The **next** method first performs a binary search on the auxiliary array, followed by a *sequential scan* of the 64 candidate postings in the postings list. Measure the average single-element lookup latency of both implementations, working with lists of n postings, for $n = 2^{12}, 2^{16}, 2^{20}, 2^{24}$. Describe and analyze your findings.

Exercise 4.3 Building an inverted index is essentially a sorting process. The lower bound for every general-purpose sorting algorithm is $\Omega(n \log(n))$. However, the merge-based index construction method from Section 4.5.3 has a running time that is linear in the size of the collection (see Table 4.7, page 130). Find at least two places where there is a hidden logarithmic factor.

Exercise 4.4 In the algorithm shown in Figure 4.12, the memory limit is expressed as the number of postings that can be stored in RAM. What is the assumption that justifies this definition of the memory limit? Give an example of a text collection or an application in which the assumption does not hold.

Exercise 4.5 In Section 4.5.1 we discussed the performance characteristics of various dictionary data structures. We pointed out that hash-based implementations offer better performance for single-term lookups (performed during index construction), while sort-based solutions are more appropriate for multi-term lookups (needed for prefix queries). Design and implement a data structure that offers better single-term lookup performance than a sort-based dictionary and better prefix query performance than a hash-based implementation.

Exercise 4.6 (project exercise) Design and implement an index construction method that creates a schema-independent index for a given text collection. The result of the index construction process is an on-disk index. The index does not need to use any of the optimizations discussed in Sections 4.3 and 4.4.

- Implement the in-memory index construction method described in Section 4.5.1. When run on typical English text, your implementation should be able to build an index for a collection containing approximately $\frac{M}{8}$ tokens, where M is the amount of available main memory, in bytes.

- Extend your implementation so that the size of the text collection is no longer limited by the amount of main memory available to the indexing process. This will probably require you to write a module that can merge two or more on-disk indices into a single index.

4.10 Bibliography

Bender, M., Michel, S., Triantafillou, P., and Weikum, G. (2007). Design alternatives for large-scale Web search: Alexander was great, Aeneas a pioneer, and Anakin has the force. In *Proceedings of the 1st Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*, pages 16–22. Amsterdam, The Netherlands.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

Brin, S., and Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117.

Büttcher, S., and Clarke, C. L. A. (2005). *Memory Management Strategies for Single-Pass Index Construction in Text Retrieval Systems.* Technical Report CS-2005-32. University of Waterloo, Waterloo, Canada.

Carterette, B., and Can, F. (2005). Comparing inverted files and signature files for searching a large lexicon. *Information Processing & Management*, 41(3):613–633.

Clark, D. R., and Munro, J. I. (1996). Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391. Atlanta, Georgia.

Faloutsos, C., and Christodoulakis, S. (1984). Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4):267–288.

Heinz, S., and Zobel, J. (2003). Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729.

Heinz, S., Zobel, J., and Williams, H. E. (2002). Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223.

Luk, R. W. P., and Lam, W. (2007). Efficient in-memory extensible inverted file. *Information Systems*, 32(5):733–754.

Manber, U., and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327. San Francisco, California.

Moffat, A., and Bell, T. A. H. (1995). In-situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550.

Moffat, A., and Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379.

Rao, J., and Ross, K. A. (1999). Cache conscious indexing for decision-support in main memory. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 78–89. Edinburgh, Scotland.

Rao, J., and Ross, K. A. (2000). Making B^+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486. Dallas, Texas.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

Weiner, P. (1973). Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11. Iowa City, Iowa.

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

Zobel, J., Heinz, S., and Williams, H. E. (2001). In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277.

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

Zobel, J., Moffat, A., and Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490.

4 Static Inverted Indices

In this chapter we describe a set of index structures that are suitable for supporting search queries of the type outlined in Chapter 2. We restrict ourselves to the case of static text collections. That is, we assume that we are building an index for a collection that never changes. Index update strategies for dynamic text collections, in which documents can be added to and removed from the collection, are the topic of Chapter 7.

For performance reasons, it may be desirable to keep the index for a text collection completely in main memory. However, in many applications this is not feasible. In file system search, for example, a full-text index for all data stored in the file system can easily consume several gigabytes. Since users will rarely be willing to dedicate the most part of their available memory resources to the search system, it is not possible to keep the entire index in RAM. And even for dedicated index servers used in Web search engines it might be economically sensible to store large portions of the index on disk instead of in RAM, simply because disk space is so much cheaper than RAM. For example, while we are writing this, a gigabyte of RAM costs around \$40 (U.S.), whereas a gigabyte of hard drive space costs only about \$0.20 (U.S.). This factor-200 price difference, however, is not reflected by the relative performance of the two storage media. For typical index operations, an in-memory index is usually between 10 and 20 times faster than an on-disk index. Hence, when building two equally priced retrieval systems, one storing its index data on disk, the other storing them in main memory, the disk-based system may actually be faster than the RAM-based one (see Bender et al. (2007) for a more in-depth discussion of this and related issues).

The general assumption that will guide us throughout this chapter is that main memory is a scarce resource, either because the search engine has to share it with other processes running on the same system or because it is more economical to store data on disk than in RAM. In our discussion of data structures for inverted indices, we focus on hybrid organizations, in which some parts of the index are kept in main memory, while the majority of the data is stored on disk. We examine a variety of data structures for different parts of the search engine and evaluate their performance through a number of experiments. A performance summary of the computer system used to conduct these experiments can be found in the appendix.

4.1 Index Components and Index Life Cycle

When we discuss the various aspects of inverted indices in this chapter, we look at them from two perspectives: the structural perspective, in which we divide the system into its components and examine aspects of an individual component of the index (e.g., an individual postings list); and the operational perspective, in which we look at different phases in the life cycle of an inverted index and discuss the essential index operations that are carried out in each phase (e.g., processing a search query).

As already mentioned in Chapter 2, the two principal components of an inverted index are the *dictionary* and the *postings lists*. For each term in the text collection, there is a postings list that contains information about the term's occurrences in the collection. The information found in these postings lists is used by the system to process search queries. The dictionary serves as a lookup data structure on top of the postings lists. For every query term in an incoming search query, the search engine first needs to locate the term's postings list before it can start processing the query. It is the job of the dictionary to provide this mapping from terms to the location of their postings lists in the index.

In addition to dictionary and postings lists, search engines often employ various other data structures. Many engines, for instance, maintain a *document map* that, for each document in the index, contains document-specific information, such as the document's URL, its length, PageRank (see Section 15.3.1), and other data. The implementation of these data structures, however, is mostly straightforward and does not require any special attention.

The life cycle of a static inverted index, built for a never-changing text collection, consists of two distinct phases (for a dynamic index the two phases coincide):

1. *Index construction*: The text collection is processed sequentially, one token at a time, and a postings list is built for each term in the collection in an incremental fashion.
2. *Query processing*: The information stored in the index that was built in phase 1 is used to process search queries.

Phase 1 is generally referred to as *indexing time*, while phase 2 is referred to as *query time*. In many respects these two phases are complementary; by performing additional work at indexing time (e.g., precomputing score contributions — see Section 5.1.3), less work needs to be done at query time. In general, however, the two phases are quite different from one another and usually require different sets of algorithms and data structures. Even for subcomponents of the index that are shared by the two phases, such as the search engine's dictionary data structure, it is not uncommon that the specific implementation utilized during index construction is different from the one used at query time.

The flow of this chapter is mainly defined by our bifocal perspective on inverted indices. In the first part of the chapter (Sections 4.2–4.4), we are primarily concerned with the query-time aspects of dictionary and postings lists, looking for data structures that are most suitable for supporting efficient index access and query processing. In the second part (Section 4.5) we focus on aspects of the index construction process and discuss how we can efficiently build the data structures outlined in the first part. We also discuss how the organization of the dictionary and the postings lists needs to be different from the one suggested in the first part of the chapter, if we want to maximize their performance at indexing time.

For the sake of simplicity, we assume throughout this chapter that we are dealing exclusively with *schema-independent* indices. Other types of inverted indices, however, are similar to the schema-independent variant, and the methods discussed in this chapter apply to all of them (see Section 2.1.3 for a list of different types of inverted indices).

4.2 The Dictionary

The *dictionary* is the central data structure that is used to manage the set of terms found in a text collection. It provides a mapping from the set of index terms to the locations of their postings lists. At query time, locating the query terms' postings lists in the index is one of the first operations performed when processing an incoming keyword query. At indexing time, the dictionary's lookup capability allows the search engine to quickly obtain the memory address of the inverted list for each incoming term and to append a new posting at the end of that list.

Dictionary implementations found in search engines usually support the following set of operations:

1. Insert a new entry for term T.
2. Find and return the entry for term T (if present).
3. Find and return the entries for all terms that start with a given prefix P.

When building an index for a text collection, the search engine performs operations of types 1 and 2 to look up incoming terms in the dictionary and to add postings for these terms to the index. After the index has been built, the search engine can process search queries, performing operations of types 2 and 3 to locate the postings lists for all query terms. Although dictionary operations of type 3 are not strictly necessary, they are a useful feature because they allow the search engine to support *prefix queries* of the form "inform$*$", matching all documents containing a term that begins with "inform".

Table 4.1 Index sizes for various index types and three example collections, with and without applying index compression techniques. In each case the first number refers to an index in which each component is stored as a simple 32-bit integer, while the second number refers to an index in which each entry is compressed using a byte-aligned encoding method.

	Shakespeare	**TREC45**	**GOV2**
Number of tokens	1.3×10^6	3.0×10^8	4.4×10^{10}
Number of terms	2.3×10^4	1.2×10^6	4.9×10^7
Dictionary (uncompr.)	0.4 MB	24 MB	1046 MB
Docid index	n/a	578 MB/200 MB	37751 MB/12412 MB
Frequency index	n/a	1110 MB/333 MB	73593 MB/21406 MB
Positional index	n/a	2255 MB/739 MB	245538 MB/78819 MB
Schema-ind. index	5.7 MB/2.7 MB	1190 MB/532 MB	173854 MB/63670 MB

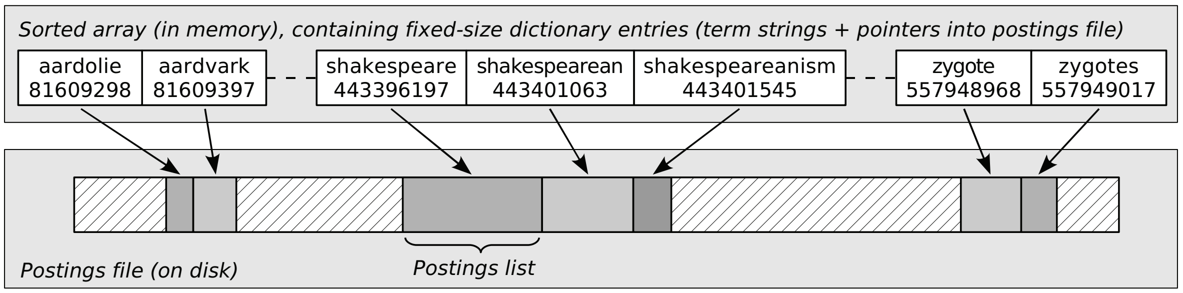


Figure 4.1 Dictionary data structure based on a sorted array (data extracted from a schema-independent index for TREC45). The array contains fixed-size dictionary entries, composed of a zero-terminated string and a pointer into the postings file that indicates the position of the term's postings list.

For a typical natural-language text collection, the dictionary is relatively small compared to the total size of the index. Table 4.1 shows this for the three example collections used in this book. The size of the uncompressed dictionary is only between 0.6% (GOV2) and 7% (Shakespeare) of the size of an uncompressed schema-independent index for the respective collection (the fact that the relative size of the dictionary is smaller for large collections than for small ones follows directly from Zipf's law — see Equation 1.2 on page 16). We therefore assume, at least for now, that the dictionary is small enough to fit completely into main memory.

The two most common ways to realize an in-memory dictionary are:

- A *sort-based* dictionary, in which all terms that appear in the text collection are arranged in a sorted array or in a search tree, in lexicographical (i.e., alphabetical) order, as shown in Figure 4.1. Lookup operations are realized through tree traversal (when using a search tree) or binary search (when using a sorted list).
- A *hash-based* dictionary, in which each index term has a corresponding entry in a hash table. Collisions in the hash table (i.e., two terms are assigned the same hash value) are resolved by means of *chaining* — terms with the same hash value are arranged in a linked list, as shown in Figure 4.2.

Storing the dictionary terms

When implementing the dictionary as a sorted array, it is important that all array entries are of the same size. Otherwise, performing a binary search may be difficult. Unfortunately, this causes some problems. For example, the longest sequence of alphanumeric characters in GOV2 (i.e., the longest term in the collection) is 74,147 bytes long. Obviously, it is not feasible to allocate 74 KB of memory for each term in the dictionary. But even if we ignore such extreme outliers and truncate each term after, say, 20 bytes, we are still wasting precious memory resources. Following the simple tokenization procedure from Section 1.3.2, the average length of a term

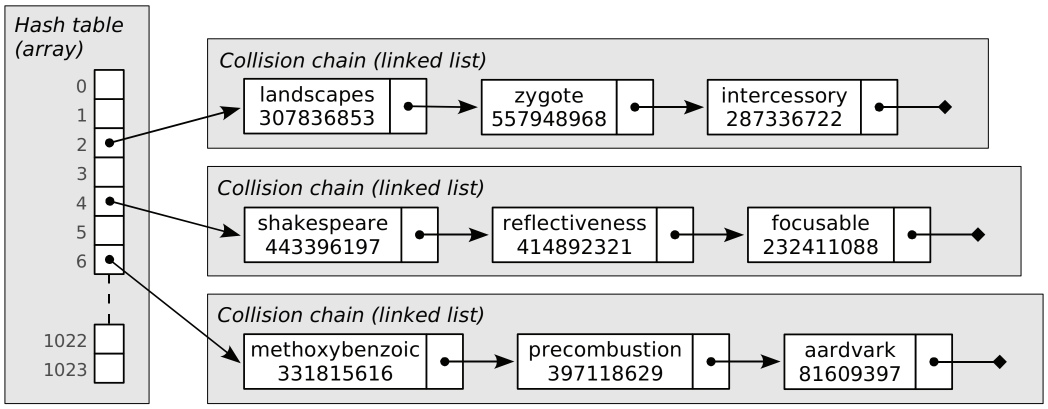


Figure 4.2 Dictionary data structure based on a hash table with $2^{10} = 1024$ entries (data extracted from schema-independent index for TREC45). Terms with the same hash value are arranged in a linked list (*chaining*). Each term descriptor contains the term itself, the position of the term's postings list, and a pointer to the next entry in the linked list.

in GOV2 is 9.2 bytes. Storing each term in a fixed-size memory region of 20 bytes wastes 10.8 bytes per term on average (*internal fragmentation*).

One way to eliminate the internal fragmentation is to not store the index terms themselves in the array, but only pointers to them. For example, the search engine could maintain a *primary* dictionary array, containing 32-bit pointers into a *secondary* array. The secondary array then contains the actual dictionary entries, consisting of the terms themselves and the corresponding pointers into the postings file. This way of organizing the search engine's dictionary data is shown in Figure 4.3. It is sometimes referred to as the *dictionary-as-a-string approach*, because there are no explicit delimiters between two consecutive dictionary entries; the secondary array can be thought of as a long, uninterrupted string.

For the GOV2 collection, the dictionary-as-a-string approach, compared to the dictionary layout shown in Figure 4.1, reduces the dictionary's storage requirements by $10.8 - 4 = 6.8$ bytes per entry. Here the term 4 stems from the pointer overhead in the primary array; the term 10.8 corresponds to the complete elimination of any internal fragmentation.

It is worth pointing out that the term strings stored in the secondary array do not require an explicit termination symbol (e.g., the "\0" character), because the length of each term in the dictionary is implicitly given by the pointers in the primary array. For example, by looking at the pointers for "shakespeare" and "shakespearean" in Figure 4.3, we know that the dictionary entry for "shakespeare" requires $16629970 - 16629951 = 19$ bytes in total: 11 bytes for the term plus 8 bytes for the 64-bit file pointer into the postings file.

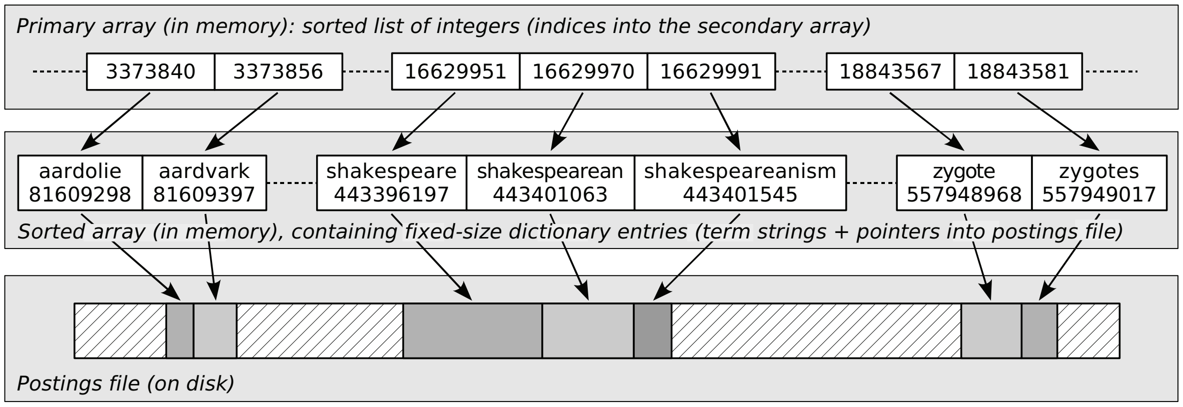

Primary array (in memory): sorted list of integers (indices into the secondary array)
3373840
3373856
16629951
16629970
16629991
18843567
18843581
aardolie 81609298
aardvark 81609397
shakespeare 443396197
shakespearean 443401063
shakespeareanism 443401545
zygote 557948968
zygotes 557949017
Sorted array (in memory), containing fixed-size dictionary entries (term strings + pointers into postings file)
Postings file (on disk)

Figure 4.3 Sort-based dictionary data structure with an additional level of indirection (the so-called *dictionary-as-a-string* approach).

Sort-based versus hash-based dictionary

For most applications a hash-based dictionary will be faster than a sort-based implementation, as it does not require a costly binary search or the traversal of a path in a search tree to find the dictionary entry for a given term. The precise speed advantage of a hash-based dictionary over a sort-based dictionary depends on the size of the hash table. If the table is too small, then there will be many collisions, potentially reducing the dictionary's performance substantially. As a rule of thumb, in order to keep the lengths of the collision chains in the hash table small, the size of the table should grow linearly with the number of terms in the dictionary.

Table 4.2 Lookup performance at query time. Average latency of a single-term lookup for a sort-based (shown in Figure 4.3) and a hash-based (shown in Figure 4.2) dictionary implementation. For the hash-based implementation, the size of the hash table (number of array entries) is varied between 2^{18} ($\approx$ 262,000) and 2^{24} ($\approx$ 16.8 million).

	Sorted	**Hashed (2^{18})**	**Hashed (2^{20})**	**Hashed (2^{22})**	**Hashed (2^{24})**
Shakespeare	0.32 μs	0.11 μs	0.13 μs	0.14 μs	0.16 μs
TREC45	1.20 μs	0.53 μs	0.34 μs	0.27 μs	0.25 μs
GOV2	2.79 μs	19.8 μs	5.80 μs	2.23 μs	0.84 μs

Table 4.2 shows the average time needed to find the dictionary entry for a random index term in each of our three example collections. A larger table usually results in a shorter lookup time, except for the *Shakespeare* collection, which is so small (23,000 different terms) that the effect of decreasing the term collisions is outweighed by the less efficient CPU cache utilization. A bigger hash table in this case results in an increased lookup time. Nonetheless, if the table size is chosen properly, a hash-based dictionary is usually at least twice as fast as a sort-based one.

Unfortunately, this speed advantage for single-term dictionary lookups has a drawback: A sort-based dictionary offers efficient support for prefix queries (e.g., "inform∗"). If the dictionary is based on a sorted array, for example, then a prefix query can be realized through two binary search operations, to find the first term T_j and the last term T_k matching the given prefix query, followed by a linear scan of all $k - j + 1$ dictionary entries between T_j and T_k. The total time complexity of this procedure is

$$\Theta(\log(|\mathcal{V}|)) + \Theta(m), \tag{4.1}$$

where $m = k - j + 1$ is the number of terms matching the prefix query and $\mathcal{V}$ is the vocabulary of the search engine.

If prefix queries are to be supported by a hash-based dictionary implementation, then this can be realized only through a linear scan of all terms in the hash table, requiring $\Theta(|\mathcal{V}|)$ string comparisons. It is therefore not unusual that a search engine employs two different dictionary data structures: a hash-based dictionary, used during the index construction process and providing efficient support of operations 1 (insert) and 2 (single-term lookup), and a sort-based dictionary that is created after the index has been built and that provides efficient support of operations 2 (single-term lookup) and 3 (prefix lookup).

The distinction between an indexing-time and a query-time dictionary is further motivated by the fact that support for high-performance single-term dictionary lookups is more important during index construction than during query processing. At query time the overhead associated with finding the dictionary entries for all query terms is negligible (a few microseconds) and is very likely to be outweighed by the other computations that have to be performed while processing a query. At indexing time, however, a dictionary lookup needs to be performed for every token in the text collection — 44 billion lookup operations in the case of GOV2. Thus, the dictionary represents a major bottleneck in the index construction process, and lookups should be as fast as possible.

4.3 Postings Lists

The actual index data, used during query processing and accessed through the search engine's dictionary, is stored in the index's postings lists. Each term's postings list contains information about the term's occurrences in the collection. Depending on the type of the index (docid, frequency, positional, or schema-independent — see Section 2.1.3), a term's postings list contains more or less detailed, and more or less storage-intensive, information. Regardless of the actual type of the index, however, the postings data always constitute the vast majority of all the data in the index. In their entirety, they are therefore usually too large to be stored in main memory and have to be kept on disk. Only during query processing are the query terms' postings lists (or small parts thereof) loaded into memory, on a by-need basis, as required by the query processing routines.

To make the transfer of postings from disk into main memory as efficient as possible, each term's postings list should be stored in a contiguous region of the hard drive. That way, when accessing the list, the number of disk seek operations is minimized. The hard drives of the computer system used in our experiments (summarized in the appendix) can read about half a megabyte of data in the time it takes to perform a single disk seek, so discontiguous postings lists can reduce the system's query performance dramatically.

Random list access: The per-term index

The search engine's list access pattern at query time depends on the type of query being processed. For some queries, postings are accessed in an almost strictly sequential fashion. For other queries it is important that the search engine can carry out efficient random access operations on the postings lists. An example of the latter type is phrase search or — equivalently — conjunctive Boolean search (processing a phrase query on a schema-independent index is essentially the same as resolving a Boolean AND on a docid index).

Recall from Chapter 2 the two main access methods provided by an inverted index: **next** and **prev**, returning the first (or last) occurrence of the given term after (or before) a given index address. Suppose we want to find all occurrences of the phrase "iterative binary search" in GOV2. After we have found out that there is exactly one occurrence of "iterative binary" in the collection, at position [33,399,564,886, 33,399,564,887], a single call to

$$\mathbf{next}(\text{“search”},\ 33{,}399{,}564{,}887)$$

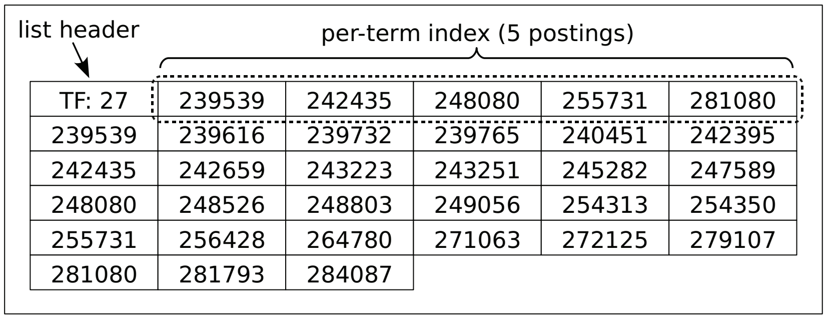
will tell us whether the phrase "iterative binary search" appears in the corpus. If the method returns 33,399,564,888, then the answer is yes. Otherwise, the answer is no.

If postings lists are stored in memory, as arrays of integers, then this operation can be performed very efficiently by conducting a binary search (or galloping search — see Section 2.1.2) on the postings array for "search". Since the term "search" appears about 50 million times in GOV2, the binary search requires a total of

$$\lceil \log_2(5 \times 10^7) \rceil = 26$$

random list accesses. For an on-disk postings list, the operation could theoretically be carried out in the same way. However, since a hard disk is not a true random access device, and a disk seek is a very costly operation, such an approach would be prohibitively expensive. With its 26 random disk accesses, a binary search on the on-disk postings list can easily take more than 200 milliseconds, due to seek overhead and rotational latency of the disk platter.

As an alternative, one might consider loading the entire postings list into memory in a single sequential read operation, thereby avoiding the expensive disk seeks. However, this is not a good solution, either. Assuming that each posting requires 8 bytes of disk space, it would take our computer more than 4 seconds to read the term's 50 million postings from disk.



TF: 27	239539	242435	248080	255731	281080
239539	239616	239732	239765	240451	242395
242435	242659	243223	243251	245282	247589
248080	248526	248803	249056	254313	254350
255731	256428	264780	271063	272125	279107
281080	281793	284087			

Figure 4.4 Schema-independent postings list for "denmark" (extracted from the Shakespeare collection) with per-term index: one synchronization point for every six postings. The number of synchronization points is implicit from the length of the list: $\lceil 27/6 \rceil = 5$.

In order to provide efficient random access into any given on-disk postings list, each list has to be equipped with an auxiliary data structure, which we refer to as the *per-term index.* This data structure is stored on disk, at the beginning of the respective postings list. It contains a copy of a subset of the postings in the list, for instance, a copy of every 5,000th posting. When accessing the on-disk postings list for a given term T, before performing any actual index operations on the list, the search engine loads T's per-term index into memory. Random-access operations of the type required by the **next** method can then be carried out by performing a binary search on the in-memory array representing T's per-term index (identifying a candidate range of 5,000 postings in the on-disk postings list), followed by loading up to 5,000 postings from the candidate range into memory and then performing a random access operation on those postings.

This approach to random access list operations is sometimes referred to as *self-indexing* (Moffat and Zobel, 1996). The entries in the per-term index are called *synchronization points.* Figure 4.4 shows the postings list for the term "denmark", extracted from the Shakespeare collection, with a per-term index of granularity 6 (i.e., one synchronization point for every six postings in the list). In the figure, a call to **next**(250,000) would first identify the postings block starting with 248,080 as potentially containing the candidate posting. It would then load this block into memory, carry out a binary search on the block, and return 254,313 as the answer. Similarly, in our example for the phrase query "iterative binary search", the random access operation into the list for the term "search" would be realized using only 2 disk seeks and loading a total of about 15,000 postings into memory (10,000 postings for the per-term index and 5,000 postings from the candidate range) — translating into a total execution time of approximately 30 ms.

Choosing the granularity of the per-term index, that is, the number of postings between two synchronization points, represents a trade-off. A greater granularity increases the amount of data between two synchronization points that need to be loaded into memory for every random access operation; a smaller granularity, conversely, increases the size of the per-term index and thus the amount of data read from disk when initializing the postings list (Exercise 4.1 asks you to calculate the optimal granularity for a given list).

In theory it is conceivable that, for a very long postings list containing billions of entries, the optimal per-term index (with a granularity that minimizes the total amount of disk activity) becomes so large that it is no longer feasible to load it completely into memory. In such a situation it is possible to build an index for the per-term index, or even to apply the whole procedure recursively. In the end this leads to a multi-level static B-tree that provides efficient random access into the postings list. In practice, however, such a complicated data structure is rarely necessary. A simple two-level structure, with a single per-term index for each on-disk postings list, is sufficient. The term "the", for instance, the most frequent term in the GOV2 collection, appears roughly 1 billion times in the collection. When stored uncompressed, its postings list in a schema-independent index consumes about 8 billion bytes (8 bytes per posting). Suppose the per-term index for "the" contains one synchronization point for every 20,000 postings. Loading the per-term index with its 50,000 entries into memory requires a single disk seek, followed by a sequential transfer of 400,000 bytes ($\approx$ 4.4 ms). Each random access operation into the term's list requires an additional disk seek, followed by loading 160,000 bytes ($\approx$ 1.7 ms) into RAM. In total, therefore, a single random access operation into the term's postings list requires about 30 ms (two random disk accesses, each taking about 12 ms, plus reading 560,000 bytes from disk). In comparison, adding an additional level of indexing, by building an index for the per-term index, would increase the number of disk seeks required for a single random access to at least three, and would therefore most likely decrease the index's random access performance.

Compared to an implementation that performs a binary search directly on the on-disk postings list, the introduction of the per-term index improves the performance of random access operations quite substantially. The true power of the method, however, lies in the fact that it allows us to store postings of variable length, for example postings of the form (*docid, tf,* $\langle$*positions*$\rangle$), and in particular compressed postings. If postings are stored not as fixed-size (e.g., 8-byte) integers, but in compressed form, then a simple binary search is no longer possible. However, by compressing postings in small chunks, where the beginning of each chunk corresponds to a synchronization point in the per-term index, the search engine can provide efficient random access even for compressed postings lists. This application also explains the choice of the term "synchronization point": a synchronization point helps the decoder establish synchrony with the encoder, thus allowing it to start decompressing data at an (almost) arbitrary point within the compressed postings sequence. See Chapter 6 for details on compressed inverted indices.

Prefix queries

If the search engine has to support prefix queries, such as "inform$*$", then it is imperative that postings lists be stored in lexicographical order of their respective terms. Consider the GOV2 collection; 4,365 different terms with a total of 67 million occurrences match the prefix query "inform$*$". By storing lists in lexicographical order, we ensure that the inverted lists for these 4,365 terms are close to each other in the inverted file and thus close to each other on disk. This decreases the seek distance between the individual lists and leads to better query performance. If lists were stored on disk in some random order, then disk seeks and rotational latency alone would account for almost a minute (4,365 × 12 ms), not taking into account any of the other operations that need to be carried out when processing the query. By arranging the inverted lists in lexicographical order of their respective terms, a query asking for all documents matching "inform∗" can be processed in less than 2 seconds when using a frequency index; with a schema-independent index, the same query takes about 6 seconds. Storing the lists in the inverted file in some predefined order (e.g., lexicographical) is also important for efficient index updates, as discussed in Chapter 7.

A separate positional index

If the search engine is based on a document-centric positional index (containing a docid, a frequency value, and a list of within-document positions for each document that a given term appears in), it is not uncommon to divide the index data into two separate inverted files: one file containing the docid and frequency component of each posting, the other file containing the exact within-document positions. The rationale behind this division is that for many queries — and many scoring functions — access to the positional information is not necessary. By excluding it from the main index, query processing performance can be increased.

4.4 Interleaving Dictionary and Postings Lists

For many text collections the dictionary is small enough to fit into the main memory of a single machine. For large collections, however, containing many millions of different terms, even the collective size of all dictionary entries might be too large to be conveniently stored in RAM. The GOV2 collection, for instance, contains about 49 million distinct terms. The total size of the concatenation of these 49 million terms (if stored as zero-terminated strings) is 482 MB. Now suppose the dictionary data structure used in the search engine is based on a sorted array, as shown in Figure 4.3. Maintaining for each term in the dictionary an additional 32-bit pointer in the primary sorted array and a 64-bit file pointer in the secondary array increases the overall memory consumption by another 12×49 = 588 million bytes (approximately), leading to a total memory requirement of 1046 MB. Therefore, although the GOV2 collection is small enough to be managed by a single machine, the dictionary may be too large to fit into the machine's main memory.

To some extent, this problem can be addressed by employing dictionary compression techniques (discussed in Section 6.4). However, dictionary compression can get us only so far. There exist situations in which the number of distinct terms in the text collection is so enormous that it becomes impossible to store the entire dictionary in main memory, even after compression. Consider, for example, an index in which each postings list represents not an individual term but a term bigram, such as "information retrieval". Such an index is very useful for processing

Table 4.3 Number of unique terms, term bigrams, and trigrams for our three text collections. The number of unique bigrams is much larger than the number of unique terms, by about one order of magnitude.

	Tokens	Unique Words	Unique Bigrams	Unique Trigrams
Shakespeare	1.3×10^6	2.3×10^4	2.9×10^5	6.5×10^5
TREC45	3.0×10^8	1.2×10^6	2.5×10^7	9.4×10^7
GOV2	4.4×10^{10}	4.9×10^7	5.2×10^8	2.3×10^9

phrase queries. Unfortunately, the number of unique bigrams in a text collection is substantially larger than the number of unique terms. Table 4.3 shows that GOV2 contains only about 49 million distinct terms, but 520 million distinct term bigrams. Not surprisingly, if trigrams are to be indexed instead of bigrams, the situation becomes even worse — with 2.3 billion different trigrams in GOV2, it is certainly not feasible to keep the entire dictionary in main memory anymore.

Storing the entire dictionary on disk would satisfy the space requirements but would slow down query processing. Without any further modifications an on-disk dictionary would add at least one extra disk seek per query term, as the search engine would first need to fetch each term's dictionary entry from disk before it could start processing the given query. Thus, a pure on-disk approach is not satisfactory, either.

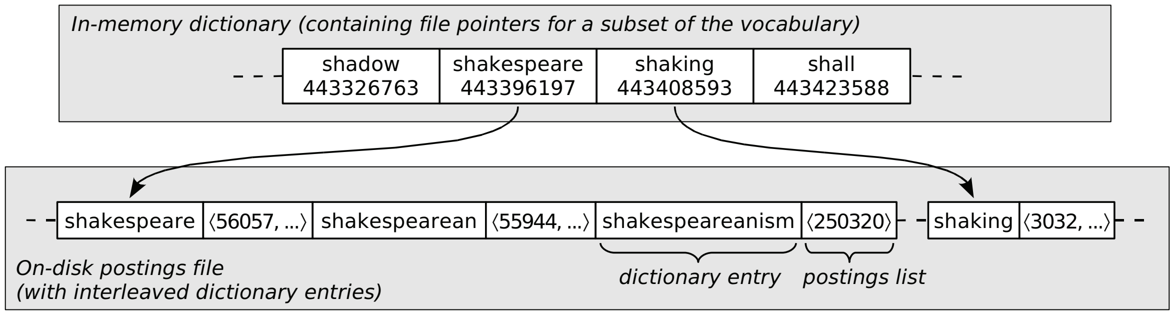


Figure 4.5 Interleaving dictionary and postings lists: Each on-disk inverted list is immediately preceded by the dictionary entry for the respective term. The in-memory dictionary contains entries for only *some* of the terms. In order to find the postings list for "shakespeareanism", a sequential scan of the on-disk data between "shakespeare" and "shaking" is necessary.

A possible solution to this problem is called *dictionary interleaving*, shown in Figure 4.5. In an interleaved dictionary all entries are stored on disk, each entry right before the respective postings list, to allow the search engine to fetch dictionary entry and postings list in one sequential read operation. In addition to the on-disk data, however, copies of *some* dictionary entries

Table 4.4 The impact of dictionary interleaving on a schema-independent index for GOV2 (49.5 million distinct terms). By choosing an index block size $B = 16{,}384$ bytes, the number of in-memory dictionary entries can be reduced by over 99%, at the cost of a minor query slowdown: 1 ms per query term.

Index Block Size (in bytes)	1,024	4,096	16,384	65,536	262,144
No. of in-memory dict. entries ($\times 10^6$)	3.01	0.91	0.29	0.10	0.04
Avg. index access latency (in ms)	11.4	11.6	12.3	13.6	14.9

(but not all of them) are kept in memory. When the search engine needs to determine the location of a term's postings list, it first performs a binary search on the sorted list of in-memory dictionary entries, followed by a sequential scan of the data found between two such entries. For the example shown in the figure, a search for "shakespeareanism" would first determine that the term's postings list (if it appears in the index) must be between the lists for "shakespeare" and "shaking". It would then load this index range into memory and scan it in a linear fashion to find the dictionary entry (and thus the postings list) for the term "shakespeareanism".

Dictionary interleaving is very similar to the self-indexing technique from Section 4.3, in the sense that random access disk operations are avoided by reading a little bit of extra data in a sequential manner. Because sequential disk operations are so much faster than random access, this trade-off is usually worthwhile, as long as the additional amount of data transferred from disk into main memory is small. In order to make sure that this is the case, we need to define an upper limit for the amount of data found between each on-disk dictionary entry and the closest preceding in-memory dictionary entry. We call this upper limit the *index block size*. For instance, if it is guaranteed for every term T in the index that the search engine does not need to read more than 1,024 bytes of on-disk data before it reaches T's on-disk dictionary entry, then we say that the index has a block size of 1,024 bytes.

Table 4.4 quantifies the impact that dictionary interleaving has on the memory consumption and the list access performance of the search engine (using GOV2 as a test collection). Without interleaving, the search engine needs to maintain about 49.5 million in-memory dictionary entries and can access the first posting in a random postings list in 11.3 ms on average (random disk seek + rotational latency). Choosing a block size of $B = 1{,}024$ bytes, the number of in-memory dictionary entries can be reduced to 3 million. At the same time, the search engine's list access latency (accessing the first posting in a randomly chosen list) increases by only 0.1 ms — a negligible overhead. As we increase the block size, the number of in-memory dictionary entries goes down and the index access latency goes up. But even for a relatively large block size of $B = 256$ KB, the additional cost — compared with a complete in-memory dictionary — is only a few milliseconds per query term.

Note that the memory consumption of an interleaved dictionary with block size B is quite different from maintaining an in-memory dictionary entry for every B bytes of index data. For example, the total size of the (compressed) schema-independent index for GOV2 is 62 GB. Choosing an index block size of $B = 64$ KB, however, does not lead to $62\,\text{GB} \,/\, 64\,\text{KB} \approx 1$ million

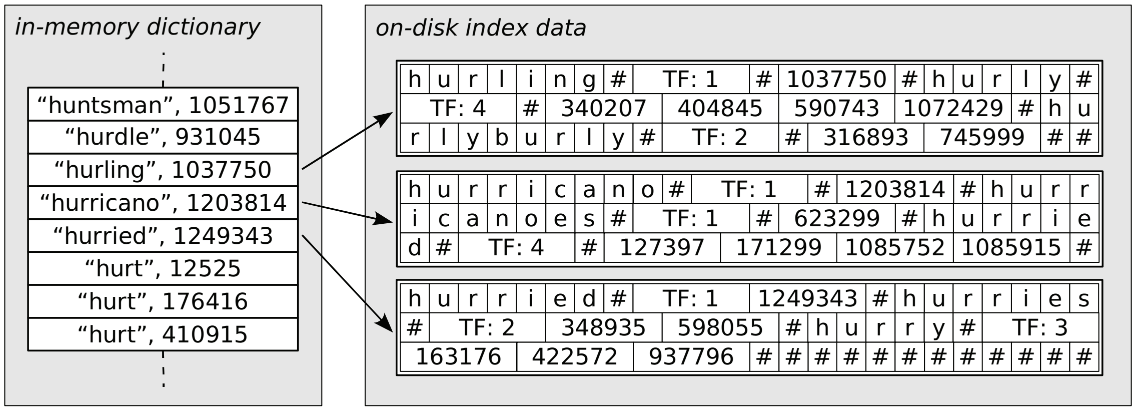

in-memory dictionary
"huntsman", 1051767
"hurdle", 931045
"hurling", 1037750
"hurricano", 1203814
"hurried", 1249343
"hurt", 12525
"hurt", 176416
"hurt", 410915
on-disk index data
h u r l i n g # TF: 1 # 1037750 # h u r l y #
TF: 4 # 340207 404845 590743 1072429 # h u
r l y b u r l y # TF: 2 # 316893 745999 # #
h u r r i c a n o # TF: 1 # 1203814 # h u r r
i c a n o e s # TF: 1 # 623299 # h u r r i e
d # TF: 4 # 127397 171299 1085752 1085915 #
h u r r i e d # TF: 1 1249343 # h u r r i e s
# TF: 2 348935 598055 # h u r r y # TF: 3
163176 422572 937796 # # # # # # # # # # # #

Figure 4.6 Combining dictionary and postings lists. The index is split into blocks of 72 bytes. Each entry of the in-memory dictionary is of the form (*term*, *posting*), indicating the first term and first posting in a given index block. The "#" symbols in the index data are record delimiters that have been inserted for better readability.

dictionary entries, but 10 times less. The reason is that frequent terms, such as "the" and "of", require only a single in-memory dictionary entry each, even though their postings lists each consume far more disk space than 64 KB (the compressed list for "the" consumes about 1 GB).

In practice, a block size between 4 KB and 16 KB is usually sufficient to shrink the in-memory dictionary to an acceptable size, especially if dictionary compression (Section 6.4) is used to decrease the space requirements of the few remaining in-memory dictionary entries. The disk transfer overhead for this range of block sizes is less than 1 ms per query term and is rather unlikely to cause any performance problems.

Dropping the distinction between terms and postings

We may take the dictionary interleaving approach one step further by dropping the distinction between terms and postings altogether, and by thinking of the index data as a sequence of pairs of the form (*term*, *posting*). The on-disk index is then divided into fixed-size index blocks, with each block perhaps containing 64 KB of data. All postings are stored on disk, in alphabetical order of their respective terms. Postings for the same term are stored in increasing order, as before. Each term's dictionary entry is stored on disk, potentially multiple times, so that there is a dictionary entry in every index block that contains at least one posting for the term. The in-memory data structure used to access data in the on-disk index then is a simple array, containing for each index block a pair of the form (*term*, *posting*), where *term* is the first term in the given block, and *posting* is the first posting for *term* in that block.

An example of this new index layout is shown in Figure 4.6 (data taken from a schema-independent index for the Shakespeare collection). In the example, a call to

$$\textbf{next}(\text{“hurried”},\ 1{,}000{,}000)$$

would load the second block shown (starting with “hurricano”) from disk, would search the block for a posting matching the query, and would return the first matching posting (1,085,752). A call to

$$\textbf{next}(\text{“hurricano”},\ 1{,}000{,}000)$$

would load the first block shown (starting with “hurling”), would not find a matching posting in that block, and would then access the second block, returning the posting 1,203,814.

The combined representation of dictionary and postings lists unifies the interleaving method explained above and the self-indexing technique described in Section 4.3 in an elegant way. With this index layout, a random access into an arbitrary term's postings list requires only a single disk seek (we have eliminated the initialization step in which the term's per-term index is loaded into memory). On the downside, however, the total memory consumption of the index is higher than if we employ self-indexing and dictionary interleaving as two independent techniques. A 62-GB index with block size B = 64 KB now in fact requires approximately 1 million in-memory entries.

4.5 Index Construction

In the previous sections of this chapter, you have learned about various components of an inverted index. In conjunction, they can be used to realize efficient access to the contents of the index, even if all postings lists are stored on disk, and even if we don't have enough memory resources to hold a complete dictionary for the index in RAM. We now discuss how to efficiently construct an inverted index for a given text collection.

From an abstract point of view, a text collection can be thought of as a sequence of term occurrences — tuples of the form (*term*, *position*), where *position* is the word count from the beginning of the collection, and *term* is the term that occurs at that position. Figure 4.7 shows a fragment of the Shakespeare collection under this interpretation. Naturally, when a text collection is read in a sequential manner, these tuples are ordered by their second component, the position in the collection. The task of constructing an index is to change the ordering of the tuples so that they are sorted by their first component, the term they refer to (ties are broken by the second component). Once the new order is established, creating a proper index, with all its auxiliary data structures, is a comparatively easy task.

In general, index construction methods can be divided into two categories: in-memory index construction and disk-based index construction. In-memory indexing methods build an index for a given text collection entirely in main memory and can therefore only be used if the collection

Text fragment:

⟨SPEECH⟩
⟨SPEAKER⟩ JULIET ⟨/SPEAKER⟩
⟨LINE⟩ O Romeo, Romeo! wherefore art thou Romeo? ⟨/LINE⟩
⟨LINE⟩ . . .

Original tuple ordering (collection order):

. . ., ("⟨speech⟩", 915487), ("⟨speaker⟩", 915488), ("juliet", 915489),
("⟨/speaker⟩", 915490), ("⟨line⟩", 915491), ("o", 915492), ("romeo", 915493),
("romeo", 915494), ("wherefore", 915495), ("art", 915496), ("thou", 915497),
("romeo", 915498), ("⟨/line⟩", 915499), ("⟨line⟩", 915500), . . .

New tuple ordering (index order):

. . ., ("⟨line⟩", 915491), ("⟨line⟩", 915500), . . .,
("romeo", 915411), ("romeo", 915493), ("romeo", 915494), ("romeo", 915498),
. . ., ("wherefore", 913310), ("wherefore", 915495), ("wherefore", 915849), . . .

Figure 4.7 The index construction process can be thought of as reordering the tuple sequence that constitutes the text collection. Tuples are rearranged from their original *collection order* (sorted by position) to their new *index order* (sorted by term).

is small relative to the amount of available RAM. They do, however, form the basis for more sophisticated, disk-based index construction methods. Disk-based methods can be used to build indices for very large collections, much larger than the available amount of main memory.

As before, we limit ourselves to schema-independent indices because this allows us to ignore some details that need to be taken care of when discussing more complicated index types, such as document-level positional indices, and lets us focus on the essential aspects of the algorithms. The techniques presented, however, can easily be applied to other index types.

4.5.1 In-Memory Index Construction

Let us consider the simplest form of index construction, in which the text collection is small enough for the index to fit entirely into main memory. In order to create an index for such a collection, the indexing process needs to maintain the following data structures:

- a dictionary that allows efficient single-term lookup and insertion operations;
- an extensible (i.e., dynamic) list data structure that is used to store the postings for each term.

Assuming these two data structures are readily available, the index construction procedure is straightforward, as shown in Figure 4.8. If the right data structures are used for dictionary and extensible postings lists, then this method is very efficient, allowing the search engine to build

```
buildIndex (inputTokenizer) ≡
    $position \leftarrow 0$
    while $inputTokenizer.hasNext()$ do
        $T \leftarrow inputTokenizer.getNext()$
        obtain dictionary entry for $T$; create new entry if necessary
        append new posting $position$ to $T$'s postings list
        $position \leftarrow position + 1$
    sort all dictionary entries in lexicographical order
    for each term $T$ in the dictionary do
        write $T$'s postings list to disk
    write the dictionary to disk
    return
```

Figure 4.8 In-memory index construction algorithm, making use of two abstract data types: in-memory dictionary and extensible postings lists.

an index for the Shakespeare collection in less than one second. Thus, there are two questions that need to be discussed: 1. What data structure should be used for the dictionary? 2. What data structure should be used for the extensible postings lists?

Indexing-time dictionary

The dictionary implementation used during index construction needs to provide efficient support for single-term lookups and term insertions. Data structures that support these kinds of operations are very common and are therefore part of many publicly available programming libraries. For example, the C++ Standard Template Library (STL) published by SGI[1] provides a `map` data structure (binary search tree), and a `hash_map` data structure (variable-size hash table) that can carry out the lookup and insert operations performed during index construction. If you are implementing your own indexing algorithm, you might be tempted to use one of these existing implementations. However, it is not always advisable to do so.

We measured the performance of the STL data structures on a subset of GOV2, indexing the first 10,000 documents in the collection. The results are shown in Table 4.5. At first sight, it seems that the lookup performance of STL's `map` and `hash_map` implementations is sufficient for the purposes of index construction. On average, `map` needs about 630 ns per lookup; `hash_map` is a little faster, requiring 240 ns per lookup operation.

Now suppose we want to index the entire GOV2 collection instead of just a 10,000-document subset. Assuming the lookup performance stays at roughly the same level (an optimistic assumption, since the number of terms in the dictionary will keep increasing), the total time spent on

[1] `www.sgi.com/tech/stl/`

`hash_map` lookup operations, one for each of the 44 billion tokens in GOV, is:

$$44 \times 10^9 \times 240 \text{ ns} \;=\; 10{,}560 \text{ sec} \;\approx\; 3 \text{ hrs.}$$

Given that the fastest publicly available retrieval systems can build an index for the same collection in about 4 hours, including reading the input files, parsing the documents, and writing the (compressed) index to disk, there seems to be room for improvement. This calls for a custom dictionary implementation.

When designing our own dictionary implementation, we should try to optimize its performance for the specific type of data that we are dealing with. We know from Chapter 1 that term occurrences in natural-language text data roughly follow a Zipfian distribution. A key property of such a distribution is that the vast majority of all tokens in the collection correspond to a surprisingly small number of terms. For example, although there are about 50 million different terms in the GOV2 corpus, more than 90% of all term occurrences correspond to one of the 10,000 most frequent terms. Therefore, the bulk of the dictionary's lookup load may be expected to stem from a rather small set of very frequent terms. If the dictionary is based on a hash table, and collisions are resolved by means of chaining (as in Figure 4.2), then it is crucial that the dictionary entries for those frequent terms are kept near the beginning of each chain. This can be realized in two different ways:

1. **The insert-at-back heuristic**
   If a term is relatively frequent, then it is likely to occur very early in the stream of incoming tokens. Conversely, if the first occurrence of a term is encountered rather late, then chances are that the term is not very frequent. Therefore, when adding a new term to an existing chain in the hash table, it should be inserted at the end of the chain. This way, frequent terms, added early on, stay near the front, whereas infrequent terms tend to be found at the end.
2. **The move-to-front heuristic**
   If a term is frequent in the text collection, then it should be at the beginning of its chain in the hash table. Like insert-at-back, the move-to-front heuristic inserts new terms at the end of their respective chain. In addition, however, whenever a term lookup occurs and the term descriptor is not found at the beginning of the chain, it is relocated and moved to the front. This way, when the next lookup for the term takes place, the term's dictionary entry is still at (or near) the beginning of its chain in the hash table.

We evaluated the lookup performance of these two alternatives (using a handcrafted hash table implementation with a fixed number of hash slots in both cases) and compared it against a third alternative that inserts new terms at the beginning of the respective chain (referred to as the *insert-at-front* heuristic). The outcome of the performance measurements is shown in Table 4.5.

The insert-at-back and move-to-front heuristics achieve approximately the performance level (90 ns per lookup for a hash table with 2^{14} slots). Only for a small table size does move-to-front have a slight edge over insert-at-back (20% faster for a table with 2^{10} slots). The insert-at-front

Table 4.5 Indexing the first 10,000 documents of GOV2 (≈ 14 million tokens; 181,334 distinct terms). Average dictionary lookup time per token in microseconds. The rows labeled "Hash table" represent a handcrafted dictionary implementation based on a fixed-size hash table with chaining.

Dictionary Implementation	Lookup Time	String Comparisons
Binary search tree (STL `map`)	0.63 μs per token	18.1 per token
Variable-size hash table (STL `hash_map`)	0.24 μs per token	2.2 per token
Hash table (2^{10} entries, insert-at-front)	6.11 μs per token	140 per token
Hash table (2^{10} entries, insert-at-back)	0.37 μs per token	8.2 per token
Hash table (2^{10} entries, move-to-front)	0.31 μs per token	4.8 per token
Hash table (2^{14} entries, insert-at-front)	0.32 μs per token	10.1 per token
Hash table (2^{14} entries, insert-at-back)	0.09 μs per token	1.5 per token
Hash table (2^{14} entries, move-to-front)	0.09 μs per token	1.3 per token

heuristic, however, exhibits a very poor lookup performance and is between 3 and 20 times slower than move-to-front. To understand the origin of this extreme performance difference, look at the table column labeled "String Comparisons". When storing the 181,344 terms from the 10,000-document subcollection of GOV2 in a hash table with 2^{14} = 16,384 slots, we expect about 11 terms per chain on average. With the insert-at-front heuristic, the dictionary — on average — performs 10.1 string comparisons before it finds the term it is looking for. Because the frequent terms tend to appear early on in the collection, insert-at-front places them at the end of the respective chain. This is the cause of the method's dismal lookup performance. Incidentally, STL's `hash_map` implementation also inserts new hash table entries at the beginning of the respective chain, which is one reason why it does not perform so well in this benchmark.

A hash-based dictionary implementation employing the move-to-front heuristic is largely insensitive to the size of the hash table. In fact, even when indexing text collections containing many millions of different terms, a relatively small hash table, containing perhaps 2^{16} slots, will be sufficient to realize efficient dictionary lookups.

Extensible in-memory postings lists

The second interesting aspect of the simple in-memory index construction method, besides the dictionary implementation, is the implementation of the extensible in-memory postings lists. As suggested earlier, realizing each postings list as a singly linked list allows incoming postings to be appended to the existing lists very efficiently. The disadvantage of this approach is its rather high memory consumption: For every 32-bit (or 64-bit) posting, the indexing process needs to store an additional 32-bit (or 64-bit) pointer, thus increasing the total space requirements by 50–200%.

Various methods to decrease the storage overhead of the extensible postings lists have been proposed. For example, one could use a fixed-size array instead of a linked list. This avoids the

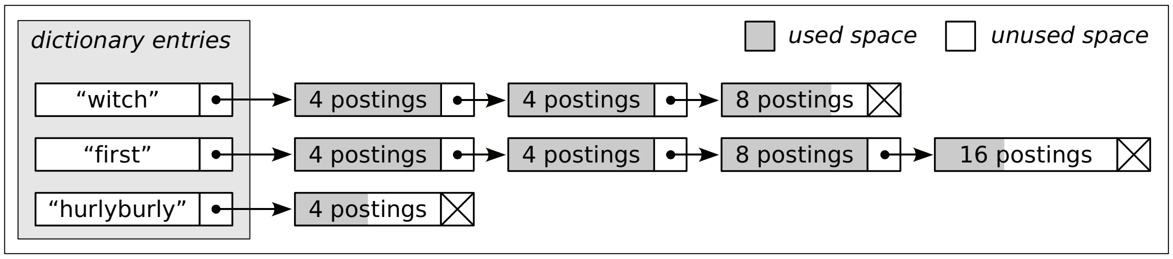


Figure 4.9 Realizing per-term extensible postings list by linking between groups of postings (*unrolled linked list*). Proportional pre-allocation (here with pre-allocation factor $k = 2$) provides an attractive trade-off between space overhead due to `next` pointers and space overhead due to internal fragmentation.

storage overhead associated with the `next` pointers in the linked list. Unfortunately, the length of each term's postings list is not known ahead of time, so this method requires an additional pass over the input data: In the first pass, term statistics are collected and space is allocated for each term's postings list; in the second pass, the pre-allocated arrays are filled with postings data. Of course, reading the input data twice harms indexing performance.

Another solution is to pre-allocate a postings array for every term in the dictionary, but to allow the indexing process to change the size of the array, for example, by performing a call to `realloc` (assuming that the programming language and run-time environment support this kind of operation). When a novel term is encountered, *init* bytes are allocated for the term's postings (e.g., $init = 16$). Whenever the capacity of the existing array is exhausted, a `realloc` operation is performed in order to increase the size of the array. By employing a proportional pre-allocation strategy, that is, by allocating a total of

$$s_{\text{new}} = \max\{s_{\text{old}} + init, k \times s_{\text{old}}\} \tag{4.2}$$

bytes in a reallocation operation, where s_{old} is the old size of the array and k is a constant, called the *pre-allocation factor*, the number of calls to `realloc` can be kept small (logarithmic in the size of each postings list). Nonetheless, there are some problems with this approach. If the pre-allocation factor is too small, then a large number of list relocations is triggered during index construction, possibly affecting the search engine's indexing performance. If it is too big, then a large amount of space is wasted due to allocated but unused memory (internal fragmentation). For instance, if $k = 2$, then 25% of the allocated memory will be unused on average.

A third way of realizing extensible in-memory postings lists with low storage overhead is to employ a linked list data structure, but to have multiple postings share a `next` pointer by linking to *groups* of postings instead of individual postings. When a new term is inserted into the dictionary, a small amount of space, say 16 bytes, is allocated for its postings. Whenever the space reserved for a term's postings list is exhausted, space for a new group of postings is

Table 4.6 Indexing the TREC45 collection. Index construction performance for various memory allocation strategies used to manage the extensible in-memory postings lists (32-bit postings; 32-bit pointers). Arranging postings for the same term in small groups and linking between these groups is faster than any other strategy. Pre-allocation factor used for both `realloc` and *grouping*: $k = 1.2$.

Allocation Strategy	**Memory Consumption**	**Time (Total)**	**Time (CPU)**
Linked list (simple)	2,312 MB	88 sec	77 sec
Two-pass indexing	1,168 MB	123 sec	104 sec
`realloc`	1,282 MB	82 sec	71 sec
Linked list (grouping)	1,208 MB	71 sec	61 sec

allocated. The amount of space allotted to the new group is

$$s_{\text{new}} = \min\{limit, \max\{16, (k-1) \times s_{\text{total}}\}\}, \tag{4.3}$$

where *limit* is an upper bound on the size of a postings group, say, 256 postings, s_{total} is the amount of memory allocated for postings so far, and k is the pre-allocation factor, just as in the case of `realloc`.

This way of combining postings into groups and having several postings share a single `next` pointer is shown in Figure 4.9. A linked list data structure that keeps more than one value in each list node is sometimes called an *unrolled linked list*, in analogy to loop unrolling techniques used in compiler optimization. In the context of index construction, we refer to this method as *grouping*.

Imposing an upper limit on the amount of space pre-allocated by the grouping technique allows to control internal fragmentation. At the same time, it does not constitute a performance problem; unlike in the case of `realloc`, allocating more space for an existing list is a very lightweight operation and does not require the indexing process to relocate any postings data.

A comparative performance evaluation of all four list allocation strategies — simple linked list, two-pass, `realloc`, and linked list with grouping — is given by Table 4.6. The two-pass method exhibits the lowest memory consumption but takes almost twice as long as the grouping approach. Using a pre-allocation factor $k = 1.2$, the `realloc` method has a memory consumption that is about 10% above that of the space-optimal two-pass strategy. This is consistent with the assumption that about half of the pre-allocated memory is never used. The grouping method, on the other hand, exhibits a memory consumption that is only 3% above the optimum. In addition, grouping is about 16% faster than `realloc` (61 sec vs. 71 sec CPU time).

Perhaps surprisingly, linking between groups of postings is also faster than linking between individual postings (61 sec vs. 77 sec CPU time). The reason for this is that the unrolled linked list data structure employed by the grouping technique not only decreases internal fragmentation, but also improves CPU cache efficiency, by keeping postings for the same term close

```
buildIndex_sortBased (inputTokenizer) ≡
    position ← 0
    while inputTokenizer.hasNext() do
        T ← inputTokenizer.getNext()
        obtain dictionary entry for T; create new entry if necessary
        termID ← unique term ID of T
        write record $R_{position} \equiv (termID, position)$ to disk
        position ← position + 1
    tokenCount ← position
    sort $R_0 \ldots R_{tokenCount-1}$ by first component; break ties by second component
    perform a sequential scan of $R_0 \ldots R_{tokenCount-1}$, creating the final index
    return
```

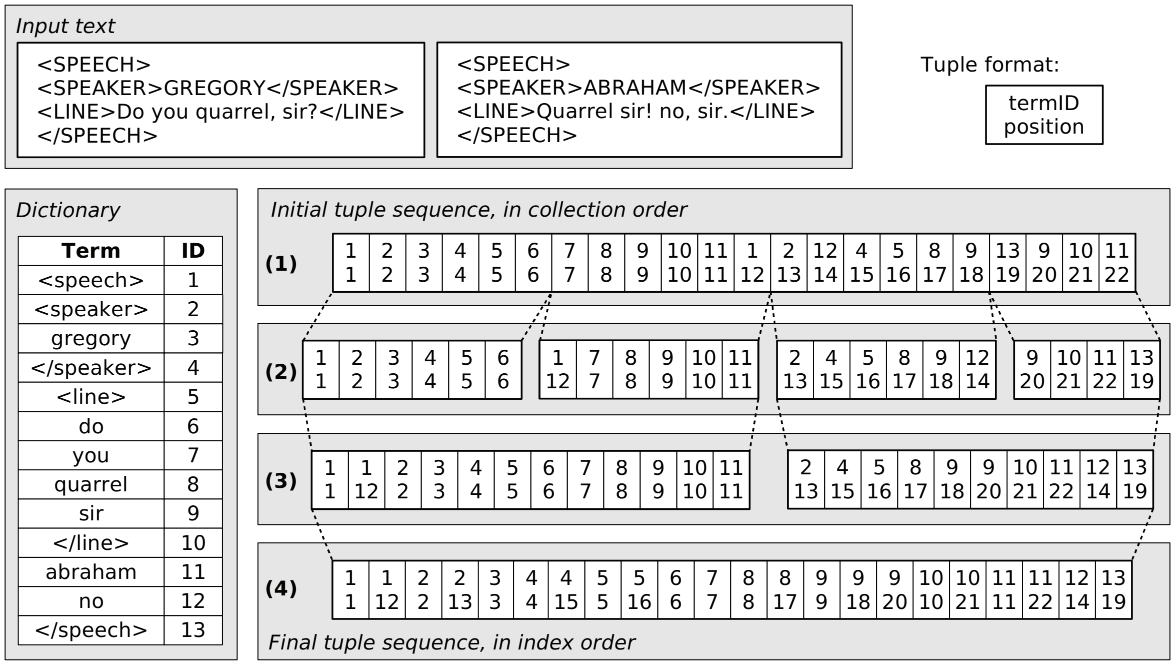
Figure 4.10 Sort-based index construction algorithm, creating a schema-independent index for a text collection. The main difficulty lies in efficiently sorting the on-disk records $R_0 \ldots R_{tokenCount-1}$.

together. In the simple linked-list implementation, postings for a given term are randomly scattered across the used portion of the computer's main memory, resulting in a large number of CPU cache misses when collecting each term's postings before writing them to disk.

4.5.2 Sort-Based Index Construction

Hash-based in-memory index construction algorithms can be extremely efficient, as demonstrated in the previous section. However, if we want to index text collections that are substantially larger than the available amount of RAM, we need to move away from the idea that we can keep the entire index in main memory, and need to look toward disk-based approaches instead. The sort-based indexing method presented in this section is the first of two disk-based index construction methods covered in this chapter. It can be used to index collections that are much larger than the available amount of main memory.

Recall from the beginning of this section that building an inverted index can be thought of as the process of reordering the sequence of term-position tuples that represents the text collection — transforming it from collection order into index order. This is a sorting process, and sort-based index construction realizes the transformation in the simplest way possible. Consuming tokens from the input files, a sort-based indexing process emits records of the form (*termID*, *position*) and writes them to disk immediately. The result is a sequence of records, sorted by their second component (*position*). When it is done processing the input data, the indexer sorts all tuples written to disk by their first component, using the second component to break ties. The result is a new sequence of records, sorted by their first component (*termID*). Transforming this new sequence into a proper inverted file, using the information found in the in-memory dictionary, is straightforward.


Input text
Tuple format:
termID
position
Dictionary
Initial tuple sequence, in collection order
(1)
(2)
(3)
(4)
Final tuple sequence, in index order

```
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>Do you quarrel, sir?</LINE>
</SPEECH>
```

```
<SPEECH>
<SPEAKER>ABRAHAM</SPEAKER>
<LINE>Quarrel sir! no, sir.</LINE>
</SPEECH>
```

Term	ID
<speech>	1
<speaker>	2
gregory	3
</speaker>	4
<line>	5
do	6
you	7
quarrel	8
sir	9
</line>	10
abraham	11
no	12
</speech>	13

```
(1)  1 2 3 4 5 6 7 8 9 10 11  1  2 12  4  5  8  9 13  9 10 11
     1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

(2)  1 2 3 4 5 6 |  1 7 8 9 10 11 |  2  4  5  8  9 12 |  9 10 11 13
     1 2 3 4 5 6 | 12 7 8 9 10 11 | 13 15 16 17 18 14 | 20 21 22 19

(3)  1  1 2 3 4 5 6 7 8 9 10 11 |  2  4  5  8  9  9 10 11 12 13
     1 12 2 3 4 5 6 7 8 9 10 11 | 13 15 16 17 18 20 21 22 14 19

(4)  1  1 2  2 3 4  4 5  5 6 7 8  8 9  9  9 10 10 11 11 12 13
     1 12 2 13 3 4 15 5 16 6 7 8 17 9 18 20 10 21 11 22 14 19
```

Figure 4.11 Sort-based index construction with global term IDs. Main memory is large enough to hold 6 (*termID*, *position*) tuples at a time. (1)→(2): sorting blocks of size ≤ 6 in memory, one at a time. (2)→(3) and (3)→(4): merging sorted blocks into bigger blocks.

The method, shown as pseudo-code in Figure 4.10, is very easy to implement and can be used to create an index that is substantially larger than the available amount of main memory. Its main limitation is the available amount of disk space.

Sorting the on-disk records can be a bit tricky. In many implementations, it is performed by loading a certain number of records, say n, into main memory at a time, where n is defined by the size of a record and the amount of available main memory. These n records are then sorted in memory and written back to disk. The process is repeated until we have $\lceil \frac{tokenCount}{n} \rceil$ blocks of sorted records. These blocks can then be combined, in a *multiway merge operation* (processing records from all blocks at the same time) or in a *cascaded merge operation* (merging, for example, two blocks at a time), resulting in a sorted sequence of *tokenCount* records. Figure 4.11 shows a sort-based indexing process that creates the final tuple sequence by means of a cascaded merge operation, merging two tuple sequences at a time. With some additional data structures, and with the *termID* component removed from each posting, this final sequence can be thought of as the index for the collection.

Despite its ability to index text collections much larger than the available amount of main memory, sort-based indexing has two essential limitations:

- It requires a substantial amount of disk space, usually at least 8 bytes per input token (4 bytes for the term ID + 4 bytes for the position), or even 12 bytes (4 + 8) for larger text collections. When indexing GOV2, for instance, the disk space required to store the temporary files is at least $12 \times 44 \times 10^9$ bytes (= 492 GB) — more than the uncompressed size of the collection itself (426 GB).
- To be able to properly sort the (*termID*, *docID*) pairs emitted in the first phase, the indexing process needs to maintain globally unique term IDs, which can be realized only through a complete in-memory dictionary. As discussed earlier, a complete dictionary for GOV2 might consume more than 1 GB of RAM, making it difficult to index the collection on a low-end machine.

There are various ways to address these issues. However, in the end they all have in common that they transform the sort-based indexing method into something more similar to what is known as *merge-based index construction*.

4.5.3 Merge-Based Index Construction

In contrast to sort-based index construction methods, the merge-base method presented in this section does not need to maintain any global data structures. In particular, there is no need for globally unique term IDs. The size of the text collection to be indexed is therefore limited only by the amount of disk space available to store the temporary data and the final index, but not by the amount of main memory available to the indexing process.

Merge-based indexing is a generalization of the in-memory index construction method discussed in Section 4.5.1, building inverted lists by means of hash table lookup. In fact, if the collection for which an index is being created is small enough for the index to fit completely into main memory, then merge-based indexing behaves exactly like in-memory indexing. If the text collection is too large to be indexed completely in main memory, then the indexing process performs a *dynamic partitioning* of the collection. That is, it starts building an in-memory index. As soon as it runs out of memory (or when a predefined memory utilization threshold is reached), it builds an on-disk inverted file by transferring the in-memory index data to disk, deletes the in-memory index, and continues indexing. This procedure is repeated until the whole collection has been indexed. The algorithm is shown in Figure 4.12.

The result of the process outlined above is a set of inverted files, each representing a certain part of the whole collection. Each such subindex is referred to as an *index partition*. In a final step, the index partitions are merged into the final index, representing the entire text collection. The postings lists in the index partitions (and in the final index) are usually stored in compressed form (see Chapter 6), in order to keep the disk I/O overhead low.

The index partitions written to disk as intermediate output of the indexing process are completely independent of each other. For example, there is no need for globally unique term IDs; there is not even a need for numerical term IDs. Each term is its own ID; the postings lists in each partition are stored in lexicographical order of their terms, and access to a term's list can

```
buildIndex_mergeBased (inputTokenizer, memoryLimit) ≡
    $n \leftarrow 0$   // initialize the number of index partitions
    $position \leftarrow 0$
    $memoryConsumption \leftarrow 0$
    while $inputTokenizer.hasNext()$ do
        $T \leftarrow inputTokenizer.getNext()$
        obtain dictionary entry for $T$; create new entry if necessary
        append new posting $position$ to $T$'s postings list
        $position \leftarrow position + 1$
        $memoryConsumption \leftarrow memoryConsumption + 1$
        if $memoryConsumption \geq memoryLimit$ then
            createIndexPartition()
    if $memoryConsumption > 0$ then
        createIndexPartition()
    merge index partitions $I_0 \ldots I_{n-1}$, resulting in the final on-disk index $I_{\text{final}}$
    return

createIndexPartition () ≡
    create empty on-disk inverted file $I_n$
    sort in-memory dictionary entries in lexicographical order
    for each term $T$ in the dictionary do
        add $T$'s postings list to $I_n$
    delete all in-memory postings lists
    reset the in-memory dictionary
    $memoryConsumption \leftarrow 0$
    $n \leftarrow n + 1$
    return
```

Figure 4.12 Merge-based indexing algorithm, creating a set of independent sub-indices (*index partitions*). The final index is generated by combining the sub-indices via a multi-way merge operation.

be realized by using the data structures described in Sections 4.3 and 4.4. Because of the lexicographical ordering and the absence of term IDs, merging the individual partitions into the final index is straightforward. Pseudo-code for a very simple implementation, performing repeated linear probing of all subindices, is given in Figure 4.13. If the number of index partitions is large (more than 10), then the algorithm can be improved by arranging the index partitions in a priority queue (e.g., a heap), ordered according to the next term in the respective partition. This eliminates the need for the linear scan in lines 7–10.

Overall performance numbers for merge-based index construction, including all components, are shown in Table 4.7. The total time necessary to build a schema-independent index for GOV2 is around 4 hours. The time required to perform the final merge operation, combining the n index partitions into one final index, is about 30% of the time it takes to generate the partitions.

```
mergeIndexPartitions ($\langle I_0, \ldots, I_{n-1} \rangle$) $\equiv$
    create empty inverted file $I_{\text{final}}$
    for $k \leftarrow 0$ to $n - 1$ do
        open index partition $I_k$ for sequential processing
    $currentIndex \leftarrow 0$
    while $currentIndex \neq nil$ do
        $currentIndex \leftarrow nil$
        for $k \leftarrow 0$ to $n - 1$ do
            if $I_k$ still has terms left then
                if $(currentIndex = nil) \lor (I_k.currentTerm < currentTerm)$ then
                    $currentIndex \leftarrow I_k$
                    $currentTerm \leftarrow I_k.currentTerm$
        if $currentIndex \neq nil$ then
            $I_{\text{final}}.addPostings(currentTerm, currentIndex.getPostings(currentTerm))$
            $currentIndex.advanceToNextTerm()$
    delete $I_0 \ldots I_{n-1}$
    return
```

Figure 4.13 Merging a set of n index partitions $I_0 \ldots I_{n-1}$ into an index I_{final}. This is the final step in merge-based index construction.

The algorithm is very scalable: On our computer, indexing the whole GOV2 collection (426 GB of text) takes only 11 times as long as indexing a 10% subcollection (43 GB of text).

There are, however, some limits to the scalability of the method. When merging the index partitions at the end of the indexing process, it is important to have at least a moderately sized read-ahead buffer, a few hundred kilobytes, for each partition. This helps keep the number of disk seeks (jumping back and forth between the different partitions) small. Naturally, the size of the read-ahead buffer for each partition is bounded from above by M/n, where M is the amount of available memory and n is the number of partitions. Thus, if n becomes too large, merging becomes slow.

Reducing the amount of memory available to the indexing process therefore has two effects. First, it decreases the total amount of memory available for the read-ahead buffers. Second, it increases the number of index partitions. Thus, reducing main memory by 50% decreases the size of each index partition's read-ahead buffer by 75%. Setting the memory limit to $M = 128$ MB, for example, results in 3,032 partitions that need to be merged, leaving each partition with a read-ahead buffer of only 43 KB. The general trend of this effect is depicted in Figure 4.14. The figure shows that the performance of the final merge operation is highly dependent on the amount of main memory available to the indexing process. With 128 MB of available main memory, the final merge takes 6 times longer than with 1,024 MB.

There are two possible countermeasures that could be taken to overcome this limitation. The first is to replace the simple multiway merge by a cascaded merge operation. For instance, if 1,024 index partitions need to be merged, then we could first perform 32 merge operations

Table 4.7 Building a schema-independent index for various text collections, using merge-based index construction with 512 MB of RAM for the in-memory index. The indexing-time dictionary is realized by a hash table with 2^{16} entries and move-to-front heuristic. The extensible in-memory postings lists are unrolled linked lists, linking between groups of postings, with a pre-allocation factor $k = 1.2$.

	Reading, Parsing & Indexing	**Merging**	**Total Time**
Shakespeare	1 sec	0 sec	1 sec
TREC45	71 sec	11 sec	82 sec
GOV2 (10%)	20 min	4 min	24 min
GOV2 (25%)	51 min	11 min	62 min
GOV2 (50%)	102 min	25 min	127 min
GOV2 (100%)	205 min	58 min	263 min

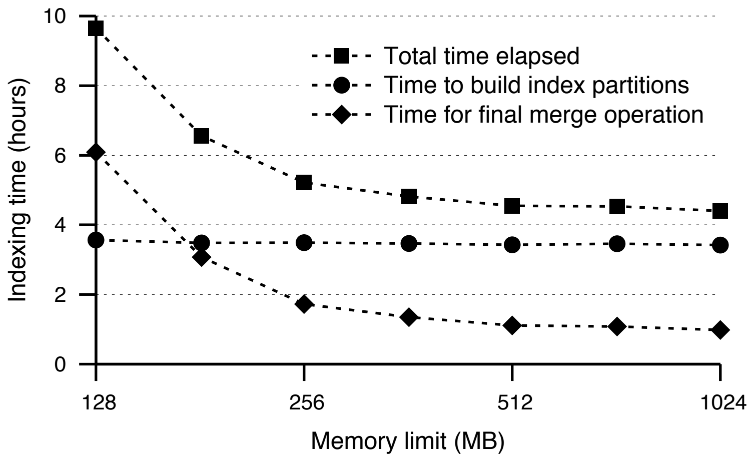


Figure 4.14 The impact of available RAM on the performance of merge-based indexing (data set: GOV2). The performance of phase 1 (building index partitions) is largely independent of the amount of available main memory. The performance of phase 2 (merging partitions) suffers severely if little main memory is available.

involving 32 partitions each, and then merge the resulting 32 new partitions into the final index. This is a generalization of the process shown in Figure 4.11, which depicts a cascaded merge operation that works on 2 partitions at a time. The second countermeasure is to decrease the space consumed by the postings lists, by means of compressed in-memory inversion. Compressing postings on-the-fly, as they enter the in-memory index, allows the indexing process to accumulate more postings before it runs out of memory, thus decreasing the number of on-disk index partitions created.

In conclusion, despite some problems with the final merge operation, merge-based index construction lets us build an inverted file for very large text collections, even on a single PC. Its advantage over the sort-based method is that it does not require globally unique term IDs. It is therefore especially attractive if the number of dictionary terms is very large. Another great advantage of this index construction algorithm over the sort-based method is that it produces an in-memory index that is immediately queriable. This feature is essential when the search engine needs to deal with dynamic text collections (see Chapter 7).

4.6 Other Types of Indices

In our discussion of index data structures, we have limited ourselves to the case of inverted indices. However, an inverted index is just one possible type of index that can be used as part of a search engine.

A *forward index* (or *direct index*) is a mapping from document IDs to the list of terms appearing in each document. Forward indices complement inverted indices. They are usually not used for the actual search process, but to obtain information about per-document term distributions at query time, which is required by query expansion techniques such as pseudo-relevance feedback (see Chapter 8), and to produce result *snippets*. Compared to extracting this information from the raw text files, a forward index has the advantage that the text has already been parsed, and the relevant data can be extracted more efficiently.

Signature files (Faloutsos and Christodoulakis, 1984) are an alternative to docid indices. Similar to a Bloom filter (Bloom, 1970), a signature file can be used to obtain a list of documents that *may* contain the given term. In order to find out whether the term actually appears in a document, the document itself (or a forward index) needs to be consulted. By changing the parameters of the signature file, it is possible to trade time for speed: A smaller index results in a greater probability of false positives, and vice versa.

Suffix trees (Weiner, 1973) and *suffix arrays* (Manber and Myers, 1990) can be used to efficiently find all occurrences of a given n-gram sequence in a given text collection. They can be used either to index character n-grams (without tokenizing the input text) or to index word n-grams (after tokenization). Suffix trees are attractive data structures for phrase search or regular expression search, but are usually larger than inverted indices and provide less efficient search operations when stored on disk instead of in RAM.

4.7 Summary

In this chapter we have covered the essential algorithms and data structures necessary to build and access inverted indices. The main points of the chapter are:

- Inverted indices are usually too large to be loaded completely into memory. It is therefore common to keep only the dictionary, which is relatively small compared to the size of the entire index, in memory, while storing the postings lists on disk (Section 4.2).
- For a large text collection even the dictionary might be too large to fit into RAM. The memory requirements of the dictionary can be substantially decreased by keeping an incomplete dictionary in memory, interleaving dictionary entries with on-disk postings lists, and exploiting the fact that all lists in the index are sorted in lexicographical order (Section 4.4).
- A sort-based dictionary implementation should be used, and postings lists should be stored on disk in lexicographical order, if prefix queries are to be supported by the search engine (Sections 4.2 and 4.3).
- For each on-disk postings list, a per-term index, containing a subset of the term's postings, can be used to realize efficient quasi-random access into the list (Section 4.3).
- Highly efficient in-memory index construction can be realized by employing a hash-based in-memory dictionary with move-to-front heuristic and by using grouped linked lists to implement extensible in-memory postings lists (Section 4.5.1).
- If the amount of main memory available to the indexing process is too small to allow the index to be built completely in RAM, then the in-memory index construction method can be extended to a merge-based method, in which the text collection is divided into subcollections, dynamically and based on the available amount of main memory. An index for each subcollection is built using the in-memory indexing method. After the entire collection has been indexed, the indices for the individual subcollections are merged in a single multiway merge operation or a cascaded merge process (Section 4.5.3).
- The performance of merge-based index construction is essentially linear in the size of the collection. However, the final merge operation may suffer severely if too little main memory is available for the subindices' read buffers (Section 4.5.3).

4.8 Further Reading

Good entry points for an in-depth understanding of the architecture and performance of inverted indices are provided by Witten et al. (1999, chapters 3 and 5) and Zobel and Moffat (2006). A high-level overview of the index data structures employed by Google around 1998 is given by Brin and Page (1998).

Moffat and Zobel (1996) discuss query-time efficiency issues of inverted files, including the per-term index data structure ("self-indexing") used for random list access that is outlined in Section 4.3. Rao and Ross (1999, 2000) demonstrate that random access issues arise not only in the context of on-disk indices but also for in-memory inverted files. They show that binary search is *not* the best way to realize random access for in-memory postings lists.

Heinz and Zobel (2003) discuss single-pass merge-based index construction and its advantages over the sort-based method. They also examine the efficiency of various in-memory dictionary implementations, including the move-to-front heuristic described in Section 4.5.1 (Zobel et al., 2001), and propose a new dictionary data structure, the *burst trie* (Heinz et al., 2002), which achieves a single-term lookup performance close to a hash table but — unlike a hash table — can also be used to resolve prefix queries.

Memory management strategies for extensible in-memory postings lists (e.g., unrolled linked lists) are examined by Büttcher and Clarke (2005) and, more recently, by Luk and Lam (2007).

With a naïve implementation of the final merge procedure in merge-based index construction (and variants of sort-based indexing), the total storage requirement is twice as large as the size of the final index. Moffat and Bell (1995) describe a clever technique that can be used to realize the merge in situ, re-using the disk space occupied by already processed parts of the input partitions to store the final index.

Faloutsos and Christodoulakis (1984) give a nice overview of signature files, including some theoretical properties. Zobel et al. (1998) discuss the relative performance of inverted files and signature files in text search. They come to the conclusion that, for many applications, inverted files are the better choice. Carterette and Can (2005) argue that, under certain circumstances, signature files can be almost as fast as inverted files.

Suffix trees made their first appearance under the name *position trees* in a paper by Weiner (1973). Ukkonen (1995) presents a linear-time construction method for suffix trees. Clark and Munro (1996) discuss a variant of suffix trees that allows efficient search operations when stored on disk instead of main memory.

4.9 Exercises

Exercise 4.1 In Section 4.3 we introduced the concept of the *per-term index* as a means to improve the index's random access performance. Suppose the postings list for some term consists of 64 million postings, each of which consumes 4 bytes. In order to carry out a single random access into the term's postings list, the search engine needs to perform two disk read operations:

1. Loading the per-term index (list of synchronization points) into RAM.
2. Loading a block B of postings into RAM, where B is identified by means of binary search on the list of synchronization points.

Let us call the number of postings per synchronization point the *granularity* of the per-term index. For the above access pattern, what is the optimal granularity (i.e., the one that minimizes disk I/O)? What is the total number of bytes read from disk?

Exercise 4.2 The introduction of the per-term index in Section 4.3 was motivated by the performance characteristics of typical hard disk drives (in particular, the high cost of disk seeks). However, the same method can also be used to improve the random access performance of in-memory indices. To confirm this claim, you have to implement two different data structures for in-memory postings lists and equip them with a **next** access method (see Chapter 2). The first data structure stores postings in a simple array of 32-bit integers. Its **next** method operates by performing a binary search on that array. In the second data structure, an auxiliary array is used to store a copy of every 64th posting in the postings list. The **next** method first performs a binary search on the auxiliary array, followed by a *sequential scan* of the 64 candidate postings in the postings list. Measure the average single-element lookup latency of both implementations, working with lists of n postings, for $n = 2^{12}, 2^{16}, 2^{20}, 2^{24}$. Describe and analyze your findings.

Exercise 4.3 Building an inverted index is essentially a sorting process. The lower bound for every general-purpose sorting algorithm is $\Omega(n \log(n))$. However, the merge-based index construction method from Section 4.5.3 has a running time that is linear in the size of the collection (see Table 4.7, page 130). Find at least two places where there is a hidden logarithmic factor.

Exercise 4.4 In the algorithm shown in Figure 4.12, the memory limit is expressed as the number of postings that can be stored in RAM. What is the assumption that justifies this definition of the memory limit? Give an example of a text collection or an application in which the assumption does not hold.

Exercise 4.5 In Section 4.5.1 we discussed the performance characteristics of various dictionary data structures. We pointed out that hash-based implementations offer better performance for single-term lookups (performed during index construction), while sort-based solutions are more appropriate for multi-term lookups (needed for prefix queries). Design and implement a data structure that offers better single-term lookup performance than a sort-based dictionary and better prefix query performance than a hash-based implementation.

Exercise 4.6 (project exercise) Design and implement an index construction method that creates a schema-independent index for a given text collection. The result of the index construction process is an on-disk index. The index does not need to use any of the optimizations discussed in Sections 4.3 and 4.4.

- Implement the in-memory index construction method described in Section 4.5.1. When run on typical English text, your implementation should be able to build an index for a collection containing approximately $\frac{M}{8}$ tokens, where M is the amount of available main memory, in bytes.

- Extend your implementation so that the size of the text collection is no longer limited by the amount of main memory available to the indexing process. This will probably require you to write a module that can merge two or more on-disk indices into a single index.

4.10 Bibliography

Bender, M., Michel, S., Triantafillou, P., and Weikum, G. (2007). Design alternatives for large-scale Web search: Alexander was great, Aeneas a pioneer, and Anakin has the force. In *Proceedings of the 1st Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*, pages 16–22. Amsterdam, The Netherlands.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

Brin, S., and Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117.

Büttcher, S., and Clarke, C. L. A. (2005). *Memory Management Strategies for Single-Pass Index Construction in Text Retrieval Systems.* Technical Report CS-2005-32. University of Waterloo, Waterloo, Canada.

Carterette, B., and Can, F. (2005). Comparing inverted files and signature files for searching a large lexicon. *Information Processing & Management*, 41(3):613–633.

Clark, D. R., and Munro, J. I. (1996). Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391. Atlanta, Georgia.

Faloutsos, C., and Christodoulakis, S. (1984). Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4):267–288.

Heinz, S., and Zobel, J. (2003). Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729.

Heinz, S., Zobel, J., and Williams, H. E. (2002). Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223.

Luk, R. W. P., and Lam, W. (2007). Efficient in-memory extensible inverted file. *Information Systems*, 32(5):733–754.

Manber, U., and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327. San Francisco, California.

Moffat, A., and Bell, T. A. H. (1995). In-situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550.

Moffat, A., and Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379.

Rao, J., and Ross, K. A. (1999). Cache conscious indexing for decision-support in main memory. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 78–89. Edinburgh, Scotland.

Rao, J., and Ross, K. A. (2000). Making B^+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486. Dallas, Texas.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

Weiner, P. (1973). Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11. Iowa City, Iowa.

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

Zobel, J., Heinz, S., and Williams, H. E. (2001). In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277.

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

Zobel, J., Moffat, A., and Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490.

6 Index Compression

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

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

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

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

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

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

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

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

6.1 General-Purpose Data Compression

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

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

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

6.2 Symbolwise Data Compression

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

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

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

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

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

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

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

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

6.2.1 Modeling and Coding

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

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

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

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

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

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

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

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

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

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

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

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

It encodes the symbol sequence "aababacd" as

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

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

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

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

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

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

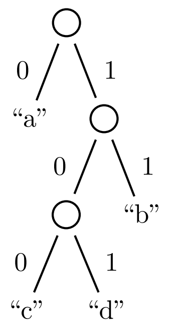


Figure 6.1 Binary tree associated with the prefix code $\mathcal{C}_0$. Codewords: $\mathcal{C}_0(\text{"a"}) = \overline{0}$, $\mathcal{C}_0(\text{"b"}) = \overline{11}$, $\mathcal{C}_0(\text{"c"}) = \overline{100}$, $\mathcal{C}_0(\text{"d"}) = \overline{101}$.

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

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

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

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

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

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

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

Let us try to find an optimal code tree for this distribution. Every node in the tree must be either a leaf node (and have a codeword associated with it) or an internal node with exactly two children. Such a tree is called a *proper binary tree*. If the tree had an internal node with only one child, then it would be possible to improve the code by removing that internal node, thus reducing the depths of its descendants by 1. Hence, the code would not be optimal.

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

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

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

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

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

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

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

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

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

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

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

6.2.2 Huffman Coding

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

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

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

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

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

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

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

Optimality

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

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

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

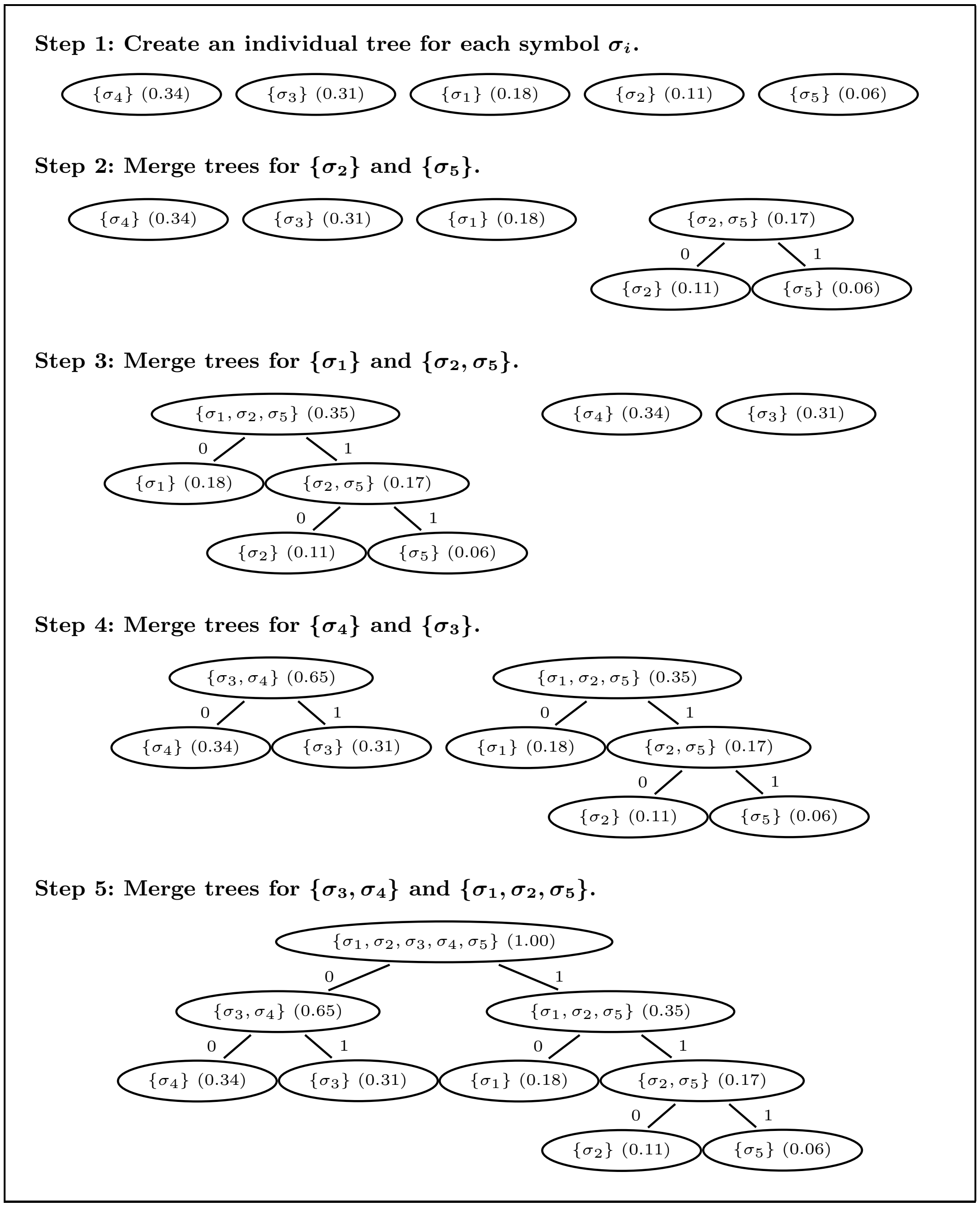

Step 1: Create an individual tree for each symbol σ_i.
$\{\sigma_4\}$ (0.34) $\{\sigma_3\}$ (0.31) $\{\sigma_1\}$ (0.18) $\{\sigma_2\}$ (0.11) $\{\sigma_5\}$ (0.06)
Step 2: Merge trees for $\{\sigma_2\}$ and $\{\sigma_5\}$.
$\{\sigma_2, \sigma_5\}$ (0.17)
Step 3: Merge trees for $\{\sigma_1\}$ and $\{\sigma_2, \sigma_5\}$.
$\{\sigma_1, \sigma_2, \sigma_5\}$ (0.35)
Step 4: Merge trees for $\{\sigma_4\}$ and $\{\sigma_3\}$.
$\{\sigma_3, \sigma_4\}$ (0.65)
Step 5: Merge trees for $\{\sigma_3, \sigma_4\}$ and $\{\sigma_1, \sigma_2, \sigma_5\}$.
$\{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ (1.00)
0
1

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

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

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

(bits per symbol). Consider the symbol set

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

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

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

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

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

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

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

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

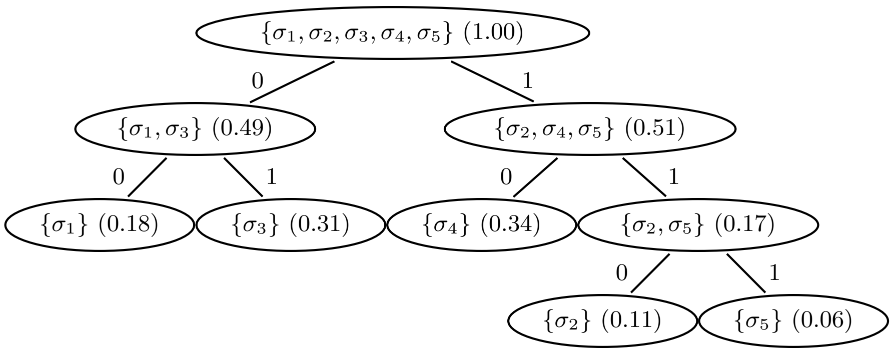
By collapsing the nodes for σ_j and σ_k in this tree, as before, we obtain a code tree for the symbol set $\mathcal{S}'$, with expected cost

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

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

Complexity

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



$\{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ (1.00)
0
1
$\{\sigma_1, \sigma_3\}$ (0.49)
$\{\sigma_2, \sigma_4, \sigma_5\}$ (0.51)
0
1
0
1
$\{\sigma_1\}$ (0.18)
$\{\sigma_3\}$ (0.31)
$\{\sigma_4\}$ (0.34)
$\{\sigma_2, \sigma_5\}$ (0.17)
0
1
$\{\sigma_2\}$ (0.11)
$\{\sigma_5\}$ (0.06)

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

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

Canonical Huffman codes

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

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

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

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

Reading the symbols σ_i from left to right in Figure 6.3 corresponds to ordering them by the length of their respective codewords (short codewords first); ties are broken according to the natural ordering of the symbols (e.g., σ_1 before σ_3). A code with this property is called a *canonical Huffman code*. For every possible Huffman code $\mathcal{C}$, there is always an equivalent canonical Huffman code $\mathcal{C}_{can}$.

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

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

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

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

It can be described by

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

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

Length-limited Huffman codes

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

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

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

6.2.3 Arithmetic Coding

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

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

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

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

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

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

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

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

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

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

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

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

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

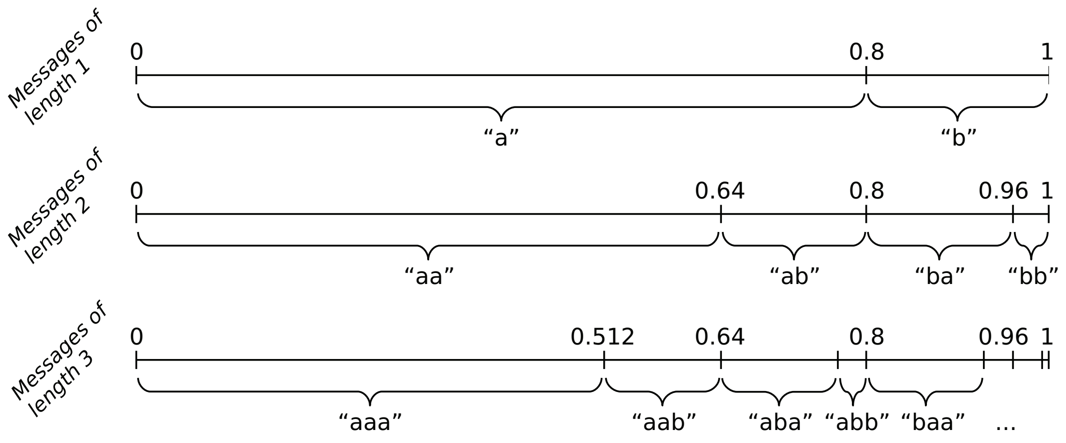


Figure 6.4 Arithmetic coding: transforming symbol sequences into subintervals of the interval $[0, 1)$. A message M (sequence of symbols) is encoded as a binary subinterval of the interval corresponding to M. Probability distribution assumed for the example: Pr["a"] = 0.8, Pr["b"] = 0.2.

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

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

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

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

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

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

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

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

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

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

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

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

Thus, in order to encode a message

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

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

Two convenient properties of arithmetic coding are:

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

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

Implementation

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

Huffman coding versus arithmetic coding

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

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

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

6.2.4 Symbolwise Text Compression

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

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

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

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

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

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

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

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

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

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

6.3 Compressing Postings Lists

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

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

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

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

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

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

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

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

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

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

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

would be transformed into the equivalent Δ-list

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

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

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

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

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

6.3.1 Nonparametric Gap Compression

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

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

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

Elias's γ code

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

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

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

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

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

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

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

δ and ω codes

γ coding is appropriate when compressing lists with predominantly small gaps (say, smaller than 32) but can be quite wasteful if applied to lists with large gaps. For such lists the δ code, a variant of γ, may be the better choice. δ coding is very similar to γ coding. However, instead of storing the selector component of a given integer in unary representation, δ encodes the selector using γ. Thus, the δ codewords of the integers 1, 5, 7, and 16 are:

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

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

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

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

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

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

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

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

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

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

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

Table 6.3 Encoding positive integers using various nonparametric codes.

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

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

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

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

6.3.2 Parametric Gap Compression

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

Parametric compression methods can be divided into two classes: *global* methods and *local* methods. A global method chooses a single parameter value that is used for all inverted lists in the index. A local method chooses a different value for each list in the index, or even for each small chunk of postings in a given list, where a chunk typically comprises a few hundred or a few thousand postings. You may recall from Section 4.3 that each postings list contains a list of synchronization points that help us carry out random access operations on the list. Synchronization points fit naturally with chunkwise list compression (in fact, this is the application from which they initially got their name); each synchronization point corresponds to the beginning of a new chunk. For heterogeneous postings lists, in which different parts of the list have different statistical characteristics, chunkwise compression can improve the overall effectiveness considerably.

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

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

Golomb/Rice codes

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

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

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

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

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

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

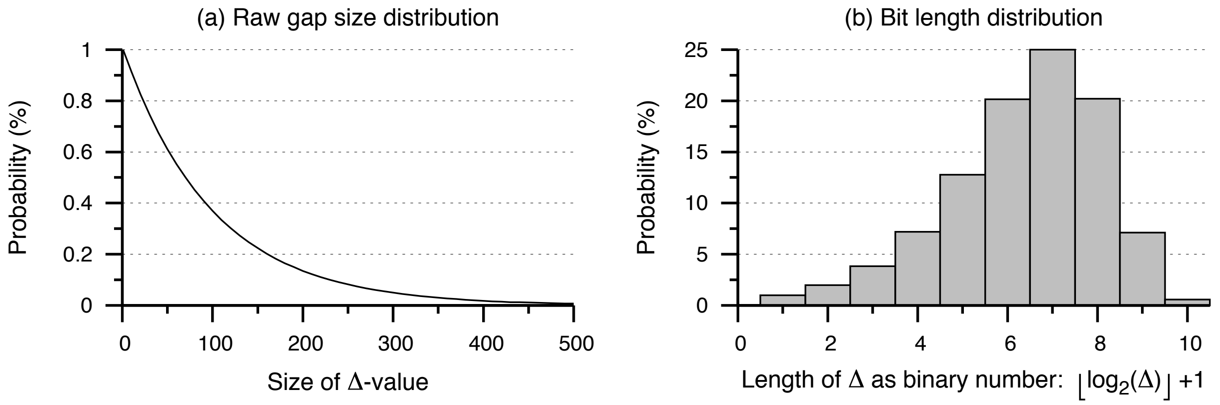


Figure 6.5 Distribution of Δ-gaps in a hypothetical postings list with $N_T/N = 0.01$ (i.e., term T appears in 1% of all documents). (a) distribution of raw gap sizes; (b) distribution of bit lengths (i.e., number of bits needed to encode each gap as a binary number).

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

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

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

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

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

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

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

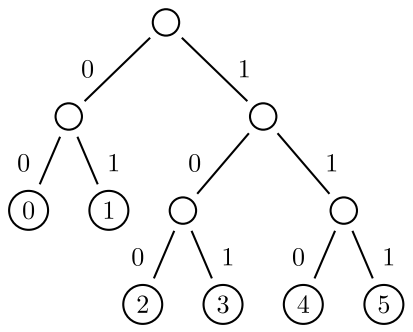


Figure 6.6 Codewords of the remainders $0 \leq r(k) < 6$, for the Golomb code with parameter $M = 6$.

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

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

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

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

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

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

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

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

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

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

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

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

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

Finding the optimal Golomb/Rice parameter value

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

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

for two symbols σ_1 and σ_2, implies

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

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

$$\Pr[\Delta = k + M^*] = \frac{1}{2} \cdot \Pr[\Delta = k] \quad\Leftrightarrow\quad (1 - N_T/N)^{k+M^*-1} = \frac{1}{2} \cdot (1 - N_T/N)^{k-1}$$

$$\Leftrightarrow\quad M^* = \frac{-\log(2)}{\log(1 - N_T/N)}. \tag{6.48}$$

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

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

always leads to the optimal code.

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

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

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

Huffman codes: LLRUN

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

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

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

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

6.3.3 Context-Aware Compression Methods

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

Huffman codes: Finite-context LLRUN

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

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

```
encodeLLRUN-2 (⟨L[1], ..., L[n]⟩, $\vartheta$, output) ≡
    let $\Delta(L)$ denote the list $\langle L[1], L[2]-L[1], \ldots, L[n]-L[n-1]\rangle$
    $\Delta_{max} \leftarrow \max_i\{\Delta(L)[i]\}$
    initialize the array bucketFrequencies$[0..1][0..\lfloor\log_2(\Delta_{max})\rfloor]$ to zero
    $c \leftarrow 0$   // the context to be used for finite-context modeling
    for i ← 1 to n do   // collect context-specific statistics
        $b \leftarrow \lfloor\log_2(\Delta(L)[i])\rfloor$   // the bucket of the current Δ-value
        bucketFrequencies[c][b] ← bucketFrequencies[c][b] + 1
        if $b < \vartheta$ then $c \leftarrow 0$ else $c \leftarrow 1$
    for i ← 0 to 1 do   // build two Huffman trees, one for each context
        $T_i$ ← buildHuffmanTree(bucketFrequencies[i])
    $c \leftarrow 0$   // reset the context
    for i ← 1 to n do   // compress the postings
        $b \leftarrow \lfloor\log_2(\Delta(L)[i])\rfloor$
        write b's Huffman codeword, according to the tree $T_c$, to output
        write $\Delta(L)[i]$ to output, as a b-bit binary number (omitting the leading $\overline{1}$)
        if $b < \vartheta$ then $c \leftarrow 0$ else $c \leftarrow 1$
    return
```

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

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

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

Interpolative coding

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

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

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

```
**encodeInterpolative** ($\langle L[1], \ldots, L[n] \rangle$, *output*) $\equiv$
    **encodeGamma** ($n$)
    **encodeGamma** ($L[1]$, *output*)
    **encodeGamma** ($L[n] - L[1]$, *output*)
    **encodeInterpolativeRecursively** ($\langle L[1], \ldots, L[n] \rangle$, *output*)

**encodeInterpolativeRecursively** ($\langle L[1], \ldots, L[n] \rangle$, *output*) $\equiv$
    **if** $n < 3$ **then**
        **return**
    $middle \leftarrow \lceil n/2 \rceil$
    $firstPossible \leftarrow L[1] + (middle - 1)$
    $lastPossible \leftarrow L[n] + (middle - n)$
    $k \leftarrow \lceil \log_2(lastPossible - firstPossible + 1) \rceil$
    write ($L[middle] - firstPossible$) to *output*, as a $k$-bit binary number
    **encodeInterpolativeRecursively** ($\langle L[1], \ldots, L[middle] \rangle$, *output*)
    **encodeInterpolativeRecursively** ($\langle L[middle], \ldots, L[n] \rangle$, *output*)
```

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

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

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

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

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

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

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

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

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

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

6.3.4 Index Compression for High Query Performance

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

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

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

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

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

Byte-aligned codes

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

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

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

$$1\ 1011000\ 0\ 0001100\ 0\ 0011010\ 1\ 1100010\ 0\ 0000001\ 0\ 1100000\ 1\ 0000000\ 0\ 0000011\ \ldots$$

(spaces inserted for better readability).

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

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

decodeVByte (inputBuffer, ⟨L[1], ..., L[n]⟩) ≡
    current ← 0
    for i ← 1 to n do
        shift ← 0
        b ← inputBuffer.readByte()
        while b ≥ 128 do
            current ← current + ((b & 127) ≪ shift)
            shift ← shift + 7
            b ← inputBuffer.readByte()
        current ← current + (b ≪ shift)
        L[i] ← current
    return
```

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

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

Word-aligned codes

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

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

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

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

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

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

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

$$0001\ 00011001011000\ 00000000011001\ \overline{0010}\ \overline{011100001}\ 001011111\ 101111111\ \mathrm{U}, \tag{6.52}$$

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

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

Efficiently decoding unaligned codes

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

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

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

into an equivalent sequence of Δ-values

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

the bit sequence produced by the γ coder is

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

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

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

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

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

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

```
decodeGamma ($inputBuffer$, $T[0..2^k - 1]$, $\langle L[1], \ldots, L[n] \rangle$) ≡
    $current \leftarrow 0$
    $bitBuffer \leftarrow 0$
    $bitsInBuffer \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        while $bitsInBuffer + 8 \leq 64$ do
            $bitBuffer \leftarrow bitBuffer + (inputBuffer.readByte() \ll bitsInBuffer)$
            $bitsInBuffer \leftarrow bitsInBuffer + 8$
        $codeLength \leftarrow T[bitBuffer \,\&\, (2^k - 1)]$
        $bitBuffer \leftarrow bitBuffer \gg (codeLength - 1)$
        $mask \leftarrow (1 \ll codeLength) - 1$
        $current \leftarrow current + (bitBuffer \,\&\, mask)$
        $bitBuffer \leftarrow bitBuffer \gg codeLength$
        $bitsInBuffer \leftarrow bitsInBuffer - 2 \times codeLength - 1$
        $L[i] \leftarrow current$
    return
```

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

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

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

6.3.5 Compression Effectiveness

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

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

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

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

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

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

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

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

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

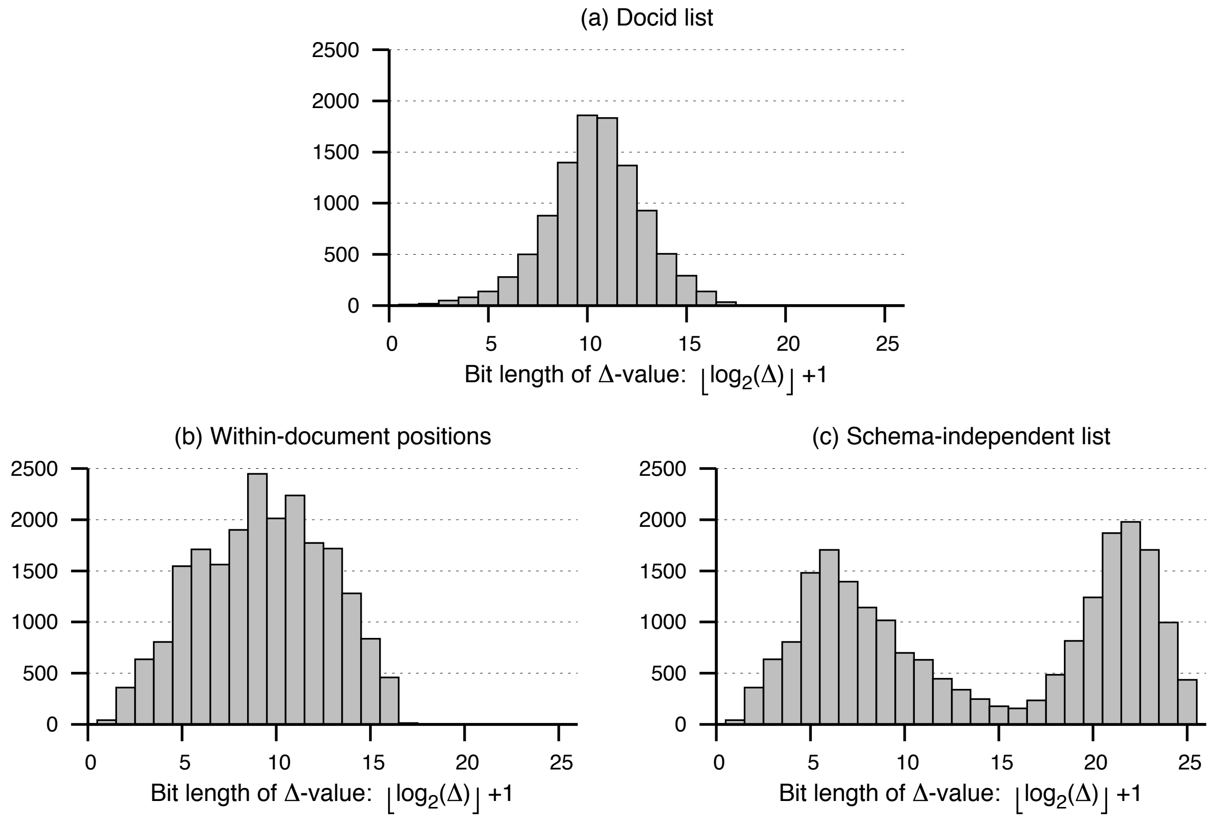


Figure 6.11 Gap distributions in different postings lists for the term “huffman” in the GOV2 collection. Vertical axis: Number of Δ-values of the given size. Only the gaps in the docid list follow a geometric distribution.

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

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

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

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

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

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

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

6.3.6 Decoding Performance

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

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

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

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

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

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

Of course, the numbers shown in the table are only representative of docid indices. For a positional or a schema-independent index, for example, the results will look more favorable for vByte. Moreover, the validity of the simplistic performance model applied here (calculating the total overhead as a sum of disk I/O and decoding operations) may be questioned in the presence of caching and background I/O. Nonetheless, the general message is clear: Compared to an uncompressed index (with either 32-bit or 64-bit postings), every compression method leads to improved query performance.

6.3.7 Document Reordering

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

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

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

with γ-code representation

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

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

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

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

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

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

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

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

is reduced from 22 to 18 (−18%).

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

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

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

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

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

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

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

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

6.4 Compressing the Dictionary

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

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

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

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

Dictionary groups

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

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

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

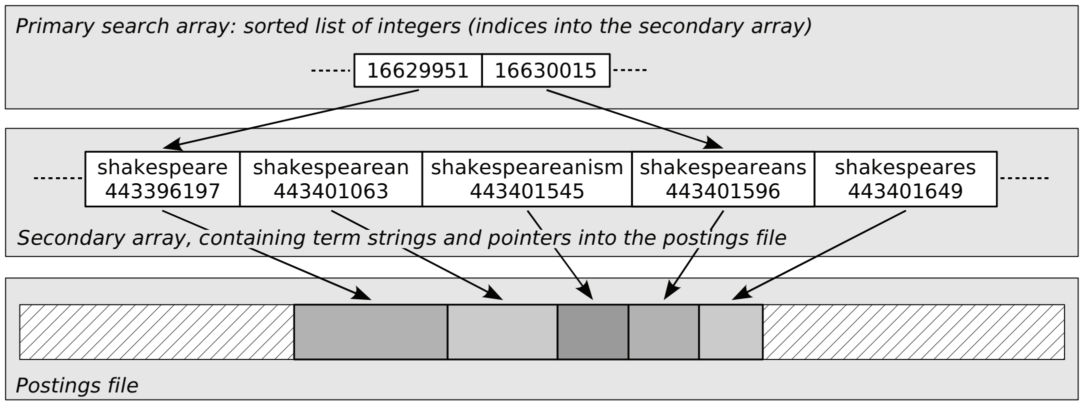

Primary search array: sorted list of integers (indices into the secondary array)
16629951
16630015
shakespeare 443396197
shakespearean 443401063
shakespeareanism 443401545
shakespeareans 443401596
shakespeares 443401649
Secondary array, containing term strings and pointers into the postings file
Postings file

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

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

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

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

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

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

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

Front coding

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[6] `www.zlib.net`

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

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

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

Combining dictionary compression and dictionary interleaving

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

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

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

6.5 Summary

The main points of this chapter are:

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

6.6 Further Reading

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

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

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

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

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

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

6.7 Exercises

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

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

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

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

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

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

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

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

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

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

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

6.8 Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14 Parallel Information Retrieval

Information retrieval systems often have to deal with very large amounts of data. They must be able to process many gigabytes or even terabytes of text, and to build and maintain an index for millions of documents. To some extent the techniques discussed in Chapters 5–8 can help us satisfy these requirements, but it is clear that, at some point, sophisticated data structures and clever optimizations alone are not sufficient anymore. A single computer simply does not have the computational power or the storage capabilities required for indexing even a small fraction of the World Wide Web.[1]

In this chapter we examine various ways of making information retrieval systems scale to very large text collections such as the Web. The first part (Section 14.1) is concerned with parallel query processing, where the search engine's service rate is increased by having multiple index servers process incoming queries in parallel. It also discusses redundancy and fault tolerance issues in distributed search engines. In the second second part (Section 14.2), we shift our attention to the parallel execution of off-line tasks, such as index construction and statistical analysis of a corpus of text. We explain the basics of MapReduce, a framework designed for massively parallel computations carried out on large amounts of data.

14.1 Parallel Query Processing

There are many ways in which parallelism can help a search engine process queries faster. The two most popular approaches are *index partitioning* and *replication*. Suppose we have a total of n index servers. Following the standard terminology, we refer to these servers as *nodes*. By creating n replicas of the index and assigning each replica to a separate node, we can realize an n-fold increase of the search engine's service rate (its theoretical throughput) without affecting the time required to process a single query. This type of parallelism is referred to as *inter-query parallelism*, because multiple queries can be processed in parallel but each individual query is processed sequentially. Alternatively, we could split the index into n parts and have each node work only on its own small part of the index. This approach is referred to as *intra-query*

[1] While nobody knows the exact size of the indexable part of the Web, it is estimated to be at least 100 billion pages. In August 2005, when Yahoo! last disclosed the size of its index, it had reached a total size of 19.2 billion documents (http://www.ysearchblog.com/archives/000172.html).

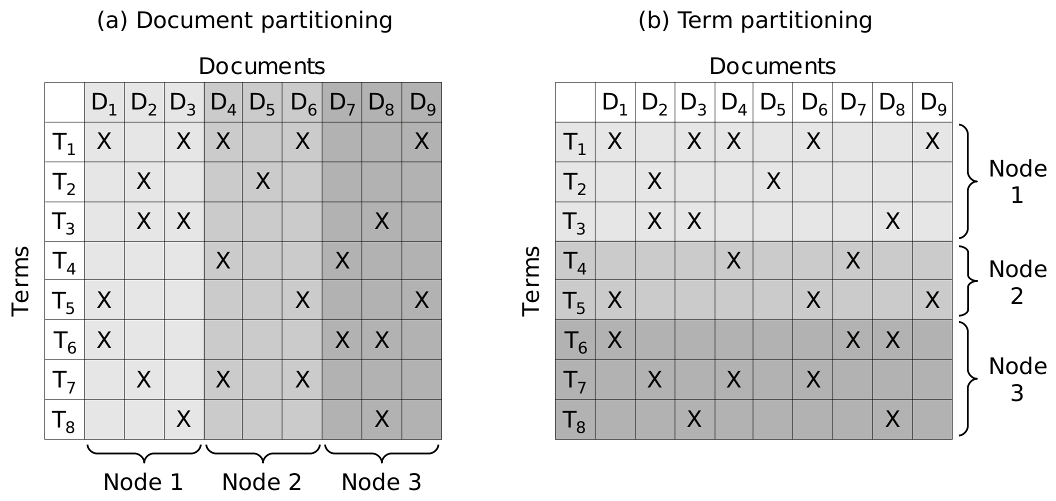


Figure 14.1 The two prevalent index partitioning schemes: document partitioning and term partitioning (shown for a hypothetical index containing 8 terms and 9 documents).

parallelism, because each query is processed by multiple servers in parallel. It improves the engine's service rate as well as the average time per query.

In this section we focus primarily on methods for intra-query parallelism. We study index partitioning schemes that divide the index into independent parts so that each node is responsible for a small piece of the overall index.

The two predominant index partitioning schemes are *document partitioning* and *term partitioning* (visualized in Figure 14.1). In a document-partitioned index, each node holds an index for a subset of the documents in the collection. For instance, the index maintained by node 2 in Figure 14.1(a) contains the following docid lists:

$$L_1 = \langle 4, 6 \rangle, \quad L_2 = \langle 5 \rangle, \quad L_4 = \langle 4 \rangle, \; L_5 = \langle 6 \rangle, \quad L_7 = \langle 4, 6 \rangle.$$

In a term-partitioned index, each node is responsible for a subset of the terms in the collection. The index stored in node 1 in Figure 14.1(b) contains the following lists:

$$L_1 = \langle 1, 3, 4, 6, 9 \rangle, \quad L_2 = \langle 2, 5 \rangle, \quad L_3 = \langle 2, 3, 8 \rangle.$$

The two partitioning strategies differ greatly in the way queries are processed by the system. In a document-partitioned search engine, each of the n nodes is involved in processing all queries received by the engine. In a term-partitioned configuration, a query is seen by a given node only if the node's index contains at least one of the query terms.

14.1.1 Document Partitioning

In a document-partitioned index, each index server is responsible for a subset of the documents in the collection. Each incoming user query is received by a frontend server, the *receptionist*, which forwards it to all n index nodes, waits for them to process the query, merges the search results received from the index nodes, and sends the final list of results to the user. A schematic of a document-partitioned search engine is shown in Figure 14.2.

The main advantage of the document-partitioned approach is its simplicity. Because all index servers operate independently of each other, no additional complexity needs to be introduced into the low-level query processing routines. All that needs to be provided is the receptionist server that forwards the query to the backends and, after receiving the top k search results from each of the n nodes, selects the top m, which are then returned to the user (where the value of m is typically chosen by the user and k is chosen by the operator of the search engine). In addition to forwarding queries and search results, the receptionist may also maintain a cache that contains the results for recently/frequently issued queries.

If the search engine maintains a dynamic index that allows updates (e.g., document insertions/deletions), then it may even be possible to carry out the updates in a distributed fashion, in which each node takes care of the updates that pertain to its part of the overall index. This approach eliminates the need for a complicated centralized index construction/maintenance process that involves the whole index. However, it is applicable only if documents may be assumed to be independent of each other, not if inter-document information, such as hyperlinks and anchor text, is part of the index.

When deciding how to divide the collection across the n index nodes, one might be tempted to bias the document-node assignment in some way — for instance, by storing similar documents in the same node. In Section 6.3.7 we have seen that the index can be compressed better if the documents in the collection are reordered according to their URL, so that documents with similar URLs receive nearby docids. Obviously, this method is most effective if all pages from the same domain are assigned to the same node. The problem with this approach is that it may create an imbalance in the load distribution of the index servers. If a given node primarily contains documents associated with a certain topic, and this topic suddenly becomes very popular among users, then the query processing load for that node may become much higher than the load of the other nodes. In the end this will lead to a suboptimal utilization of the available resources. To avoid this problem, it is usually best to not try anything fancy but to simply split the collection into n completely random subsets.

Once the index has been partitioned and each subindex has been loaded into one of the nodes, we have to make a decision regarding the per-node result set size k. How should k be chosen with respect to m, the number of search results requested by the user? Suppose the user has asked for $m = 100$ results. If we want to be on the safe side, we can have each index node return $k = 100$ results, thus making sure that all of the top 100 results overall are received by the receptionist. However, this would be a poor decision, for two reasons:

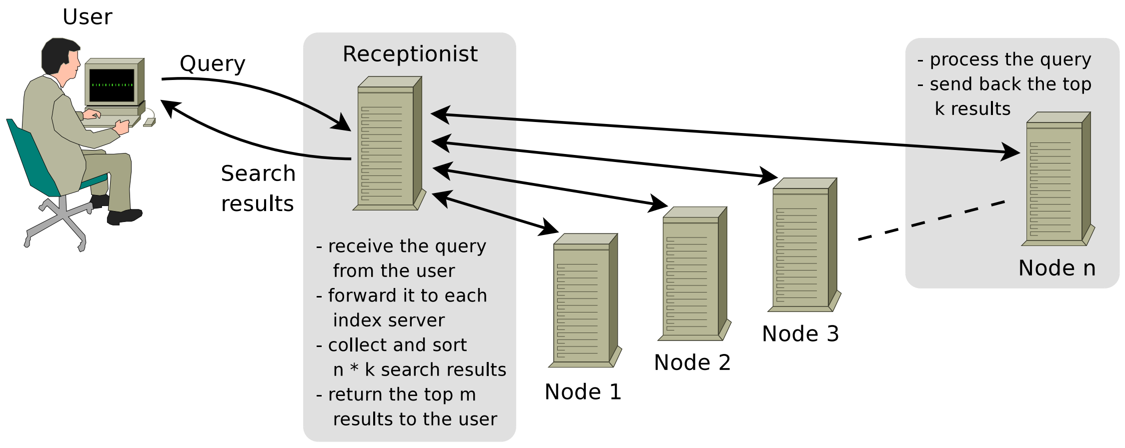

User
Query
Search results
Receptionist
- receive the query from the user
- forward it to each index server
- collect and sort n * k search results
- return the top m results to the user
- process the query
- send back the top k results
Node 1
Node 2
Node 3
Node n

Figure 14.2 Document-partitioned query processing with one receptionist and several index servers. Each incoming query is forwarded to all index servers.

1. It is quite unlikely that all of the top 100 results come from the same node. By having each index server return 100 results to the receptionist, we put more load on the network than necessary.
2. The choice of k has a non-negligible effect on query processing performance, because of performance heuristics such as MaxScore (see Section 5.1.1). Table 5.1 on page 144 shows that decreasing the result set size from $k = 100$ to $k = 10$ can reduce the average CPU time per query by around 15%, when running queries against a frequency index for GOV2.

Clarke and Terra (2004) describe a method to compute the probability that the receptionist sees at least the top m results overall, given the number n of index servers and the per-node result set size k. Their approach is based on the assumption that each document was assigned to a random index node when the collection was split into n subsets, and thus that each node is equally likely to return the best, second-best, third-best, ... result overall.

Consider the set $\mathcal{R}_m = \{r_1, r_2, \ldots, r_m\}$ composed of the top m search results. For each document r_i the probability that it is found by a particular node is $1/n$. Hence, the probability that exactly l of the top m results are found by that node is given by the binomial distribution

$$b(n, m, l) \;=\; \binom{m}{l} \cdot \left(\frac{1}{n}\right)^{l} \cdot \left(1 - \frac{1}{n}\right)^{m-l}. \tag{14.1}$$

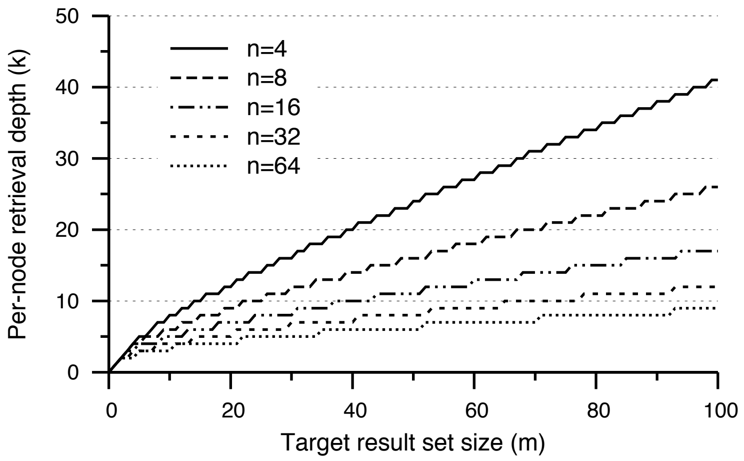


Figure 14.3 Choosing the minimum retrieval depth k that returns the top m results with probability $p(n, m, k) > 99.9\%$, where n is the number of nodes in the document-partitioned index.

The probability that all members of $\mathcal{R}_m$ are discovered by requesting the top k results from each of the n index nodes can be calculated according to the following recursive formula:

$$p(n, m, k) = \begin{cases} 1 & \text{if } m \leq k; \\ 0 & \text{if } m > k \text{ and } n = 1; \\ \sum_{l=0}^{k} b(n, m, l) \cdot p(n-1, m-l, k) & \text{if } m > k \text{ and } n > 1. \end{cases} \tag{14.2}$$

The two base cases are obvious. In the recursive case, we consider the first node in the system and compute the probability that $l = 0, 1, 2, \ldots, k$ of the top m results overall are retrieved by that node (i.e., $b(n, m, l)$). For each possible value for l, this probability is multiplied by the probability $p(n - 1, m - l, k)$ that the remaining $m - l$ documents in $\mathcal{R}_m$ are found by the remaining $n - 1$ index nodes. Equation 14.2 does not appear to have a closed-form solution but can be solved through the application of dynamic programming in time $\Theta(n \cdot m \cdot k)$.

Figure 14.3 shows the per-node retrieval depth k required to find the top m results with probability at least 99.9%. For $m = 100$ and $n = 4$, a per-node retrieval depth of $k = 41$ achieves the desired probability level. If we decide to relax our correctness requirements from 99.9% to 95%, k can even be decreased a little further, from 41 to 35.

It can sometimes be beneficial to optimize the retrieval depth k for a different value m than the user has asked for. For example, if the receptionist maintains a cache for recently issued queries, it may be worthwhile to obtain the top 20 results for a given query even if the user has asked for only the top 10, so that a click on the "next page" link can be processed from the cache. Allowing the receptionist to inspect a result set $\mathcal{R}_{m'}$ for $m' > m$ is also useful because it facilitates the application of diversity-seeking reranking techniques, such as Carbonell and Goldstein's (1998) *maximal marginal relevance* or Google's *host crowding* heuristic.[2]

14.1.2 Term Partitioning

Although document partitioning is often the right choice and scales almost linearly with the number of nodes, it can unfold its true potential only if the index data found in the individual nodes are stored in main memory or any other low-latency random-access storage medium, such as flash memory, but not if it is stored on disk.

Consider a document-partitioned search engine in which all postings lists are stored on disk. Suppose each query contains 3 words on average, and we want the search engine to handle a peak query load of 100 queries per second. Recall from Section 13.2.3 that, due to queueing effects, we usually cannot sustain a utilization level above 50%, unless we are willing to accept occasional latency jumps. Thus, a query load of 100 qps translates into a required service rate of at least 200 qps or — equivalently — 600 random access operations per second (one for each query term). Assuming an average disk seek latency of 10 ms, a single hard disk drive cannot perform more than 100 random access operations per second, a factor of 6 less than what is required to achieve the desired throughput. We could try to circumvent this limitation by adding more disks to each index node, but equipping each server with six hard disks may not always be practical. Moreover, it is obvious that we will never be able to handle loads of more than a few hundred queries per second, regardless of how many nodes we add to the system.

Term partitioning addresses the disk seek problem by splitting the collection into sets of terms instead of sets of documents. Each index node v_i is responsible for a certain term set $\mathcal{T}_i$ and is involved in processing a given query only if one or more query terms are members of $\mathcal{T}_i$. Our discussion of term-partitioned query processing is based upon the pipelined architecture proposed by Moffat et al. (2007). In this architecture, queries are processed in a term-at-a-time fashion (see Section 5.1.2 for details on term-at-a-time query processing strategies).

Suppose a query contains q terms $t_1, t_2, \ldots, t_q$. Then the receptionist will forward the query to the node $v(t_1)$ responsible for the term t_1. After creating a set of document score accumulators from t_1's postings list, $v(t_1)$ forwards the query, along with the accumulator set, to the node $v(t_2)$ responsible for t_2, which updates the accumulator set, sends it to $v(t_3)$, and so forth. When the last node in this pipeline, $v(t_q)$, is finished, it sends the final accumulator set to the receptionist. The receptionist then selects the top m search results and returns them to the user. A schematic of this approach is shown in Figure 14.4.

Many of the optimizations used for sequential term-at-a-time query processing also apply to term-partitioned query processing: infrequent query terms should be processed first; accumulator pruning strategies should be applied to keep the size of the accumulator set, and thus the

[2] `www.mattcutts.com/blog/subdomains-and-subdirectories/`

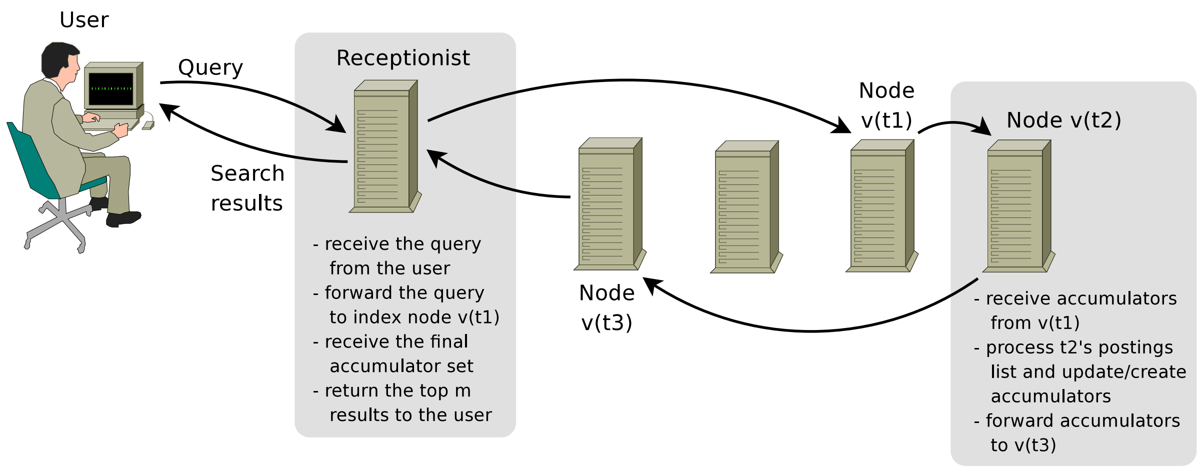

User
Query
Search results
Receptionist
- receive the query from the user
- forward the query to index node v(t1)
- receive the final accumulator set
- return the top m results to the user
Node v(t1)
Node v(t2)
Node v(t3)
- receive accumulators from v(t1)
- process t2's postings list and update/create accumulators
- forward accumulators to v(t3)

Figure 14.4 Term-partitioned query processing with one receptionist and four index servers. Each incoming query is passed from index server to index server, depending on the terms found in the query (shown for a query containing three terms).

overall network traffic, under control; impact ordering can be used to efficiently identify the most important postings for a given term.

Note that the pipelined query processing architecture outlined above does not use intra-query parallelism, as each query — at a given point in time — is processed by only a single node. Therefore, although it can help increase the system's theoretical throughput, term partitioning, in the simple form described here, does not necessarily decrease the search engine's response time. To some extent this limitation can be addressed by having the receptionist send prefetch instructions to the nodes $v(t_2), \ldots, v(t_q)$ at the same time it forwards the query to $v(t_1)$. This way, when a node $v(t_i)$ receives the accumulator set, some of t_i's postings may already have been loaded into memory and the query can be processed faster.

Despite its potential performance advantage over the document-partitioned approach, at least for on-disk indices, term partitioning has several shortcomings that make it difficult to use the method in practice:

- **Scalability.** As the collection becomes bigger, so do the individual postings lists. For a corpus composed of a billion documents, the postings list for a frequent term, such as "computer" or "internet", can easily occupy several hundred megabytes. Processing a query that contains one or more of these frequent terms will require at least a few seconds, far more than what most users are accustomed to. In order to solve this problem, large postings lists need to be cut into smaller chunks and divided among several nodes, with each node taking care of a small part of the postings list and all of them working in parallel. Unfortunately, this complicates the query processing logic quite a bit.

- **Load Imbalance.** Term partitioning suffers from an uneven load across the index nodes. The load corresponding to a single term is a function of the term's frequency in the collection as well as its frequency in users' queries. If a term has a long postings list and is also very popular in search queries, then the corresponding index node may experience a load that is much higher than the average load in the system. To address this problem, postings lists that are responsible for a high fraction of the overall load should be replicated and distributed across multiple nodes. This way, the computational load associated with a given term can be shared among several machines, albeit at the cost of increased storage requirements.
- **Term-at-a-Time.** Perhaps the most severe limitation of the term-partitioned approach is its inability to support efficient document-at-a-time query processing. In order to realize document-at-a-time scoring on top of a term-partitioned index, entire postings lists, as opposed to pruned accumulator sets, would need to be sent across the network. This is impractical, due to the size of the postings lists. Therefore, ranking methods that necessitate a document-at-a-time approach, such as the proximity ranking function from Section 2.2.2, are incompatible with a term-partitioned index.

Even with all these shortcomings, term partitioning can sometimes be the right choice. Recall, for instance, the three-level cache hierarchy from Section 13.4.1. The second level in this hierarchy caches list intersections (in search engines with Boolean-AND query semantics; see Section 2.2). Instead of following a document partitioning approach and equipping each index node with its own intersection cache, we may choose a term-partitioned index and treat the cached intersections just like ordinary postings lists. In the example shown in Figure 14.4, if we have already seen the query $\langle t_1, t_2 \rangle$ before, we may have cached the intersected list $(t_1 \wedge t_2)$ in index node $v(t_2)$ and may skip $v(t_1)$ when processing the new query $\langle t_1, t_2, t_3 \rangle$.

More generally, term partitioning suggests itself if postings lists are relatively short — either due to the nature of the information they represent (as in the case of list intersections) or because they are artificially shortened (for instance, by applying index pruning techniques; see Section 5.1.5 for details).

14.1.3 Hybrid Schemes

Consider a distributed index with n nodes, for a collection of size $|\mathcal{C}|$. Document partitioning becomes inefficient if $|\mathcal{C}|/n$ is too small and disk seeks dominate the overall query processing cost. Term partitioning, on the other hand, becomes impractical if $|\mathcal{C}|$ is too large, as the time required to process a single query is likely to exceed the users' patience.

Xi et al. (2002) propose a hybrid architecture in which the collection is divided into p subcollections according to a standard document partitioning scheme. The index for each subcollection is then term-partitioned across n/p nodes, so that each node in the system is responsible for all occurrences of a set of terms within one of the p subcollections. With the right load balancing policies in place, this can lead roughly to a factor-n increase in throughput and a factor-p latency reduction.

As an alternative to the hybrid term/document partitioning, we may also consider a hybrid of document partitioning and replication. Remember that the primary objective of term partitioning is to increase throughput, not to decrease latency. Thus, instead of term-partitioning each of the p subindices, we may achieve the same performance level by simply replicating each subindex n/p times and load-balancing the queries among n/p identical replicas. At a high level, this is the index layout that was used by Google around 2003 (Barroso et al., 2003). The overall impact on maximum throughput and average latency is approximately the same as in the case of the hybrid document/term partitioning. The storage requirements are likely to be a bit higher, due to the (n/p)-way replication. If the index is stored on disk, this is usually not a problem.

14.1.4 Redundancy and Fault Tolerance

When operating a large-scale search engine with thousands of users, reliability and fault tolerance tend to be of similar importance as response time and result quality. As we increase the number of machines in the search engine, so as to scale to higher query loads, it becomes more and more likely that one of them will fail at some point in time. If the system has been designed with fault tolerance in mind, then the failure of a single machine may cause a small reduction in throughput or search quality. If it has not been designed with fault tolerance in mind, a single failure may bring down the entire search engine.

Let us compare the simple, replication-free document partitioning and term partitioning schemes for a distributed search engine with 32 nodes. If one of the nodes in the term-partitioned index fails, the engine will no longer be able to process queries containing any of the terms managed by that node. Queries that do not contain any of those terms are unaffected. For a random query q containing three words (the average number of query terms for Web queries), the probability that q can be processed by the remaining 31 nodes is

$$\left(\frac{31}{32}\right)^3 \approx 90.9\%. \tag{14.3}$$

If one of the nodes in the document-partitioned index fails, on the other hand, the search engine will still be able to process all incoming queries. However, it will miss some search results. If the partitioning was performed bias-free, then the probability that j out of the top k results for a random query are missing is

$$\binom{k}{j} \cdot \left(\frac{1}{32}\right)^j \cdot \left(\frac{31}{32}\right)^{k-j}. \tag{14.4}$$

Thus, the probability that the top 10 results are unaffected by the failure is

$$\left(\frac{31}{32}\right)^{10} \approx 72.8\%. \tag{14.5}$$

It therefore might seem that the impact of the machine failure is lower for the term-partitioned than for the document-partitioned index. However, the comparison is somewhat unfair, because the inability to process a query is a more serious problem than losing one of the top 10 search results. If we look at the probability that at least 2 of the top 10 results are lost due to the missing index node, we obtain

$$1 - \left(\frac{31}{32}\right)^{10} - \binom{10}{1} \cdot \left(\frac{1}{32}\right) \cdot \left(\frac{31}{32}\right)^{9} \approx 3.7\%. \tag{14.6}$$

Thus, the probability that a query is impacted severely by a single node failure is in fact quite small, and most users are unlikely to notice the difference. Following this line of argument, we may say that a document-partitioned index degrades more gracefully than a term-partitioned one in the case of a single node failure.

For informational queries, where there are often multiple relevant results, this behavior might be good enough. For navigational queries, however, this is clearly not the case (see Section 15.2 for the difference between informational and navigational queries). As an example, consider the navigational query ⟨“white”, “house”, “website”⟩. This query has a single vital result (http://www.whitehouse.gov/) that *must* be present in the top search results. If 1 of the 32 nodes in the document-partitioned index fails, then for each navigational query there is a 3.2% chance that the query’s vital result is lost (if there is a vital result for the query). There are many ways to address this problem. Three popular ones are the following:

- **Replication.** We can maintain multiple copies of the same index node, as described in the previous section on hybrid partitioning schemes, and have them process queries in parallel. If one of the r replicas for a given index node fails, the remaining $r-1$ will take over the load of the missing replica. The advantage of this approach is its simplicity (and its ability to improve throughput and fault tolerance at the same time). Its disadvantage is that, if the system is operating near its theoretical throughput and r is small, the remaining $r-1$ replicas may become overloaded.
- **Partial Replication.** Instead of replicating the entire index r times, we may choose to replicate index information only for important documents. The rationale behind this strategy is that most search results aren’t vital and can easily be replaced by an equally relevant document. The downside of this approach is that it can be difficult to predict which documents may be targeted by navigational queries. Query-independent signals such as PageRank (Section 15.3.1) can provide some guidance.

- **Dormant Replication.** Suppose the search engine comprises a total of n nodes. We can divide the index found on each node v_i into $n-1$ fragments and distribute them evenly among the $n-1$ remaining nodes, but leave them dormant (on disk) and not use them for query processing. Only when v_i fails will the corresponding $n-1$ fragments be activated inside the remaining nodes and loaded into memory for query processing. It is important that the fragments are loaded into memory, for otherwise we will double the overall number of disk seeks per query. Dormant replication roughly causes a factor-2 storage overhead, because each of the n nodes has to store $n-1$ additional index fragments.

It is possible to combine the above strategies — for example, by employing dormant replication of partial indices. Instead of replicating the whole index found in a given node, we replicate only the part that corresponds to important documents. This reduces the storage overhead and limits the impact on the search engine's throughput in case of a node failure.

14.2 MapReduce

Apart from processing search queries, there are many other data-intensive tasks that need to be carried out by a large-scale search engine. Such tasks include building and updating the index; identifying duplicate documents in the corpus; and analyzing the link structure of the document collection (e.g., PageRank; see Section 15.3.1).

MapReduce is a framework developed at Google that is designed for massively parallel computations (thousands of machines) on very large amounts of data (many terabytes), and that can accomplish all of the tasks listed above. MapReduce was first presented by Dean and Ghemawat (2004). In addition to a high-level overview of the framework, their paper includes information about many interesting implementation details and performance optimizations.

14.2.1 The Basic Framework

MapReduce was inspired by the *map* and *reduce* functions found in functional programming languages, such as Lisp. The map function takes as its arguments a function f and a list of elements $l = \langle l_1, l_2, \ldots, l_n \rangle$. It returns a new list

$$map(f, l) \;=\; \langle\, f(l_1),\; f(l_2),\; \ldots,\; f(l_n)\,\rangle. \tag{14.7}$$

The reduce function (also known as *fold* or *accumulate*) takes a function g and a list of elements $l = \langle l_1, l_2, \ldots, l_n \rangle$. It returns a new element l', such that

$$l' \;=\; reduce(g, l) \;=\; g(l_1,\; g(l_2,\; g(l_3,\; \ldots))). \tag{14.8}$$

When people refer to the map function in the context of MapReduce, they usually mean the function f that gets passed to *map* (where *map* itself is provided by the framework). Similarly,

```
map ($k$, $v$) ≡
    split $v$ into tokens
    for each token $t$ do
        output($t, 1$)
    return

reduce ($k$, $\langle v_1, v_2, \ldots, v_n \rangle$) ≡
    $count \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        $count \leftarrow count + v_i$
    output($count$)
    return
```

Figure 14.5 A MapReduce that counts the number of occurrences of each term in a given corpus of text. The input values processed by the map function are documents or other pieces of text. The input keys are ignored. The outcome of the MapReduce is a sequence of (t, f_t) tuples, where t is a term, and f_t is the number of times t appeared in the input.

when they refer to the reduce function, they mean the function g that gets passed to *reduce*. We will follow this convention.

From a high-level point of view, a MapReduce program (often simply called "a MapReduce") reads a sequence of key/value pairs, performs some computations on them, and outputs another sequence of key/value pairs. Keys and values are often strings, but may in fact be any data type. A MapReduce consists of three distinct phases:

- In the *map phase*, key/value pairs are read from the input and the map function is applied to each of them individually. The function is of the general form
$$map: \ (k, v) \ \mapsto \ \langle\, (k_1, v_1), \ (k_2, v_2), \ \ldots \rangle. \tag{14.9}$$
That is, for each key/value pair, map outputs a sequence of key/value pairs. This sequence may or may not be empty, and the output keys may or may not be identical to the input key (they usually aren't).
- In the *shuffle phase*, the pairs produced during the map phase are sorted by their key, and all values for the same key are grouped together.
- In the *reduce phase*, the reduce function is applied to each key and its values. The function is of the form
$$reduce: \ (k, \ \langle v_1, v_2, \ldots \rangle) \ \mapsto \ (k, \ \langle v_1', v_2', \ldots \rangle). \tag{14.10}$$
That is, for each key the reduce function processes the list of associated values and outputs another list of values. The output values may or may not be the same as the input values. The output key usually has to be the same as the input key, although this depends on the implementation.

Figure 14.5 shows the map and reduce functions of a MapReduce that counts the number of occurrences of all terms in a given corpus of text. In the reduce function, the output key is omitted, as it is implicit from the input key.

MapReduces are highly parallelizable, because both map and reduce can be executed in parallel on many different machines. Suppose we have a total of $n = m + r$ machines, where m is the number of *map workers* and r is the number of *reduce workers*. The input of the MapReduce is broken into small pieces called *map shards*. Each shard typically holds between 16 and 64 MB of data. The shards are treated independently, and each shard is assigned to one of the m map workers. In a large MapReduce, it is common to have dozens or hundreds of map shards assigned to each map worker. A worker usually works on only 1 shard at a time, so all its shards have to be processed sequentially. However, if a worker has more than 1 CPU, it may improve performance to have it work on multiple shards in parallel.

In a similar fashion, the output is broken into separate *reduce shards*, where the number of reduce shards is often the same as r, the number of reduce workers. Each key/value pair generated by the map function is sent to one of the r reduce shards. Typically, the shard that a given key/value pair is sent to depends only on the key. For instance, if we have r reduce shards, the target shard for each pair could be chosen according to

$$shard(key, value) = hash(key) \bmod r, \tag{14.11}$$

where *hash* is an arbitrary hash function. Assigning the map output to different reduce shards in this manner guarantees that all values for the same key end up in the same reduce shard. Within each reduce shard, incoming key/value pairs are sorted by their key (this is the shuffle phase), and are eventually fed into the reduce function to produce the final output of the MapReduce.

Figure 14.6 shows the data flow for the MapReduce from Figure 14.5, for three small text fragments from the Shakespeare corpus. Each fragment represents a separate map shard. The key/value pairs emitted by the map workers are partitioned onto the three reduce shards based on the hash of the respective key. For the purpose of the example, we assume *hash*("heart") mod 3 = 0, *hash*("soul") mod 3 = 1, and so forth.

The map phase may overlap with the shuffle phase, and the shuffle phase may overlap with the reduce phase. However, the map phase may never overlap with the reduce phase. The reason for this is that the reduce function can only be called after all values for a given key are available. Since, in general, it is impossible to predict what keys the map workers will emit, the reduce phase cannot commence before the map phase is finished.

14.2.2 Combiners

In many MapReduce jobs, a single map shard may produce a large number of key/value pairs for the same key. For instance, if we apply the counting MapReduce from Figure 14.5 to a typical corpus of English text, 6–7% of the map outputs will be for the key "the". Forwarding all these tuples to the reduce worker responsible for the term "the" wastes network and storage resources. More important, however, it creates an unhealthy imbalance in the overall load distribution. Regardless of how many reduce workers we assign to the job, one of them will end up doing at least 7% of the overall reduce work.

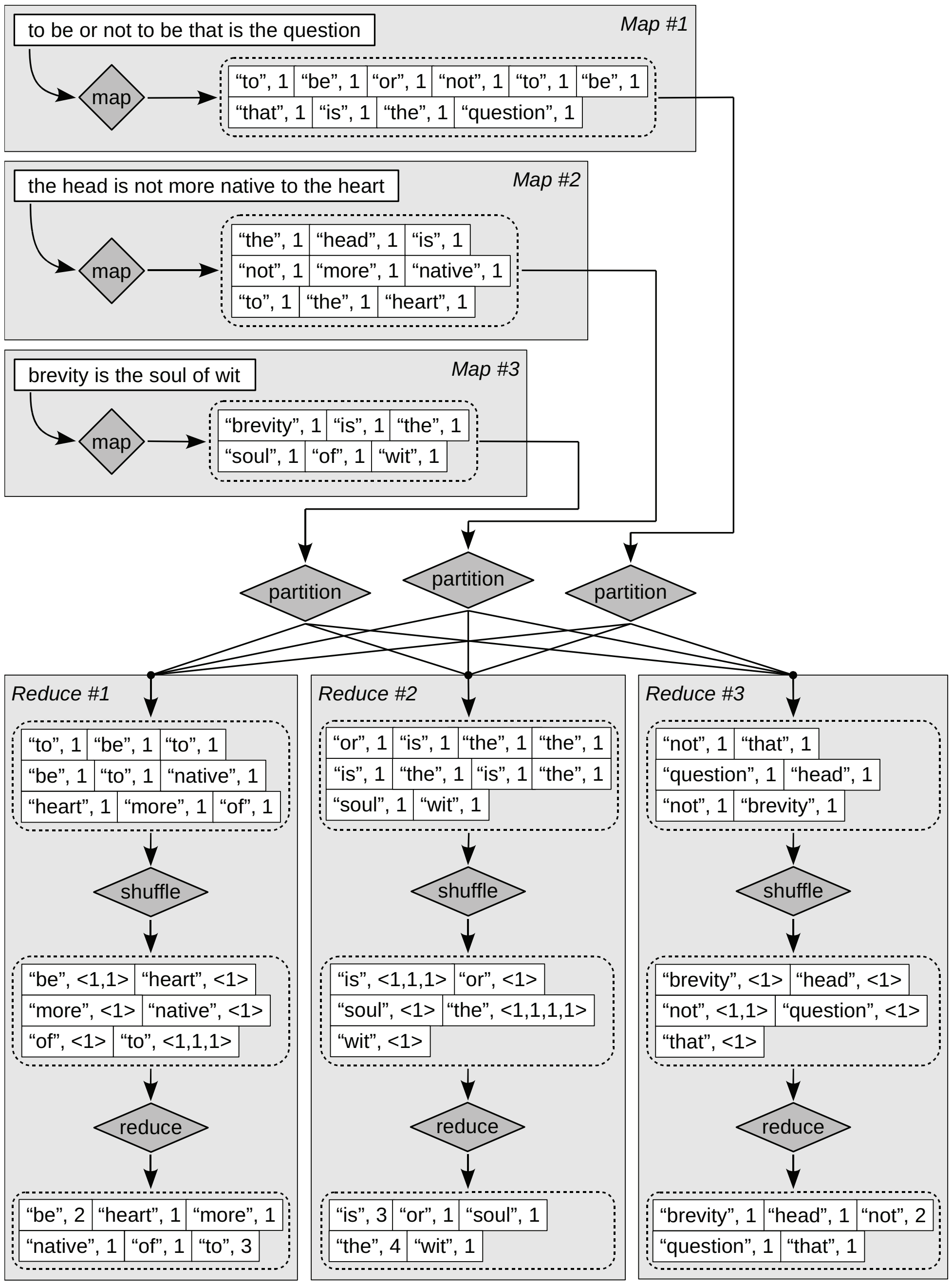

Map #1
to be or not to be that is the question
map
"to", 1 "be", 1 "or", 1 "not", 1 "to", 1 "be", 1
"that", 1 "is", 1 "the", 1 "question", 1
Map #2
the head is not more native to the heart
map
"the", 1 "head", 1 "is", 1
"not", 1 "more", 1 "native", 1
"to", 1 "the", 1 "heart", 1
Map #3
brevity is the soul of wit
map
"brevity", 1 "is", 1 "the", 1
"soul", 1 "of", 1 "wit", 1
partition
partition
partition
Reduce #1
"to", 1 "be", 1 "to", 1
"be", 1 "to", 1 "native", 1
"heart", 1 "more", 1 "of", 1
shuffle
"be", <1,1> "heart", <1>
"more", <1> "native", <1>
"of", <1> "to", <1,1,1>
reduce
"be", 2 "heart", 1 "more", 1
"native", 1 "of", 1 "to", 3
Reduce #2
"or", 1 "is", 1 "the", 1 "the", 1
"is", 1 "the", 1 "is", 1 "the", 1
"soul", 1 "wit", 1
shuffle
"is", <1,1,1> "or", <1>
"soul", <1> "the", <1,1,1,1>
"wit", <1>
reduce
"is", 3 "or", 1 "soul", 1
"the", 4 "wit", 1
Reduce #3
"not", 1 "that", 1
"question", 1 "head", 1
"not", 1 "brevity", 1
shuffle
"brevity", <1> "head", <1>
"not", <1,1> "question", <1>
"that", <1>
reduce
"brevity", 1 "head", 1 "not", 2
"question", 1 "that", 1

Figure 14.6 Data flow for the MapReduce definition shown in Figure 14.5, using 3 map shards and 3 reduce shards.

To overcome this problem, we could modify the map workers so that they accumulate per-shard term counts in a local hash table and output one pair of the form (t, f_t) instead of f_t pairs of the form $(t, 1)$ when they are finished with the current shard. This approach has the disadvantage that it requires extra implementation effort by the programmer.

As an alternative to the hash table method, it is possible to perform a local shuffle/reduce phase for each map shard before forwarding any data to the reduce workers. This approach has about the same performance effect as accumulating local counts in a hash table, but is often preferred by developers because it does not require any changes to their implementation. A reduce function that is applied to a map shard instead of a reduce shard is called a *combiner*. Every reduce function can serve as a combiner as long as its input values are of the same type as its output values, so that it can be applied to its own output. In the case of the counting MapReduce, this requirement is met, as all input and output values in the reduce phase are integers.

14.2.3 Secondary Keys

The basic MapReduce framework, as described so far, does not make any guarantees regarding the relative order in which values for the same key are fed into the reduce function. Often this is not a problem, because the reduce function can inspect and reorder the values in any way it wishes. However, for certain tasks the number of values for a given key may be so large that they cannot all be loaded into memory at the same time, thus making reordering inside the reduce function difficult. In that situation it helps to have the MapReduce framework sort each key's values in a certain way before they are passed to the reduce function.

An example of a task in which it is imperative that values arrive at the reduce function in a certain, predefined order is index construction. To create a docid index for a given text collection, we might define a map function that, for each term t encountered in a document d, outputs the key/value pair $(t, \mathit{docid}(d))$. The reduce function then builds t's postings list by concatenating all its postings (and potentially compressing them). Obviously, for this to be possible, the reduce input has to arrive in increasing order of $\mathit{docid}(d)$.

MapReduce supports the concept of *secondary keys* that can be used to define the order in which values for the same key arrive at the reduce function. In the shuffle phase, key/value pairs are sorted by their key, as usual. However, if there is more than one value for a given key, the key's values are sorted according to their secondary key. In the index construction MapReduce, the secondary key of a key/value pair would be the same as the pair's value (i.e., the docid of the document that contains the given term).

14.2.4 Machine Failures

When running a MapReduce that spans across hundreds or thousands of computers, some of the machines occasionally experience problems and have to be shut down. The MapReduce framework assumes that the map function behaves strictly deterministically, and that the map output for a given map shard depends only on that one shard (i.e., no information may be exchanged between two shards processed by the same map worker). If this assumption holds, then a map worker failure can be dealt with by assigning its shards to a different machine and reprocessing them.

Dealing with a reduce worker failure is slightly more complicated. Because the data in each reduce shard may depend on data in every map shard, assigning the reduce shard to a different worker may necessitate the re-execution of all map shards. In order to avoid this, the output of the map phase is usually not sent directly to the reduce workers but is temporarily stored in a reliable storage layer, such as a dedicated storage server or the Google file system (Ghemawat et al., 2003), from where it can be read by the new reduce worker in case of a worker failure. However, even if the map output is stored in a reliable fashion, a reduce worker failure will still require the re-execution of the shuffle phase for the failed shard. If we want to avoid this, too, then the reduce worker needs to send the output of the shuffle phase back to the storage server before it enters the reduce phase. Because, for a given shard, the shuffle phase and the reduce phase take place on the same machine, the additional network traffic is usually not worth the time savings unless machine failures occur frequently.

14.3 Further Reading

Compared with other topics covered by this book, the literature on parallel information retrieval is quite sparse. Existing publications are mostly limited to small or mid-size compute clusters comprising not more than a few dozen machines (the exception being occasional publications by some of the major search engine companies). In some cases it may therefore be difficult to assess the scalability of a proposed architecture. For instance, although the basic version of term partitioning discussed in Section 14.1.2 might work well on an 8-node cluster, it is quite obvious that it does not scale to a cluster containing hundreds or thousands of nodes. Despite this caveat, however, some of the results obtained in small-scale experiments may still be applicable to large-scale parallel search engines.

Load balancing issues for term-partitioned query evaluation are investigated by Moffat et al. (2006). Their study shows that, with the right load balancing policies in place, term partitioning can lead to almost the same query performance as document partitioning. Marín and Gil-Costa (2007) conduct a similar study and come to the conclusion that a term-partitioned index can sometimes outperform a document-partitioned one. Abusukhon et al. (2008) examine a variant of term partitioning in which terms with long postings lists are distributed across multiple index nodes.

Puppin et al. (2006) discuss a document partitioning scheme in which documents are not assigned to nodes at random but based on the queries for which they are ranked highly (according to an existing query log). Documents that rank highly for the same set of queries tend to be assigned to the same node. Each incoming query is forwarded only to those index nodes that are likely to return good results for the query. Xi et al. (2002) and Marín and Gil-Costa (2007) report on experiments conducted with hybrid partitioning schemes, combining term partitioning and document partitioning. A slightly different view of parallel query processing is presented by Marín and Navarro (2003), who discuss distributed query processing based on suffix arrays instead of inverted files.

Barroso et al. (2003) provide an overview of distributed query processing at Google. Other instruments for large-scale data processing at Google are described by Ghemawat et al. (2003), Dean and Ghemawat (2004, 2008), and Chang et al. (2008).

Hadoop[3] is an open-source framework for parallel computations that was inspired by Google's MapReduce and GFS technologies. Among other components, Hadoop includes HDFS (a distributed file system) and a MapReduce implementation. The Hadoop project was started by Doug Cutting, who also created the Lucene search engine. Yahoo is one of the main contributors to the project and is believed to be running the world's largest Hadoop installation, comprising several thousand machines.

Recently the use of graphics processing units (GPUs) for general-purpose, non-graphics-related computations has received some attention. Due to their highly parallel nature, GPUs can easily beat ordinary CPUs in applications in which long sequences of data have to be processed sequentially or cosequentially, such as sorting (Govindaraju et al., 2006; Sintorn and Assarsson, 2008) and disjunctive (i.e., Boolean-OR) query processing (Ding et al., 2009).

14.4 Exercises

Exercise 14.1 When replicating a distributed search engine, the replication may take place either at the node level (i.e., a single cluster with $2n$ index nodes, where two nodes share the query load for a given index shard), or at the cluster level (i.e., two identical clusters, but no replication within each cluster). Discuss the advantages and disadvantages of each approach.

Exercise 14.2 Describe possible scalability issues that may arise in the context of document-partitioned indices even if the index is held in main memory. (Hint: Consider query processing operations whose complexity is sublinear in the size of the index.)

Exercise 14.3 Given a document-partitioned index with $n = 200$ nodes and a target result set size of $m = 50$, what is the minimum per-node result set size k required to obtain the correct top m results with probability 99%?

Exercise 14.4 Generalize the dormant replication strategy from Section 14.1.4 so that it can deal with k simultaneous machine failures. How does this affect the overall storage requirements?

[3] `hadoop.apache.org`

Exercise 14.5 Describe how dormant replication for a term-partitioned index differs from dormant replication for a document-partitioned index.

Exercise 14.6 (a) Design a MapReduce (i.e., a map function and a reduce function) that computes the average document length (number of tokens per document) in a given corpus of text. (b) Revise your reduce function so that it can be used as a combiner.

Exercise 14.7 Design a MapReduce that computes the conditional probability $\Pr[t_1|t_2]$ of seeing the term t_1 in a document that contains the term t_2. You may find it useful to have your map function emit secondary keys to enforce a certain ordering among all values for a given key.

Exercise 14.8 (project exercise) Simulate a document-partitioned search engine using the BM25 ranking function you implemented for Exercise 5.9. Build an index for 100%, 50%, 25%, and 12.5% of the GOV2 collection. For each index size, measure the average time per query (for some standard query set). What do you observe? How does this affect the scalability of document partitioning?

14.5 Bibliography

Abusukhon, A., Talib, M., and Oakes, M. P. (2008). An investigation into improving the load balance for term-based partitioning. In *Proceedings of the 2nd International United Information Systems Conference*, pages 380–392. Klagenfurt, Austria.

Barroso, L. A., Dean, J., and Hölzle, U. (2003). Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28.

Carbonell, J. G., and Goldstein, J. (1998). The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 335–336. Melbourne, Australia.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):1–26.

Clarke, C. L. A., and Terra, E. L. (2004). Approximating the top-m passages in a parallel question answering system. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management*, pages 454–462. Washington, D.C.

Dean, J., and Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 137–150. San Francisco, California.

Dean, J., and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Ding, S., He, J., Yan, H., and Suel, T. (2009). Using graphics processors for high performance IR query processing. In *Proceedings of the 18th International Conference on World Wide Web*, pages 421–430. Madrid, Spain.

Ghemawat, S., Gobioff, H., and Leung, S. T. (2003). The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43. Bolton Landing, New York.

Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. (2006). GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 325–336. Chicago, Illinois.

Marín, M., and Gil-Costa, V. (2007). High-performance distributed inverted files. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, pages 935–938. Lisbon, Portugal.

Marín, M., and Navarro, G. (2003). Distributed query processing using suffix arrays. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval*, pages 311–325. Manaus, Brazil.

Moffat, A., Webber, W., and Zobel, J. (2006). Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 348–355. Seattle, Washington.

Moffat, A., Webber, W., Zobel, J., and Baeza-Yates, R. (2007). A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10(3):205–231.

Puppin, D., Silvestri, F., and Laforenza, D. (2006). Query-driven document partitioning and collection selection. In *Proceedings of the 1st International Conference on Scalable Information Systems*. Hong Kong, China.

Sintorn, E., and Assarsson, U. (2008). Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388.

Xi, W., Sornil, O., Luo, M., and Fox, E. A. (2002). Hybrid partition inverted files: Experimental validation. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries*, pages 422–431. Rome, Italy.

15 Web Search

Apart from an occasional example or exercise, the preceding chapters deal with information retrieval in a generic context. We assume the IR system contains a collection of documents, with each document represented by a sequence of tokens. Markup may indicate titles, authors, and other structural elements. We assume nothing further about the IR system's operating environment or the documents it contains.

In this chapter we consider IR in the specific context of Web search, the context in which it may be most familiar to you. Assuming this specific context provides us with the benefit of document features that cannot be assumed in the generic context. One of the most important of these features is the structure supplied by hyperlinks. These links from one page to another, often labeled by an image or anchor text, provide us with valuable information concerning both the individual pages and the relationship between them.

Along with these benefits come various problems, primarily associated with the relative "quality", "authority" or "popularity" of Web pages and sites, which can range from the carefully edited pages of a major international news agency to the personal pages of a high school student. In addition, many Web pages are actually *spam* — malicious pages deliberately posing as something that they are not in order to attract unwarranted attention of a commercial or other nature. Although the owners of most Web sites wish to enjoy a high ranking from the major commercial search engines, and may take whatever steps are available to maximize their ranking, the creators of spam pages are in an adversarial relationship with the search engine's operators. The creators of these pages may actively attempt to subvert the features used for ranking, by presenting a false impression of content and quality.

Other problems derive from the scale of the Web — billions of pages scattered among millions of hosts. In order to index these pages, they must be gathered from across the Web by a *crawler* and stored locally by the search engine for processing. Because many pages may change daily or hourly, this snapshot of the Web must be refreshed on a regular basis. While gathering data the crawler may detect duplicates and near-duplicates of pages, which must be dealt with appropriately. For example, the standard documentation for the Java programming language may be found on many Web sites, but in response to a query such as ⟨"java", "vector", "class"⟩ it might be best for a search engine to return only the official version on the `java.sun.com` site.

Another consideration is the volume and variety of queries commercial Web search engines receive, which direct reflect the volume and variety of information on the Web itself. Queries are often short — one or two terms — and the search engine may know little or nothing about the user entering a query or the context of the user's search. A user entering the query ⟨"UPS"⟩ may be interested in tracking a package sent by the courier service, purchasing a universal power supply, or attending night classes at the University of Puget Sound. Although such query ambiguity is a consideration in all IR applications, it reaches an extreme level in Web IR.

15.1 The Structure of the Web

Figure 15.1 provides an example of the most important features related to the structure of the Web. It shows Web pages on three sites: W, M, and H.[1] Site W provides a general encyclopedia including pages on Shakespeare (`w0.html`) and two of his plays, *Hamlet* (`w1.html`) and *Macbeth* (`w2.html`). Site H provides historical information including information on Shakespeare's wife (`h0.html`) and son (`h1.html`). Site M provides movie and TV information including a page on the 1971 movie version of *Macbeth* directed by Roman Polanski (`m0.html`).

The figure illustrates the link structure existing between these pages and sites. For example, the HTML *anchor* on page `w0.html`

```
<a href="http://H/h0.html">Anne Hathaway</a>
```

establishes a link between that page and `h0.html` on site H. The anchor text associated with this link ("Anne Hathaway") provides an indication of the content on the target page.

15.1.1 The Web Graph

The relationship between sites and pages indicated by these hyperlinks gives rise to what is called a *Web graph*. When it is viewed as a purely mathematical object, each page forms a node in this graph and each hyperlink forms a directed edge from one node to another. Figure 15.2 shows the Web graph corresponding to Figure 15.1. For convenience we have simplified the labels on the pages, with `http://W/w0.html` becoming w_0, for example.

From a practical standpoint it is important to remember that when we refer to a Web graph we are referring to more than just this mathematical abstraction — that pages are grouped into sites, and that anchor text and images may be associated with each link. Links may reference a labeled position within a page as well as the page as a whole. The highly dynamic and fluid nature of the Web must also be remembered. Pages and links are continuously added and removed at sites around the world; it is never possible to capture more than a rough approximation of the entire Web graph.

At times our interest may be restricted to a subset of the Web, perhaps to all the pages from a single site or organization. Under these circumstances, capturing an accurate snapshot of the Web graph for that site or organization is easier than for the entire Web. Nonetheless, most Web sites of any size grow and change on a continuous basis, and must be regularly recrawled for the snapshot to remain accurate.

[1] For simplicity we use single-letter site names and simplified page names rather than actual site and page names — for instance, `http://W/w0.html` instead of a full URL such as `http://en.wikipedia.org/wiki/William_Shakespeare`.

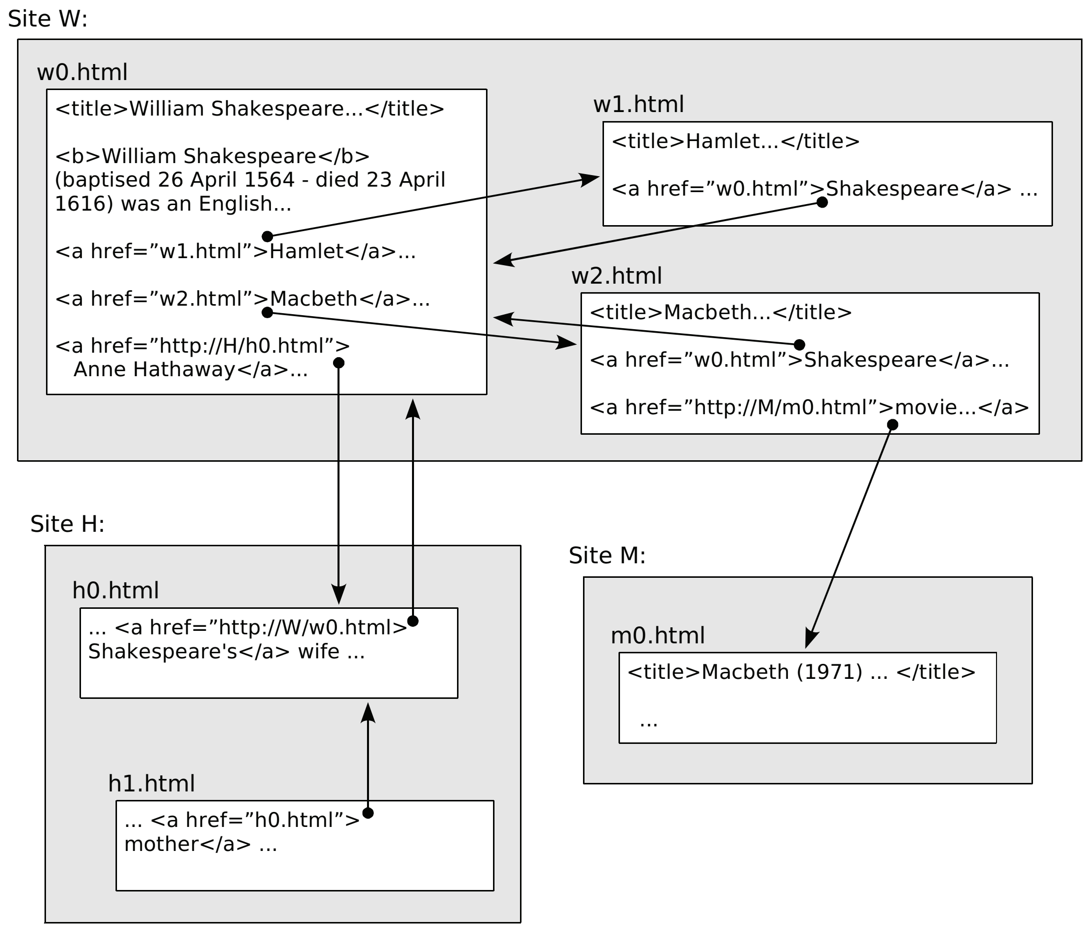

Site W:
w0.html
<title>William Shakespeare...</title>
<b>William Shakespeare</b> (baptised 26 April 1564 - died 23 April 1616) was an English...
<a href="w1.html">Hamlet</a>...
<a href="w2.html">Macbeth</a>...
<a href="http://H/h0.html"> Anne Hathaway</a>...
w1.html
<title>Hamlet...</title>
<a href="w0.html">Shakespeare</a> ...
w2.html
<title>Macbeth...</title>
<a href="w0.html">Shakespeare</a>...
<a href="http://M/m0.html">movie...</a>
Site H:
h0.html
... <a href="http://W/w0.html> Shakespeare's</a> wife ...
h1.html
... <a href="h0.html"> mother</a> ...
Site M:
m0.html
<title>Macbeth (1971) ... </title>
...

Figure 15.1 Structure on the Web. Pages can link to each other (`<a href=...>`) and can associate each link with anchor text (e.g., "mother").

We now introduce a formal notation for Web graphs that we use in later sections. Let Φ be the set of all pages in a Web graph, let $N = |\Phi|$ be the number of pages, and E the number of links (or edges) in the graph. Given a page $\alpha \in \Phi$, we define *out*(α) as the number of *out-links* from α to other pages (the *out-degree*). Similarly, we define *in*(α) as the number of *in-links* from other pages to α (the *in-degree*). In the Web graph of Figure 15.2, *out*$(w_0) = 3$ and *in*$(h_0) = 2$. If *in*$(\alpha) = 0$, then α is called a *source*; if *out*$(\alpha) = 0$, it is called a *sink*. We define Γ as the set of sinks. In the Web graph of Figure 15.2, page m_0 is a sink and page h_1 is a source.

The individual Web pages themselves may be highly complex and structured objects. They often include menus, images, and advertising. Scripts may be used to generate content when the page is first loaded and to update the content as the user interacts with it. A page may

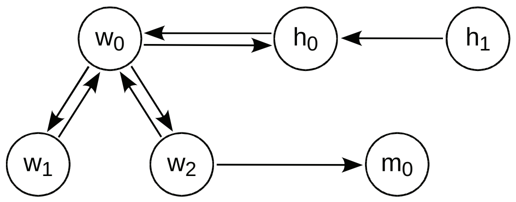


Figure 15.2 A Web graph for the Web pages shown in Figure 15.1. Each link corresponds to a directed edge in the graph.

serve as a frame to hold other pages or may simply redirect the browser to another page. These redirections may be implemented through scripts or through special HTTP responses, and may be repeated multiple times, thus adding further complexity. Along with HTML pages the Web includes pages in PDF, Microsoft Word, and many other formats.

15.1.2 Static and Dynamic Pages

It is not uncommon to hear Web pages described as "static" or "dynamic". The HTML for a static Web page is assumed to be generated in advance of any request, placed on disk, and transferred to the browser or Web crawler on demand. The home page of an organization is a typical example of a static page. A dynamic Web page is assumed to be generated at the time the request is made, with the contents of the page partially determined by the details of the request. A search engine result page (SERP) is a typical example of a dynamic Web page for which the user's query itself helps to determine the content.

There is often the implication in this dichotomy that static Web pages are more important for crawling and indexing than dynamic Web pages, and it is certainly the case that many types of dynamic Web pages are not suitable for crawling and indexing. For example, on-line calendar systems, which maintain appointments and schedules for people and events, will often serve up dynamically generated pages for dates far into the past and the future. You can reach the page for any date if you follow the links far enough. Although the flexibility to book appointments 25 years from now is appropriate for an online calendar; indexing an endless series of empty months is not appropriate for a general Web search engine.

Nonetheless, although many dynamic Web pages should not be crawled and indexed, many should. For example, catalog pages on retail sites are often generated dynamically by accessing current products and prices in a relational database. The result is then formatted into HTML and wrapped with menus and other fixed information for display in a browser. To meet the needs of a consumer searching for a product, a search engine must crawl and index these pages.

A dynamic page can sometimes be identified by features of its URL. For example, servers accessed through the Common Gateway Interface (CGI) may contain the path element `cgi-bin` in their URLs. Pages dynamically generated by Microsoft's Active Server Pages technology include the extension `.asp` or `.aspx` in their URLs. Unfortunately, these URL features are not always present. In principle any Web page can be static or dynamic, and there is no sure way to tell. For crawling and indexing, it is the content that is important, not the static or dynamic nature of the page.

15.1.3 The Hidden Web

Many pages are part of the so-called "hidden" or "invisible" or "deep" Web. This hidden Web includes pages that have no links referencing them, those that are protected by passwords, and those that are available only by querying a digital library or database. Although these pages can contain valuable content, they are difficult or impossible for a Web crawler to locate.

Pages in *intranets* represent a special case of the hidden Web. Pages in a given intranet are accessible only within a corporation or similar entity. An enterprise search engine that indexes an intranet may incorporate many of the same retrieval techniques that are found in general Web search engines, but may be tuned to exploit specific features of that intranet.

15.1.4 The Size of the Web

Even if we exclude the hidden Web, it is difficult to compute a meaningful estimate for the size of the Web. Adding or removing a single host can change the number of accessible pages by an arbitrary amount, depending on the contents of that host. Some hosts contain millions of pages with valuable content; others contain millions of pages with little or no useful content (see Exercise 15.14).

However, many Web pages would rarely, if ever, appear near the top of search results. Excluding those pages from a search engine would have little impact. Thus, we may informally define what is called the *indexable Web* as being those pages that should be considered for inclusion in a general-purpose Web search engine (Gulli and Signorini, 2005). This indexable Web would comprise all those pages that could have a substantial impact on search results.

If we may assume that any page included in the index of a major search engine forms part of the indexable Web, a lower bound for the size of the indexable Web may be determined from the combined coverage of the major search engines. More specifically, if the sets $A_1, A_2, \ldots$ represent the sets of pages indexed by each of these search engines, a lower bound on the size of the indexable Web is the size of the union of these sets $|\cup A_i|$.

Unfortunately, it is difficult to explicitly compute this union. Major search engines do not publish lists of the pages they index, or even provide a count of the number of pages, although it is usually possible to check if a given URL is included in the index. Even if we know the number of pages each engine contains, the size of the union will be smaller than the sum of the sizes because there is considerable overlap between engines.

Bharat and Broder (1998) and Lawrence and Giles (1998) describe a technique for estimating the combined coverage of major search engines. First, a test set of URLs is generated. This step may be achieved by issuing a series of random queries to the engines and selecting a random URL from the results returned by each. We assume that this test set represents a uniform

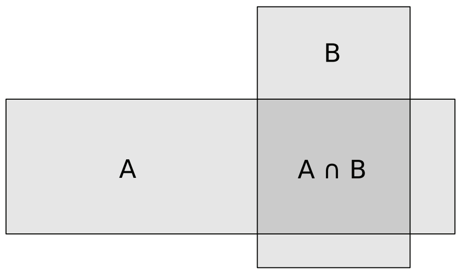


Figure 15.3 The collection overlap between search engines A and B can be used to estimate the size of the indexable Web.

sample of the pages indexed by the engines. Second, each URL from the test set is checked against each engine and the engines that contain it are recorded.

Given two engines, A and B, the relationship between their collections is illustrated by Figure 15.3. Sampling with our test set of URLs allows us to estimate $\Pr[A \cap B \,|\, A]$, the probability that a URL is contained in the intersection if it is contained in A:

$$\Pr[A \cap B \,|\, A] \;=\; \frac{\text{\# of test URLs contained in both } A \text{ and } B}{\text{\# of test URLs contained in } A}\,. \tag{15.1}$$

We may estimate $\Pr[A \cap B \,|\, B]$ in a similar fashion. If we know the size of A, we may then estimate the size of the intersection as

$$|A \cap B| \;=\; |A| \cdot \Pr[A \cap B \,|\, A]\,. \tag{15.2}$$

and the size of B as

$$|B| \;=\; \frac{|A \cap B|}{\Pr[A \cap B \,|\, B]}\,. \tag{15.3}$$

Thus, the size of the union may be estimated as

$$|A \cup B| \;=\; |A| + |B| - |A \cap B|\,. \tag{15.4}$$

If sizes for both A and B are available, the size of the intersection may be estimated from both sets, and the average of these estimates may be used to estimate the size of the union

$$|A \cup B| \;=\; |A| + |B| - \frac{1}{2}\left(|A| \cdot \Pr[A \cap B \,|\, A] + |B| \cdot \Pr[A \cap B \,|\, B]\right)\,. \tag{15.5}$$

This technique may be extended to multiple engines. Using a variant of this method, Bharat and Broder (1998) estimated the size of the indexable Web in mid-1997 at 160 million pages. In a study conducted at roughly the same time, Lawrence and Giles (1998) estimated the size at 320 million pages. Approximately a year later the size had grown to 800 million pages (Lawrence and Giles, 1999). By mid-2005 it was 11.5 billion (Gulli and Signorini, 2005).

15.2 Queries and Users

Web queries are short. Several studies of query logs from major search engines are summarized by Spink and Jansen (2004). Although the exact numbers differ from study to study, these studies consistently reveal that many queries are just one or two terms long, with a mean query length between two and three terms. The topics of these queries range across the full breadth of human interests, with sex, health, commerce, and entertainment representing some of the major themes.

Perhaps it is not surprising that Web queries are short. Until quite recently, the query processing strategies of Web search engines actually discouraged longer queries. As we described in Section 2.3, these processing strategies filter the collection against a Boolean conjunction of the query terms prior to ranking. For a page to be included in the result set, all query terms must be associated with it in some way, either by appearing on the page itself or by appearing in the anchor text of links referencing it. As a result, increasing the length of the query by adding related terms could cause relevant pages to be excluded if they are missing one or more of the added terms. This strict filtering has been relaxed in recent years. For example, synonyms may be accepted in place of exact matches — the term "howto" might be accepted in place of the query term "FAQ". Nonetheless, for efficiency reasons filtering still plays a role in query processing, and Web search engines may still perform poorly on longer queries.

The distribution of Web queries follows Zipf's law (see Figure 15.4). In a representative log of 10 million queries, the single most frequent query may represent more than 1% of the total, but nearly half of the queries will occur only once. The tuning and evaluation of a search engine must consider this "long tail" of Zipf's law. In the aggregate, infrequent queries are at least as important as the frequent ones.

15.2.1 User Intent

Several researchers have examined collections of queries issued to Web search engines in an attempt to characterize the user intent underlying these queries. Broder (2002) surveyed users of the Altavista search engine and examined its query logs to develop a taxonomy of Web search. He classified Web queries into three categories reflecting users' apparent intent, as follows:

- The intent behind a *navigational* query is to locate a specific page or site on the Web. For example, a user intending to locate the home page of the CNN news network might enter the query ⟨"CNN"⟩. A navigational query usually has a single correct result. However, this correct result may vary from user to user. A Spanish-speaking user from the United States might want the CNN en Español home page (`www.cnn.com/espanol`); a user from Tunisia may want the Arabic edition (`arabic.cnn.com`).

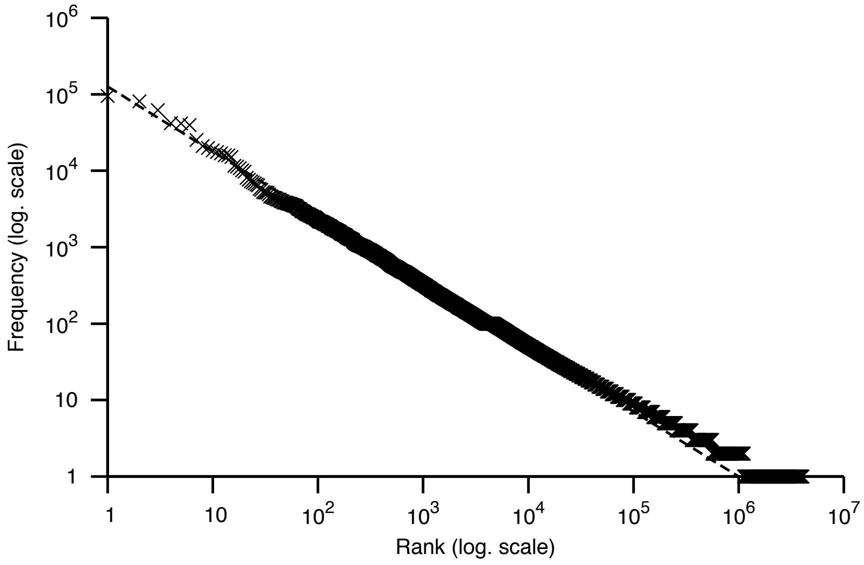


Figure 15.4 Frequency of queries by rank order, based on 10 million queries taken from the logs of a commercial search engine. The dashed line corresponds to Zipf's law with $\alpha = 0.85$.

- A user issuing an *informational* query is interested in learning something about a particular topic and has a lesser regard for the source of the information, provided it is reliable. The topics forming the test collections introduced in Chapter 1 and used for evaluation throughout the first four parts of the book reflect informational intent. A user entering the query ⟨"president", "obama"⟩ might find the information she is seeking on the CNN Web site, on Wikipedia, or elsewhere. Perhaps she will browse and read a combination of these before finding all the information she requires. The need behind an informational query such as this one may vary from user to user, and may be broad or narrow in scope. A user may be seeking a detailed description of government policy, a short biography, or just a birth date.
- A user issuing a *transactional* query intends to interact with a Web site once she finds it. This interaction may involve activities such as playing games, purchasing items, booking travel, or downloading images, music, and videos. This category may also include queries seeking services such as maps and weather, provided the user is not searching for a specific site providing that service.

Rose and Levinson (2004) extended the work of Broder by expanding his three categories into a hierarchy of user goals. They retained Broder's categories at the top level of their hierarchy but renamed the transactional category as the "resource" category. Under both the informational and the transactional categories they identified a number of subcategories. For example, the goal of a user issuing a *directed informational* query is the answer to a particular question ("When was President Obama born?"), whereas the goal of a user issuing an *undirected informational* query is simply to learn about the topic ("Tell me about President Obama."). In turn, directed informational queries may be classified as being *open* or *closed*, depending on whether the question is open-ended or has a specific answer.

The distinction between navigational and informational queries is relatively straightforward: Is the user seeking a specific site or not? The distinction between transactional queries and the other two categories is not always as clear. We can assume the query ⟨"mapquest"⟩ is navigational, seeking the site `www.mapquest.com`, but it is likely that the user will then interact with the site to obtain directions and maps. A user entering the query ⟨"travel", "washington"⟩ may be seeking both tourist information about Washington, D.C., and planning to book a hotel, making the intent both informational and transactional. In cases such as these, it may be reasonable to view such queries as falling under a combination of categories, as navigational/transactional and informational/transactional, respectively.

According to Broder (2002), navigational queries comprise 20–25% of all Web queries. Of the remainder, transactional queries comprise at least 22%, and the rest are informational queries. According to Rose and Levinson (2004), 12–15% of queries are navigational and 24–27% are transactional. A more recent study by Jansen et al. (2007) slightly contradicts these numbers. Their results indicate that more than 80% of queries are informational, with the remaining queries split roughly equally between the navigational and transactional categories. Nonetheless, these studies show that all three categories represent a substantial fraction of Web queries and suggest that Web search engines must be explicitly aware of the differences in user intent.

Referring to "navigational queries" and "informational queries" is common jargon. This usage is understandable when the goal underlying a query is the same, or similar, regardless of the user issuing it. But for some queries the category may differ from user to user. As a result we stress that these query categories fundamentally describe the goals and intentions of the users issuing the queries, and are not inherent properties of the queries themselves. For example, a user issuing our example query ⟨"UPS"⟩ may have

- Informational intent, wanting to know how universal power supplies work
- Transactional intent, wanting to purchase an inexpensive UPS for a personal computer
- Transactional/navigational intent, wanting to track a package or
- Navigational/informational intent, wanting information on programs offered by the University of Puget Sound.

Although this query is atypical, falling into so many possible categories, the assignment of any given query to a category (by anyone other than the user) may be little more than an educated guess.

15.2.2 Clickthrough Curves

The distinction between navigational and informational queries is visible in user behavior. Lee et al. (2005) examined *clickthroughs* as one feature that may be used to infer user intent. A clickthrough is the act of clicking on a search result on a search engine result page. Clickthroughs are often logged by commercial search engines as a method for measuring performance (see Section 15.5.2).

Although almost all clicks occur on the top ten results (Joachims et al., 2005; Agichtein et al., 2006b), the pattern of clicks varies from query to query. By examining clickthroughs from a large number of users issuing the same query, Lee et al. (2005) identified clickthrough distributions that are typical of informational and navigational queries. For navigational queries the clickthroughs are skewed toward a single result; for informational queries the clickthrough distribution is flatter.

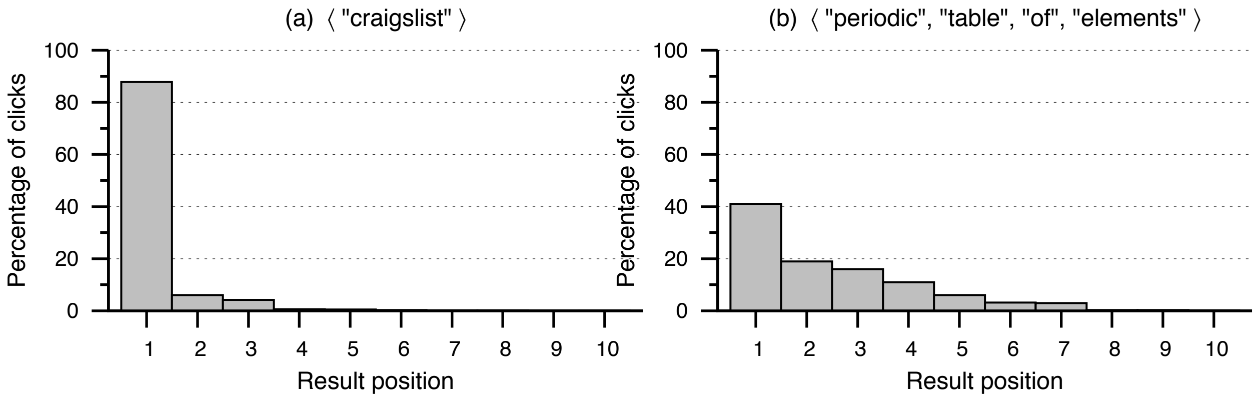


Figure 15.5 Clickthrough curve for a typical navigational query (⟨“craigslist”⟩) and a typical informational query (⟨“periodic”, “table”, “of”, “elements”⟩).

Clarke et al. (2007) analyzed logs from a commercial search engine and provided further analysis and examples. The plots in Figure 15.5 are derived from examples in that paper. Both plots show the percentage of clickthroughs for results ranked 1 to 10. Each clickthrough represents the first result clicked by a different user entering the query. Figure 15.5(a) shows a clickthrough distribution for a stereotypical navigational query, exhibiting a spike at `www.craigslist.org`, a classified advertising site and the presumed target of the query. Figure 15.5(b) shows a clickthrough distribution for a typical informational query. For both queries the number of clickthroughs decreases with rank; the informational query receives proportionally more clicks at lower ranks.

15.3 Static Ranking

Web retrieval works in two phases. The first phase takes place during the indexing process, when each page is assigned a *static rank* (Richardson et al., 2006). Informally, this static rank may be viewed as reflecting the quality, authority, or popularity of the page. Ideally, static rank may correspond to a page's prior probability of relevance — the higher the probability, the higher the static rank.

Static rank is independent of any query. At query time the second phase of Web retrieval takes place. During this phase the static rank is combined with query-dependent features, such as term proximity and frequency, to produce a *dynamic rank*.

Assigning a static rank during indexing allows a Web search engine to consider features that would be impractical to consider at query time. The most important of these features are those derived from *link analysis* techniques. These techniques extract information encoded in the structure of the Web graph. Other features may also contribute to static rank, including features derived from the content of the pages and from user behavior (Section 15.3.5).

Our presentation of static ranking begins with the fundamentals of PageRank, easily the most famous of the link analysis techniques, in Section 15.3.1. Although this section presents the version of the algorithm generally known by the name PageRank, its practical value in Web search may be relatively limited because it depends on naïve assumptions regarding the structure of the Web. Sections 15.3.2 and 15.3.3 present and analyze an extended version of the algorithm, which accommodates a more sophisticated view of Web structure. Section 15.3.4 provides an overview of other link analysis techniques and Section 15.3.5 briefly discusses other features applicable to static ranking. Dynamic ranking is covered in Section 15.4.

15.3.1 Basic PageRank

PageRank was invented in the mid-1990s through the efforts of Larry Page and Sergey Brin, two Stanford computer science graduate students. The algorithm became a key element of their Backrub search engine, which quickly matured into Google.

The classic intuition behind the PageRank algorithm imagines a person surfing the Web at random. At any point a Web page is visible in her browser. As a next step she can either

1. Follow a link from the current page by clicking on it or
2. Select a page uniformly at random and jump to it, typing its URL into the address bar.

At any step the probability she will follow a link is fixed as δ. Thus, the probability of a jump is $1 - \delta$. Reasonable values for δ might range from 0.75 to 0.90, with 0.85 being the value quoted most often in the research literature. For simplicity we use $\delta = 3/4$ for our experiments and examples, unless otherwise stated.

Sinks in the Web graph force a jump: When our surfer reaches a sink, she always takes the second option. Assuming that our surfer could surf quickly and tirelessly for a long period of time, the value of PageRank $r(\alpha)$ for a page α would indicate the relative frequency at which the surfer visits that page.

The probability δ is sometimes referred to as the *restart probability* or *damping factor*, because it reduces the probability that the surfer will follow a link. The use of a damping factor to allow random jumps is known to improve the stability of PageRank, in a statistical sense, in comparison to an equivalent algorithm without jumps (Section 15.3.3). Informally, small changes to the Web graph will not generate large changes in PageRank.

The value of $r(\alpha)$ may be expressed in terms of the PageRank values of the pages that link to it and the PageRank values of the sinks in the graph, as follows:

$$r(\alpha) \;=\; \delta \cdot \left(\sum_{\beta \to \alpha} \frac{r(\beta)}{out(\beta)} + \sum_{\gamma \in \Gamma} \frac{r(\gamma)}{N} \right) + (1-\delta) \cdot \sum_{\alpha \in \Phi} \frac{r(\alpha)}{N} \tag{15.6}$$

For simplicity, and because the choice is arbitrary, we assume

$$\sum_{\alpha \in \Phi} r(\alpha) \;=\; N, \tag{15.7}$$

reducing the equation slightly to

$$r(\alpha) \;=\; \delta \cdot \left(\sum_{\beta \to \alpha} \frac{r(\beta)}{out(\beta)} + \sum_{\gamma \in \Gamma} \frac{r(\gamma)}{N} \right) + (1-\delta). \tag{15.8}$$

Because $\sum_{\alpha \in \Phi} r(\alpha)/N \;=\; 1$, the probability that the random surfer will be at page α at any given point is thus $r(\alpha)/N$.

Let us consider the composition of Equation 15.8 in detail. First, the value $(1-\delta)$ reflects the contribution from random jumps to α. The other contributions to the PageRank value of α — from links and sinks — are more complex. In the case of links, the page α may be reached by following a link from a page β. Because β may have links to multiple pages, its PageRank value is distributed according to its out-degree *out(β)*. Finally, jumps from sinks contribute in a small way to the PageRank value of α. Because a jump from sink γ may target any of the N pages in the graph — even itself — its contribution is $r(\gamma)/N$.

The application of Equation 15.8 to the Web graph of 15.2 gives the following set of equations:

$$\begin{aligned}
r(w_0) &= \delta \cdot \left(r(w_1) + \frac{r(w_2)}{2} + r(h_0) + \frac{r(m_0)}{6} \right) + (1-\delta) \\
r(w_1) &= \delta \cdot \left(\frac{r(w_0)}{3} + \frac{r(m_0)}{6} \right) + (1-\delta)
\end{aligned}$$

$$\begin{aligned}
r(w_2) &= \delta \cdot \left(\frac{r(w_0)}{3} + \frac{r(m_0)}{6} \right) + (1-\delta) \\
r(h_0) &= \delta \cdot \left(\frac{r(w_0)}{3} + r(h_1) + \frac{r(m_0)}{6} \right) + (1-\delta) \\
r(h_1) &= \delta \cdot \left(\frac{r(m_0)}{6} \right) + (1-\delta) \\
r(m_0) &= \delta \cdot \left(\frac{r(w_2)}{2} + \frac{r(m_0)}{6} \right) + (1-\delta)
\end{aligned}$$

Setting $\delta = 3/4$ and simplifying gives

$$\begin{aligned}
r(w_0) &= \frac{3\,r(w_1)}{4} + \frac{3\,r(w_2)}{8} + \frac{3\,r(h_0)}{4} + \frac{r(m_0)}{8} + \frac{1}{4} \\
r(w_1) &= \frac{r(w_0)}{4} + \frac{r(m_0)}{8} + \frac{1}{4} \\
r(w_2) &= \frac{r(w_0)}{4} + \frac{r(m_0)}{8} + \frac{1}{4} \\
r(h_0) &= \frac{r(w_0)}{4} + \frac{3\,r(h_1)}{4} + \frac{r(m_0)}{8} + \frac{1}{4} \\
r(h_1) &= \frac{r(m_0)}{8} + \frac{1}{4} \\
r(m_0) &= \frac{3\,r(w_2)}{8} + \frac{r(m_0)}{8} + \frac{1}{4}
\end{aligned}$$

To compute PageRank, we solve for the six variables in the resulting system of linear equations $r(w_0)$, $r(w_1)$, $r(w_2)$, $r(h_0)$, $r(h_1)$, and $r(m_0)$. Many algorithms exist for the numerical solution of linear systems such as this one. The method particularly suited to PageRank is a form of *fixed-point iteration.*

Fixed point iteration is a general technique for solving systems of equations, linear and otherwise. To apply it, each variable must be expressed as a function of the other variables and itself. The PageRank equations are already expressed in this form, with the PageRank value for each page appearing alone on the left-hand side, and functions of these PageRank values appearing on the right-hand side.

The algorithm begins by making an initial guess for the value of each variable. These values are substituted into the right-hand equations, generating new approximations for the variables. We repeat this process, substituting the current values to generate new approximations. If the values converge, staying the same from iteration to iteration, we have solved the system of equations.

The method produces a series of approximations to $r(\alpha)$ for each page α: $r^{(0)}(\alpha)$, $r^{(1)}(\alpha)$, $r^{(2)}(\alpha), \ldots$ If we choose as our initial guess $r^{(0)}(\alpha) = 1$ for all pages, we have

$$\sum_{\alpha \in \Phi} r^{(0)}(\alpha) \;=\; N, \tag{15.9}$$

as required by Equation 15.7. From Equation 15.8 we then compute new approximations from existing approximations with the equation:

$$r^{(n+1)}(\alpha) \;=\; \delta \cdot \left(\sum_{\beta \to \alpha} \frac{r^{(n)}(\beta)}{out(\beta)} + \sum_{\gamma \in \Gamma} \frac{r^{(n)}(\gamma)}{N} \right) + (1 - \delta). \tag{15.10}$$

To compute PageRank for the Web graph of Figure 15.2, we iterate the following equations:

$$\begin{aligned}
r^{(n+1)}(w_0) &= \frac{3\,r^{(n)}(w_1)}{4} + \frac{3\,r^{(n)}(w_2)}{8} + \frac{3\,r^{(n)}(h_0)}{4} + \frac{r^{(n)}(m_0)}{8} + \frac{1}{4} \\
r^{(n+1)}(w_1) &= \frac{r^{(n)}(w_0)}{4} + \frac{r^{(n)}(m_0)}{8} + \frac{1}{4} \\
r^{(n+1)}(w_2) &= \frac{r^{(n)}(w_0)}{4} + \frac{r^{(n)}(m_0)}{8} + \frac{1}{4} \\
r^{(n+1)}(h_0) &= \frac{r^{(n)}(w_0)}{4} + \frac{3\,r^{(n)}(h_1)}{4} + \frac{r^{(n)}(m_0)}{8} + \frac{1}{4} \\
r^{(n+1)}(h_1) &= \frac{r^{(n)}(m_0)}{8} + \frac{1}{4} \\
r^{(n+1)}(m_0) &= \frac{3\,r^{(n)}(w_2)}{8} + \frac{r^{(n)}(m_0)}{8} + \frac{1}{4}
\end{aligned}$$

Table 15.1 shows the successive approximations to PageRank for this system of equations. After 18 iterations the values have converged to within three decimal places.

Figure 15.6 presents a detailed algorithm for computing PageRank. The algorithm assumes that pages in the Web graph are numbered consecutively from 1 to N. The array *link*, of length E, stores the links in the Web graph, with *link*$[i]$.*from* indicating the source of the link and *link*$[i]$.*to* indicating the destination of the link. The array R stores the current approximation to PageRank; the array R' accumulates the new approximation.

The loop over lines 1–2 sets R to the initial approximation. At the end of each iteration of the main loop (lines 3–14) the array R contains the next approximation. As written, these lines form an infinite loop. In practice this loop would be terminated when the approximation converges or after a fixed number of iterations. We omit the exact termination condition because it depends on the accuracy desired and the precision of the computations. Over most Web graphs, acceptable values can be achieved after a few hundred iterations. Each iteration of the main loop takes $O(N + E)$ time; the time complexity of the overall algorithm depends on the number of iterations of the main loop.

Table 15.1 Iterative computation of PageRank for the Web graph of Figure 15.1.

n	$r^{(n)}(w_0)$	$r^{(n)}(w_1)$	$r^{(n)}(w_2)$	$r^{(n)}(h_0)$	$r^{(n)}(h_1)$	$r^{(n)}(m_0)$
0	1.000	1.000	1.000	1.000	1.000	1.000
1	2.250	0.625	0.625	1.375	0.375	0.750
2	2.078	0.906	0.906	1.188	0.344	0.578
3	2.232	0.842	0.842	1.100	0.322	0.662
4	2.104	0.891	0.891	1.133	0.333	0.648
5	2.183	0.857	0.857	1.107	0.331	0.665
6	2.128	0.879	0.879	1.127	0.333	0.655
7	2.166	0.864	0.864	1.114	0.332	0.661
8	2.140	0.874	0.874	1.123	0.333	0.657
9	2.158	0.867	0.867	1.116	0.332	0.660
10	2.145	0.872	0.872	1.121	0.332	0.658
11	2.154	0.868	0.868	1.118	0.332	0.659
12	2.148	0.871	0.871	1.120	0.332	0.658
13	2.152	0.869	0.869	1.119	0.332	0.659
14	2.149	0.870	0.870	1.120	0.332	0.658
15	2.151	0.870	0.870	1.119	0.332	0.659
16	2.150	0.870	0.870	1.119	0.332	0.658
17	2.151	0.870	0.870	1.119	0.332	0.659
18	2.150	0.870	0.870	1.119	0.332	0.658
19	2.150	0.870	0.870	1.119	0.332	0.659
20	2.150	0.870	0.870	1.119	0.332	0.658
...	...	...	...	...	...	...

Lines 4–5 initialize R' by storing the contribution from jumps. Lines 6–9 consider each link in turn, computing the contribution of the source page to the destination's PageRank value. Because $\sum_{\alpha \in \Phi} r(\alpha) = N$, the set of sinks need not be explicitly considered. Instead, after initializing R' and applying links we sum the elements of R'. The difference between the resulting sum and N represents $\sum_{\gamma \in \Gamma} r(\gamma)$, the contribution from sinks in the graph.

As an alternative to lines 6–9 we might consider iterating over all nodes. For each node we would then sum the PageRank contributions for the nodes linking to it, directly implementing Equation 15.8. Although this alternative is reasonable for small Web graphs, the current approach of iterating over links requires simpler data structures and saves memory, an important consideration for larger Web graphs.

Iterating over links also allows us to store the links in a file on disk, further reducing memory requirements. Instead of iterating over an array, the loop of lines 6–9 would sequentially read this file of links from end to end during each iteration of the main loop. If links are stored in a file, the algorithm requires only the $O(N)$ memory to store the arrays R and R'.

```
for $i \leftarrow 1$ to $N$ do
    $R[i] \leftarrow 1$
loop
    for $i \leftarrow 1$ to $N$ do
        $R'[i] \leftarrow 1 - \delta$
    for $k \leftarrow 1$ to $E$ do
        $i \leftarrow link[k].from$
        $j \leftarrow link[k].to$
        $R'[j] \leftarrow R'[j] + \frac{\delta \cdot R[i]}{out(i)}$
    $s \leftarrow N$
    for $i \leftarrow 1$ to $N$ do
        $s \leftarrow s - R'[i]$
    for $i \leftarrow 1$ to $N$ do
        $R[i] \leftarrow R'[i] + s/N$
```

Figure 15.6 Basic PageRank algorithm. The array *link* contains the links in the Web graph. Lines 3-14 form an infinite loop. At the end of each interation of this loop, the array R contains a new estimate of PageRank for each page. In practice the loop would be terminated when the change in PageRank from iteration to iteration drops below a threshold or after a fixed number of iterations.

To demonstrate PageRank over a larger Web graph, we applied the algorithm to the English Wikipedia (see Exercise 1.9). The top 12 pages are listed in Table 15.2. Unsurprisingly, given that the collection includes only the English Wikipedia, major English-speaking countries figure prominently on the list. The remaining countries have large economies and close ties to the United States and to other English-speaking countries. Dates are often linked in Wikipedia, which explains the high ranks given to recent years. The high rank given to World War II, the pivotal event of the twentieth century, is also unsurprising. Overall, these results reflect the culture and times of Wikipedia's authors, an appropriate outcome.

15.3.2 Extended PageRank

The limitations of basic PageRank become evident if we revisit and reconsider its original motivation, the random surfer. No real user accesses the Web in the fashion of this random surfer. Even if we view the random surfer as an artificial construct — as a homunculus exhibiting the average behavior of Web users as a group — its selection of links and jumps uniformly at random remains unrealistic.

More realistically, our random surfer should have preferences for both links and jumps. For example, she might prefer navigational links over links to ads, prefer links at the top of the page to those at the bottom, and prefer links in readable fonts to those in tiny or invisible fonts. When the random surfer is jumping randomly, top-level pages may be preferred over deeply nested pages, longstanding pages may be preferred over newly minted pages, and pages with more text may be preferred over pages with less.

Table 15.2 The top 12 pages from a basic PageRank of Wikipedia.

Page: α	**PageRank: $r(\alpha)$**	**Probability: $r(\alpha)/N$**
United States	10509.50	0.004438
United Kingdom	3983.74	0.001682
2006	3781.65	0.001597
England	3421.03	0.001445
France	3340.53	0.001411
2007	3301.65	0.001394
2005	3290.57	0.001389
Germany	3218.33	0.001359
Canada	3090.20	0.001305
2004	2742.86	0.001158
Australia	2441.65	0.001031
World War II	2417.38	0.001021

Fortunately, we can easily extend PageRank to accommodate these preferences. To accommodate jump preferences, we define a *teleport vector* or *jump vector* J, of length N, where the element $J[i]$ indicates the probability that the random surfer will target page i when jumping. Because J is a vector of probabilities, we require $\sum_{i=1}^{N} J[i] = 1$. It is acceptable for some elements of J to be 0, but this may result in some pages having a zero PageRank value. If all elements of J are positive, $J[i] > 0$ for all $1 \leq i \leq N$, then all pages will have non-zero PageRank (Section 15.3.3).

To accommodate link preferences, we define an $N \times N$ *follow matrix* F, where element $F[i, j]$ indicates the probability that our random surfer will following a link to page j from page i. Unless page i is a sink, the elements in its row of F must sum to 1, $\sum_{i=1}^{N} F[i, j] = 1$. If page i is a sink, all elements of the row are 0.

F is *sparse*. At most E of the N^2 elements are non-zero, with one element corresponding to each link. We exploit this property of F in our implementation of the extended PageRank algorithm. The algorithm is given in Figure 15.7. It is similar to the basic PageRank algorithm of Figure 15.6, with lines 5, 9, and 14 generalized to use the jump vector and follow matrix. This generalization has no impact on the algorithm's time complexity, with each iteration of the main loop requiring $O(N + E)$ time.

Because F is sparse, the elements of F may be stored with their corresponding links. More specifically, we may add a *follow* field to each element of the *link* array, defined so that

$$\mathit{link}[k].\mathit{follow} = F[\mathit{link}[k].\mathit{from}, \mathit{link}[k].\mathit{to}].$$

```
1   for $i \leftarrow 1$ to $N$ do
2       $R[i] \leftarrow J[i] \cdot N$
3   loop
4       for $i \leftarrow 1$ to $N$ do
5           $R'[i] \leftarrow (1 - \delta) \cdot J[i] \cdot N$
6       for $k \leftarrow 1$ to $E$ do
7           $i \leftarrow link[k].from$
8           $j \leftarrow link[k].to$
9           $R'[j] \leftarrow R'[j] + \delta \cdot R[i] \cdot F[i, j]$
10      $s \leftarrow N$
11      for $i \leftarrow 1$ to $N$ do
12          $s \leftarrow s - R'[i]$
13      for $i \leftarrow 1$ to $N$ do
14          $R[i] \leftarrow R'[i] + s \cdot J[i]$
```

Figure 15.7 Extended PageRank algorithm. Each element $J[i]$ of the jump vector J indicates the probabilty of reaching page i when jumping randomly. Each element $F[i,j]$ of the follow matrix F indicates the probabilty of reaching page i from page j when following a link.

This extended link array could then be stored in a file on disk, with the loop of lines 6–9 changed to iterate over this file. With the links stored on disk, the algorithm requires only $O(N)$ RAM to store the arrays R and R' and the jump vector J.

To compute basic PageRank using this extended algorithm, we construct a jump vector with all elements equal to $1/N$. For the follow matrix we set its elements equal to $1/out(\alpha)$ for all pages to which a page α links, and to 0 otherwise. If we number the nodes in the graph of Figure 15.2 from 1 to 6 in the order w_0, w_1, w_2, h_0, h_1, and m_0, we obtain the corresponding follow matrix and jump vector for basic PageRank:

$$F = \begin{pmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{2} \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad J = \begin{pmatrix} \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \end{pmatrix}. \tag{15.11}$$

Now, suppose we know from external information that site W contains only high-quality and carefully edited information, and the relative quality of the information on the other sites remains unknown. We might adjust the follow matrix and jump vector to reflect this knowledge by assuming the random surfer is twice as likely to link or jump to site W than to any other site, as follows:

$$F = \begin{pmatrix} 0 & \frac{2}{5} & \frac{2}{5} & \frac{1}{5} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ \frac{2}{3} & 0 & 0 & 0 & 0 & \frac{1}{3} \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad J = \begin{pmatrix} \frac{2}{9} \\ \frac{2}{9} \\ \frac{2}{9} \\ \frac{1}{9} \\ \frac{1}{9} \\ \frac{1}{9} \end{pmatrix}. \tag{15.12}$$

For general Web retrieval there are numerous adjustments we might make to the follow matrix and jump vector to reflect external knowledge. In general, however, we should not make adjustments to reflect properties of the Web graph itself. For example, if many pages link to a given page, we might view that page as having high "popularity" and be tempted to increase its probability in the jump vector. Avoid this temptation. Adjustments of this type are not required or recommended. Focus on external sources when adjusting the follow matrix and jump vector. It is the job of PageRank to account for the properties of the Web graph itself.

Many external sources can be enlisted when setting the follow matrix and jump vector. Although space does not allow us to describe and analyze all possibilities, we list a number of the possible sources below. This list is not exhaustive (see Exercise 15.6) nor is it the case that all (or any) of these suggestions are useful in practice.

- **Page content and structure.** During crawling and indexing, a search engine might analyze the structure and organization of a page as it would appear in a browser. This analysis might then be used to assign probabilities to links based on their layout and appearance. For example, users may be more likely to follow links near the top of the page or in menus. Various aspects of page content might also be interpreted as indicators of quality, thus influencing the jump vector by making the page a likelier target for a jump. Large blocks of readable text might be interpreted positively, while a low ratio of text to HTML tags might be interpreted negatively. Text in tiny and unreadable fonts may indicate an attempt to trick the search engine by misrepresenting what a user would actually see when the page is displayed.
- **Site content and structure.** Users may be more likely to jump to a site's top-level page than to a deeply nested page. Moreover, the longer the URL, the less likely a user will be to remember it or be willing to type it directly into a browser's address bar. The number of pages in a site may also be a factor when setting the jump vector: A site should not receive a high jump probability just because it contains a large number of pages. Depending on the site, users may be more likely to follow a link within a site, as they navigate about, than to follow a link off-site. On the other hand, a link between an educational site and a commercial site may represent a strong recommendation because these links are not usually motivated by commercial considerations. Older, more established, sites may be preferred over newer sites.

- **Explicit judgments.** Human editors might be recruited to manually identify (and classify) high-quality sites, which then receive higher jump probabilities. These editors could be professionals working for the search service itself or volunteers working for a Web directory such as the Open Directory Project (ODP—see Open Directory Project[2]). A major search engine might employ in-house editors to maintain their own Web directories as part of their overall service.
- **Implicit feedback.** A clickthrough on a Web search result might be interpreted as an implicit judgment regarding the quality of that page (see Section 15.5.2). Many of the major search services offer *toolbars*. These browser extensions provide additional features complementing the basic search service. With a user's explicit permission, a toolbar transmits information regarding the user's browsing behavior back to the search service. Pages and sites that the user visits might be recorded and then used to adjust probabilities in the follow matrix and jump vector. Many search services also provide free e-mail accounts. Links sent through these e-mail services might be accessible to these search services and may represent recommendations regarding site and page quality. Of course, any such use of implicit feedback must respect the privacy of users.

Adjustments in the jump vector may also be used to compute special variants of PageRank. To compute a *personalized PageRank* we assign high jump probabilities to pages of personal interest to an individual user (Page et al., 1999). These pages may be extracted from the user's browser bookmarks or her home page, or determined by monitoring her browsing behavior over a period of time. A *topic-oriented PageRank*, or *focused PageRank*, assigns high jump probabilities to pages known to be related to a specified topic, such as sports or business (Haveliwala, 2002). These topic-oriented pages may be taken from a Web directory such as the ODP.

For example, we can select a page from our Wikipedia corpus and generate a topic-specific PageRank focused just on that page. To generate the jump vector we assign a jump probability of 50% to the selected page and assign the remaining 50% uniformly to the other pages. We assign non-zero values to all elements of the jump vector in order to ensure that every page receives a non-zero PageRank value, but the outcome does not change substantially if we assign 100% probability to the selected page.

Table 15.3 shows the top 12 pages for a pair of topics: "William Shakespeare" and "Information Retrieval". The topic-specific PageRank for the topic "Information Retrieval" assigns high ranks to many pages related to that topic: The names Karen Spärck Jones and Gerard Salton should be familiar from Chapters 2 and 8; Rutgers University, City University London, and the University of Glasgow are home to prominent information retrieval research groups; the presence of Google and SIGIR is hardly unexpected. Surprisingly, however, the topic-specific PageRank for the topic "William Shakespeare" appears to be little different from the general

[2] `www.dmoz.org`

Table 15.3 The top 12 pages from a focused PageRank over Wikipedia for two topics.

William Shakespeare		**Information Retrieval**	
Article	PageRank	Article	PageRank
William Shakespeare	303078.44	Information retrieval	305677.03
United States	7200.15	United States	8831.25
England	5357.85	Association for Computing Machinery	6238.30
London	3637.60	Google	5510.16
United Kingdom	3320.49	GNU General Public License	4811.08
2007	3185.71	World Wide Web	4696.78
France	2965.52	SIGIR	4456.67
English language	2714.88	Rutgers University	4389.07
2006	2702.72	Karen Spärck Jones	4282.03
Germany	2490.50	City University, London	4274.76
2005	2377.21	University of Glasgow	4222.44
Canada	2058.84	Gerard Salton	4171.45

PageRank shown in Figure 15.2. Apart from a page on the "English language" and the Shakespeare page itself, all the pages appear in the top twelve of the general PageRank. Despite our topic-specific focus, the page for United States appears second in both lists.

A topic-specific PageRank and a general PageRank both represent probability distributions over the same set of pages, and we may obtain clearer results by comparing these distributions. In Section 9.4 (page 296) we defined the Kullback-Leibler divergence, or KL divergence, between two discrete probability distributions f and g as

$$\sum_{x} f(x) \cdot \log \frac{f(x)}{g(x)} \,. \tag{15.13}$$

Here we define f to be the topic-specific PageRank and g to be the general PageRank, normalized to represent probabilities. For each page α we rerank the page according to its *contribution* to the KL divergence (pointwise KL divergence) between f and g:

$$f(\alpha) \cdot \log \frac{f(\alpha)}{g(\alpha)} \,. \tag{15.14}$$

Recall that KL divergence indicates the average number of extra bits per symbol needed to compress a message if we assume its symbols are distributed according to g instead of the correct distribution f. Equation 15.14 indicates the extra bits due to α. Table 15.4 shows the impact of adjusting a focused PageRank using Equation 15.14.

All the top pages are now on-topic. Andrew Cecil Bradley, a Professor at Oxford University, was famous for his books on Shakespeare's plays and poetry. Ben Jonson and George Wilkins

Table 15.4 Reranking a focused PageRank according to each page's contribution to the KL divergence between the focused PageRank and the general PageRank.

William Shakespeare		**Information Retrieval**	
Article	KL divergence	Article	KL divergence
William Shakespeare	1.505587	Information retrieval	2.390223
First Folio	0.007246	SIGIR	0.027820
Andrew Cecil Bradley	0.007237	Karen Spärck Jones	0.026368
King's Men (playing company)	0.005955	C. J. van Rijsbergen	0.024170
Twelfth Night, or What You Will	0.005939	Gerard Salton	0.024026
Lord Chamberlain's Men	0.005224	Text Retrieval Conference	0.023260
Ben Jonson	0.005095	Cross-language information retrieval	0.022819
Stratford-upon-Avon	0.004927	Relevance (information retrieval)	0.022121
Richard Burbage	0.004794	Assoc. for Computing Machinery	0.022051
George Wilkins	0.004746	Sphinx (search engine)	0.021963
Henry Condell	0.004712	Question answering	0.021773
Shakespeare's reputation	0.004710	Divergence from randomness model	0.021620

were playwrights and contemporaries of Shakespeare. Richard Burbage and Henry Condell were actors and members of the King's Men playing company, for which Shakespeare wrote and acted. On the information retrieval side, Spärck Jones and Salton are joined by C. J. (Keith) van Rijsbergen, another great pioneer of the field. Sphinx is an open-source search engine targeted at relational database systems. The remaining topics should be familiar to readers of this book.

15.3.3 Properties of PageRank

In our presentation of PageRank we have been ignoring several important considerations: Will the fixed-point iteration procedure of Figure 15.7 always converge, or will the algorithm fail to work for some Web graphs? If it does converge, how quickly will it converge? Will it always convergence to the same PageRank vector, regardless of the initial estimate?

Fortunately the PageRank algorithm possesses properties that guarantee good behavior. To simplify our discussion of these properties, we combine the follow matrix and the jump vector into a single *transition matrix*. First we extend the follow matrix to handle sinks. Let F' be the $N \times N$ matrix

$$F'[i,j] \;=\; \begin{cases} J[j] & \text{if } i \text{ is a sink,} \\ F[i,j] & \text{otherwise.} \end{cases} \tag{15.15}$$

Next, let J' be the $N \times N$ matrix with each row equal to the jump vector. Finally, we define the transition matrix as

$$M \;=\; \delta \cdot F' + (1 - \delta) \cdot J' \,. \tag{15.16}$$

For pages i and j, $M[i,j]$ represents the probability that a random surfer on page i will transition to page j without considering if this transition is made through a jump or by following a link. For example, the transition matrix corresponding to F and J in Equation 15.12 is

$$M = \begin{pmatrix} \frac{1}{18} & \frac{16}{45} & \frac{16}{45} & \frac{37}{180} & \frac{1}{18} & \frac{1}{18} \\ \frac{29}{36} & \frac{1}{18} & \frac{1}{18} & \frac{1}{18} & \frac{1}{18} & \frac{1}{18} \\ \frac{5}{9} & \frac{1}{18} & \frac{1}{18} & \frac{1}{18} & \frac{1}{18} & \frac{11}{36} \\ \frac{7}{9} & \frac{1}{36} & \frac{1}{36} & \frac{1}{36} & \frac{1}{36} & \frac{1}{36} \\ \frac{1}{36} & \frac{1}{36} & \frac{1}{36} & \frac{7}{9} & \frac{1}{36} & \frac{1}{36} \\ \frac{7}{36} & \frac{7}{36} & \frac{7}{36} & \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}. \tag{15.17}$$

You may recognize M as a stochastic matrix representing the transition matrix of a Markov chain, as introduced in Section 1.3.4. Each page corresponds to a state; the surfer's initial starting location corresponds to a starting state. Moreover, the PageRank vector R has the property that

$$M^T R = R. \tag{15.18}$$

You may recognize R as being an eigenvector[3] of M^T, with a corresponding eigenvalue of 1. Given an $n \times n$ matrix A, recall that $\vec{x}$ is an *eigenvector* of A and λ is the corresponding *eigenvalue* of A, if $A\vec{x} = \lambda\vec{x}$.

A considerable amount is known about Markov chains and their eigenvectors, and we can turn to this knowledge to determine the properties of PageRank. For simplicity we temporarily assume that all elements of the jump vector are positive, but these properties also hold when elements are 0. We consider jump vectors with 0 elements later in the section.

Underlying PageRank is the idea that as our random surfer surfs for longer periods of time, the relative time spent on each page converges to a value that is independent of the surfer's starting location. This informal notion corresponds to an important property of Markov chains known as *ergodicity.* A Markov chain is *ergodic* if two properties hold, both of which trivially hold for M when J contains only positive values. The first property, *irreducibility*, captures the idea that the random surfer can get from any state to any other state after some finite number of steps. More specifically for any states i and j, if the surfer is currently in state i, there is a positive probability that she can reach state j after taking no more than k steps, where $k \leq N$. If the jump vector contains only positive elements, then $M[i,j] > 0$ for all states i and j, and there is a positive probability that the surfer will reach j from i after $k = 1$ step.

The second property, that the transition matrix be *aperiodic*, eliminates those Markov chains in which the probability of being in a given state cycles between multiple values. A state i is *periodic* with period k if it is possible to return to it only after a number of steps that is a multiple of k. For example, if the surfer can return to a state only after an even number of

[3] If linear algebra is a distant memory, you may safely skip to the start of Section 15.3.4.

steps, then the state is periodic with period 2. If all states in a matrix have period 1, then it is aperiodic. Because $M[i, j] > 0$ for all i, the surfer can reach any state from any other state. Thus, the period of all states is 1 and the property holds.

Let the eigenvectors of M^T be $\vec{x}_1$, $\vec{x}_2$,..., $\vec{x}_N$, with associated eigenvalues λ_1,..., λ_N. Following convention we order these eigenvectors so that $|\lambda_1| \geq |\lambda_2| \geq ... \geq |\lambda_N|$. Because all elements of M^T are positive and each row of M sums to 1, a theorem known as the *Perron-Frobenius theorem* tells us that $\lambda_1 = 1$, that all other eigenvectors satisfy $|\lambda_i| < 1$, and that all elements of $\vec{x}_1$ are positive. Thus, *the principal eigenvector of M^T is the PageRank vector*. Following convention, eigenvectors are scaled to unit length, with $\|\vec{x}_i\| = 1$ for all i, so that $\vec{x}_1 = R/N$.

We now re-express our algorithm for computing PageRank in matrix notation. Let $\vec{x}^{(0)}$ be our initial estimate of PageRank (normalized to have a length of 1). Each iteration of the main loop in Figure 15.7 multiplies M^T by our current estimate to produce a new estimate. The first iteration produces the estimate $\vec{x}^{(1)} = M^T\vec{x}^{(0)}$, the second iteration produces the estimate $\vec{x}^{(2)} = M^T\vec{x}^{(1)}$, and so on. After n iterations we have the estimate

$$\vec{x}^{(n)} \;=\; \left(M^T\right)^n \vec{x}^{(0)}. \tag{15.19}$$

Thus, the algorithm computes PageRank if

$$\lim_{n\to\infty} \vec{x}^{(n)} \;=\; \vec{x}_1. \tag{15.20}$$

The ergodicity of M guarantees this convergence. In the terminology of Markov chains, $\vec{x}_1$ is called the *stationary distribution* of M. This algorithm for computing the principal eigenvector of a matrix is called the *power method* (Golub and Van Loan, 1996).

To determine the rate of convergence, suppose the initial estimate $\vec{x}^{(0)}$ is expressed as a linear combination of the (unknown) eigenvectors.

$$\vec{x}^{(0)} \;=\; \vec{x}_1 + a_2\,\vec{x}_2 + a_3\,\vec{x}_3 + \cdots + a_N\,\vec{x}_N \tag{15.21}$$

After the first iteration we have

$$\begin{aligned}
\vec{x}^{(1)} &= M^T\vec{x}^{(0)} \\
&= M^T\left(\vec{x}_1 + a_2\,\vec{x}_2 + \cdots + a_N\,\vec{x}_N\right) \\
&= M^T\vec{x}_1 + a_2\,M^T\vec{x}_2 + \cdots + a_N\,M^T\vec{x}_N \\
&= \vec{x}_1 + a_2\,\lambda_2\,\vec{x}_2 + \cdots + a_N\,\lambda_N\,\vec{x}_N.
\end{aligned}$$

After n iterations, and recalling that λ_2 is the second-largest eigenvalue after λ_1, we have

$$\begin{aligned} \vec{x}^{(n)} &= \vec{x}_1 + a_2\,\lambda_2^n\,\vec{x}_2 + \cdots + a_N\,\lambda_N^n\,\vec{x}_N \\ &\leq \vec{x}_1 + \lambda_2^n\,(a_2\,\vec{x}_2 + \cdots + a_N\,\vec{x}_N) \\ &= \vec{x}_1 + O(\lambda_2^n). \end{aligned}$$

Thus, the rate of convergence depends on the value of the second eigenvalue of M^T. The smaller the value, the faster the convergence.

For PageRank, Haveliwala and Kamvar (2003) proved the remarkable result that the value of the second eigenvector is δ, the damping factor.[4] Provided that δ is not too close to 1, convergence should be acceptably fast. More important, the rate of convergence does not depend on characteristics of the Web graph. If convergence is acceptably fast for one Web graph, it should remain acceptably fast as the Web graph evolves over time.

Haveliwala and Kamvar note that the stability of PageRank is also associated with δ. With smaller values of δ, PageRank becomes less sensitive to small changes in the Web graph. Of course, if the value of δ is close to 0, PageRank may fail to capture any useful information regarding the Web graph because a random jump will happen at nearly every step. The traditional value of $\delta = 0.85$ appears to be a reasonable compromise, giving good stability and convergence properties while allowing PageRank to do its job.

Thus, it is the random jumps of the imaginary surfer that determines the convergence and stability of PageRank. The jump vector guarantees the ergodicity of M, and therefore the convergence of the PageRank algorithm. The damping factor determines stability and the rate of convergence. The structure of the Web graph itself does not play a major role.

The above discussion assumes that all elements of the jump vector are positive. If this assumption does not hold, and the jump vector contains a zero, M may not be irreducible. An element with value zero in the jump vector indicates a page to which the surfer will never jump at random. Suppose there is a page α that cannot be reached by following links from any of the pages with a non-zero jump probability. Once the surfer makes a random jump after leaving α, she can never return to it.

As the length of the surf goes to infinity, the probability that the surfer eventually makes a jump goes to one. After making a jump the probability of visiting α is zero. Thus, we can set the PageRank value for α to zero, without explicitly computing it.

Let Φ' be the set of pages reachable after a jump (Figure 15.8). The pages outside Φ' have a PageRank value of zero, and do not need to be involved in the PageRank computation for the pages in Φ'. Let M' be the transition matrix corresponding to Φ' and let J' be its jump vector. J' may still contain elements with value zero, but the pages associated with these elements will be reachable from pages with non-zero values. Because a jump is possible at any step, and any

[4] More accurately, $|\lambda_2| \leq \delta$, but $\lambda_2 = \delta$ for any realistic Web graph.

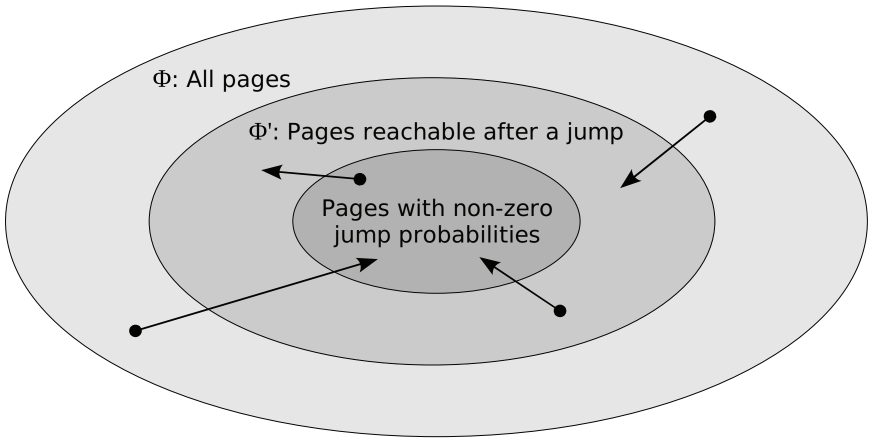

Φ: All pages
Φ': Pages reachable after a jump
Pages with non-zero jump probabilities

Figure 15.8 Pages reachable after a jump.

page in Φ' may be reached after a jump, M' is irreducible. The possibility of a jump at any step also helps to guarantee that M' is aperiodic (Exercise 15.5). Because M' is irreducible and aperiodic, it is ergodic and PageRank will converge. The other stability and convergence properties can also be shown to hold for M'.

15.3.4 Other Link Analysis Methods: HITS and SALSA

In addition to PageRank a number of other link analysis techniques have been proposed in the context of the Web. Two of these methods are Kleinberg's HITS algorithm (Kleinberg, 1998, 1999) and the related SALSA algorithm due to Lempel and Moran (2000). HITS was developed independently of PageRank during roughly the same time frame. SALSA was developed shortly after, in an attempt to combine features from both algorithms.

The intuition underlying HITS derives from a consideration of the roles a page may play on the Web with respect to a given topic. One role is that of an *authority*, a page that contains substantial and reliable information on the topic. A second role is that of a *hub*, a page that aggregates links to pages related to the topic. A good hub points to many authorities; a good authority is referenced by many hubs. We emphasize that these are idealized roles; a page may be both a hub and an authority to some extent.

In contrast to PageRank, which was envisioned as operating over the entire Web, Kleinberg envisioned HITS as operating over smaller Web graphs. These graphs may be generated by executing a query on a search engine to retrieve the contents of the top few hundred pages, along with pages in their immediate neighborhood. Based on the retrieved pages, HITS computes hubs and authorities with respect to the query. At the time HITS was invented, Web search engines were not known to incorporate link analysis techniques (Marchiori, 1997), and HITS provided a way of obtaining the benefit of these techniques without explicit support from the engine.

For a given page α, HITS computes two values: an authority value $a(\alpha)$ and a hub value $h(\alpha)$. The authority value of a page is derived from the hub values of the pages that link to it:

$$a(\alpha) \;=\; w_a \cdot \sum_{\beta \to \alpha} h(\beta). \tag{15.22}$$

The hub value of a page is derived from the authority values of the pages it references.

$$h(\alpha) \;=\; w_h \cdot \sum_{\alpha \to \beta} h(\beta). \tag{15.23}$$

The significance of the weights w_a and w_h is discussed shortly. Links from a page to itself (*self-loops*) and links within sites are ignored because these are assumed to represent navigational relationships rather than hub-authority relationships.

We may express Equations 15.22 and 15.23 in matrix/vector notation by defining $\vec{a}$ to be the authority values and $\vec{h}$ to be the hub values for the N pages in the Web graph.[5]. Let W be the *adjacency matrix* of the Web graph, where $W[i,j] = 1$ if there is a link from the ith page to the jth page, and $W[i,j] = 0$ otherwise. For example, the adjacency matrix for the Web graph in Figure 15.2 (retaining links within sites) is

$$W \;=\; \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{15.24}$$

In matrix/vector notation, Equations 15.22 and 15.23 become

$$\vec{a} \;=\; w_a \cdot W^T \vec{h} \quad \text{and} \quad \vec{h} \;=\; w_h \cdot W \vec{a}\,. \tag{15.25}$$

Substituting gives

$$\vec{a} \;=\; w_a \cdot w_h \cdot W^T W \vec{a} \quad \text{and} \quad \vec{h} \;=\; w_a \cdot w_h \cdot W W^T \vec{h}\,. \tag{15.26}$$

If we define $A = W^T W$, $H = W W^T$, and $\lambda = 1/(w_a \cdot w_h)$, we have

$$\lambda \vec{a} \;=\; A \vec{a} \quad \text{and} \quad \lambda \vec{h} \;=\; H \vec{h}\,. \tag{15.27}$$

Thus, the authority vector $\vec{a}$ is an eigenvector of A and the hub vector $\vec{h}$ is an eigenvector of H.

[5] If you wish to avoid the details, you may safely skip to Section 15.3.5.

The matrices A and H have interesting interpretations taken from the field of bibliometrics (Lempel and Moran, 2000; Langville and Meyer, 2005): A is the *co-citation matrix* for the Web graph, where $A[i, j]$ is the number of pages that link to both i and j; H is the *co-reference* or *coupling matrix* for the Web graph, where $H[i, j]$ is the number of pages referenced by both i and j. Both A and H are symmetric, a property of importance for the computation of $\vec{a}$ and $\vec{h}$.

As we did for PageRank, we may apply fixed-point iteration — in the form of the power method — to compute $\vec{a}$ and $\vec{h}$. Let $\vec{a}^{(0)}$ and $\vec{h}^{(0)}$ be initial estimates for $\vec{a}$ and $\vec{h}$. The nth estimate, $\vec{a}^{(n)}$ and $\vec{h}^{(n)}$, may be computed from the previous estimate by the equations

$$\vec{a}^{(n)} \;=\; W^T\vec{h}^{(n-1)}/\|W^T\vec{h}^{(n-1)}\| \quad \text{and} \quad \vec{h}^{(n)} \;=\; W\vec{a}^{(n-1)}/\|W\vec{a}^{(n-1)}\|\,. \tag{15.28}$$

Normalization allows us to avoid explicit computation of the eigenvalue by ensuring that each estimate has unit length. Successive application of these equations will compute $\vec{a}$ and $\vec{h}$ if

$$\lim_{n\to\infty} \vec{a}^{(n)} = \vec{a} \quad \text{and} \quad \lim_{n\to\infty} \vec{h}^{(n)} = \vec{h}\,. \tag{15.29}$$

For PageRank the ergodicity of M guarantees convergence. For HITS the symmetric property of A and H provides this guarantee, as long as the initial estimates have a component in the direction of the principal eigenvector (Kleinberg, 1999; Golub and Van Loan, 1996). Any unit vector with all positive elements will suffice to satisfy this requirement, such as

$$\vec{a}^{(0)} \;=\; \vec{h}^{(0)} \;=\; \langle\, 1/\sqrt{N},\, 1/\sqrt{N}, \ldots \rangle\,.$$

Unfortunately, convergence to a unique solution is not guaranteed (Langville and Meyer, 2005). Depending on the initial estimate, HITS may convergence to different solutions (see Exercise 15.9).

SALSA, the *stochastic approach for link-structure analysis*, introduces PageRank's random surfer into HITS (Lempel and Moran, 2000). The creation of SALSA was motivated by the observation that a small set of highly interconnected sites (or "tightly connected communities") can have a disproportionate influence on hub and authority values, by generating inappropriately high values for members of these communities.

SALSA imagines a random surfer following both links and *backlinks* (i.e., reversed links), alternating between them. On odd-numbered steps the surfer chooses an outgoing link from the current page uniformly at random and follows it. On even-numbered steps the surfer chooses an incoming link uniformly at random, from a page linking to the current page, and follows it backwards to its source. As the number of steps increases, the relative time spent on a page just before taking an odd-numbered step represents the page's hub score; the relative time spent on a page just before taking an even-numbered step represents the page's authority score.

Although we do not provide the details, SALSA may be formulated as an eigenvector problem. Lempel and Moran (2000) discuss properties that simplify the computation of SALSA. Rafiei and Mendelzon (2000) propose the introduction of random jumps into a SALSA-like algorithm, with the possibility for a random jump occurring after each odd-even pair of steps.

15.3.5 Other Static Ranking Methods

Static ranking provides a query-independent score for each Web page. Although link analysis is an essential component in the computation of static rank, other features may contribute.

As we mentioned with respect to link analysis in Section 15.3.2, some of the most important features are provided through *implicit user feedback*. By "implicit" we mean that the user provides this feedback while engaged in other activities, perhaps without being aware that she is providing feedback. For example, a click on a Web search result may indicate a preference for that site or page. If higher-ranked results were skipped, the click may also indicate a negative assessment of these skipped pages (Joachims et al., 2005). The toolbars provided by commercial search services track the sites visited by a user and return this information to the search service (with the user's permission). The popularity of a site may be measured through the number of users visiting it, and through the frequency and length of their visits (Richardson et al., 2006).

The content of the pages themselves may also contribute to their static rank. Ivory and Hearst (2002) describe how quantitative measures of page content and structure predict the quality scores assigned by expert assessors. These measures consider the quantity and complexity of text, the placement and formatting of graphical elements, and the choice of fonts and colors.

Finally, the content and structure of a URL may be considered. Short and simple URLs might be favored over long and complex ones, particularly for navigational queries (Upstill et al., 2003). Sites in the `com` domain might be more appropriate for commercial queries than sites in the `edu` domain; the opposite is true for academic queries.

Richardson et al. (2006) apply machine learning techniques, similar to those described in Section 11.7, to the computation of static rank. By combining popularity, page content, and URL features with basic PageRank, they demonstrate substantial improvements over basic PageRank alone. In Section 15.3.2 we discussed how these same features might be applied to adjust the follow and jump vectors in extended PageRank. The best methods for exploiting these features in the computation of static rank remains an area open for exploration.

15.4 Dynamic Ranking

At query time the search engine combines each page's static rank with query-dependent features — such as term frequency and proximity — to generate a dynamic ranking for that query. Although the dynamic ranking algorithms used in commercial search engines are grounded in the theory of Part III, the details of these algorithms vary greatly from one Web search engine to another. Moreover, these algorithms evolve continuously, reflecting the experience gained from the tremendous numbers of queries and users these search engines serve.

Although the scope of this book does not allow us to provide the details of the dynamic ranking algorithms used in specific Web search engines, two aspects of dynamic ranking deserve some attention. The first of these aspects — the availability of anchor text — represents a ranking feature of particular importance in Web search. Anchor text often provides a description of

Table 15.5 Anchor text for in-links to the Wikipedia page `en.wikipedia.org/wiki/William_Shakespeare`, ordered by number of appearances.

#	Anchor Text	#	Anchor Text
3123	Shakespeare	2	Will
3008	William Shakespeare	2	Shakesperean
343	Shakespeare's	2	Shakespere
210	Shakespearean	2	Shakespearean studies
58	William Shakespeare's	2	Shakepeare
52	Shakespearian	2	Bill Shakespeare...
10	W. Shakespeare	1	the Bard's
7	Shakespeare, William	1	lost play of Shakespeare
3	William Shakepeare	1	Shakespearesque
3	Shakesphere	1	Shakespearean theatre
3	Shakespeare's Theatre	1	Shakespearean plays
3	Bard		...
2	the Bard		...

the page referenced by its link, which may be exploited to improve retrieval effectiveness. The importance of the second aspect — novelty — derives from the scale and structure of the Web. Although many pages on a given site may be relevant to a query, it may be better to return one or two results from several sites rather than many pages from a single site, thereby providing more diverse information.

15.4.1 Anchor Text

Anchor text often provides a label or description for the target of its link. Table 15.5 lists the anchor text linking to the page `en.wikipedia.org/wiki/William_Shakespeare` from other pages within Wikipedia, ordered by the number of times each string appears. Pages outside of Wikipedia may also link to this page, but that anchor text can be discovered only through a crawl of the general Web. The table shows a mixture of text, including misspellings and nicknames ("the Bard"). In all, 71 different strings are used in 6889 anchors linking to the page. However, more than 45% of these anchors use the text "Shakespeare", and nearly as many use the text "William Shakespeare".

On the general Web the number of links to a page may vary from one to many millions. When a page is linked many times, the anchor text often repeats, and when it repeats, it usually represents an accurate description of the page (but see Exercise 15.12). Misspellings, nicknames, and similar anchor text can also prove valuable because these terms may not appear on the page, but may be entered by a user as part of a query.

For anchor text to be usable as a ranking feature, it must be associated with the target of the link. Before building a Web index, anchor text must be extracted from each page to create a set of tuples of the form

$$\langle\, \textit{target URL}, \textit{anchor text} \rangle\,.$$

These tuples are sorted by URL. The anchor text for each URL is then merged with the page contents for that URL to form a composite document for indexing purposes.

For retrieval, anchor text may be treated as a document field, much as we did for titles and other fields in Section 8.7. When computing term weights, anchor text may be weighted in the same way as other text, or term frequencies may be adjusted to dampen the influence of repeated terms (Hawking et al., 2004). The static rank of the page on which the anchor appears may also play a role in term weights (Robertson et al., 2004). Anchor text appearing on pages with high static rank may be given greater weight than text appearing on pages with low static rank.

15.4.2 Novelty

Given the variety and volume of information on the Web, it is important for a search engine to present a diverse set of results to the user. Given our example query ⟨"UPS"⟩, a search engine might best return a mix of results related to the parcel service, to power supplies, and to the university. Simple ways to reduce redundancy include the identification of duplicate pages in the index and post-retrieval filtering to eliminate excessive results from a single Web site.

Although the Web contains many pages that are duplicates (or near-duplicates) of others, the result of a search should usually contain only a single copy. These duplicated pages occur on the Web for several reasons. Many sites return a default page when a URL path is invalid, and all invalid URLs from these sites appear to be duplicates of this default page. Certain types of content are frequently mirrored across multiple sites. For example, a newswire article might appear on the Web sites of multiple newspapers. These duplicates may be detected and flagged by the Web crawler during the crawl (Section 15.6.3), and static rank may be used to choose the best copy at query time.

In general, duplicated pages within a given site should be eliminated from the index. However, it may be reasonable to retain duplicates when they appear on different sites. Most commercial Web search engines allow searches to be restricted to a specified site or domain. For example, including the term `site:wikipedia.org` in a query would restrict a search to Wikipedia. Retaining pages duplicated on other sites allows these searches to be handled correctly.

Another method for improving diversity in search results is to apply post-retrieval filtering. After dynamic ranking, most commercial search engines filter the results to reduce redundancy.

For example, the documentation for the (now-deprecated) Google SOAP API[6] describes one simple post-retrieval filtering algorithm. After retrieval, results may be filtered in two ways:

1. If several results have identical titles and snippets, then only one is retained.
2. Only the two best results from a given Web site are retained.

The API calls this second filter "host crowding". Interestingly, the first filter may eliminate relevant results that are not in fact redundant in order to reduce the *appearance* of redundancy in the search results.

15.5 Evaluating Web Search

In principle it is possible to apply the traditional IR evaluation framework (e.g., P@10 and MAP) to Web search. However, the volume of material available on the Web introduces some problems. For example, an informational query could have hundreds or thousands of relevant documents. It is not uncommon that all of the top ten results returned for such a query are relevant. In that situation the binary relevance assessments ("relevant"/"not relevant") underlying the traditional methodology might not be a sufficient basis for a meaningful evaluation; graded relevance assessments may be more appropriate.

When graded relevance values are available, evaluation measures such as nDCG (Section 12.5.1) may be applied to take advantage of them (Richardson et al., 2006). For example, Najork (2007) uses nDCG in a Web search evaluation based on more than 28,000 queries taken from the Windows Live search engine. Nearly half a million manual judgments were obtained for these queries. These judgments were made on a six-point scale: definitive, excellent, good, fair, bad, and detrimental.

The nature of the Web introduces several novel aspects of evaluation. As we discussed in Section 15.2, many Web queries are navigational. For these queries only a single specific page may be relevant. In Section 15.5.1 we present an evaluation methodology addressing this query type. In addition, the volume of queries and users handled by commercial Web search services provides an opportunity to infer relevance judgments from user behavior, which may be used to augment or replace manual judgments. In Section 15.5.2 we provide an overview of techniques for interpreting clickthroughs and similar user actions for evaluation purposes.

15.5.1 Named Page Finding

A *named page finding* task is a Web evaluation task that imagines a user searching for a specific page, perhaps because she has seen it in the past or otherwise learned about it. The user enters a query describing the content of the page and expects to receive it as the first (or only) search

[6] `code.google.com/apis/soapsearch/reference.html` (accessed Dec 23, 2009)

Table 15.6 Named page finding results for selected retrieval methods from Part III. The best run from TREC 2006 (Metzler et al., 2006) is included for comparison.

Method	**MRR**	**% Top 10**	**% Not Found**
BM25 (Ch. 8)	0.348	50.8	16.0
BM25F (Ch. 8)	0.421	58.6	16.6
LMD (Ch. 9)	0.298	48.1	15.5
DFR (Ch. 9)	0.306	45.9	19.9
Metzler et al. (2006)	0.512	69.6	13.8

result. For example, the query ⟨“Apollo”, “11”, “mission”⟩ might be used to describe NASA’s history page on the first moon landing. Although other pages might be related to this topic, only that page would be considered correct. Named page finding tasks were included in TREC as part of the Web Track from 2002 to 2004 and in the Terabyte Track in 2005 and 2006 (Hawking and Craswell, 2001; Craswell and Hawking, 2004; Clarke et al., 2005; Büttcher et al., 2006).

The premise behind named page finding represents an important subset of navigational queries (see page 515). Nonetheless, many navigational queries (e.g., ⟨“UPS”⟩) are seeking specific sites rather than specific content. To address this issue the scope of named page finding may be expanded to include “home page finding” queries (Craswell and Hawking, 2004).

For the 2006 Terabyte Track, 181 topics were created by track participants. For each topic the creator specified the answer page within the GOV2 collection. Each submitted run consisted of up to 1000 results. During the evaluation of the runs, near-duplicates of answer pages were detected using an implementation of Bernstein and Zobel’s (2005) DECO algorithm, a variant of the near-duplicate detection algorithm from Section 15.6.3. All near-duplicates of a correct answer page were also considered correct. Three measures were used in the evaluation:

- **MRR**: The *mean reciprocal rank* of the first correct answer.
- **% Top 10**: The proportion of queries for which a correct answer was found in the top ten search results.
- **% Not Found**: The proportion of queries for which no correct answer was found in the top 1000 search results.

For a given topic, reciprocal rank is the inverse of the rank at which the answer first appears. If the answer is the top result, its reciprocal rank is 1; if the answer is the fifth result, its reciprocal rank is 1/5. If the answer does not appear in the result list, it may be assigned a reciprocal rank of $1/\infty = 0$. Mean reciprocal rank is the average of reciprocal rank across all topics.

Full results for the track are provided by Büttcher et al. (2006). Table 15.6 shows the results of applying the retrieval methods from Part III to this task. For this task, BM25F improves upon BM25. For comparison we include the best result from the track itself (Metzler et al., 2006). This run incorporated a number of Web-specific techniques, such as link analysis (static ranking) and anchor text. The gap between the standard retrieval methods from Part III and Metzler’s run illustrates the importance of these techniques for navigational search tasks.

15.5.2 Implicit User Feedback

Implicit feedback is provided by users as a side effect of their interaction with a search engine. Clickthroughs provide one important and readily available source of this implicit feedback. Whenever a user enters a query and clicks on the link to a result, a record of that clickthrough may be sent by the browser to the search engine, where it is logged for later analysis.

For a given query the clickthroughs from many users may be combined into a *clickthrough curve* showing the pattern of clicks for that query. The plots in Figure 15.5 provide examples of stereotypical clickthrough curves for navigational and informational queries. In both plots the number of clickthroughs decreases with rank. If we interpret a clickthrough as a positive preference on the part of the user, the shapes of these curves are just as we would expect: Higher-ranked results are more likely to be relevant and to receive more clicks. Even when successive results have the same relevance, we expect the higher ranked result to receive more clicks. If the user scans the results in order, higher-ranked results will be seen before lower-ranked ones. There is also a *trust bias* (Joachims et al., 2005) in user behavior: Users expect search engines to return the best results first, and therefore tend to click on highly ranked results even though they might not contain the information sought.

Figure 15.9 presents a third clickthrough curve, taken from the same paper as the previous two (Clarke et al., 2007). This figure plots clickthroughs for the informational/transactional query ⟨“kids”, “online”, “games”⟩. The plot includes a number of *clickthrough inversions*, in which a particular result received fewer clicks than the result ranked immediately below it (e.g., the result at rank 2 versus rank 3, or rank 7 versus rank 8).

Clickthrough inversions may indicate a suboptimal ranking, in which a less relevant document is ranked above a more relevant one. However, they may also arise for reasons unrelated to relevance. For example, the title and snippet of the higher-ranked result may not accurately describe the underlying page (Clarke et al., 2007; Dupret et al., 2007). If a user does not understand how or why a result is relevant after reading its title and snippet, she may be inclined to ignore it and to move on to other results. Nonetheless, when the titles and snippets provide accurate descriptions of results, a clickthrough inversion may be interpreted as a pairwise preference for the lower-ranked result (Joachims and Radlinski, 2007).

Query *abandonment* — issuing a query but not clicking on any result — suggests user dissatisfaction of a more substantial nature (Joachims and Radlinski, 2007). When many users abandon a given query, this behavior may indicate that none of the top results are relevant. In some cases an abandonment is followed by a query reformulation in which the user adds, removes, or corrects terms. A search service may capture these query reformulations by tracking search activity through browser cookies and similar mechanisms.[7]

By examining query reformulations from large numbers of users, it is possible to correct spelling errors, to suggest expansion terms, and to identify acronyms (Cucerzan and Brill, 2004; Jones et al., 2006). For example, a user entering the query ⟨“brittany”, “spears”⟩ may correct

[7] `www.w3.org/Protocols/rfc2109/rfc2109`

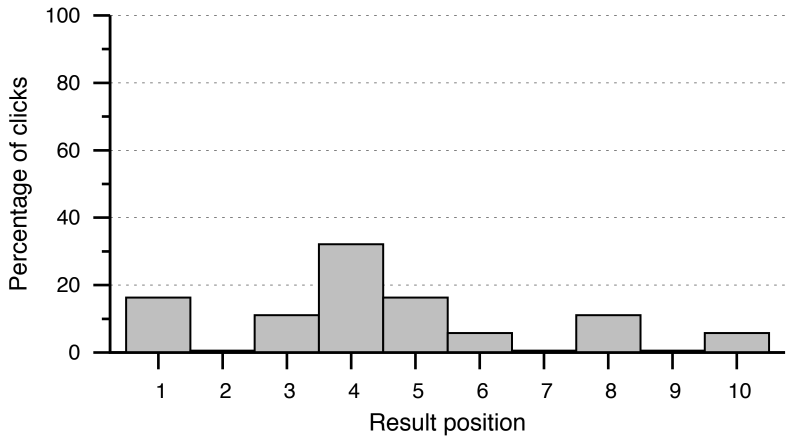


Figure 15.9 Clickthrough curve for the query ⟨“kids”, “online”, “games”⟩. Inversions (e.g., rank 2 versus rank 3) indicate a suboptimal ranking of search results.

it to ⟨“britney”, “spears”⟩.[8] A click after a reformulation may indicate a page that is relevant to the original query. For example, Joachims and Radlinski (2007) report that users frequently reformulate ⟨“oed”⟩ to ⟨“oxford”, “english”, “dictionary”⟩ and then click on the first result.

Capturing other forms of implicit feedback requires the cooperation of the user, who must install a browser toolbar or similar application, and give explicit permission to capture this feedback (Kellar et al., 2007). Page *dwell time* provides one example of feedback that may be captured in this way. If a user clicks on a result and then immediately backtracks to the result page, this action may indicate that the result is not relevant.

Taken together, these and other sources of implicit feedback can lead to substantial improvements in the quality of Web search results, both by facilitating evaluation (Agichtein et al., 2006b) and by providing additional ranking features (Agichtein et al., 2006a).

15.6 Web Crawlers

The functioning of a Web crawler is similar to that of a user surfing the Web, but on a much larger scale. Just as a user follows links from page to page, a crawler successively downloads pages, extracts links from them, and then repeats with the new links. Many of the challenges associated with the development of a Web crawler relate to the scale and speed at which this process must take place.

[8] `labs.google.com/britney.html` (accessed Dec 23, 2009)

Suppose our goal is to download an 8-billion-page snapshot of the Web over the course of a week (a small snapshot by the standards of a commercial Web search engine). If we assume an average page is 64 KB in size, we must download data at a steady rate of

$$\begin{aligned} 64 \text{ KB/page} \cdot 8 \text{ giga-pages/week} &= 512 \text{ TB/week} \\ &= 888 \text{ MB/second} \end{aligned}$$

To achieve this tremendous download rate, the crawler must download pages from multiple sites concurrently because it may take an individual site several seconds to respond to a single download request. As it downloads pages, the crawler must track its progress, avoiding pages it has already downloaded and retrying pages when a download attempt fails.

The crawler must take care that its activities do not interfere with the normal operation of the sites it visits. Simultaneous downloads should be distributed across the Web, and visits to a single site must be carefully spaced to avoid overload. The crawler must also respect the wishes of the sites it visits by following established conventions that may exclude the crawler from certain pages and links.

Many Web pages are updated regularly. After a period of time the crawler must return to capture new versions of these pages, or the search engine's index will become stale. To avoid this problem we might recrawl the entire Web each week, but this schedule wastes resources and does not match the practical requirements of Web search. Some pages are updated regularly with "popular" or "important" information and should be recrawled more frequently, perhaps daily or hourly. Other pages rarely change or contain information of minimal value, and it may be sufficient to revisit these pages less frequently, perhaps biweekly or monthly.

To manage the order and frequency of crawling and recrawling, the crawler must maintain a priority queue of seen URLs. The ordering of URLs in the priority queue should reflect a combination of factors including their relative importance and update frequency.

15.6.1 Components of a Crawler

In this section we examine the individual steps and components of a crawler by tracing the journey of a single URL (`en.wikipedia.org/wiki/William_Shakespeare`) as it travels through the crawling process. Tracing a single URL allows us to simplify our presentation, but we emphasize that crawling at the rates necessary to support a Web search engine requires these steps to take place concurrently for large numbers of URLs.

Figure 15.10 provides an overview of these steps and components. The crawling process begins when a URL is removed from the front of the priority queue and ends when it is returned to the queue along with other URLs extracted from the visited page. Although specific implementation details vary from crawler to crawler, all crawlers will include these steps in one form or another. For example, a single thread may execute all steps for a single URL, with thousands of such threads executing concurrently. Alternatively, URLs may be grouped into large batches, with

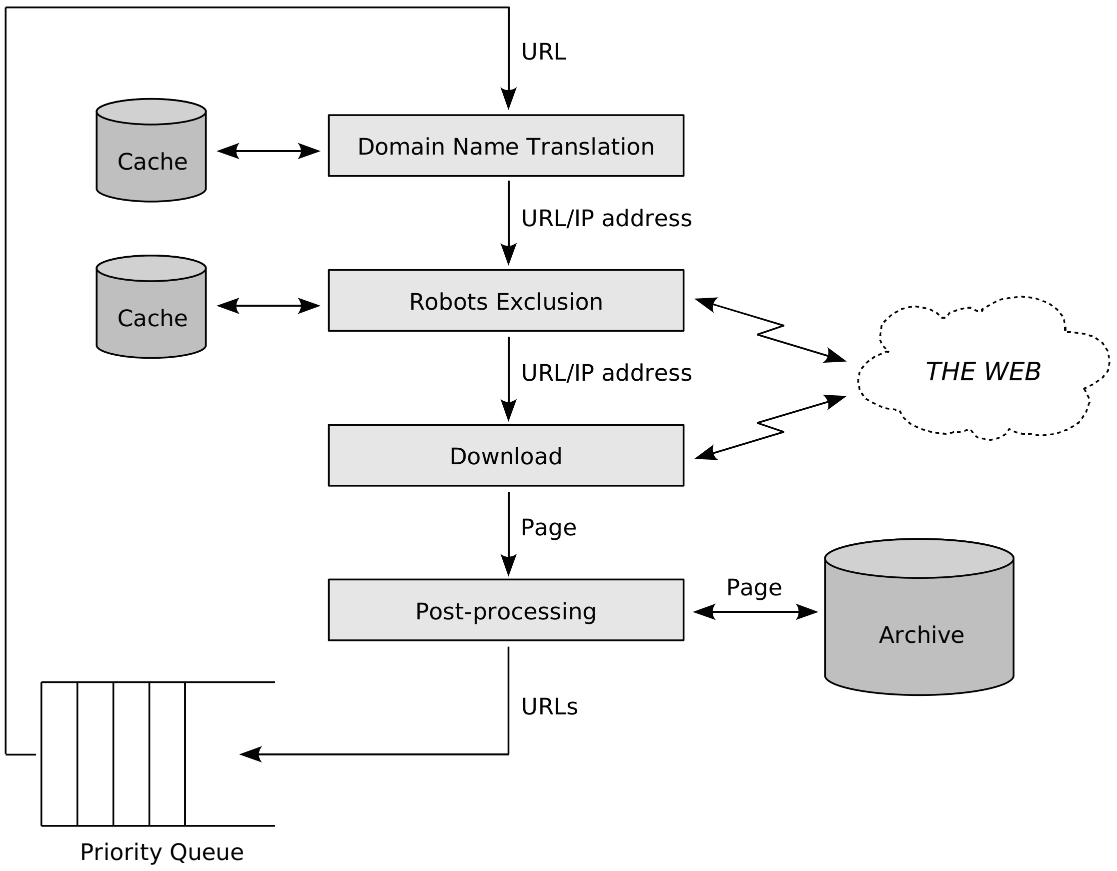


Figure 15.10 Components of a Web crawler.

each step executed for all URLs in the batch before the crawler moves on to execute the next step for the same batch.

Like the search engine it supports (Section 14.1), the activities of a large-scale Web crawler must be distributed across multiple machines in order to sustain the necessary download rates. This distribution may be achieved by assigning subsets of URLs to each machine, essentially giving each machine responsibility for a portion of the Web. The construction of these subsets may be based on host name, IP address, or other factors (Chung and Clarke, 2002). For example, we might assign each machine responsibility for certain hosts. The priority queue might be centralized at a single machine that would distribute URLs to other machines for crawling, or each machine might implement its own priority queue for its assigned subset.

Domain name translation

The processing of a URL begins by translating its host name into a 32-bit IP address. For the host `en.wikipedia.org` the corresponding IP address is `208.80.152.2` (at time and place of writing). This address will be downloaded to contact the machine hosting the page.

As it is for browsers and other Internet applications, this translation is effected through the Domain Name System (DNS), a standard Internet service that is implemented by a distributed hierarchy of servers and caches spread across the Internet. Although the speed of DNS is acceptable for most applications, its speed may be insufficient to satisfy the demands of a Web crawler, which may require thousands of translations per second. To satisfy this translation rate, the crawler may need to maintain its own translation cache. This cache would maintain mappings between host names and IP addresses, expiring and refreshing these entries as they grow stale. Although the inclusion of a custom DNS cache represents a minor part of a crawler, the need for this cache illustrates the problems encountered when crawling at high speeds.

Robots exclusion protocol

With the IP address available the crawler must check that access to the page is permitted by the Web site. This permission is determined through an unofficial (but well-established) convention known as the *robots exclusion protocol* (more informally, the "`robots.txt` protocol"). For a given site, access permissions are obtained by appending the path `"/robots.txt"` to the host name and downloading the corresponding page.

In our example the URL is `http://en.wikipedia.org/robots.txt`. Figure 15.11 shows a small portion of the corresponding page. The page is structured as a series of comments and instructions. A "`User-agent`" instruction begins a list that applies to the crawler named on the line, with a "`*`" indicating that the instructions apply to all crawlers. For example, a "`Disallow`" instruction indicates a path prefix to which access is disallowed. Downloading any page having a path starting with a disallowed prefix is not permitted. The example in the figure denies all access to the `wget` program, and requests other crawlers to avoid the random article link and the search link, both of which generate content dynamically. Full details on the robots exclusion protocol may be found on the Web Robots site.[9]

A Web crawler will generally cache the `robots.txt` page for each host it encounters in order to avoid repeated downloads whenever a URL from that host is processed. This cached information should be expired on a regular basis, perhaps after a few hours or days.

When access by the crawler is not permitted, the URL may be returned to the priority queue and labeled to indicate that download was disallowed. The crawler may retry the URL at a future time, checking first that access has been permitted, or the URL may be permanently flagged as disallowed, never to be retried but maintained as a placeholder to prevent further attempts.

[9] `www.robotstxt.org`

```
#
# robots.txt for http://www.wikipedia.org/ and friends
#
...

#
# Sorry, wget in its recursive mode is a frequent problem.
#
User-agent: wget
Disallow: /
...

#
# Friendly, low-speed bots are welcome viewing article pages, but not
# dynamically-generated pages please.
#
User-agent: *
Disallow: /wiki/Special:Random
Disallow: /wiki/Special:Search
...
```

Figure 15.11 Extracts from a `robots.txt` file.

Download

After confirming that download is permitted, the crawler accesses the Web site via the HTTP protocol and downloads the page. The page may be formatted as HTML (e.g., Figure 8.1 on page 278) or in other formats, such as PDF. Associated pages may be identified and downloaded at the same time, or they may be left for a future crawling cycle. For example, if the page defines a frame set, all pages in the set might be downloaded together. Images may also be downloaded, perhaps in support of an image search service.

During download the crawler may be required to resolve redirections, which indicate that the actual content of the page is found elsewhere. These redirections can add considerable complexity to the download process, in part because redirections may be implemented in multiple ways. At the HTTP protocol level various responses generated by the Web server may indicate that a page has moved permanently or temporarily to a new location. Redirections may also be specified by tags appearing in HTML pages. Finally, JavaScript loaded with HTML pages may generate redirections during execution.

For example, Wikipedia uses an HTTP "301 Moved Permanently" response to redirect the URL `en.wikipedia.org/wiki/william_shakespeare` to `en.wikipedia.org/wiki/William_Shakespeare`. Redirections to correct spelling and capitalization are common in Wikipedia.

Other sites may use HTTP redirections when sites are redesigned or restructured, so that older URLs continue to work.

The use of JavaScript to generate redirections causes the most trouble for Web crawlers because correct determination of the redirection's target potentially requires the crawler to execute the JavaScript. Moreover, this JavaScript may redirect to different pages depending on factors such as the user's browser type. Covering all possibilities theoretically requires the crawler to repeatedly execute the JavaScript under different configurations until all possible redirection targets are determined.

Given resource limitations, it may not be feasible for a crawler to execute JavaScript for millions of downloaded pages. Instead the crawler may attempt partial evaluation or other heuristics, which may be sufficient to determine redirection targets in many cases. On the other hand, if the crawler ignores JavaScript redirections, it may be left with little useful content, perhaps no more than a message suggesting that JavaScript be enabled.

Post-processing

After download, the crawler stores the page in an archive for indexing by the search engine. Older copies of the page may be retained in this archive, thus allowing the crawler to estimate the frequency and types of changes the page undergoes. If the download fails, perhaps because the site is temporarily inaccessible, these older copies remain available for indexing. After a failed download the URL may be returned to the priority queue and retried at a later time. If several download attempts fail over a period of days, the older copies may be allowed to expire, thus removing the page from the index.

In addition to being stored in the archive, the page is analyzed, or *scraped*, to extract any URLs it contains. Much like a browser, the crawler parses the HTML to locate anchor tags and other elements containing links. Anchor text and other information required for indexing may be extracted at the same time, then stored in the archive for the convenience of the search engine. Pages that are duplicates (or near-duplicates) of the downloaded page may also be identified during this analysis (see Section 15.6.3).

During post-processing, the crawler must respect conventions requesting that it not index a page or follow certain links. If the tag

```
<meta name="robots" content="noindex">
```

appears in the header of page, it indicates that a search engine should not include the page in its index. The "`rel=nofollow`" attribute appearing in an anchor tag indicates that the crawler should essentially ignore the link. For example, the following external link appeared on the Wikipedia Shakespeare page at the time of writing:

```
<a href="http://www.opensourceshakespeare.org" rel="nofollow">
  Open Source Shakespeare
</a>
```

The crawler should crawl this page only if it finds a link elsewhere, and the link should not influence ranking in any way. Sites such as blogs and wikis, which allow their users to create external links, may automatically add "`rel=nofollow`" to all such links. This policy is intended to discourage the creation of inappropriate links solely for the purpose of increasing the PageRank value of the target page. With the addition of "`rel=nofollow`" users generating this link spam will garner no benefit from it.

As it did during download, JavaScript poses problems during post-processing. When it is executed, JavaScript is capable of completely rewriting a page with new content and links. If execution is not feasible, the crawler may apply heuristics to extract whatever URLs and other information it can, but without any guarantee of success.

Priority queue

URLs extracted during post-processing are inserted into the priority queue. If a URL already appears in the queue, information gathered during post-processing may alter its position.

Implementing the priority queue is a challenging problem. If we assume an average URL is 64 bytes in length, a priority queue for 8 billion pages requires half a terabyte just to store the URLs (ignoring compression). These storage requirements, coupled with the need to support millions of updates per second, may limit the sophistication of strategies used to manage the priority queue. In Section 15.6.2, where these strategies are discussed, we ignore implementation difficulties. However, when deploying an operational crawler, these difficulties cannot be ignored.

15.6.2 Crawl Order

Suppose we are crawling the Web for the first time, with the goal of capturing and maintaining a multibillion-page snapshot. We have no detailed knowledge of the sites and pages contained on the Web, which remain to be discovered. To begin, we might start with a small *seed set* of well-known URLs including major portals, retail sites, and news services. The pages linked from the ODP[10] or other Web directory could form one possible seed set. If we then proceed in a breadth-first manner, we should encounter many high-quality pages early in the crawl (Najork and Wiener, 2001).

As we conduct the crawl, our knowledge of the Web increases. At some point, especially if the crawler is feeding an operational search engine, it becomes important to revisit pages as well as to crawl new URLs. Otherwise, the index of the search service will become stale. From this point forward, crawling proceeds *incrementally*, with the crawler continuously expanding and updating its snapshot of the Web. As it visits and revisits pages, new URLs are discovered and deleted pages are dropped.

[10] `www.dmoz.org`

The activities of the crawler are then determined by its *refresh policy* (Olston and Pandey, 2008; Pandey and Olston, 2008; Cho and Garcia-Molina, 2000, 2003; Wolf et al., 2002; Edwards et al., 2001). Two factors feed into this policy: (1) the frequency and nature of changes to pages and (2) the impact that these changes have on search results, compared to the impact of crawling new URLs. The simplest refresh policy is to revisit all pages at a fixed rate, once every X weeks, while continuing to crawl new URLs in breadth-first order. However, although this policy is appropriate for pages that change infrequently, high-impact pages that undergo frequent alterations (e.g., `www.cnn.com`) should be revisited more often. Moreover, priority should be placed on crawling those new URLs that have the highest potential to affect search results. This potential may, for instance, be estimated by processing existing query logs and analyzing user behavior (e.g., clickthrough data). It is the role of the refresh policy to determine revisit rates for existing pages and the crawl order for new URLs.

The Web changes continuously. Ntoulas et al. (2004) tracked the changes to 154 Web sites over the course of a year and estimated that new pages were created at a rate of 8% per week and new links at a rate of 25% per week. The deletion rate was also high. After one year only 20% of the initial pages remained available. However, once created, most pages changed very little until they were deleted. Even after a year less than half of the pages changed by more than 5% before deletion, as determined through a measure based on TF-IDF. When pages do change, Cho and Garcia-Molina (2003) demonstrate that the frequency of these changes may be modeled by the Poisson distribution (see page 268). Thus, if we know the history of a page, we can predict how often future changes will take place.

Even when a page does change frequently, the determination of the revisit rate depends on the nature of the changes. Olston and Pandey (2008) recognize that different parts of a Web page have different update characteristics, and these characteristics must be considered when establishing a refresh policy for that page. Some parts of a Web page may remain static, while other parts exhibit *churning behavior*. For example, the ads on a page may change each time it is accessed, or a page may update daily with a "quote of the day" while its other content remains the same. Blogs and forums may exhibit *scrolling behavior*, with new items pushing down older items that eventually fall away. Some pages, such as the home pages of news services, may have the bulk of their content revised several times an hour.

Both revisit rates and the ordering of unvisited URLs must be determined in light of the impact they have on search results. Pandey and Olston (2008) define impact in terms of the number of times a page appears, or would appear, in the top results for the queries received by the search engine. Their work suggests that when choosing a new URL to visit, priority should be given to URLs that are likely to improve the search results for one or more queries. Similarly, we may consider impact when determining revisit rates. It might be appropriate to revisit a news service every hour or more, because the top news services change constantly and have high impact. A user searching on the topic of a breaking story wants to know the latest news. On the other hand, a daily visit to a page featuring a "quote of the day" may be unnecessary if the rest of the page remains unchanged and the searches returning the page as a top result depend only on this static content.

Impact is related to static rank. Pages with high static rank may often be good candidates for frequent visits. However, impact is not equivalent to static rank (Pandey and Olston, 2008). For example, a page may have high static rank because it is linked by other pages on the same site with higher static rank. However, if it always appears well below other pages from the same site in search results, changes to that page will have little or no impact. On the other hand, a page may have a low static rank because its topic is relatively obscure, of interest only to a small subset of users. But for those users the absence of the page would have high impact on their search results.

15.6.3 Duplicates and Near-Duplicates

Roughly 30–40% of Web pages are exact duplicates of other pages and about 2% are near-duplicates (Henzinger, 2006). The volume of this data represents a major problem for search engines because retaining unnecessary duplicates increases storage and processing costs. More important, duplicates and near-duplicates can impact novelty and lead to less-than-desirable search results. Although host crowding or similar post-retrieval filtering methods might ameliorate the problem, the performance costs must still be paid and near-duplicates may still slip past the filter.

Detecting exact duplicates of pages is relatively straightforward. The entire page, including tags and scripts, may be passed to a hash function. The resulting integer hash value may then be compared with the hash value for other pages. One possible hash function for detecting duplicates is provided by the MD5 algorithm (Rivest, 1992). Given a string, MD5 computes a 128-bit “message digest” corresponding to that string. MD5 is often used to validate the integrity of files after a data transfer, providing a simple way of checking that the entire file was transferred correctly. With a length of 128 bits inadvertent collisions are unlikely (however, see Exercise 15.11).

Detection of exact duplicates through these hash values is sufficient to handle many common sources of duplication, even though pages will match only if they are byte-for-byte the same. Pages mirrored across multiple sites, such as the Java documentation, may be identified in this way. Many sites will return the same “Not Found” page for every invalid URL. Some URLs that contain a user or session id may represent the same content regardless of its value. Ideally, exact duplicates should be detected during the crawling processing. Once a mirrored page is detected, other mirrored pages linked from it can be avoided.

From the perspective of a user, two pages may be considered duplicates because they contain essentially the same material even if they are not byte-for-byte copies of one another. To detect these near-duplicates, pages may be canonicalized into a stream of tokens, reducing their content to a standard form much as we do for indexing. This canonicalization usually includes the removal of tags and scripts. In addition we may remove punctuation, capitalization, and extraneous white space. For example, the HTML document of Figure 8.1 (page 278) might be canonicalized to:

> william shakespeare wikipedia the free encyclopedia william shakespeare william shakespeare baptised 26 April 1564 died 23 April 1616 was an english poet and playwright he is widely regarded as the...

After canonicalization a hash function may be applied to detect duplicates of the remaining content.

However, near-duplicates often contain minor changes and additions, which would not be detected by a hash value over the entire page. Menus, titles, and other boilerplate may differ. Material copied from one site to another may have edits applied. Detecting these near-duplicates requires that we compare documents by comparing the substrings they contain. For example, a newswire article appearing on multiple sites will have a large substring (the news story) in common, but other parts of the pages will differ.

The degree to which a page is a copy of another may be measured by comparing the substrings they have in common. Consider a newswire story appearing on two sites; the page on the first site is 24 KB in size and the page on the second site is 32 KB, after canonicalization. If the story itself has a size of 16 KB, then we might compute the similarity of the two pages in terms of this duplicate content as

$$\frac{16\text{ KB}}{24\text{ KB} + 32\text{ KB} - 16\text{ KB}} = 50\%.$$

Thus, the news story represents 50% of the combined content of the pages.

Broder et al. (1997) describe a method to permit the computation of content similarity and the detection of near-duplicates on the scale of the Web. Their method extracts substrings known as *shingles* from each canonicalized page and compares pages by measuring the overlap between these shingles. We provide a simplified presentation of this method; full details can be found in Broder et al. (1997) and in related papers (Henzinger, 2006; Bernstein and Zobel, 2005; Charikar, 2002).

The shingles of length w from a page (the w-shingles) consist of all substrings of length w tokens appearing in the document. For example, consider the following three lines from *Hamlet*:

#1: To be, or not to be: that is the question
#2: To sleep: perchance to dream: ay, there's the rub
#3: To be or not to be, ay there's the point

The first two lines are taken from the standard edition of Shakespeare's works used throughout the book. The last is taken from what may be a "pirated" copy of the play, reconstructed from memory by a minor actor.[11]

[11] `internetshakespeare.uvic.ca/Library/SLT/literature/texts+1.html` (accessed Dec 23, 2009)

After canonicalization we have the following 2-shingles for each line, where the shingles have been sorted alphabetically and duplicates have been removed:

#1: be or, be that, is the, not to, or not, that is, the question, to be
#2: ay there, dream ay, perchance to, s the, sleep perchance, the rub, there s, to dream, to sleep
#3: ay there, be ay, be or, not to, or not, s the, the point, there s, to be

For this example we use $w = 2$. For Web pages $w = 11$ might be an appropriate choice (Broder et al., 1997).

Given two sets of shingles A and B, we define the *resemblance* between them according to the number of shingles they have in common

$$\frac{|A \cap B|}{|A \cup B|}\,. \tag{15.30}$$

Resemblance ranges between 0 and 1 and indicates the degree to which A and B contain duplicate content. To simplify the computation of resemblance, we calculate a hash value for each shingle; 64-bit hash values are usually sufficient for this purpose (Henzinger, 2006). Although the range of a 64-bit value is small enough that the Web is likely to contain different shingles that hash to the same value, pages are unlikely to contain multiple matching shingles unless they contain duplicate content. For the purpose of our example we assign 8-bit hash values arbitrarily, giving the following sets of shingle values:

#1: 43, 14, 109, 204, 26, 108, 154, 172
#2: 132, 251, 223, 16, 201, 118, 93, 197, 217
#3: 132, 110, 43, 204, 26, 16, 207, 93, 172.

A Web page may consist of a large number of shingles. One way of reducing this number is to eliminate all but those that are equal to $0 \mod m$, where $m = 25$ might be appropriate for Web data (Broder et al., 1997). Another approach is to keep the smallest s shingles. Resemblance is then estimated by comparing the remaining shingles. For our example we are following the first approach with $m = 2$, leaving us with the even-valued shingles:

#1: 14, 204, 26, 108, 154, 172
#2: 132, 16, 118
#3: 132, 110, 204, 26, 16, 172.

For this example it is easy to make pairwise comparisons between the sets of shingles. However, for billions of Web documents pairwise comparisons are not feasible. Instead, we construct tuples consisting of pairs of the form

$\langle$ *shingle value*, *document id* $\rangle$

and sort the pairs by shingle value (essentially constructing an inverted index). For our example this gives the following tuples:

$\langle 14, 1\rangle$, $\langle 16, 2\rangle$, $\langle 16, 3\rangle$, $\langle 26, 1\rangle$, $\langle 26, 3\rangle$ $\langle 108, 1\rangle$, $\langle 110, 3\rangle$, $\langle 118, 2\rangle$, $\langle 132, 2\rangle$, $\langle 132, 3\rangle$, $\langle 154, 1\rangle$, $\langle 172, 1\rangle$, $\langle 172, 3\rangle$, $\langle 204, 1\rangle$, $\langle 204, 3\rangle$.

Our next step is to combine tuples with the same shingle value to create tuples of the form

$\langle$ *id1*, *id2* $\rangle$,

where each pair indicates that the two documents share a shingle. In constructing these pairs we place the lower-valued document identifier first. The actual shingle values may now be ignored because there will be one pair for each shingle the documents have in common. For our example we have the following pairs:

$\langle 2, 3\rangle$, $\langle 1, 3\rangle$, $\langle 2, 3\rangle$, $\langle 1, 3\rangle$, $\langle 1, 3\rangle$.

Sorting and counting gives triples of the form

$\langle$ *id1*, *id2*, *count* $\rangle$.

For our example these triples are

$\langle 1, 3, 3\rangle$, $\langle 2, 3, 2\rangle$.

From these triples we may estimate the resemblance between #1 and #3 as

$$\frac{3}{6+6-3} = \frac{1}{3},$$

and the resemblance between #2 and #3 as

$$\frac{2}{3+6-2} = \frac{2}{7}.$$

The resemblance between #1 and #2 is 0.

15.7 Summary

Although this chapter is long and covers a wide range of topics, three factors occur repeatedly: scale, structure, and users.

- The Web contains an enormous volume of material on all topics and in many languages, and is constantly growing and changing. The scale of the Web implies that for many queries there are large numbers of relevant pages. With so many relevant pages, factors such as novelty and quality become important considerations for ranking. The same query will mean different things to different users. Maintaining an accurate and appropriate index for millions of sites requires substantial planning and resources.
- The structure represented by HTML tags and links provides important features for ranking. Link analysis methods have been extensively explored as methods for determining the relative quality of pages. The weighting of terms appearing in titles and anchor text may be adjusted to reflect their status. Without links, Web crawling would be nearly impossible.
- User experience drives Web search. Search engines must be aware of the range of interpretations and intent underlying user queries. Informational queries require different responses than navigational queries, and search engines must be evaluated with respect to their performance on both query types. Clickthroughs and other implicit user feedback captured in search engine logs may be analyzed to evaluate and improve performance.

15.8 Further Reading

A Web search engine is a mysterious and complex artifact that undergoes constant tuning and enhancement by large teams of dedicated engineers and developers. In this chapter we have covered only a small fraction of the technology underlying Web search.

The commercial importance of Web search and Web advertising is now significant enough that a large community of search engine marketing and search engine optimization (SEO) companies have grown around it. These SEOs advise Web site owners on how to obtain higher rankings from the major engines by means both fair and (occasionally) foul. Members of this community often blog about their work, and the technical tidbits appearing in these blogs can make for fascinating reading. A good place to start is the Search Engine Roundtable.[12] The personal blog of Matt Cutts, the head of Google's Webspam team, is another good entry point.[13]

[12] `www.seroundtable.com`
[13] `www.mattcutts.com`

The first Web search engines appeared in the early 1990s, soon after the genesis of the Web itself. By the mid-1990s commercial search services, such as Excite, Lycos, Altavista, and Yahoo!, were handling millions of queries per day. A history of these early search engines may be found at Search Engine Watch, a site that has tracked the commercial aspects of search engine technology since 1996.[14] Along with the content-based features discussed in Part III, these early engines incorporated simple link-based features — for example, using a page's in-degree as an indicator of its popularity (Marchiori, 1997).

15.8.1 Link Analysis

By the late 1990s the importance of static ranking and link analysis was widely recognized. In 1997 Marchiori outlined a link analysis technique that foreshadowed the intuition underlying PageRank. In the same year Carrière and Kazman (1997) presented a tool for exploring the linkage relationships between Web sites.

On 15 April 1998, at the 7th World Wide Web Conference in Brisbane, Australia, Brin and Page presented their now-classic paper describing the basic PageRank algorithm and the architecture of their nascent Google search engine (Brin and Page, 1998). In papers published soon afterwards, they and their colleagues built upon this work, describing foundational algorithms for personalized PageRank and Web data mining (Brin et al., 1998; Page et al., 1999). While Brin and Page were creating PageRank (and the Google search engine), Kleinberg independently developed the HITS algorithm and presented it at the 9th Annual Symposium on Discrete Algorithms in January 1998 (Kleinberg, 1998, 1999). A paper co-authored by Kleinberg, extending HITS to accommodate anchor text, was presented at the 7th World Wide Web Conference on the same day as Brin and Page's paper (Chakrabarti et al., 1998). Together the work of Brin, Page, and Kleinberg engendered a flood of research into link analysis techniques.

Bharat and Henzinger (1998) combined HITS with content analysis. They also recognized the problems that tightly connected communities cause for HITS and suggested the solution that Lempel and Moran (2000) later developed into SALSA. Rafiei and Mendelzon (2000) extended PageRank along the lines of SALSA to determine the topics for which a page is known (its "reputation"). Davidson (2000) recognized that some links should be assigned lower weights than others and trained a classifier to recognize links arising from commercial relationships rather than solely from merit. Cohn and Chang (2000) applied *principal component analysis* (PCA) to extract multiple eigenvectors from the HITS matrix. A number of other authors explore efficient methods for computing personalized and topic-oriented PageRank (Haveliwala, 2002; Jeh and Widom, 2003; Chakrabarti, 2007).

Ng et al. (2001a,b) compare the stability of HITS and PageRank, demonstrating the role of PageRank's jump vector and suggesting the addition of a jump vector to HITS. Borodin et al. (2001) provide a theoretical analysis of HITS and SALSA and suggest further improvements.

[14] `searchenginewatch.com/showPage.html?page=3071951` (accessed Dec 23, 2009)

Richardson and Domingos (2002) describe a query-dependent version of PageRank in which the random surfer is more likely to follow links to pages related to the query. Kamvar et al. (2003) present a method for accelerating the computation of PageRank.

More recently, Langville and Meyer (2005, 2006) provide a detailed and readable survey of the mathematics underlying PageRank and HITS. Bianchini et al. (2005) explore further properties of PageRank. Craswell et al. (2005) discuss the extension of BM25F to incorporate static ranking. Baeza-Yates et al. (2006) generalize the role of the jump vector and replace it with other damping functions to create a family of algorithms related to PageRank. Najork et al. (2007) compare the retrieval effectiveness of HITS and basic PageRank, both alone and in combination with BM25F, illustrating the limitations of basic PageRank. In a related study Najork (2007) compares the retrieval effectiveness of HITS and SALSA, finding that SALSA outperforms HITS as a static ranking feature.

Gyöngyi et al. (2004) present a link analysis technique called *TrustRank* for identifying Web spam. A related paper by Gyöngyi and Garcia-Molina (2005) provides a general overview and discussion of the Web spam problem. In addition to link analysis techniques, content-oriented techniques such as the e-mail spam filtering methods described in Chapter 10 may be applied to identify Web spam. The AIRWeb[15] workshops provide a forum for the testing and evaluating methods for detecting Web spam as part of a broader theme of adversarial IR on the Web.

Golub and Van Loan (1996) is the bible of numerical methods for matrix computations. They devote two long chapters to the solution of eigenvalue problems, including a thorough discussion of the power method.

15.8.2 Anchor Text

Anchor text was used as a ranking feature in the earliest Web search engines (Brin and Page, 1998). Craswell et al. (2001) demonstrate its importance in comparison to content features. Robertson et al. (2004) describe the incorporation of anchor text into the probabilistic retrieval model. Hawking et al. (2004) present a method for attenuating weights for anchor text, adjusting term frequency values when anchor text is repeated many times.

15.8.3 Implicit Feedback

Joachims and Radlinski (2007) provide an overview of methods for interpreting implicit feedback; Kelly and Teevan (2003) provide a bibliography of early work. Dupret et al. (2007), Liu et al. (2007), and Carterette and Jones (2007) all discuss the use of clickthroughs for Web search evaluation. Agichtein et al. (2006b) learn relevance judgments by combining multiple browsing and clickthrough features. Qiu and Cho (2006) present a method for determining a user's interests from clickthroughs, thus allowing search results to be personalized to reflect these interests.

[15] `airweb.cse.lehigh.edu`

15.8.4 Web Crawlers

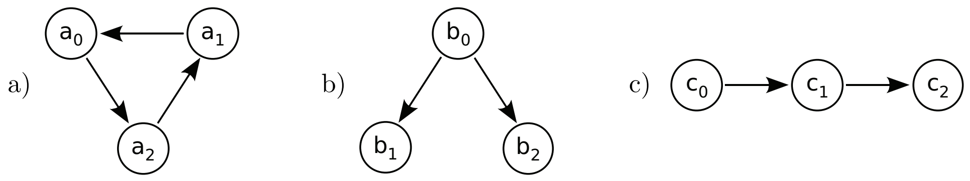
A basic architecture for a Web crawler is described by Heydon and Najork (1999). Olston and Najork (2010) provide a recent and thorough survey of the area.

The problem of optimizing the refresh policy of an incremental crawler has been studied by several groups (Edwards et al., 2001; Wolf et al., 2002; Cho and Garcia-Molina, 2003; Olston and Pandey, 2008). Dasgupta et al. (2007) examine the trade-offs between visiting new URLs and revisiting previously crawled pages. Pandey and Olston (2008) consider the impact of new pages on search results. Chakrabarti et al. (1999) describe focused crawlers that are dedicated to crawling pages related to a specific topic. Little has been published regarding the efficient implementation of priority queues for Web crawlers, but Yi et al. (2003) provide a possible starting point for investigation.

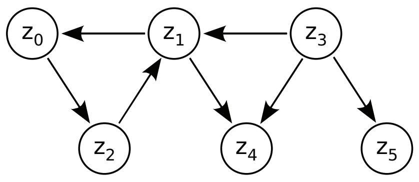
Broder et al. (1997) introduced shingles as a method for detecting near-duplicates. Henzinger (2006) experimentally compared this method with a competing method by Charikar (2002) and introduced a combined algorithm that outperforms both. Bernstein and Zobel (2005) used a version of shingling to evaluate the impact of near-duplicates on retrieval effectiveness.

15.9 Exercises

Exercise 15.1 Compute basic PageRank for the following Web graphs. Assume $\delta = 3/4$.



Exercise 15.2 Compute basic PageRank for the following Web graph. Assume $\delta = 0.85$.



Exercise 15.3 Given Equations 15.9 and 15.10, show that for all $n \geq 0$

$$\sum_{\alpha \in \Phi} r^{(n)}(\alpha) \;=\; N.$$

Exercise 15.4 Compute extended PageRank for the follow matrix and jump vector in Equation 15.12 (page 527). Assume $\delta = 0.85$.

Exercise 15.5 Show that the transition matrix M' (page 533) is aperiodic for any Web graph.

Exercise 15.6 Suggest additional sources of information, beyond those listed on page 527, that might be enlisted when setting the follow matrix and jump vector

Exercise 15.7 Compute the co-citation matrix $A = W^T W$ and the bibliographic coupling matrix $H = WW^T$ for the adjacency matrix W in Equation 15.24 (page 535).

Exercise 15.8 Compute the authority vector $\vec{a}$ and hub vector $\vec{h}$ for the adjacency matrix W in Equation 15.24 (page 535).

Exercise 15.9 Langville and Meyer (2005) suggest the following example to illustrate the convergence properties of HITS. Compute the authority vector $\vec{a}$ for the following adjacency matrix, starting with the initial estimate $\vec{a}^{(0)} = \langle 1/2, 1/2, 1/2, 1/2 \rangle^T$.

$$W = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \qquad (15.31)$$

Recompute the authority vector $\vec{a}$ starting with the initial estimate $\vec{a}^{(0)} = \langle 1/\sqrt{3}, 1/3, 1/3, 2/3 \rangle^T$.

Exercise 15.10 View the `robots.txt` file on well-known Web sites. Are any pages or crawlers disallowed? Why?

Exercise 15.11 The MD5 algorithm is known to be insecure. Given an MD5 value for a string, it is possible to construct a different string that gives the same value. How could this vulnerability be exploited by a malicious site to cause problems for a Web search service? Search this topic and suggest possible solutions.

Exercise 15.12 Search the term "Google bombing". Suggest ways in which a search engine might cope with this problem.

Exercise 15.13 (project exercise) Using the technique described by Bharat and Broder (1998), as updated by Gulli and Signorini (2005), estimate the size of the indexable Web.

Exercise 15.14 (project exercise) Build a "crawler trap", which generates dynamic context to make a Web site appear much larger than it really is (billions of pages). The trap should generate random pages, containing random text (Exercise 1.13) with random links to other random pages in the trap. URLs for the trap should have seeds for a random number generator embedded within them, so that visiting a page will consistently give the same contents.

Warning: Deploying the trap on a live Web site may negatively impact search results for that site. Be cautious. Ask for permission if necessary. If you do deploy the trap, we suggest that you add a `robots.txt` entry to steer well-behaved crawlers away from it.

15.10 Bibliography

Agichtein, E., Brill, E., and Dumais, S. (2006a). Improving Web search ranking by incorporating user behavior information. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 19–26. Seattle, Washington.

Agichtein, E., Brill, E., Dumais, S., and Ragno, R. (2006b). Learning user interaction models for predicting Web search result preferences. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3–10. Seattle, Washington.

Baeza-Yates, R., Boldi, P., and Castillo, C. (2006). Generalizing PageRank: Damping functions for link-based ranking algorithms. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 308–315. Seattle, Washington.

Bernstein, Y., and Zobel, J. (2005). Redundant documents and search effectiveness. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, pages 736–743. Bremen, Germany.

Bharat, K., and Broder, A. (1998). A technique for measuring the relative size and overlap of public Web search engines. In *Proceedings of the 7th International World Wide Web Conference*, pages 379–388. Brisbane, Australia.

Bharat, K., and Henzinger, M. R. (1998). Improved algorithms for topic distillation in a hyperlinked environment. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 104–111. Melbourne, Australia.

Bianchini, M., Gori, M., and Scarselli, F. (2005). Inside PageRank. *ACM Transactions on Internet Technology*, 5(1):92–128.

Borodin, A., Roberts, G. O., Rosenthal, J. S., and Tsaparas, P. (2001). Finding authorities and hubs from link structures on the World Wide Web. In *Proceedings of the 10th International World Wide Web Conference*, pages 415–429. Hong Kong, China.

Brin, S., Motwani, R., Page, L., and Winograd, T. (1998). What can you do with a Web in your pocket? *Data Engineering Bulletin*, 21(2):37–47.

Brin, S., and Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the 7th International World Wide Web Conference*, pages 107–117. Brisbane, Australia.

Broder, A. (2002). A taxonomy of Web search. *ACM SIGIR Forum*, 36(2):3–10.

Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G. (1997). Syntactic clustering of the Web. In *Proceedings of the 6th International World Wide Web Conference*, pages 1157–1166. Santa Clara, California.

Büttcher, S., Clarke, C. L. A., and Soboroff, I. (2006). The TREC 2006 Terabyte Track. In *Proceedings of the 15th Text REtrieval Conference*. Gaithersburg, Maryland.

Carrière, J., and Kazman, R. (1997). WebQuery: Searching and visualizing the Web through connectivity. In *Proceedings of the 6th International World Wide Web Conference*, pages 1257–1267.

Carterette, B., and Jones, R. (2007). Evaluating search engines by modeling the relationship between relevance and clicks. In *Proceedings of the 21st Annual Conference on Neural Information Processing Systems*. Vancouver, Canada.

Chakrabarti, S. (2007). Dynamic personalized PageRank in entity-relation graphs. In *Proceedings of the 16th International World Wide Web Conference*. Banff, Canada.

Chakrabarti, S., Dom, B., Raghavan, P., Rajagopalan, S., Gibson, D., and Kleinberg, J. (1998). Automatic resource list compilation by analyzing hyperlink structure and associated text. In *Proceedings of the 7th International World Wide Web Conference*. Brisbane, Australia.

Chakrabarti, S., van den Burg, M., and Dom, B. (1999). Focused crawling: A new approach to topic-specific Web resource discovery. In *Proceedings of the 8th International World Wide Web Conference*, pages 545–562. Toronto, Canada.

Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 380–388. Montreal, Canada.

Cho, J., and Garcia-Molina, H. (2000). The evolution of the Web and implications for an incremental crawler. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 200–209.

Cho, J., and Garcia-Molina, H. (2003). Effective page refresh policies for Web crawlers. *ACM Transactions on Database Systems*, 28(4):390–426.

Chung, C., and Clarke, C. L. A. (2002). Topic-oriented collaborative crawling. In *Proceedings of the 11th International Conference on Information and Knowledge Management*, pages 34–42. McLean, Virginia.

Clarke, C. L. A., Agichtein, E., Dumais, S., and White, R. W. (2007). The influence of caption features on clickthrough patterns in Web search. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 135–142. Amsterdam, The Netherlands.

Clarke, C. L. A., Scholer, F., and Soboroff, I. (2005). The TREC 2005 Terabyte Track. In *Proceedings of the 14th Text REtrieval Conference*. Gaithersburg, Maryland.

Cohn, D., and Chang, H. (2000). Learning to probabilistically identify authoritative documents. In *Proceedings of the 17th International Conference on Machine Learning*, pages 167–174.

Craswell, N., and Hawking, D. (2004). Overview of the TREC 2004 Web Track. In *Proceedings of the 13th Text REtrieval Conference*. Gaithersburg, Maryland.

Craswell, N., Hawking, D., and Robertson, S. (2001). Effective site finding using link anchor information. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 250–257. New Orleans, Louisiana.

Craswell, N., Robertson, S., Zaragoza, H., and Taylor, M. (2005). Relevance weighting for query independent evidence. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 416–423. Salvador, Brazil.

Cucerzan, S., and Brill, E. (2004). Spelling correction as an iterative process that exploits the collective knowledge of Web users. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 293–300.

Dasgupta, A., Ghosh, A., Kumar, R., Olston, C., Pandey, S., and Tomkins, A. (2007). The discoverability of the Web. In *Proceedings of the 16th International World Wide Web Conference*. Banff, Canada.

Davidson, B. D. (2000). Recognizing nepotistic links on the Web. In *Proceedings of the AAAI-2000 Workshop on Artificial Intelligence for Web Search*, pages 23–28.

Dupret, G., Murdock, V., and Piwowarski, B. (2007). Web search engine evaluation using clickthrough data and a user model. In *Proceedings of the 16th International World Wide Web Conference Workshop on Query Log Analysis: Social and Technological Challenges*. Banff, Canada.

Edwards, J., McCurley, K., and Tomlin, J. (2001). An adaptive model for optimizing performance of an incremental Web crawler. In *Proceedings of the 10th International World Wide Web Conference*, pages 106–113. Hong Kong, China.

Golub, G. H., and Van Loan, C. F. (1996). *Matrix Computations* (3rd ed.). Baltimore, Maryland: Johns Hopkins University Press.

Gulli, A., and Signorini, A. (2005). The indexable Web is more than 11.5 billion pages. In *Proceedings of the 14th International World Wide Web Conference*. Chiba, Japan.

Gyöngyi, Z., and Garcia-Molina, H. (2005). Spam: It's not just for inboxes anymore. *Computer*, 38(10):28–34.

Gyöngyi, Z., Garcia-Molina, H., and Pedersen, J. (2004). Combating Web spam with TrustRank. In *Proceedings of the 30th International Conference on Very Large Databases*, pages 576–584.

Haveliwala, T., and Kamvar, S. (2003). *The Second Eigenvalue of the Google Matrix*. Technical Report 2003-20. Stanford University.

Haveliwala, T. H. (2002). Topic-sensitive PageRank. In *Proceedings of the 11th International World Wide Web Conference*. Honolulu, Hawaii.

Hawking, D., and Craswell, N. (2001). Overview of the TREC-2001 Web Track. In *Proceedings of the 10th Text REtrieval Conference*. Gaithersburg, Maryland.

Hawking, D., Upstill, T., and Craswell, N. (2004). Toward better weighting of anchors. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 512–513. Sheffield, England.

Henzinger, M. (2006). Finding near-duplicate Web pages: A large-scale evaluation of algorithms. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and development in Information Retrieval*, pages 284–291. Seattle, Washington.

Heydon, A., and Najork, M. (1999). Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229.

Ivory, M. Y., and Hearst, M. A. (2002). Statistical profiles of highly-rated Web sites. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 367–374. Minneapolis, Minnesota.

Jansen, B. J., Booth, D., and Spink, A. (2007). Determining the user intent of Web search engine queries. In *Proceedings of the 16th International World Wide Web Conference*, pages 1149–1150. Banff, Canada.

Jeh, G., and Widom, J. (2003). Scaling personalized Web search. In *Proceedings of the 12th International World Wide Web Conference*, pages 271–279. Budapest, Hungary.

Joachims, T., Granka, L., Pan, B., Hembrooke, H., and Gay, G. (2005). Accurately interpreting clickthrough data as implicit feedback. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 154–161. Salvador, Brazil.

Joachims, T., and Radlinski, F. (2007). Search engines that learn from implicit feedback. *IEEE Computer*, 40(8):34–40.

Jones, R., Rey, B., Madani, O., and Greiner, W. (2006). Generating query substitutions. In *Proceedings of the 15th International World Wide Web Conference*, pages 387–396. Edinburgh, Scotland.

Kamvar, S. D., Haveliwala, T. H., Manning, C. D., and Golub, G. H. (2003). Extrapolation methods for accelerating PageRank computations. In *Proceedings of the 12th International World Wide Web Conference*, pages 261–270. Budapest, Hungary.

Kellar, M., Watters, C., and Shepherd, M. (2007). A field study characterizing web-based information-seeking tasks. *Journal of the American Society for Information Science and Technology*, 58(7):999–1018.

Kelly, D., and Teevan, J. (2003). Implicit feedback for inferring user preference: A bibliography. *ACM SIGIR Forum*, 37(2):18–28.

Kleinberg, J. M. (1998). Authoritative sources in a hyperlinked environment. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 668–677. San Francisco, California.

Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632.

Langville, A. N., and Meyer, C. D. (2005). A survey of eigenvector methods of Web information retrieval. *SIAM Review*, 47(1):135–161.

Langville, A. N., and Meyer, C. D. (2006). *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, New Jersey: Princeton University Press.

Lawrence, S., and Giles, C. L. (1998). Searching the World Wide Web. *Science*, 280:98–100.

Lawrence, S., and Giles, C. L. (1999). Accessibility of information on the Web. *Nature*, 400:107–109.

Lee, U., Liu, Z., and Cho, J. (2005). Automatic identification of user goals in Web search. In *Proceedings of the 14th International World Wide Web Conference*, pages 391–400. Chiba, Japan.

Lempel, R., and Moran, S. (2000). The stochastic approach for link-structure analysis (SALSA) and the TKC effect. *Computer Networks*, 33(1-6):387–401.

Liu, Y., Fu, Y., Zhang, M., Ma, S., and Ru, L. (2007). Automatic search engine performance evaluation with click-through data analysis. In *Proceedings of the 16th International World Wide Web Conference Workshop on Query Log Analysis: Social and Technological Challenges*, pages 1133–1134. Banff, Canada.

Marchiori, M. (1997). The quest for correct information on the Web: Hyper search engines. In *Proceedings of the 6th International World Wide Web Conference*. Santa Clara, California.

Metzler, D., Strohman, T., and Croft, W. (2006). Indri TREC notebook 2006: Lessons learned from three Terabyte Tracks. In *Proceedings of the 15th Text REtrieval Conference*. Gaithersburg, Maryland.

Najork, M., and Wiener, J.L. (2001). Breadth-first search crawling yields high-quality pages. In *Proceedings of the 10th International World Wide Web Conference*. Hong Kong, China.

Najork, M.A. (2007). Comparing the effectiveness of HITS and SALSA. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, pages 157–164. Lisbon, Portugal.

Najork, M.A., Zaragoza, H., and Taylor, M.J. (2007). HITS on the Web: How does it compare? In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 471–478. Amsterdam, The Netherlands.

Ng, A.Y., Zheng, A.X., and Jordan, M.I. (2001a). Link analysis, eigenvectors and stability. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 903–910. Seattle, Washington.

Ng, A.Y., Zheng, A.X., and Jordan, M.I. (2001b). Stable algorithms for link analysis. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 258–266. New Orleans, Louisiana.

Ntoulas, A., Cho, J., and Olston, C. (2004). What's new on the Web?: The evolution of the web from a search engine perspective. In *Proceedings of the 13th International World Wide Web Conference*, pages 1–12.

Olston, C., and Najork, M. (2010). Web crawling. *Foundations and Trends in Information Retrieval*.

Olston, C., and Pandey, S. (2008). Recrawl scheduling based on information longevity. In *Proceedings of the 17th International World Wide Web Conference*, pages 437–446. Beijing, China.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab.

Pandey, S., and Olston, C. (2008). Crawl ordering by search impact. In *Proceedings of the 1st ACM International Conference on Web Search and Data Mining*. Palo Alto, California.

Qiu, F., and Cho, J. (2006). Automatic identification of user interest for personalized search. In *Proceedings of the 15th International World Wide Web Conference*, pages 727–736. Edinburgh, Scotland.

Rafiei, D., and Mendelzon, A. O. (2000). What is this page known for? Computing Web page reputations. In *Proceedings of the 9th International World Wide Web Conference*, pages 823–835. Amsterdam, The Netherlands.

Richardson, M., and Domingos, P. (2002). The intelligent surfer: Probabilistic combination of link and content information in PageRank. In *Advances in Neural Information Processing Systems 14*, pages 1441–1448.

Richardson, M., Prakash, A., and Brill, E. (2006). Beyond PageRank: Machine learning for static ranking. In *Proceedings of the 15th International World Wide Web Conference*, pages 707–715. Edinburgh, Scotland.

Rivest, R. (1992). *The MD5 Message-Digest Algorithm.* Technical Report 1321. Internet RFC.

Robertson, S., Zaragoza, H., and Taylor, M. (2004). Simple BM25 extension to multiple weighted fields. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management*, pages 42–49. Washington, D.C.

Rose, D. E., and Levinson, D. (2004). Understanding user goals in web search. In *Proceedings of 13th International World Wide Web Conference*, pages 13–19. New York.

Spink, A., and Jansen, B. J. (2004). A study of Web search trends. *Webology*, 1(2).

Upstill, T., Craswell, N., and Hawking, D. (2003). Query-independent evidence in home page finding. *ACM Transactions on Information Systems*, 21(3):286–313.

Wolf, J. L., Squillante, M. S., Yu, P. S., Sethuraman, J., and Ozsen, L. (2002). Optimal crawling strategies for Web search engines. In *Proceedings of the 11th International World Wide Web Conference*, pages 136–147. Honolulu, Hawaii.

Yi, K., Yu, H., Yang, J., Xia, G., and Chen, Y. (2003). Efficient maintenance of materialized top-k views. In *Proceedings of the 19th International Conference on Data Engineering*, pages 189–200.