2 The term vocabulary and postings lists

Recall the major steps in inverted index construction:

1. Collect the documents to be indexed.
2. Tokenize the text.
3. Do linguistic preprocessing of tokens.
4. Index the documents that each term occurs in.

In this chapter, we first briefly mention how the basic unit of a document can be defined and how the character sequence that it comprises is determined (Section 2.1). We then examine in detail some of the substantive linguistic issues of tokenization and linguistic preprocessing, which determine the vocabulary of terms that a system uses (Section 2.2). *Tokenization* is the process of chopping character streams into tokens; linguistic preprocessing then deals with building equivalence classes of tokens, which are the set of terms that are indexed. Indexing itself is covered in Chapters 1 and 4. Then we return to the implementation of postings lists. In Section 2.3, we examine an extended postings list data structure that supports faster querying, and Section 2.4 covers building postings data structures suitable for handling phrase and proximity queries, of the sort that commonly appear in both extended Boolean models and on the web.

2.1 Document delineation and character sequence decoding

2.1.1 Obtaining the character sequence in a document

Digital documents that are the input to an indexing process are typically bytes in a file or on a web server. The first step of processing is to convert this byte sequence into a linear sequence of characters. For the case of plain English text in ASCII encoding, this is trivial. But often things get much more complex. The sequence of characters may be encoded by one of various single-byte or multibyte encoding schemes, such as Unicode UTF-8,

كِتابٌ ⇐ ك ِ ت ا ب ٌ
un b ā t i k
/kitābun/ *'a book'*

Figure 2.1 An example of a vocalized Modern Standard Arabic word. The writing is from right to left and letters undergo complex mutations as they are combined. The representation of short vowels (here, /i/ and /u/) and the final /n/ (nunation) departs from strict linearity by being represented as diacritics above and below letters. Nevertheless, the represented text is still clearly a linear ordering of characters representing sounds. Full vocalization, as here, normally appears only in the Koran and children's books. Day-to-day text is unvocalized (short vowels are not represented, but the letter for ā would still appear) or partially vocalized, with short vowels inserted in places where the writer perceives ambiguities. These choices add further complexities to indexing.

or various national or vendor-specific standards. We need to determine the correct encoding. This can be regarded as a machine learning classification problem, as discussed in Chapter 13,[1] but is often handled by heuristic methods, user selection, or using provided document metadata. Once the encoding is determined, we decode the byte sequence to a character sequence. We might save the choice of encoding because it gives some evidence about what language the document is written in.

The characters may have to be decoded out of some binary representation like Microsoft Word DOC files and/or a compressed format such as zip files. Again, we must determine the document format, and then an appropriate decoder has to be used. Even for plain text documents, additional decoding may need to be done. In XML documents (Section 10.1, page 180), character entities, such as `&`, need to be decoded to give the correct character, namely, `&` for `&`. Finally, the textual part of the document may need to be extracted out of other material that will not be processed. This might be the desired handling for XML files, if the markup is going to be ignored; we would almost certainly want to do this with postscript or PDF files. We do not deal further with these issues in this book, and assume henceforth that our documents are a list of characters. Commercial products usually need to support a broad range of document types and encodings, because users want things to just work with their data as is. Often, they just think of documents as text inside applications and are not even aware of how it is encoded on disk. This problem is usually solved by licensing a software library that handles decoding document formats and character encodings.

The idea that text is a linear sequence of characters is also called into question by some writing systems, such as Arabic, where text takes on some two-dimensional and mixed-order characteristics, as shown in Figures 2.1 and 2.2. But, despite some complicated writing system conventions, there is an underlying sequence of sounds being represented and hence an

[1] A classifier is a function that takes objects of some sort and assigns them to one of a number of distinct classes. Usually classification is done by machine learning methods such as probabilistic models, but it can also be done by hand-written rules.

استقلت الجزائر في سنة 1962 بعد 132 عاما من الاحتلال الفرنسي.

← → ← → ← START

'Algeria achieved its independence in 1962 after 132 years of French occupation.'

Figure 2.2 The conceptual linear order of characters is not necessarily the order that you see on the page. In languages that are written right to left, such as Hebrew and Arabic, it is quite common to also have left-to-right text interspersed, such as numbers and dollar amounts. With modern Unicode representation concepts, the order of characters in files matches the conceptual order, and the reversal of displayed characters is handled by the rendering system, but this may not be true for documents in older encodings.

essentially linear structure remains. This is what is represented in the digital representation of Arabic, as shown in Figure 2.1.

2.1.2 Choosing a document unit

DOCUMENT UNIT

The next phase is to determine what the *document unit* for indexing is. Thus far, we have assumed that documents are fixed units for the purposes of indexing. For example, we take each file in a folder as a document. But there are many cases in which you might want to do something different. A traditional Unix (mbox-format) email file stores a sequence of email messages (an email folder) in one file, but you might wish to regard each email message as a separate document. Many email messages now contain attached documents, and you might then want to regard the email message and each contained attachment as separate documents. If an email message has an attached zip file, you might want to decode the zip file and regard each file it contains as a separate document. Going in the opposite direction, various pieces of web software (such as latex2html) take things that you might regard as a single document (e.g., a Powerpoint file or a LaTeX document) and split them into separate HTML pages for each slide or subsection, stored as separate files. In these cases, you might want to combine multiple files into a single document.

INDEXING GRANULARITY

More generally, for very long documents, the issue of *indexing granularity* arises. For a collection of books, it would usually be a bad idea to index an entire book as a document. A search for Chinese toys might bring up a book that mentions China in the first chapter and toys in the last chapter, but this does not make it relevant to the query. Instead, we may well wish to index each chapter or paragraph as a mini-document. Matches are then more likely to be relevant, and because the documents are smaller it will be much easier for the user to find the relevant passages in the document. But why stop there? We could treat individual sentences as mini-documents. It becomes clear that there is a precision/recall tradeoff here. If the units get too small, we are likely to miss important passages because terms were distributed over several mini-documents, whereas if units are too large we tend to get spurious matches and the relevant information is hard for the user to find.

The problems with large document units can be alleviated by use of explicit or implicit proximity search (Sections 2.4.2 and 7.2.2), and the tradeoffs in resulting system performance that we are hinting at are discussed in Chapter 8. The issue of index granularity, and in particular a need to simultaneously index documents at multiple levels of granularity, appears prominently in XML retrieval, and is taken up again in Chapter 10. An information retrieval (IR) system should be designed to offer choices of granularity. For this choice to be made well, the person who is deploying the system must have a good understanding of the document collection, the users, and their likely information needs and usage patterns. For now, we assume that a suitable size document unit has been chosen, together with an appropriate way of dividing or aggregating files, if needed.

2.2 Determining the vocabulary of terms

2.2.1 *Tokenization*

Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called *tokens*, perhaps at the same time throwing away certain characters, such as punctuation. Here is an example of tokenization:

Input: Friends, Romans, Countrymen, lend me your ears;
Output: | Friends | Romans | Countrymen | lend | me | your | ears |

These tokens are often loosely referred to as terms or words, but it is sometimes important to make a type/token distinction. A *token* is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. A *type* is the class of all tokens containing the same character sequence. A *term* is a (perhaps normalized) type that is included in the IR system's dictionary. The set of index terms could be entirely distinct from the tokens, for instance, they could be semantic identifiers in a taxonomy, but in practice in modern IR systems they are strongly related to the tokens in the document. However, rather than being exactly the tokens that appear in the document, they are usually derived from them by various normalization processes which are discussed in Section 2.2.3.[2] For example, if the document to be indexed is *to sleep perchance to dream*, then there are five tokens, but only four types (because there are two instances of *to*). However, if *to* is omitted from the index (as a stop word; see

TOKEN
TYPE
TERM

[2] That is, as defined here, tokens that are not indexed (stop words) are not terms, and if multiple tokens are collapsed together via normalization, they are indexed as one term, under the normalized form. However, we later relax this definition when discussing classification and clustering in Chapters 13–18, where there is no index. In these chapters, we drop the requirement of inclusion in the dictionary. A *term* means a normalized word.

Section 2.2.2 (page 25)), then there are only three terms: *sleep*, *perchance*, and *dream*.

The major question of the tokenization phase is what are the correct tokens to use? In this example, it looks fairly trivial: you chop on whitespace and throw away punctuation characters. This is a starting point, but even for English there are a number of tricky cases. For example, what do you do about the various uses of the apostrophe for possession and contractions?

> Mr. O'Neill thinks that the boys' stories about Chile's capital aren't amusing.

For *O'Neill*, which of the following is the desired tokenization?

[neill]
[oneill]
[o'neill]
[o'] [neill]
[o] [neill] ?

And for *aren't*, is it:

[aren't]
[arent]
[are] [n't]
[aren] [t] ?

A simple strategy is to just split on all nonalphanumeric characters, but although [o] [neill] looks okay, [aren] [t] looks intuitively bad. For all of them, the choices determine which Boolean queries match. A query of neill AND capital matches in three cases but not the other two. In how many cases would a query of o'neill AND capital match? If no preprocessing of a query is done, then it would match in only one of the five cases. For either Boolean or free text queries, you always want to do the exact same tokenization of document and query words, generally by processing queries with the same tokenizer. This guarantees that a sequence of characters in a text will always match the same sequence typed in a query.[3]

LANGUAGE IDENTIFICATION

These issues of tokenization are language specific. It thus requires the language of the document to be known. *Language identification* based on classifiers that use short character subsequences as features is highly effective; most languages have distinctive signature patterns (see page 43 for references).

[3] For the free text case, this is straightforward. The Boolean case is more complex; this tokenization may produce multiple terms from one query word. This can be handled by combining the terms with an AND or as a phrase query (see Section 2.4, page 36). It is harder for a system to handle the opposite case, where the user enters as two terms something that was tokenized together in the document processing.

For most languages, and for particular domains within them, there are unusual specific tokens that we wish to recognize as terms, such as the programming languages C and C#, aircraft names like B-52, or a television show name such as MASH – which is sufficiently integrated into popular culture that you find usages such as *M*A*S*H-style hospitals*. Computer technology has introduced new types of character sequences that a tokenizer should probably tokenize as a single token, including email addresses (blackmail.yahoo.com), web URLs (http://stuff.big.com/new/specials.html), numeric IP addresses (142.32.48.231), package tracking numbers (19999W99845399981), and more. One possible solution is to omit from indexing tokens such as monetary amounts, numbers, and URLs, because their presence greatly expands the size of the vocabulary. However, this comes at a high cost in restricting what people can search for. For instance, people might want to search in a bug database for the line number where an error occurs. Items such as the date of an email, which have a clear semantic type, are often indexed separately as document metadata (see Section 6.1, page 101).

HYPHENS

In English, *hyphenation* is used for various purposes ranging from splitting up vowels in words (*co-education*) to joining nouns as names (*Hewlett-Packard*) to a copyediting device to show word grouping (*the hold-him-back-and-drag-him-away maneuver*). It is easy to feel that the first example should be regarded as one token (and is indeed more commonly written as just *coeducation*), the last should be separated into words, and that the middle case is unclear. Handling hyphens automatically can thus be complex: it can either be handled as a classification problem, or more commonly by some heuristic rules, such as allowing short hyphenated prefixes on words, but not longer hyphenated forms.

Conceptually, splitting on white space can also split what should be regarded as a single token. This occurs most commonly with names (*San Francisco, Los Angeles*) but also with borrowed foreign phrases (*au fait*) and compounds that are sometimes written as a single word and sometimes space separated (such as *white space* vs. *whitespace*). Other cases with internal spaces that we might wish to regard as a single token include phone numbers [(800) 234-2333] and dates (Mar 11, 1983). Splitting tokens on spaces can cause bad retrieval results, for example, if a search for York University mainly returns documents containing *New York University*. The problems of hyphens and nonseparating whitespace can even interact. Advertisements for air fares frequently contain items like *San Francisco-Los Angeles*, where simply doing whitespace splitting would give unfortunate results. In such cases, issues of tokenization interact with handling phrase queries (which we discuss in Section 2.4 (page 36)), particularly if we would like queries for all of *lowercase*, *lower-case* and *lower case* to return the same results. The last two can be handled by splitting on hyphens and using a phrase index. Getting the first case right would depend on knowing that it is sometimes written as two words

莎拉波娃现在居住在美国东南部的佛罗里达。今年４月
９日，莎拉波娃在美国第一大城市纽约度过了１８岁生
日。生日派对上，莎拉波娃露出了甜美的微笑。

Figure 2.3 The standard unsegmented form of Chinese text using the simplified characters of mainland China. There is no whitespace between words, not even between sentences – the apparent space after the Chinese period (。) is just a typographical illusion caused by placing the character on the left side of its square box. The first sentence is just words in Chinese characters with no spaces between them. The second and third sentences include Arabic numerals and punctuation breaking up the Chinese characters.

and also indexing it in this way. One effective strategy in practice, which is used by some Boolean retrieval systems such as Westlaw and Lexis-Nexis (Example 1.1), is to encourage users to enter hyphens wherever they may be possible, and whenever there is a hyphenated form, the system will generalize the query to cover all three of the one word, hyphenated, and two word forms, so that a query for over-eager will search for over-eager OR "over eager" OR overeager. However, this strategy depends on user training; if you query using either of the other two forms, you get no generalization.

Each new language presents some new issues. For instance, French has a variant use of the apostrophe for a reduced definite article "the" before a word beginning with a vowel (e.g., *l'ensemble*) and has some uses of the hyphen with postposed clitic pronouns in imperatives and questions (e.g., *donne-moi* – "give me"). Getting the first case correct affects the correct indexing of a fair percentage of nouns and adjectives: you would want documents mentioning both *l'ensemble* and *un ensemble* to be indexed under *ensemble*. Other languages make the problem harder in new ways. German writes *compound nouns* without spaces (e.g., *Computerlinguistik* – "computational linguistics"; *Lebensversicherungsgesellschaftsangestellter* – "life insurance company employee"). Retrieval systems for German greatly benefit from the use of a *compound splitter* module, which is usually implemented by seeing if a word can be subdivided into multiple words that appear in a vocabulary. This phenomenon reaches its limit case with major East Asian Languages (e.g., Chinese, Japanese, Korean, and Thai), where text is written without any spaces between words. An example is shown in Figure 2.3. One approach here is to perform *word segmentation* as prior linguistic processing. Methods of word segmentation vary from having a large vocabulary and taking the longest vocabulary match with some heuristics for unknown words to the use of machine learning sequence models, such as hidden Markov models or conditional random fields, trained over hand-segmented words (see the references in Section 2.5). Because there are multiple possible segmentations of character sequences (Figure 2.4), all such methods make mistakes sometimes, and so you are never guaranteed a consistent unique tokenization. The other approach is to abandon word-based indexing and to do all indexing via just short subsequences of characters (character k-grams), regardless of

COMPOUNDS

COMPOUND SPLITTER

WORD SEGMENTATION

和尚

Figure 2.4 Ambiguities in Chinese word segmentation. The two characters can be treated as one word meaning "monk" or as a sequence of two words meaning "and" and "still."

whether particular sequences cross word boundaries or not. Three reasons why this approach is appealing are that an individual Chinese character is more like a syllable than a letter and usually has some semantic content, that most words are short (the commonest length is two characters), and that, given the lack of standardization of word breaking in the writing system, it is not always clear where word boundaries should be placed anyway. Even in English, some cases of where to put word boundaries are just orthographic conventions – think of *notwithstanding* versus *not to mention* or *into* versus *on to* – but people are educated to write the words with consistent use of spaces.

2.2.2 *Dropping common terms: stop words*

STOP WORDS

COLLECTION FREQUENCY

STOP LIST

Sometimes, some extremely common words that would appear to be of little value in helping select documents matching a user need are excluded from the vocabulary entirely. These words are called *stop words*. The general strategy for determining a stop list is to sort the terms by *collection frequency* (the total number of times each term appears in the document collection), and then to take the most frequent terms, often hand-filtered for their semantic content relative to the domain of the documents being indexed, as a *stop list*, the members of which are then discarded during indexing. An example of a stop list is shown in Figure 2.5. Using a stop list significantly reduces the number of postings that a system has to store; we present some statistics on this in Chapter 5 (see Table 5.1, page 80). And a lot of the time not indexing stop words does little harm: keyword searches with terms like the and by don't seem very useful. However, this is not true for phrase searches. The phrase query "President of the United States," which contains two stop words, is more precise than President AND "United States." The meaning of ights to London is likely to be lost if the word to is stopped out. A search for Vannevar Bush's article *As we may think* will be difficult if the first three words are stopped out, and the system searches simply for documents containing the word think. Some special query types are disproportionately affected. Some song titles and well-known pieces of verse consist entirely of words that are commonly on stop lists (*To be or not to be*, *Let It Be*, *I don't want to be*, ...).

a	an	and	are	as	at	be	by	for	from
has	he	in	is	it	its	of	on	that	the
to	was	were	will	with					

Figure 2.5 A stop list of twenty-five semantically nonselective words that are common in Reuters-RCV1.

The general trend in IR systems over time has been from standard use of quite large stop lists (200–300 terms) to very small stop lists (7–12 terms) to no stop list whatsoever. Web search engines generally do not use stop lists. Some of the design of modern IR systems has focused precisely on how we can exploit the statistics of language so as to be able to cope with common words in better ways. We show in Section 5.3 (page 87) how good compression techniques greatly reduce the cost of storing the postings for common words. Section 6.2.1 (page 108) then discusses how standard term weighting leads to very common words having little impact on document rankings. Finally, Section 7.1.5 (page 129) shows how an IR system with impact-sorted indexes can terminate scanning a postings list early when weights get small, and hence common words do not cause a large additional processing cost for the average query, even though postings lists for stop words are very long. So for most modern IR systems, the additional cost of including stop words is not that high – either in terms of index size or in terms of query processing time.

2.2.3 Normalization (equivalence classing of terms)

Having broken up our documents (and also our query) into tokens, the easy case is if tokens in the query just match tokens in the token list of the document. However, there are many cases when two character sequences are not quite the same but you would like a match to occur. For instance, if you search for *USA*, you might hope to also match documents containing *U.S.A.*

TOKEN NORMALIZATION

Token normalization is the process of canonicalizing tokens so that matches occur despite superficial differences in the character sequences of the tokens.[4] The most standard way to normalize is to implicitly create *equivalence classes*, which are normally named after one member of the set. For instance, if the tokens *anti-discriminatory* and *antidiscriminatory* are both mapped onto the term antidiscriminatory, in both the document text and queries, then searches for one term will retrieve documents that contain either.

EQUIVALENCE CLASSES

The advantage of just using mapping rules that remove characters like hyphens is that the equivalence classing to be done is implicit, rather than being fully calculated in advance: the terms that happen to become identical as the result of these rules are the equivalence classes. It is only easy to write rules of this sort that remove characters. Because the equivalence classes are implicit, it is not obvious when you might want to add characters. For instance, it would be hard to know to turn *antidiscriminatory* into *anti-discriminatory*.

An alternative to creating equivalence classes is to maintain relations between unnormalized tokens. This method can be extended to hand-constructed lists of synonyms such as *car* and *automobile*, a topic we discuss further in Chapter 9. These term relationships can be achieved in two

[4] It is also often referred to as *term normalization*, but we prefer to reserve the name *term* for the output of the normalization process.

Query term	**Terms in documents that should be matched**
Windows	Windows
windows	Windows, windows, window
window	window, windows

Figure 2.6 An example of how asymmetric expansion of query terms can usefully model users' expectations.

ways. The usual way is to index unnormalized tokens and to maintain a query expansion list of multiple vocabulary entries to consider for a certain query term. A query term is then effectively a disjunction of several postings lists. The alternative is to perform the expansion during index construction. When the document contains automobile, we index it under car as well (and, usually, also vice versa). Use of either of these methods is considerably less efficient than equivalence classing, because there are more postings to store and merge. The first method adds a query expansion dictionary and requires more processing at query time, whereas the second method requires more space for storing postings. Traditionally, expanding the space required for the postings lists was seen as more disadvantageous, but with modern storage costs, the increased flexibility that comes from distinct postings lists is appealing.

These approaches are more flexible than equivalence classes because the expansion lists can overlap while not being identical. This means there can be an asymmetry in expansion. An example of how such an asymmetry can be exploited is shown in Figure 2.6: if the user enters windows, we wish to allow matches with the capitalized *Windows* operating system, but this is not plausible if the user enters window, even though it is plausible for this query to also match lowercase *windows*.

The best amount of equivalence classing or query expansion to do is a fairly open question. Doing some definitely seems a good idea. But doing a lot can easily have unexpected consequences of broadening queries in unintended ways. For instance, equivalence-classing *U.S.A.* and *USA* to the latter by deleting periods from tokens might at first seem very reasonable, given the prevalent pattern of optional use of periods in acronyms. However, if I put in as my query term *C.A.T.*, I might be rather upset if it matches every appearance of the word *cat* in documents.[5]

Below we present some of the forms of normalization that are commonly employed and how they are implemented. In many cases they seem helpful, but they can also do harm. In fact, you can worry about many details of equivalence classing, but it often turns out that providing processing is done consistently to the query and to documents, the fine details may not have much aggregate effect on performance.

[5] At the time we wrote this chapter (August 2005), this was actually the case on Google: the top result for the query *C.A.T.* was a site about cats, the Cat Fanciers Web Site www.fanciers.com/.

Accents and diacritics. Diacritics on characters in English have a fairly marginal status, and we might well want *cliché* and *cliche* to match, or *naive* and *naïve*. This can be done by normalizing tokens to remove diacritics. In many other languages, diacritics are a regular part of the writing system and distinguish different sounds. Occasionally words are distinguished only by their accents. For instance, in Spanish, *peña* is "a cliff," whereas *pena* is "sorrow." Nevertheless, the important question is usually not prescriptive or linguistic, but is a question of how users are likely to write queries for these words. In many cases, users enter queries for words without diacritics, whether for reasons of speed, laziness, limited software, or habits born of the days when it was hard to use non-ASCII text on many computer systems. In these cases, it might be best to equate all words to a form without diacritics.

CASE-FOLDING **Capitalization/case-folding.** A common strategy is to do *case-folding* by reducing all letters to lower case. Often this is a good idea: it allows instances of *Automobile* at the beginning of a sentence to match with a query of *automobile*. It also helps on a web search engine when most of your users type in *ferrari* when they are interested in a *Ferrari* car. On the other hand, such case folding can equate words that might better be kept apart. Many proper nouns are derived from common nouns and so are distinguished only by case, including companies (*General Motors*, *The Associated Press*), government organizations (*the Fed* vs. *fed*) and person names (*Bush*, *Black*). We already mentioned an example of unintended query expansion with acronyms, which involved not only acronym normalization ($C.A.T. \rightarrow CAT$) but also case-folding ($CAT \rightarrow cat$).

For English, an alternative to making every token lowercase is to just make some tokens lowercase. The simplest heuristic is to convert to lowercase words at the beginning of a sentence and all words occurring in a title that is all uppercase or in which most or all words are capitalized. These words are usually ordinary words that have been capitalized. Midsentence capitalized words are left as capitalized (which is usually correct). This mostly avoids case-folding in cases where distinctions should be kept apart. The same task can be done more accurately by a machine learning sequence model that uses more features to make the decision of when to case-fold. This is known as *truecasing*. However, trying to get capitalization right in this way probably doesn't help if your users usually use lowercase regardless of the correct case of words. Thus, lowercasing everything often remains the most practical solution.

TRUECASING

Other issues in English. Other possible normalizations are quite idiosyncratic and particular to English. For instance, you might wish to equate *ne'er* and *never* or the British spelling *colour* and the American spelling *color*. Dates,

ノーベル平和賞を受賞したワンガリ・マータイさんが名誉会長を務めるＭＯＴＴＡＩＮＡＩキャンペーンの一環として、毎日新聞社とマガジンハウスは「私 の、もったいない」を募集します。皆様が日ごろ「もったいない」と感じて実践していることや、それにまつわるエピソードを８００字以内の文章にまとめ、簡 単な写真、イラスト、図などを添えて１０月２０日までにお送りください。大賞受賞者には、５０万円相当の旅行券とエコ製品２点の副賞が贈られます。

Figure 2.7 Japanese makes use of multiple intermingled writing systems and, like Chinese, does not segment words. This text is mainly Chinese characters with the hiragana syllabary for inflectional endings and function words. The part in latin letters is actually a Japanese expression, but has been taken up as the name of an environmental campaign by 2004 Nobel Peace Prize winner Wangari Maathai. His name is written using the katakana syllabary in the middle of the first line. The first four characters of the final line express a monetary amount that we would want to match with ¥500,000 (500,000 Japanese yen).

times, and similar items come in multiple formats, presenting additional challenges. You might wish to collapse together *3/12/91* and *Mar. 12, 1991*. However, correct processing here is complicated by the fact that in the United States, *3/12/91* is *Mar. 12, 1991*, whereas in Europe it is *3 Dec. 1991*.

Other languages. English has maintained a dominant position on the WWW; approximately 60% of web pages are in English (Gerrand 2007). But that still leaves 40% of the web, and the non-English portion might be expected to grow over time, because less than one third of Internet users and less than 10% of the world's population primarily speak English. And there are signs of change: Sifry (2007) reports that only about one third of blog posts are in English.

Other languages again present distinctive issues in equivalence classing. The French word for *the* has distinctive forms based not only on the gender (masculine or feminine) and number of the following noun, but also depending on whether the following word begins with a vowel: *le*, *la*, *l'*, *les*. We may well wish to equivalence class these various forms of *the*. German has a convention whereby vowels with an umlaut can be rendered instead as a two-vowel digraph. We would want to treat *Schütze* and *Schuetze* as equivalent.

Japanese is a well-known difficult writing system, as illustrated in Figure 2.7. Modern Japanese is standardly an intermingling of multiple alphabets, principally Chinese characters, two syllabaries (hiragana and katakana), and Western characters (Latin letters, Arabic numerals, and various symbols). Although there are strong conventions and standardization through the education system over the choice of writing system, in many cases the same word can be written with multiple writing systems. For example, a word may be written in katakana for empasis (somewhat like italics). Or a word may sometimes be written in hiragana and sometimes in Chinese characters. Successful retrieval thus requires complex equivalence classing

across the writing systems. In particular, an end user might commonly present a query entirely in hiragana, because it is easier to type, just as Western end users commonly use all lowercase letters.

Document collections being indexed can include documents from many different languages. Or a single document can easily contain text from multiple languages. For instance, a French email might quote clauses from a contract document written in English. Most commonly, the language is detected and language-particular tokenization and normalization rules are applied at a predetermined granularity, such as whole documents or individual paragraphs, but this still does not correctly deal with cases where language changes occur for brief quotations. When document collections contain multiple languages, a single index may have to contain terms of several languages. One option is to run a language identification classifier on documents and then to tag terms in the vocabulary for their language. Or this tagging can simply be omitted, because it is relatively rare for the exact same character sequence to be a word in different languages.

When dealing with foreign or complex words, particularly foreign names, the spelling may be unclear or there may be variant transliteration standards giving different spellings (e.g., *Chebyshev* and *Tchebycheff* or *Beijing* and *Peking*). One way of dealing with this is to use heuristics to equivalence class or expand terms with phonetic equivalents. The traditional and best known such algorithm is the Soundex algorithm, which we cover in Section 3.4 (page 58).

2.2.4 *Stemming and lemmatization*

For grammatical reasons, documents are going to use different forms of a word, such as *organize*, *organizes*, and *organizing*. Additionally, there are families of derivationally related words with similar meanings, such as *democracy*, *democratic*, and *democratization*. In many situations, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set.

The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. For instance:

am, are, is $\Rightarrow$ be
car, cars, car's, cars' $\Rightarrow$ car

The result of this mapping of text will be something like:

the boy's cars are different colors $\Rightarrow$
the boy car be differ color

STEMMING

However, the two words differ in their flavor. *Stemming* usually refers to a crude heuristic process that chops off the ends of words in the hope of

achieving this goal correctly most of the time, and often includes the removal of derivational affixes. *Lemmatization* usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the *lemma*. If confronted with the token *saw*, stemming might return just *s*, whereas lemmatization would attempt to return either *see* or *saw*, depending on whether the use of the token was as a verb or a noun. The two may also differ in that stemming most commonly collapses derivationally related words, whereas lemmatization commonly only collapses the different inflectional forms of a lemma. Linguistic processing for stemming or lemmatization is often done by an additional plug-in component to the indexing process, and a number of such components exist, both commercial and open source.

LEMMATIZATION

LEMMA

The most common algorithm for stemming English, and one that has repeatedly been shown to be empirically very effective, is *Porter's algorithm* (Porter 1980). The entire algorithm is too long and intricate to present here, but we indicate its general nature. Porter's algorithm consists of five phases of word reductions, applied sequentially. Within each phase, there are various conventions to select rules, such as selecting the rule from each rule group that applies to the longest suffix. In the first phase, this convention is used with the following rule group:

PORTER STEMMER

(2.1)

Rule			**Example**		
SSES	→	SS	caresses	→	caress
IES	→	I	ponies	→	poni
SS	→	SS	caress	→	caress
S	→		cats	→	cat

Many of the later rules use a concept of the *measure* of a word, which loosely checks the number of syllables to see whether a word is long enough that it is reasonable to regard the matching portion of a rule as a suffix rather than as part of the stem of a word. For example, the rule:

$(m > 1)$ EMENT →

would map *replacement* to *replac*, but not *cement* to *c*. The official site for the Porter Stemmer is:

www.tartarus.org/ martin/PorterStemmer/

Other stemmers exist, including the older, one-pass Lovins stemmer (Lovins 1968), and newer entrants like the Paice/Husk stemmer (Paice 1990); see:

www.cs.waikato.ac.nz/eibe/ stemmers/
www.comp.lancs.ac.uk/computing/research/stemming/

Sample text: Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

Lovins stemmer: such an analys can reve featur that ar not eas vis from th vari in th individu gen and can lead to a pictur of expres that is mor biolog transpar and acces to interpres

Porter stemmer: such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret

Paice stemmer: such an analys can rev feat that are not easy vis from the vary in the individ gen and can lead to a pict of express that is mor biolog transp and access to interpret

Figure 2.8 A comparison of three stemming algorithms on a sample text.

Figure 2.8 presents an informal comparison of the different behaviors of these stemmers. Stemmers use language-specific rules, but they require less knowledge than a lemmatizer, which needs a complete vocabulary and morphological analysis to accurately lemmatize words. Particular domains may also require special stemming rules. However, the exact stemmed form does not matter, only the equivalence classes it forms.

LEMMATIZER

Rather than using a stemmer, you can use a *lemmatizer*, a tool from Natural Language Processing, that does full morphological analysis to accurately identify the lemma for each word. Doing full morphological analysis produces at most very modest benefits for retrieval. It is hard to say more, because either form of normalization tends not to improve English information retrieval performance in aggregate – at least not by very much. Although it helps a lot for some queries, it equally hurts performance a lot for others. Stemming increases recall while harming precision. As an example of what can go wrong, note that the Porter stemmer stems all of the following words:

operate operating operates operation operative operatives operational

to oper. However, because *operate* in its various forms is a common verb, we would expect to lose considerable precision on queries such as the following with Porter stemming:

operational AND research
operating AND system
operative AND dentistry

For a case like this, moving to using a lemmatizer does not completely fix the problem because particular inflectional forms are used in particular collocations: a sentence with the words *operate* and *system* is not a good match for the query operating AND system. Getting better value from term normalization depends more on pragmatic issues of word use than on formal issues of linguistic morphology.

The situation is different for languages with much more morphology (such as Spanish, German, and Finnish). Results in the European CLEF evaluations have repeatedly shown quite large gains from the use of stemmers (and compound splitting for languages like German); see the references in Section 2.5.

?

Exercise 2.1 [⋆] Are the following statements true or false?

a. In a Boolean retrieval system, stemming never lowers precision.

b. In a Boolean retrieval system, stemming never lowers recall.

c. Stemming increases the size of the vocabulary.

d. Stemming should be invoked at indexing time, but not while processing a query.

Exercise 2.2 [⋆] Suggest what normalized form should be used for these words (including the word itself as a possibility):

a. ’Cos

b. Shi’ite

c. cont’d

d. Hawai’i

e. O’Rourke

Exercise 2.3 [⋆] The following pairs of words are stemmed to the same form by the Porter stemmer. Which pairs, would you argue, should not be conflated? Give your reasoning.

a. abandon/abandonment

b. absorbency/absorbent

c. marketing/markets

d. university/universe

e. volume/volumes

Exercise 2.4 [⋆] For the Porter stemmer rule group shown in (2.1):

a. What is the purpose of including an identity rule such as SS → SS?

b. Applying just this rule group, what will the following words be stemmed to?

circus *canaries* *boss*

c. What rule should be added to correctly stem *pony*?

d. The stemming for *ponies* and *pony* might seem strange. Does it have a deleterious effect on retrieval? Why or why not?

2.3 Faster postings list intersection via skip pointers

In the remainder of this chapter, we discuss extensions to postings list data structures and ways to increase the efficiency of using postings lists. Recall the basic postings list intersection operation from Section 1.3 (page 9): we walk through the two postings lists simultaneously, in time linear in the total number of postings entries. If the list lengths are m and n, the intersection

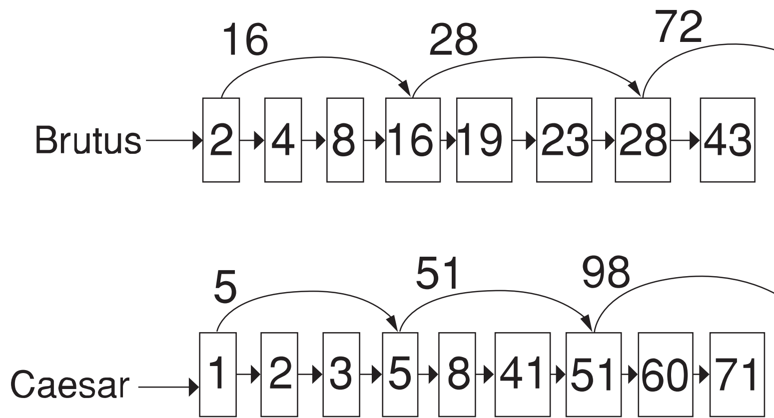


Figure 2.9 Postings lists with skip pointers. The postings intersection can use a skip pointer when the end point is still less than the item on the other list.

takes $O(m+n)$ operations. Can we do better than this? That is, empirically, can we usually process postings list intersection in sublinear time? We can, if the index isn't changing too fast.

SKIP LIST

One way to do this is to use a *skip list* by augmenting postings lists with skip pointers (at indexing time), as shown in Figure 2.9. Skip pointers are effectively shortcuts that allow us to avoid processing parts of the postings list that will not figure in the search results. The two questions are then where to place skip pointers and how to do efficient merging using skip pointers.

Consider first efficient merging, with Figure 2.9 as an example. Suppose we've stepped through the lists in the figure until we have matched 8 on each list and moved it to the results list. We advance both pointers, giving us 16 on the upper list and 41 on the lower list. The smallest item is then the element 16 on the top list. Rather than simply advancing the upper pointer, we first check the skip list pointer and note that 28 is also less than 41. Hence we can follow the skip list pointer, and then we advance the upper pointer to 28. We thus avoid stepping to 19 and 23 on the upper list. A number of variant versions of postings list intersection with skip pointers is possible depending on when exactly you check the skip pointer. One version is shown in Figure 2.10. Skip pointers will only be available for the original postings lists. For an intermediate result in a complex query, the call *hasSkip*(p) will always return false. Finally, note that the presence of skip pointers only helps for AND queries, not for OR queries.

Where do we place skips? There is a tradeoff. More skips means shorter skip spans, and that we are more likely to skip. But it also means lots of comparisons to skip pointers, and lots of space storing skip pointers. Fewer skips means few pointer comparisons, but then long skip spans, which means that there will be fewer opportunities to skip. A simple heuristic for placing skips, which has been found to work well in practice, is that for a postings list of length P, use $\sqrt{P}$ evenly spaced skip pointers. This heuristic can be improved upon; it ignores any details of the distribution of query terms.

```
IntersectWithSkips(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ nil and p2 ≠ nil
 3  do if docID(p1) = docID(p2)
 4        then Add(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8               then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
 9                       then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
10                              do p1 ← skip(p1)
11                       else p1 ← next(p1)
12               else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
13                       then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
14                              do p2 ← skip(p2)
15                       else p2 ← next(p2)
16  return answer
```

Figure 2.10 Postings lists intersection with skip pointers.

Building effective skip pointers is easy if an index is relatively static; it is harder if a postings list keeps changing because of updates. A malicious deletion strategy can render skip lists ineffective.

Choosing the optimal encoding for an inverted index is an ever-changing game for the system builder, because it is strongly dependent on underlying computer technologies and their relative speeds and sizes. Traditionally, CPUs were slow, and so highly compressed techniques were not optimal. Now CPUs are fast and disk is slow, so reducing disk postings list size dominates. However, if you're running a search engine with everything in memory, then the equation changes again. We discuss the impact of hardware parameters on index construction time in Section 4.1 (page 62) and the impact of index size on system speed in Chapter 5.

?

Exercise 2.5 [⋆] Why are skip pointers not useful for queries of the form x OR y?

Exercise 2.6 [⋆] We have a two-word query. For one term the postings list consists of the following 16 entries:

[4,6,10,12,14,16,18,20,22,32,47,81,120,122,157,180]

and for the other it is the one entry postings list:

[47].

Work out how many comparisons would be done to intersect the two postings lists with the following two strategies. Briefly justify your answers:

a. Using standard postings lists

b. Using postings lists stored with skip pointers, with the suggested skip length of $\sqrt{P}$.

Exercise 2.7 [⋆] Consider a postings intersection between this postings list, with skip pointers:

3 5 9 15 24 39 60 68 75 81 84 89 92 96 97 100 115

and the following intermediate result postings list (which hence has no skip pointers):

3 5 89 95 97 99 100 101

Trace through the postings intersection algorithm in Figure 2.10.

a. How often is a skip pointer followed (i.e., p_1 is advanced to $skip(p_1)$)?

b. How many postings comparisons will be made by this algorithm while intersecting the two lists?

c. How many postings comparisons would be made if the postings lists are intersected without the use of skip pointers?

2.4 Positional postings and phrase queries

Many complex or technical concepts and many organization and product names are multiword compounds or phrases. We would like to be able to pose a query such as Stanford University by treating it as a phrase so that a sentence in a document like *The inventor Stanford Ovshinsky never went to university* is not a match. Most recent search engines support a double quotes syntax ("stanford university") for *phrase queries*, which has proven to be very easily understood and successfully used by users. As many as 10% of web queries are phrase queries, and many more are implicit phrase queries (such as person names), entered without use of double quotes. To be able to support such queries, it is no longer sufficient for postings lists to be simply lists of documents that contain individual terms. In this section, we consider two approaches to supporting phrase queries and their combination. A search engine should not only support phrase queries, but implement them efficiently. A related but distinct concept is term proximity weighting, where a document is preferred to the extent that the query terms appear close to each other in the text. This technique is covered in Section 7.2.2 (page 132) in the context of ranked retrieval.

PHRASE QUERIES

2.4.1 *Biword indexes*

One approach to handling phrases is to consider every pair of consecutive terms in a document as a phrase. For example the text *Friends, Romans, Countrymen* would generate the *biwords*:

BIWORD INDEX

friends romans
romans countrymen

In this model, we treat each of these biwords as a vocabulary term. Being able to process two-word phrase queries is immediate. Longer phrases can

be processed by breaking them down. The query stanford university palo alto can be broken into the Boolean query on biwords:

"stanford university" AND "university palo" AND "palo alto"

This query could be expected to work fairly well in practice, but there can and will be occasional false positives. Without examining the documents, we cannot verify that the documents matching the above Boolean query do actually contain the original four-word phrase.

Among possible queries, nouns and noun phrases have a special status in describing the concepts people are interested in searching for. But related nouns can often be divided from each other by various function words, in phrases such as *the abolition of slavery* or *renegotiation of the constitution*. These needs can be incorporated into the biword indexing model in the following way. First, we tokenize the text and perform part-of-speech tagging.[6] We can then group terms into nouns, including proper nouns, (N) and function words, including articles and prepositions, (X), among other classes. Now deem any string of terms of the form NX*N to be an extended biword. Each such extended biword is made a term in the vocabulary. For example:

renegotiation	of	the	constitution
N	X	X	N

To process a query using such an extended biword index, we need to also parse it into *N*s and *X*s, and then segment the query into extended biwords, which can be looked up in the index.

This algorithm does not always work in an intuitively optimal manner when parsing longer queries into Boolean queries. Using the above algorithm, the query

cost overruns on a power plant

is parsed into

"cost overruns" AND "overruns power" AND "power plant"

whereas it might seem a better query to omit the middle biword. Better results can be obtained by using more precise part-of-speech patterns that define which extended biwords should be indexed.

The concept of a biword index can be extended to longer sequences of words, and if the index includes variable length word sequences, it is generally referred to as a *phrase index*. Indeed, searches for a single term are not naturally handled in a biword index (you would need to scan the dictionary for all biwords containing the term), and so we also need to have

PHRASE INDEX

[6] Part-of-speech taggers classify words as nouns, verbs, and so on – or, in practice, often as finer grained classes like "plural proper noun." Many fairly accurate (c. 96% per-tag accuracy) part-of-speech taggers now exist, usually trained by machine learning methods on hand-tagged text. See, for instance, Manning and Schütze (1999, ch. 10).

to, 993427:
⟨ 1, 6: ⟨7, 18, 33, 72, 86, 231⟩;
2, 5: ⟨1, 17, 74, 222, 255⟩;
4, 5: ⟨8, 16, 190, 429, 433⟩;
5, 2: ⟨363, 367⟩;
7, 3: ⟨13, 23, 191⟩; . . . ⟩

be, 178239:
⟨ 1, 2: ⟨17, 25⟩;
4, 5: ⟨17, 191, 291, 430, 434⟩;
5, 3: ⟨14, 19, 101⟩; . . . ⟩

Figure 2.11 Positional index example. The word to has a document frequency 993,477, and occurs six times in document 1 at positions 7, 18, 33, and so on.

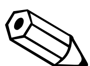
an index of single-word terms. Although there is always a chance of false-positive matches, the chance of a false-positive match on indexed phrases of length three or more becomes very small indeed. But on the other hand, storing longer phrases has the potential to greatly expand the vocabulary size. Maintaining exhaustive phrase indexes for phrases of length greater than two is a daunting prospect, and even use of an exhaustive biword dictionary greatly expands the size of the vocabulary. However, toward the end of this section, we discuss the utility of the strategy of using a partial phrase index in a compound indexing scheme.

2.4.2 Positional indexes

POSITIONAL INDEX

For the reasons given, a biword index is not the standard solution. Rather, a *positional index* is most commonly employed. Here, for each term in the vocabulary, we store postings of the form docID: ⟨position1, position2, . . . ⟩, as shown in Figure 2.11, where each position is a token index in the document. Each posting will also usually record the term frequency, for reasons discussed in Chapter 6.

To process a phrase query, you still need to access the inverted index entries for each distinct term. As before, you start with the least frequent term and then work to further restrict the list of possible candidates. In the merge operation, the same general technique is used as before, but rather than simply checking that both terms are in a document, you also need to check that their positions of appearance in the document are compatible with the phrase query being evaluated. This requires working out offsets between the words.



Example 2.1: Satisfying phrase queries. Suppose the postings lists for to and be are as in Figure 2.11, and the query is "to be or not to be." The postings lists to access are: to, be, or, not. We examine intersecting the postings lists for to and be. We first look for documents that contain both terms. Then,

```
PositionalIntersect(p1, p2, k)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ nil and p2 ≠ nil
 3  do if docID(p1) = docID(p2)
 4        then l ← ⟨ ⟩
 5              pp1 ← positions(p1)
 6              pp2 ← positions(p2)
 7              while pp1 ≠ nil
 8              do while pp2 ≠ nil
 9                 do if |pos(pp1) − pos(pp2)| ≤ k
10                       then Add(l, pos(pp2))
11                       else if pos(pp2) > pos(pp1)
12                               then break
13                    pp2 ← next(pp2)
14                 while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
15                 do Delete(l[0])
16                 for each ps ∈ l
17                 do Add(answer, ⟨docID(p1), pos(pp1), ps⟩)
18                 pp1 ← next(pp1)
19              p1 ← next(p1)
20              p2 ← next(p2)
21        else if docID(p1) < docID(p2)
22                then p1 ← next(p1)
23                else p2 ← next(p2)
24  return answer
```

Figure 2.12 An algorithm for proximity intersection of postings lists p_1 and p_2. The algorithm finds places where the two terms appear within k words of each other and returns a list of triples giving docID and the term position in p_1 and p_2.

we look for places in the lists where there is an occurrence of *be* with a token index one higher than a position of *to*, and then we look for another occurrence of each word with token index four higher than the first occurrence. In the above lists, the pattern of occurrences that is a possible match is:

to: ⟨...; 4:⟨...,429,433⟩; ...⟩
be: ⟨...; 4:⟨...,430,434⟩; ...⟩

The same general method is applied for within k word proximity searches, of the sort we saw in Example 1.1 (page 14):

employment /3 place

Here, $/k$ means "within k words of (on either side)." Clearly, positional indexes can be used for such queries; biword indexes cannot. We show in Figure 2.12 an algorithm for satisfying within k word proximity searches; it is further discussed in Exercise 2.12.

Positional index size. Adopting a positional index expands required postings storage significantly, even if we compress position values/offsets as we discuss in Section 5.3 (page 87). Indeed, moving to a positional index also changes the asymptotic complexity of a postings intersection operation, because the number of items to check is now bounded not by the number of documents but by the total number of tokens in the document collection T. That is, the complexity of a Boolean query is $\Theta(T)$ rather than $\Theta(N)$. However, most applications have little choice but to accept this, because most users now expect to have the functionality of phrase and proximity searches.

Let's examine the space implications of having a positional index. A posting now needs an entry for each occurrence of a term. The index size thus depends on the average document size. The average web page has fewer than 1,000 terms, but documents like SEC stock filings, books, and even some epic poems easily reach 100,000 terms. Consider a term with frequency 1 in 1,000 terms on average. The result is that large documents cause an increase of two orders of magnitude in the space required to store the postings list:

Document size	Expected postings	Expected entries in positional posting
1,000	1	1
100,000	1	100

Although the exact numbers depend on the type of documents and the language being indexed, some rough rules of thumb are to expect a positional index to be two to four times as large as a nonpositional index, and to expect a compressed positional index to be about one third to one half the size of the raw text (after removal of markup, etc.) of the original uncompressed documents. Specific numbers for an example collection are given in Table 5.1 (page 80) and Table 5.6 (page 95).

2.4.3 *Combination schemes*

The strategies of biword indexes and positional indexes can be fruitfully combined. If users commonly query on particular phrases, such as Michael Jackson, it is quite inefficient to keep merging positional postings lists. A combination strategy uses a phrase index, or just a biword index, for certain queries and uses a positional index for other phrase queries. Good queries to include in the phrase index are ones known to be common based on recent querying behavior. But this is not the only criterion: the most expensive phrase queries to evaluate are ones where the individual words are common but the desired phrase is comparatively rare. Adding *Britney Spears* as a phrase index entry may only give a speedup factor to that query of about

three, because most documents that mention either word are valid results, whereas adding *The Who* as a phrase index entry may speed up that query by a factor of 1,000. Hence, having the latter is more desirable, even if it is a relatively less common query.

Williams et al. (2004) evaluate an even more sophisticated scheme that uses indexes of both these sorts as well as a partial next word index as a halfway house between the first two strategies. For each term, a *next word index* records terms that follow it in a document. They conclude that such a strategy allows a typical mixture of web phrase queries to be completed in one quarter of the time taken by use of a positional index alone, while taking up 26% more space than use of a positional index alone.

NEXT WORD INDEX

?

Exercise 2.8 [⋆] Assume a biword index. Give an example of a document that will be returned for a query of New York University but is actually a false positive that should not be returned.

Exercise 2.9 [⋆] Shown below is a portion of a positional index in the format: term: doc1: ⟨position1, position2, . . . ⟩; doc2: ⟨position1, position2, . . . ⟩; etc.

angels: 2: ⟨36,174,252,651⟩; 4: ⟨12,22,102,432⟩; 7: ⟨17⟩;
fools: 2: ⟨1,17,74,222⟩; 4: ⟨8,78,108,458⟩; 7: ⟨3,13,23,193⟩;
fear: 2: ⟨87,704,722,901⟩; 4: ⟨13,43,113,433⟩; 7: ⟨18,328,528⟩;
in: 2: ⟨3,37,76,444,851⟩; 4: ⟨10,20,110,470,500⟩; 7: ⟨5,15,25,195⟩;
rush: 2: ⟨2,66,194,321,702⟩; 4: ⟨9,69,149,429,569⟩; 7: ⟨4,14,404⟩;
to: 2: ⟨47,86,234,999⟩; 4: ⟨14,24,774,944⟩; 7: ⟨199,319,599,709⟩;
tread: 2: ⟨57,94,333⟩; 4: ⟨15,35,155⟩; 7: ⟨20,320⟩;
where: 2: ⟨67,124,393,1001⟩; 4: ⟨11,41,101,421,431⟩; 7: ⟨16,36,736⟩;

Which document(s) if any match each of the following queries, where each expression within quotes is a phrase query?

a. "fools rush in"
b. "fools rush in" AND "angels fear to tread"

Exercise 2.10 [⋆] Consider the following fragment of a positional index with the format:

word: document: ⟨position, position, . . .⟩; document: ⟨position, . . .⟩
. . .
Gates: 1: ⟨3⟩; 2: ⟨6⟩; 3: ⟨2,17⟩; 4: ⟨1⟩;
IBM: 4: ⟨3⟩; 7: ⟨14⟩;
Microsoft: 1: ⟨1⟩; 2: ⟨1,21⟩; 3: ⟨3⟩; 5: ⟨16,22,51⟩;

The /*k* operator, word1 /*k* word2 finds occurrences of word1 within *k* words of word2 (on either side), where *k* is a positive integer argument. Thus $k = 1$ demands that word1 be adjacent to word2.

a. Describe the set of documents that satisfy the query Gates /2 Microsoft.

b. Describe each set of values for k for which the query Gates /k Microsoft returns a different set of documents as the answer.

Exercise 2.11 [⋆⋆] Consider the general procedure for merging two positional postings lists for a given document, to determine the document positions where a document satisfies a $/k$ clause (in general, there can be multiple positions at which each term occurs in a single document). We begin with a pointer to the position of occurrence of each term and move each pointer along the list of occurrences in the document, checking as we do so whether we have a hit for $/k$. Each move of either pointer counts as a step. Let L denote the total number of occurrences of the two terms in the document. What is the big-O complexity of the merge procedure, if we wish to have postings including positions in the result?

Exercise 2.12 [⋆⋆] Consider the adaptation of the basic algorithm for intersection of two postings lists (Figure 1.6, page 11) to the one in Figure 2.12 (page 39), which handles proximity queries. A naive algorithm for this operation could be $O(PL_{\max}{}^2)$, where P is the sum of the lengths of the postings lists (i.e., the sum of document frequencies) and $L_{\max}$ is the maximum length of a document (in tokens).

a. Go through this algorithm carefully and explain how it works.

b. What is the complexity of this algorithm? Justify your answer carefully.

c. For certain queries and data distributions, would another algorithm be more efficient? What complexity does it have?

Exercise 2.13 [⋆⋆] Suppose we wish to use a postings intersection procedure to determine simply the list of documents that satisfy a $/k$ clause, rather than returning the list of positions, as in Figure 2.12 (page 39). For simplicity, assume $k \geq 2$. Let L denote the total number of occurrences of the two terms in the document collection (i.e., the sum of their collection frequencies). Which of the following is true? Justify your answer.

a. The merge can be accomplished in a number of steps linear in L and independent of k, and we can ensure that each pointer moves only to the right.

b. The merge can be accomplished in a number of steps linear in L and independent of k, but a pointer may be forced to move nonmonotonically (i.e., to sometimes back up).

c. The merge can require kL steps in some cases.

Exercise 2.14 [⋆⋆] How could an IR system combine use of a positional index and use of stop words? What is the potential problem, and how could it be handled?

2.5 References and further reading

EAST ASIAN LANGUAGES

Exhaustive discussion of the character-level processing of East Asian languages can be found in Lunde (1998). Character bigram indexes are perhaps the most standard approach to indexing Chinese, although some systems use word segmentation. Due to differences in the language and writing system, word segmentation is most usual for Japanese (Luk and Kwok 2002; Kishida et al. 2005). The structure of a character k-gram index over unsegmented text differs from that in Section 3.2.2 (page 50): there the k-gram dictionary points to postings lists of entries in the regular dictionary, whereas here it points directly to document postings lists. For further discussion of Chinese word segmentation, see (Sproat et al. 1996; Sproat and Emerson 2003; Tseng et al. 2005; and Gao et al. 2005).

Lita et al. (2003) present a method for truecasing. Natural language processing work on computational morphology is presented in (Sproat 1992; Beesley and Karttunen 2003).

Language identification was perhaps first explored in cryptography; for example, Konheim (1981) presents a character-level k-gram language identification algorithm. Although other methods such as looking for particular distinctive function words and letter combinations have been used, with the advent of widespread digital text, many people have explored the character n-gram technique, and found it to be highly successful (Beesley 1998; Cavnar and Trenkle 1994; Dunning 1994). Written language identification is regarded as a fairly easy problem, whereas spoken language identification remains more difficult; see Hughes et al. (2006) for a recent survey.

Experiments on and discussion of the positive and negative impact of stemming in English can be found in the following works: Salton (1989), Harman (1991), Krovetz (1995), and Hull (1996). Hollink et al. (2004) provide detailed results for the effectiveness of language-specific methods on eight European languages. In terms of percent change in mean average precision (see page 147) over a baseline system, diacritic removal gains up to 23% (being especially helpful for Finnish, French, and Swedish). Stemming helped markedly for Finnish (30% improvement) and Spanish (10% improvement), but for most languages, including English, the gain from stemming was in the range 0–5%, and results from a lemmatizer were poorer still. Compound splitting gained 25% for Swedish and 15% for German, but only 4% for Dutch. Rather than language-particular methods, indexing character k-grams (as we suggested for Chinese) could often give as good or better results: using within-word character four-grams rather than words gave gains of 37% in Finnish, 27% in Swedish, and 20% in German, while even being slightly positive for other languages, such as Dutch, Spanish, and English. Tomlinson (2003) presents broadly similar results. Bar-Ilan and Gutman (2005) suggest that, at the time of their study (2003), the major commercial web search engines suffered from lacking decent language-particular

processing; for example, a query on www.google.fr for l'electricite did not separate off the article *l'* but only matched pages with precisely this string of article+noun.

SKIP LIST

The classic presentation of skip pointers for IR can be found in Moffat and Zobel (1996). Extended techniques are discussed in Boldi and Vigna (2005). The main paper in the algorithms literature is Pugh (1990), which uses multilevel skip pointers to give expected $O(\log P)$ list access (the same expected efficiency as using a tree data structure) with less implementational complexity. In practice, the effectiveness of using skip pointers depends on various system parameters. Moffat and Zobel (1996) report conjunctive queries running about five times faster with the use of skip pointers, but Bahle et al. (2002, p. 217) report that, with modern CPUs, using skip lists instead slows down search because it expands the size of the postings list (i.e., disk I/O dominates performance). In contrast, Strohman and Croft (2007) again show good performance gains from skipping, in a system architecture designed to optimize for the large memory spaces and multiple cores of recent CPUs.

Johnson et al. (2006) report that 11.7% of all queries in two 2002 web query logs contained phrase queries, although Kammenhuber et al. (2006) report only 3% phrase queries for a different data set. Silverstein et al. (1999) note that many queries without explicit phrase operators are actually implicit phrase searches.

3 *Dictionaries and tolerant retrieval*

In Chapters 1 and 2, we developed the ideas underlying inverted indexes for handling Boolean and proximity queries. Here, we develop techniques that are robust to typographical errors in the query, as well as alternative spellings. In Section 3.1, we develop data structures that help the search for terms in the vocabulary in an inverted index. In Section 3.2, we study the idea of a *wildcard query*: a query such as aeiou , which seeks documents containing any term that includes all the five vowels in sequence. The symbol indicates any (possibly empty) string of characters. Users pose such queries to a search engine when they are uncertain about how to spell a query term, or seek documents containing variants of a query term; for instance, the query automat seeks documents containing any of the terms automatic, automation, and automated.

WILDCARD QUERY

We then turn to other forms of imprecisely posed queries, focusing on spelling errors in Section 3.3. Users make spelling errors either by accident, or because the term they are searching for (e.g., Herman) has no unambiguous spelling in the collection. We detail a number of techniques for correcting spelling errors in queries, one term at a time as well as for an entire string of query terms. Finally, in Section 3.4 we study a method for seeking vocabulary terms that are phonetically close to the query term(s). This can be especially useful in cases like the Herman example, where the user may not know how a proper name is spelled in documents in the collection.

Because we develop many variants of inverted indexes in this chapter, we sometimes use the phrase *standard inverted index* to mean the inverted index developed in Chapters 1 and 2, in which each vocabulary term has a postings list with the documents in the collection.

3.1 Search structures for dictionaries

Given an inverted index and a query, our first task is to determine whether each query term exists in the vocabulary and, if so, identify the pointer to the

corresponding postings. This vocabulary lookup operation uses a classical data structure called the *dictionary* and has two broad classes of solutions: hashing and search trees. In the literature of data structures, the entries in the vocabulary (in our case, terms) are often referred to as *keys*. The choice of solution (hashing or search trees) is governed by a number of questions: (1) How many keys are we likely to have? (2) Is the number likely to remain static, or change a lot – and in the case of changes, are we likely to only have new keys inserted, or to also have some keys in the dictionary be deleted? (3) What are the relative frequencies with which various keys will be accessed?

Hashing has been used for dictionary lookup in some search engines. Each vocabulary term (key) is hashed into an integer over a large enough space that hash collisions are unlikely; collisions are resolved by auxiliary structures that can demand care to maintain.[1] At query time, we hash each query term separately and, following a pointer to the corresponding postings, taking into account any logic for resolving hash collisions. There is no easy way to find minor variants of a query term (such as the accented and unaccented versions of a word like resume), because these could be hashed to very different integers. In particular, we cannot seek (for instance) all terms beginning with the prefix automat, an operation that we require in Section 3.2. Finally, in a setting (such as the Web), where the size of the vocabulary keeps growing, a hash function designed for current needs may not suffice in a few years' time.

Search trees overcome many of these issues – for instance, they permit us to enumerate all vocabulary terms beginning with automat. The best-known search tree is the *binary tree*, in which each internal node has two children. The search for a term begins at the root of the tree. Each internal node (including the root) represents a binary test, based on whose outcome the search proceeds to one of the two subtrees below that node. Figure 3.1 gives an example of a binary search tree used for a dictionary. Efficient search (with a number of comparisons that is $O(\log M)$) hinges on the tree being balanced: the numbers of terms under the two subtrees of any node are either equal or differ by 1. The principal issue here is that of rebalancing; as terms are inserted into or deleted from the binary search tree, it needs to be rebalanced so that the balance property is maintained.

BINARY TREE

To mitigate rebalancing, one approach is to allow the number of subtrees under an internal node to vary in a fixed interval. A search tree commonly used for a dictionary is the *B-tree* – a search tree in which every internal node has a number of children in the interval $[a,b]$, where a and b are appropriate positive integers; Figure 3.2 shows an example with $a = 2$ and $b = 4$. Each branch under an internal node again represents a test for a range of character sequences, as in the binary tree example of Figure 3.1. A B-tree may

B-TREE

[1] So-called perfect hash functions are designed to preclude collisions, but are rather more complicated both to implement and to compute.

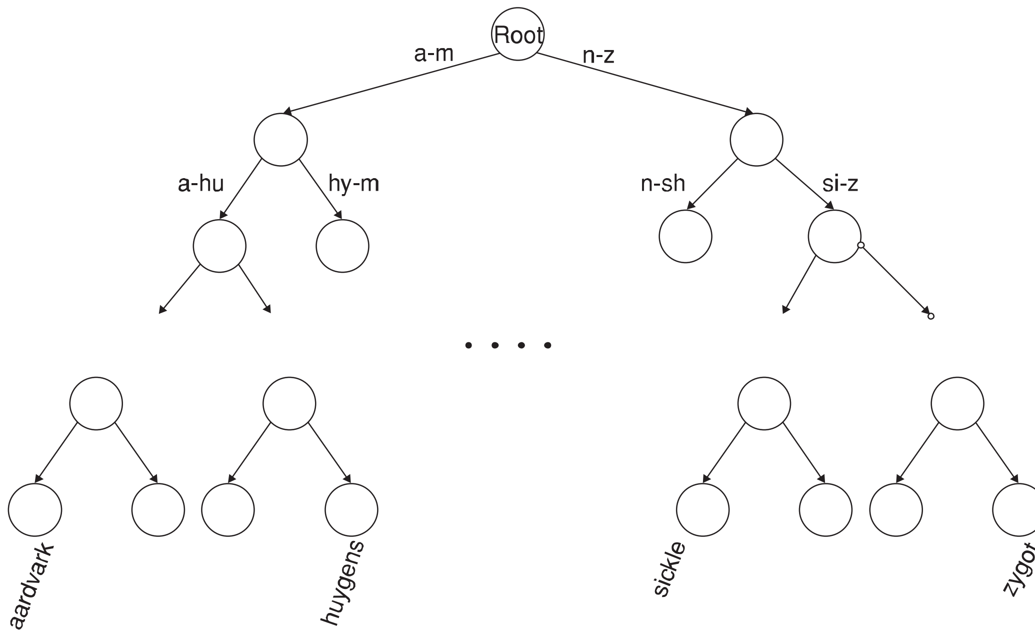


Figure 3.1 A binary search tree. In this example, the branch at the root partitions vocabulary terms into two subtrees, those whose first letter is between a and m, and the rest.

be viewed as "collapsing" multiple levels of the binary tree into one; this is especially advantageous when some of the dictionary is disk resident, in which case this collapsing serves the function of prefetching imminent binary tests. In such cases, the integers a and b are determined by the sizes of disk blocks. Section 3.5 contains pointers to further background on search trees and B-trees.

It should be noted that, unlike hashing, search trees demand that the characters used in the document collection have a prescribed ordering; for instance, the 26 letters of the English alphabet are always listed in the specific order A through . Some Asian languages such as Chinese do not always have a unique ordering, although by now all languages (including Chinese and Japanese) have adopted a standard ordering system for their character sets.

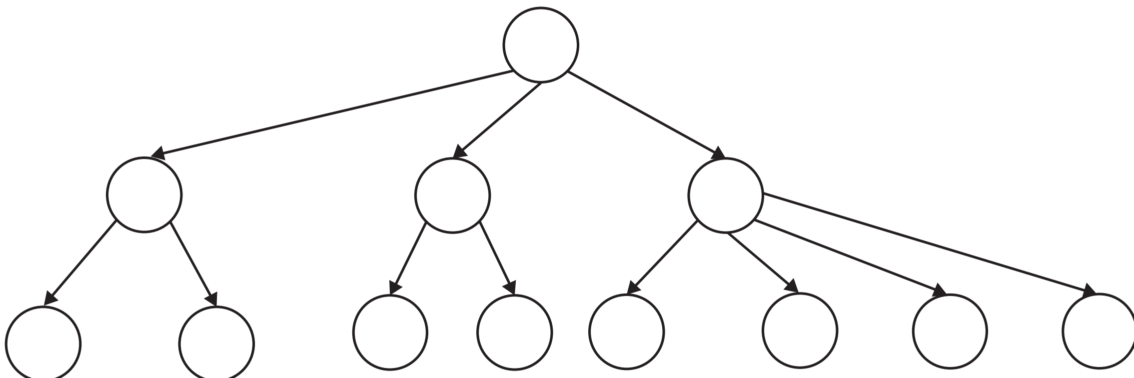


Figure 3.2 A B-tree. In this example every internal node has between 2 and 4 children.

3.2 Wildcard queries

Wildcard queries are used in any of the following situations: (1) the user is uncertain of the spelling of a query term (e.g., Sydney vs. Sidney, which leads to the wildcard query Sdne y); (2) the user is aware of multiple variants of spelling a term and (consciously) seeks documents containing any of the variants (e.g., color vs. colour); (3) the user seeks documents containing variants of a term that would be caught by stemming, but is unsure whether the search engine performs stemming (e.g., udicial vs. udiciar y, leading to the wildcard query udicia); or (4) the user is uncertain of the correct rendition of a foreign word or phrase (e.g., the query Universit Stuttgart).

WILDCARD QUERY

A query such as mon is known as a *trailing wildcard query*, because the symbol occurs only once, at the end of the search string. A search tree on the dictionary is a convenient way of handling trailing wildcard queries: we walk down the tree following the symbols m, o, and n in turn, at which point we can enumerate the set W of terms in the dictionary with the prefix mon. Finally, we use $|W|$ lookups on the standard inverted index to retrieve all documents containing any term in W.

But what about wildcard queries in which the symbol is not constrained to be at the end of the search string? Before handling this general case, we mention a slight generalization of trailing wildcard queries. First, consider *leading wildcard queries*, or queries of the form mon . Consider a *reverse B-tree* on the dictionary – one in which each root-to-leaf path of the B-tree corresponds to a term in the dictionary written *backwards:* thus, the term lemon would, in the B-tree, be represented by the path root-n-o-m-e-l. A walk down the reverse B-tree then enumerates all terms R in the vocabulary with a given prefix.

In fact, using a regular B-tree together with a reverse B-tree, we can handle an even more general case: wildcard queries in which there is a single symbol, such as semon . To do this, we use the regular B-tree to enumerate the set W of dictionary terms beginning with the prefix se and a non-empty suffix, then the reverse B-tree to enumerate the set R of terms ending with the suffix mon. Next, we take the intersection $W \cap R$ of these two sets, to arrive at the set of terms that begin with the prefix se and end with the suffix mon. Finally, we use the standard inverted index to retrieve all documents containing any terms in this intersection. We can thus handle wildcard queries that contain a single symbol using two B-trees, the normal B-tree and a reverse B-tree.

3.2.1 *General wildcard queries*

We now study two techniques for handling general wildcard queries. Both techniques share a common strategy: express the given wildcard query q_w as a Boolean query Q on a specially constructed index, such that the answer to

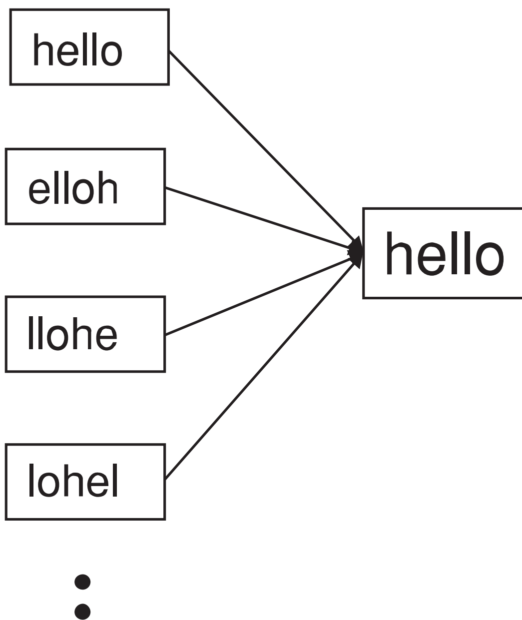


Figure 3.3 A portion of a permuterm index.

Q is a superset of the set of vocabulary terms matching q_w. Then, we check each term in the answer to Q against q_w, discarding those vocabulary terms that do not match q_w. At this point, we have the vocabulary terms matching q_w and can resort to the standard inverted index.

Permuterm indexes

PERMUTERM INDEX

Our first special index for general wildcard queries is the *permuterm index*, a form of inverted index. First, we introduce a special symbol into our character set, to mark the end of a term. Thus, the term hello is shown here as the augmented term hello . Next, we construct a permuterm index, in which the various rotations of each term (augmented with) all link to the original vocabulary term. Figure 3.3 gives an example of such a permuterm index entry for the term hello.

We refer to the set of rotated terms in the permuterm index as the *permuterm vocabulary*.

How does this index help us with wildcard queries? Consider the wildcard query mn . The key is to *rotate* such a wildcard query so that the symbol appears at the end of the string; thus, the rotated wildcard query becomes nm . Next, we look up this string in the permuterm index, where seeking nm (via a search tree) leads to rotations of (among others) the terms man and moron.

Now that the permuterm index enables us to identify the original vocabulary terms matching a wildcard query, we look up these terms in the standard inverted index to retrieve matching documents. We can thus handle any wildcard query with a single symbol. But what about a query such as moer ? In this case, we first enumerate the terms in the dictionary that are in the permuterm index of er . Not all such dictionary terms have the string

Figure 3.4 Example of a postings list in a 3-gram index. Here the 3-gram etr is illustrated. Matching vocabulary terms are lexicographically ordered in the postings.

mo in the middle – we filter these out by exhaustive enumeration, checking each candidate to see if it contains mo. In this example, the term shmonger would survive this filtering but lib uster would not. We then run the surviving terms through the standard inverted index for document retrieval. One disadvantage of the permuterm index is that its dictionary becomes quite large, including as it does all rotations of each term.

Notice the close interplay between the B-tree and the permuterm index above. Indeed, it suggests that the structure should perhaps be viewed as a permuterm B-tree. However, we follow traditional terminology here in describing the permuterm index as distinct from the B-tree that allows us to select the rotations with a given prefix.

3.2.2 *k-Gram indexes for wildcard queries*

Whereas the permuterm index is simple, it can lead to a considerable blowup from the number of rotations per term; for a dictionary of English terms, this can represent an almost tenfold space increase. We now present a second technique, known as the k-gram index, for processing wildcard queries. We also use k-gram indexes in Section 3.3.4. A *k-gram* is a sequence of k characters. Thus cas, ast and stl are all 3-grams occurring in the term castle. We use a special character to denote the beginning or end of a term, so the full set of 3-grams generated for castle is: ca , cas, ast, stl, tle, le .

k-GRAM INDEX

In a *k-gram index*, the dictionary contains all k-grams that occur in any term in the vocabulary. Each postings list points from a k-gram to all vocabulary terms containing that k-gram. For instance, the 3-gram etr would point to vocabulary terms such as metric and retrieval. An example is given in Figure 3.4.

How does such an index help us with wildcard queries? Consider the wildcard query rev e. We are seeking documents containing any term that begins with re and ends with ve. Accordingly, we run the Boolean query re AND VE .
This is looked up in the 3-gram index and yields a list of matching terms such as relive, remove, and retrieve. Each of these matching terms is then looked up in the standard inverted index to yield documents matching the query.

There is, however, a difficulty with the use of k-gram indexes that demands one further step of processing. Consider using the 3-gram index described for the query red . Following the process described, we first issue the Boolean query re AND RED to the 3-gram index. This leads to a match on terms such as retired, which contain the conjunction of the two 3-grams re and red, yet do not match the original wildcard query red .

To cope with this, we introduce a *postfiltering* step, in which the terms enumerated by the Boolean query on the 3-gram index are checked individually against the original query red . This is a simple string-matching operation and weeds out terms such as retired that do not match the original query. Terms that survive are then searched in the standard inverted index as usual.

We have seen that a wildcard query can result in multiple terms being enumerated, each of which becomes a single-term query on the standard inverted index. Search engines do allow the combination of wildcard queries using Boolean operators, for example, red AND fer i. What are the appropriate semantics for such a query? Because each wildcard query turns into a disjunction of single-term queries, the appropriate interpretation of this example is that we have a conjunction of disjunctions: we seek all documents that contain any term matching red *and* any term matching fe ri.

Even without Boolean combinations of wildcard queries, the processing of a wildcard query can be quite expensive, because of the added lookup in the special index, filtering, and finally the standard inverted index. A search engine may support such rich functionality, but most commonly, the capability is hidden behind an interface (say an "Advanced Query" interface) that most users never use. Exposing such functionality in the search interface often encourages users to invoke it even when they do not require it (say, by typing a prefix of their query followed by a *), increasing the processing load on the search engine.

?

Exercise 3.1 In the permuterm index, each permuterm vocabulary term points to the original vocabulary term(s) from which it was derived. How many original vocabulary terms can there be in the postings list of a permuterm vocabulary term?

Exercise 3.2 Write down the entries in the permuterm index dictionary that are generated by the term mama.

Exercise 3.3 If you wanted to search for sng in a permuterm wildcard index, what key(s) would one do the lookup on?

Exercise 3.4 Refer to Figure 3.4; it is pointed out in the caption that the vocabulary terms in the postings are lexicographically ordered. Why is this ordering useful?

Exercise 3.5 Consider again the query moer from Section 3.2.1. What Boolean query on a bigram index would be generated for this query? Can you think of a term that matches the permuterm query in Section 3.2.1, but does not satisfy this Boolean query?

Exercise 3.6 Give an example of a sentence that falsely matches the wildcard query monh if the search were to simply use a conjunction of bigrams.

3.3 Spelling correction

We next look at the problem of correcting spelling errors in queries. For instance, we may wish to retrieve documents containing the term carrot when the user types the query carot. Google reports (www.google.com/obs/ britney.html) that the following are all treated as misspellings of the query britney spears: britian spears, britney's spears, brandy spears, and prittany spears. We look at two steps to solving this problem: the first based on *edit distance* and the second based on *k-gram overlap*. Before getting into the algorithmic details of these methods, we first review how search engines provide spell correction as part of a user experience.

3.3.1 Implementing spelling correction

There are two basic principles underlying most spelling correction algorithms.

1. Of various alternative correct spellings for a misspelled query, choose the "nearest" one. This demands that we have a notion of nearness or proximity between a pair of queries. We develop these proximity measures in Section 3.3.3.
2. When two correctly spelled queries are tied (or nearly tied), select the one that is more common. For instance, grunt and grant both seem equally plausible as corrections for grnt. Then, the algorithm should choose the more common of grunt and grant as the correction. The simplest notion of more common is to consider the number of occurrences of the term in the collection; thus if grunt occurs more often than grant, it is the chosen correction. A different notion of more common is employed in many search engines, especially on the web. The idea is to use the correction that is most common among queries typed in by other users. The idea here is that if grunt is typed as a query more often than grant, then it is more likely that the user who typed grnt intended to type the query grunt.

Beginning in Section 3.3.3, we describe notions of proximity between queries, as well as their efficient computation. Spelling correction algorithms build on these computations of proximity; their functionality is then exposed to users in one of several ways:

1. On the query carot always retrieve documents containing carot as well as any "spell-corrected" version of carot, including carrot and tarot.
2. As in (1) above, but only when the query term carot is not in the dictionary.
3. As in (1) above, but only when the original query returned fewer than a preset number of documents (say fewer than five documents).

4. When the original query returns fewer than a preset number of documents, the search interface presents a *spelling suggestion* to the end user: this suggestion consists of the spell-corrected query term(s). Thus, the search engine might respond to the user: "Did you mean carrot?"

3.3.2 Forms of spelling correction

We focus on two specific forms of spelling correction that we refer to as *isolated-term* correction and *context-sensitive* correction. In isolated-term correction, we attempt to correct a single query term at a time – even when we have a multiple-term query. The carot example demonstrates this type of correction. Such isolated-term correction fails to detect, for instance, that the query e w form Heathrow contains a misspelling of the term from – because each term in the query is correctly spelled in isolation.

We begin by examining two techniques for addressing isolated-term correction: edit distance and k-gram overlap. We then proceed to context-sensitive correction.

3.3.3 Edit distance

EDIT DISTANCE Given two character strings s_1 and s_2, the *edit distance* between them is the minimum number of *edit operations* required to transform s_1 into s_2. Most commonly, the edit operations allowed for this purpose are (i) insert a character into a string, (ii) delete a character from a string, and (iii) replace a character of a string by another character; for these operations, edit distance is sometimes known as *Levenshtein distance*. For example, the edit distance between cat and dog is three. In fact, the notion of edit distance can be generalized to allowing different weights for different kinds of edit operations; for instance, a higher weight may be placed on replacing the character s by the character p, than on replacing it by the character a (the latter being closer to *s* on the keyboard). Setting weights in this way – depending on the likelihood of letters substituting for each other – is very effective in practice (see Section 3.4 for the separate issue of phonetic similarity). However, the remainder of our treatment here focus on the case in which all edit operations have the same weight.

LEVENSHTEIN DISTANCE

It is well-known how to compute the (weighted) edit distance between two strings in time $O(|s_1| \times |s_2|)$, where $|s_i|$ denotes the length of a string s_i. The idea is to use the dynamic programming algorithm in Figure 3.5, where the characters in s_1 and s_2 are given in array form. The algorithm fills the (integer) entries in a matrix m whose two dimensions equal the lengths of the two strings whose edit distances is being computed; the (i, j) entry of the matrix holds (after the algorithm is executed) the edit distance between

```
EditDistance(s1, s2)
 1  int m[|s1|, |s2|] = 0
 2  for i ← 1 to |s1|
 3  do m[i, 0] = i
 4  for j ← 1 to |s2|
 5  do m[0, j] = j
 6  for i ← 1 to |s1|
 7  do for j ← 1 to |s2|
 8      do m[i, j] = min{m[i − 1, j − 1] + if (s1[i] = s2[j]) then 0 else 1fi,
 9                       m[i − 1, j] + 1,
10                       m[i, j − 1] + 1}
11  return m[|s1|, |s2|]
```

Figure 3.5 Dynamic programming algorithm for computing the edit distance between strings s_1 and s_2.

the strings consisting of the first i characters of s_1 and the first j characters of s_2. The central dynamic programming step is depicted in lines 8–10 of Figure 3.5, where the three quantities whose minimum is taken correspond to substituting a character in s_1, inserting a character in s_1, and inserting a character in s_2.

Figure 3.6 shows an example Levenshtein distance computation of Figure 3.5. The typical cell $[i, j]$ has four entries formatted as a 2×2 cell. The lower right entry in each cell is the min of the other three, corresponding to the main dynamic programming step in Figure 3.5. The other three entries are the three entries $m[i-1, j-1] + 0$ or 1 depending on whether $s_1[i] = s_2[j]$, $m[i-1, j] + 1$ and $m[i, j-1] + 1$. The cells with numbers in italics depict the path by which we determine the Levenshtein distance.

			f		a		s		t	
		0	1	1	2	2	3	3	4	4
c		1	1	2	2	3	3	4	4	5
		1	2	1	2	2	3	3	4	4
a		2	2	2	1	3	3	4	4	5
		2	3	2	3	1	2	2	3	3
t		3	3	3	3	2	2	3	2	4
		3	4	3	4	2	3	2	3	2
s		4	4	4	4	3	2	3	3	3
		4	5	4	5	3	4	2	3	3

Figure 3.6 Example Levenshtein distance computation. The 2×2 cell in the $[i, j]$ entry of the table shows the three numbers whose minimum yields the fourth. The cells in italics determine the edit distance in this example.

The spelling correction problem however demands more than computing edit distance: given a set $\mathcal{S}$ of strings (corresponding to terms in the vocabulary) and a query string q, we seek the string(s) in V of least edit distance from q. We may view this as a decoding problem, in which the codewords (the strings in V) are prescribed in advance. The obvious way of doing this is to compute the edit distance from q to each string in V, before selecting the string(s) of minimum edit distance. This exhaustive search is inordinately expensive. Accordingly, a number of heuristics are used in practice to efficiently retrieve vocabulary terms likely to have low edit distance to the query term(s).

The simplest such heuristic is to restrict the search to dictionary terms beginning with the same letter as the query string; the hope is that spelling errors do not occur in the first character of the query. A more sophisticated variant of this heuristic is to use a version of the permuterm index, in which we omit the end-of-word symbol \$. Consider the set of all rotations of the query string q. For each rotation r from this set, we traverse the B-tree into the permuterm index, thereby retrieving all dictionary terms that have a rotation beginning with r. For instance, if q is mase and we consider the rotation $r =$ sema, we would retrieve dictionary terms such as semantic and semaphore, which do not have a small edit distance to q. Unfortunately, we would miss more pertinent dictionary terms such as mare and mane. To address this, we refine this rotation scheme: for each rotation, we omit a suffix of ℓ characters before performing the B-tree traversal. This ensures that each term in the set R of terms retrieved from the dictionary includes a "long" substring in common with q. The value of ℓ could depend on the length of q. Alternatively, we may set it to a fixed constant such as 2.

3.3.4 *k-Gram indexes for spelling correction*

To further limit the set of vocabulary terms for which we compute edit distances to the query term, we now show how to invoke the k-gram index of Section 3.2.2 (page 50) to assist with retrieving vocabulary terms with low edit distance to the query q. Once we retrieve such terms, we can then find the ones of least edit distance from q.

In fact, we use the k-gram index to retrieve vocabulary terms that have many k-grams in common with the query. We argue that, for reasonable definitions of "many k-grams in common," the retrieval process is essentially that of a single scan through the postings for the k-grams in the query string q.

The 2-gram (or *bigram*) index in Figure 3.7 shows (a portion of) the postings for the three bigrams in the query bord. Suppose we wanted to retrieve vocabulary terms that contained at least two of these three bigrams. A single scan of the postings (much as in Chapter 1) would let us enumerate all such

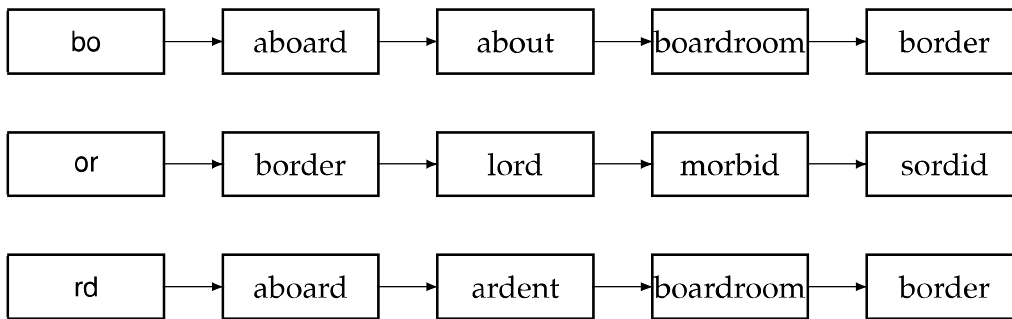


Figure 3.7 Matching at least two of the three 2-grams in the query bord.

terms; in the example of Figure 3.7, we would enumerate aboard, boardroom, and border.

This straightforward application of the linear scan intersection of postings immediately reveals the shortcoming of simply requiring matched vocabulary terms to contain a fixed number of k-grams from the query q: terms like boardroom, an implausible "correction" of bord, get enumerated. Consequently, we require more nuanced measures of the overlap in k-grams between a vocabulary term and q. The linear scan intersection can be adapted when the measure of overlap is the *Jaccard coefficient* for measuring the overlap between two sets A and B, defined to be $|A \cap B|/|A \cup B|$. The two sets we consider are the set of k-grams in the query q, and the set of k-grams in a vocabulary term. As the scan proceeds, we proceed from one vocabulary term t to the next, computing on the fly the Jaccard coefficient between q and t. If the coefficient exceeds a preset threshold, we add t to the output; if not, we move on to the next term in the postings. To compute the Jaccard coefficient, we need the set of k-grams in q and t.

JACCARD COEFFICIENT

Because we are scanning the postings for all k-grams in q, we immediately have these k-grams on hand. What about the k-grams of t? In principle, we could enumerate these on the fly from t. In practice, this is not only slow but potentially infeasible; in all likelihood, the postings entries themselves do not contain the complete string t but rather some encoding of t. The crucial observation is that to compute the Jaccard coefficient, we only need the length of the string t. To see this, recall the example of Figure 3.7 and consider the point when the postings scan for query $q =$ bord reaches term $t =$ boardroom. We know that two bigrams match. If the postings stored the (precomputed) number of bigrams in boardroom (namely, 8), we have all the information we require to compute the Jaccard coefficient to be $2/(8+3-2)$; the numerator is obtained from the number of postings hits (2, from bo and rd); the denominator is the sum of the number of bigrams in bord and boardroom, less the number of postings hits.

We could replace the Jaccard coefficient by other measures that allow efficient on the fly computation during postings scans. How do we use these for spelling correction? One method that has some empirical support is to first use the k-gram index to enumerate a set of candidate vocabulary terms

that are potential corrections of q. We then compute the edit distance from q to each term in this set, selecting terms from the set with small edit distance to q.

3.3.5 Context-sensitive spelling correction

Isolated-term correction would fail to correct typographical errors such as e w form Heathrow, where all three query terms are correctly spelled. When a phrase such as this retrieves few documents, a search engine may like to offer the corrected query e w from Heathrow. The simplest way to do this is to enumerate corrections of each of the three query terms (using the methods leading up to Section 3.3.4) even though each query term is correctly spelled, then try substitutions of each correction in the phrase. For the example e w form Heathrow, we enumerate such phrases as ed form Heathrow and e w fore Heathrow. For each such substitute phrase, the search engine runs the query and determines the number of matching results.

This enumeration can be expensive if we find many corrections of the individual terms; we could encounter a large number of combinations of alternatives. Several heuristics are used to trim this space. In this example, as we expand the alternatives for e w and form, we retain only the most frequent combinations in the collection or in the query logs, which contain previous queries by users. For instance, we would retain e w from as an alternative to try and extend to a three-term corrected query, but perhaps not ed fore or ea form. In this example, the biword ed fore is likely to be rare compared with the biword e w from. Then, we only attempt to extend the list of top biwords (such as e w from), to corrections of Heathrow. As an alternative to using the biword statistics in the collection, we may use the logs of queries issued by users; these could of course include queries with spelling errors.

?

Exercise 3.7 If $|s_i|$ denotes the length of string s_i, show that the edit distance between s_1 and s_2 is never more than $\max\{|s_1|, |s_2|\}$.

Exercise 3.8 Compute the edit distance between paris and alice. Write down the 5×5 array of distances between all prefixes as computed by the algorithm in Figure 3.5.

Exercise 3.9 Write pseudocode showing the details of computing on the fly the Jaccard coefficient while scanning the postings of the k-gram index, as mentioned on page 56.

Exercise 3.10 Compute the Jaccard coefficients between the query bord and each of the terms in Figure 3.7 that contain the bigram or.

Exercise 3.11 Consider the four-term query catched in the rye and suppose that each of the query terms has five alternative terms suggested by isolated-term correction. How many possible corrected phrases must we consider

if we do not trim the space of corrected phrases, but instead try all six variants for each of the terms?

Exercise 3.12 For each of the prefixes of the query – catched, catched in, and catched in the – we have a number of substitute prefixes arising from each term and its alternatives. Suppose that we were to retain only the top ten of these substitute prefixes, as measured by its number of occurrences in the collection. We eliminate the rest from consideration for extension to longer prefixes: thus, if batched in is not one of the ten most common two-term queries in the collection, we do not consider any extension of batched in as possibly leading to a correction of catched in the rye. How many of the possible substitute prefixes are we eliminating at each phase?

Exercise 3.13 Are we guaranteed that retaining and extending only the ten commonest substitute prefixes of catched in will lead to one of the ten commonest substitute prefixes of catched in the?

3.4 Phonetic correction

Our final technique for tolerant retrieval has to do with *phonetic* correction: misspellings that arise because the user types a query that sounds like the target term. Such algorithms are especially applicable to searches on the names of people. The main idea here is to generate, for each term, a "phonetic hash" so that similar-sounding terms hash to the same value. The idea owes its origins to work in international police departments from the early 20th century, seeking to match names for wanted criminals despite the names being spelled differently in different countries. It is mainly used to correct phonetic misspellings in proper nouns.

SOUNDEX ALGORITHMS

Algorithms for such phonetic hashing are commonly collectively known as *soundex algorithms*. However, there is an original soundex algorithm, with various variants, built on the following scheme:

1. Turn every term to be indexed into a four-character reduced form. Build an inverted index from these reduced forms to the original terms; call this the soundex index.
2. Do the same with query terms.
3. When the query calls for a soundex match, search this soundex index.

The variations in different soundex algorithms have to do with the conversion of terms to four-character forms. A commonly used conversion results in a four-character code, with the first character being a letter of the alphabet and the other three being digits between 0 and 9.

1. Retain the first letter of the term.
2. Change all occurrences of the following letters to '0' (zero): A, E, I, O, U, H, W, and Y.

3. Change letters to digits as follows:
   B, F, P, V to 1.
   C, G, J, K, Q, S, X, Z to 2.
   D, T to 3.
   L to 4.
   M, N to 5.
   R to 6.
4. Repeatedly remove one out of each pair of consecutive identical digits.
5. Remove all zeros from the resulting string. Pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits.

For an example of a soundex map, Hermann maps to H655. Given a query (say herman), we compute its soundex code and then retrieve all vocabulary terms matching this soundex code from the soundex index, before running the resulting query on the standard inverted index.

This algorithm rests on a few observations: (1) vowels are viewed as interchangeable, in transcribing names; (2) consonants with similar sounds (e.g., D and T) are put in equivalence classes. This leads to related names often having the same soundex codes. Although these rules work for many cases, especially European languages, such rules tend to be writing-system dependent. For example, Chinese names can be written in Wade-Giles or Pinyin transcription. Although soundex works for some of the differences in the two transcriptions – for instance, mapping both Wade-Giles hs and Pinyin x to 2 – it fails in other cases – for example, Wade-Giles and Pinyin r are mapped differently.

Exercise 3.14 Find two differently spelled proper nouns whose soundex codes are the same.

Exercise 3.15 Find two phonetically similar proper nouns whose soundex codes are different.

3.5 References and further reading

Knuth (1997) is a comprehensive source for information on search trees, including B-trees and their use in searching through dictionaries.

Garfield (1976) gives one of the first complete descriptions of the permuterm index. Ferragina and Venturini (2007) give an approach to addressing the space blowup in permuterm indexes.

One of the earliest formal treatments of spelling correction was due to Damerau (1964). The notion of edit distance that we have used is due to Levenshtein (1965) and the algorithm in Figure 3.5 is due to Wagner and Fischer (1974). Peterson (1980) and Kukich (1992) developed variants of methods based on edit distances, culminating in a detailed empirical study of several

methods by Zobel and Dart (1995), which shows that k-gram indexing is very effective for finding candidate mismatches, but should be combined with a more fine-grained technique such as edit distance to determine the most likely misspellings. Gusfield (1997) is a standard reference on string algorithms such as edit distance.

Probabilistic models ("noisy channel" models) for spelling correction were pioneered by Kernighan et al. (1990) and further developed by Brill and Moore (2000) and Toutanova and Moore (2002). In these models, the misspelled query is viewed as a probabilistic corruption of a correct query. They have a similar mathematical basis to the language model methods presented in Chapter 12, and also provide ways of incorporating phonetic similarity, closeness on the keyboard, and data from the actual spelling mistakes of users. Many would regard them as the state-of-the-art approach. Cucerzan and Brill (2004) show how this work can be extended to learning spelling correction models based on query reformulations in search engine logs.

The soundex algorithm is attributed to Margaret K. Odell and Robert C. Russelli (from U.S. patents granted in 1918 and 1922); the version described here draws on Bourne and Ford (1961). Zobel and Dart (1996) evaluate various phonetic matching algorithms, finding that a variant of the soundex algorithm performs poorly for general spelling correction, but that other algorithms based on the phonetic similarity of term pronunciations perform well.