2 The dictionary and postings lists

Recall the major steps in inverted index construction:

1. Collect the documents to be indexed.
2. Tokenize the text.
3. Linguistic preprocessing.
4. Indexing.

In this chapter we will first examine some of the substantive linguistic issues of tokenization and linguistic preprocessing, which determine which terms are indexed in the dictionary. Tokenization is the process of chopping character streams into *tokens*, while 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. In the second half of this chapter, we then start to examine extended postings list data structures that support faster querying and extended Boolean models, such as handling phrase and proximity queries.

TOKEN
TERM

2.1 Document delineation and character sequence decoding

2.1.1 Obtaining the character sequence in a document

Digital documents that are 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, or various national or vendor-specific standards. The correct encoding has to be determined. 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 by using provided document metadata. Once the encoding is determined, decoding to a character sequence has to be performed. The choice of encoding might be saved as it gives some evidence about what language the document is written in.

Even for text-format files, additional decoding may then need to be done. For instance, XML character entities such as `&` need to be decoded to give the & character. Alternatively, 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, the document format has to be determined, and then an appropriate decoder has to be used. Finally, the textual part of the document may need to be extracted out of other material that will not be processed. You might want to do this with XML files if the markup is going to be ignored; you would almost certainly want to do this with postscript or PDF files. We will not deal further with these issues here, and will assume henceforth that our documents are a list of characters. But commercial products can easily live or die by the range of document types and encodings that they support. Users want things to just work with their data as is. Often, they just see documents as text inside applications and are not even aware of how it is encoded on disk.

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

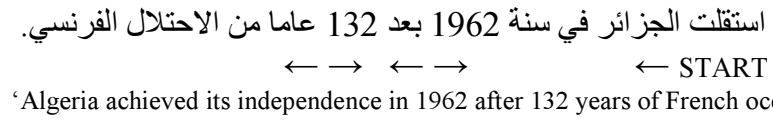
2.1.2 Choosing a document unit

DOCUMENT

The next phase is to determine what the *document* unit for indexing is. In the simplest case, each file in a document collection is 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 (a folder) in one file, but one 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, one might want to decode the zip file and regard each file it contains as a

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

► **Figure 2.1** Arabic script example. This is 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. Additionally, 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.



► **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, as shown in the Arabic example above. 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. You need at least to ensure that indexed text and query terms are representing characters in the same order.

separate document. Sometimes people index each paragraph of a document as a separate pseudo-document, because they believe it will be more helpful for retrieval to be returning small pieces of text so that the user can find the relevant sentences of a document more easily. Going in the opposite direction, various pieces of web software take things that you might regard as a single document (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. For now, we will assume that some suitable size unit has been chosen, perhaps together with an appropriate way of dividing or aggregating files. See Chapter 10 for discussion of simultaneously indexing documents at multiple levels of granularity.

2.2 Determining dictionary terms

2.2.1 Tokenization

TOKENS
TOKEN
TYPE

Given a character sequence and a defined document unit, tokenization is the job of chopping it up into pieces, called *tokens*, perhaps at the same time throwing away certain characters, such as punctuation. 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 character sequence in some particular document. A *type* is the class of all tokens containing the same character sequence. A term is a (perhaps normalized) type that is indexed in the IR system's dictionary. Here is an example of tokenization:

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

Each such token is now a candidate for an index entry, after further processing is done, which we describe later in the chapter.

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

APOSTROPHE

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 ?

The last looks intuitively bad. For all of them, the choices determine which Boolean queries will match. A query of neill AND capital will match 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 terms. This can be done by processing queries with the same tokenizer that is used for documents.[2] 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 clearly language-specific. It thus requires the language of the document to be known. *Language identification* based on classifiers that look at character-sequence level features is highly effective; most languages have distinctive signature patterns.

For most languages and 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 T.V. show name such as M*A*S*H – which is sufficiently integrated into popular culture that one finds 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 (jblack@mail.yahoo.com), web URLs (http://stuff.big.com/new/specials.html), numeric IP addresses (142.32.48.231), package tracking numbers (1Z9999W99845399981), and more. One possible solution is to omit from indexing tokens such as monetary amounts, numbers, and URLs, since their presence greatly expands the size of the dictionary. However, this comes at a large cost in restricting what people can search for. For instance, people might want to search in a bug database for something like a line number where an error occurs. For items such as dates, if they have a clear semantic type, such as the date of an email, then they are often indexed separately as document metadata.

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 kind of unclear. Handling hyphens automatically can thus be complex: it can either be done as a classification problem, or more commonly by some heuristic rules, such as allowing short hyphenated prefixes on words, but

2. For the free-text case, this is straightforward. The Boolean case is more complex: if terms in the input are tokenized into multiple tokens, this can be represented by combining them with an AND or as a phrase query, but it is harder to correct things if the user has separated terms that were tokenized together in the document processing.

3. This tokenization may result in multiple words from one query word, which is unproblematic in the free text case, and which in the Boolean case should be regarded as terms joined by an AND or as a multiterm phrase query, which we discuss in Section 2.3.2.

not longer hyphenated forms. One effective strategy in practice is to encourage users to enter hyphens wherever they may be possible, and then for, say, *over-eager* to search for over-eager OR over eager OR overeager.

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*). Another case with internal spaces that one might wish to regard as a single token includes 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 non-separating 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 later in the chapter), 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, as discussed later in this chapter. Also getting the first case right would depend on knowing that it is sometimes written as two words and also indexing it in this way. For some Boolean retrieval systems, such as WestLaw and Lexis-Nexis, if you search using the hyphenated form, then it will generalize the query to cover all three of the one word, hyphenated, and two word forms, but 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 will affect 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 extend the problem space in new ways. German writes *compound nouns* without spaces (e.g., *Computerlinguistik* 'computational linguistics'; *Lebensversicherungsgesellschaftsangestellter* 'life insurance company employee'). 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 dictionary and taking the longest dictionary 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. All such

COMPOUNDS

WORD SEGMENTATION

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** Chinese text example. The figure shows the standard unsegmented form of Chinese text (using the simplified characters that are standard in mainland China). There is no whitespace between words, nor 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 show arabic numerals and punctuation occurring to break up the Chinese characters, but there are still no spaces between words.

a and as at be by for from has he
in is it its of on s said that the to
was will with year

► **Figure 2.4** A stop list of 25 words common in Reuters-RCV1.

methods make mistakes sometimes, and so one is 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 whether particular sequences cross word boundaries or not. Two 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, 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* vs. *not to mention* or *into* vs. *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

Sometimes, some extremely common and semantically non-selective words are excluded from the dictionary entirely. These words are called *stop words*. The general strategy for determining a stop list is to sort the terms by frequency, 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 2.4. Using a stop list significantly reduces the number of postings that a system has to store: we will present

STOP LIST

some statistics on this in Chapter 5 (see Table 5.1, page 68). 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 flights 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 usually on stop lists (*To be or not to be*, *Let It Be*, *I don't want to be*, ...).

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 often 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 will show in Chapter 5 how good compression techniques greatly reduce the cost of storing the postings for common words. And we will discuss in Section 7.2.1 (page 112) how an IR system with impact-sorted indexes can terminate scanning a postings list early when weights get small, and hence it does not incur a large additional cost on 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 big – neither in terms of index size nor in terms of query processing time.

2.2.3 Normalization (equivalence classing of terms)

Having cut our documents (and also our query, if it is a free text 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 things 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

EQUIVALENCE CLASSES

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 latter, in both the document text and queries, then searches for one term will retrieve documents that contain either.

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, windows

► **Figure 2.5** An example of how asymmetric expansion of query terms can usefully model users' expectations.

The advantage to doing things by just having mapping rules that remove things 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 stuff. Since the equivalence classes are implicit, it is not obvious when you might want to add things. For instance, it would be hard to know to turn *antidiscriminatory* into *antidiscriminatory*.

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 ways. The usual way is to index unnormalized tokens and to maintain a query expansion list of multiple dictionary 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, as one has more postings to store and merge. The first method adds a query expansion dictionary and requires more processing at query time, while the second method requires more space for storing postings. In most circumstances, expanding the space required for the postings lists is seen as more disadvantageous.

However, 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 2.5: 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 other forms of normalization that are commonly employed and how they are implemented. In many cases they seem helpful, but they can also do harm. One 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.

Accents and diacritics. Diacritics on characters in English have a fairly marginal status, and one might well want *resume* and *résumé* 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. 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 will 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 useful to equate all words to a form without diacritics.

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 will allow instances of *Automobile* at the beginning of a sentence to match with a query of *automobile*. It will also help 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, this case folding can equate things 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 lowercasing every token is to just lowercase some tokens. The simplest heuristic is to lowercase words at the beginning of sentences, which are usually ordinary words that have been capitalized,

5. At the time we wrote this chapter (Aug. 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 http://www.fanciers.com/.

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 2.6** Example of Japanese text. As well as not segmenting words like Chinese, Japanese makes use of multiple writing systems, intermingled together. The 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 shows a monetary amount that one would like to match with ¥500,000 (500,000 Japanese yen).

and also everything in a title that is all uppercase or in title case where non-function words are all capitalized, leaving mid-sentence capitalized words as capitalized (which is usually correct). This will mostly avoid case-folding in cases where distinctions should be kept apart. The same task can be done more accurately by a machine learning sequence model which uses more features to make the decision of when to case-fold. This is known as *truecasing*. However, trying to get things 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, times and similar items come in multiple formats, presenting additional challenges. You might wish to collapse together *3/12/91* and *Mar. 12, 1991*.

Other languages. 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. In such cases one may wish to equivalence class these various forms of *the*: *le*, *la*, *l'*, *les*.

Japanese is a well-known difficult writing system, as illustrated in Figure 2.6. Modern Japanese is standardly an intermingling of multiple alphabets, principally Chinese characters, the two syllabaries (hiragana and katakana) and western characters (Latin letters, Arabic numerals, and various symbols). While 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 emphasis (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.

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 will 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 dictionary for their language. Or this tagging can simply be omitted, since collisions across languages are relatively rare.

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 (for example, *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 3.3 (page 49).

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 ⇒ be
car, cars, car's, cars' ⇒ car

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

the boy's cars are different colors ⇒
the boy car be differ color

STEMMING

LEMMATIZATION

However, the two terms differ in their flavor. *Stemming* suggests a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, whereas *lemmatization* suggests doing this properly with the use of a dictionary and morphological analysis of words, normally aiming to return the base or dictionary form of a word. 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 at one of these levels is often supplied as additional plug-in components to the indexing process, and a number of such components exist, both commercial and open-source.

PORTER STEMMER

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 will indicate its general nature. Porter's algorithm consists of 5 phases of word reductions, applied sequentially. Within each phase there are various conventions to select rules, such as selecting the one that applies to the longest suffix. In the first phase, this convention is used with the following rules:

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

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

$(m > 1)$ EMENT →

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 2.7** A comparison of three stemming algorithms on a sample text.

would map *replacement* to *replac*, but not *cement* to *c*.

The canonical site for the Porter Stemmer (Porter 1980) is:

http://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:

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

Figure 2.7 presents an informal comparison of the different behavior of these stemmers. Stemmers use language-particular rules, but they require less knowledge than a complete dictionary and morphological analysis to correctly 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, one can use a *lemmatizer*, a tool from Natural Language Processing which does full morphological analysis and can 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 information retrieval performance in aggregate – at least not by very much. While 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, since *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 would not fix things: particular inflectional forms are used in particular collocations.

2.3 Postings lists, revisited

In the remainder of this chapter, we will discuss extensions to postings list structure and ways to increase the efficiency of using postings lists.

2.3.1 Faster postings merges: Skip pointers

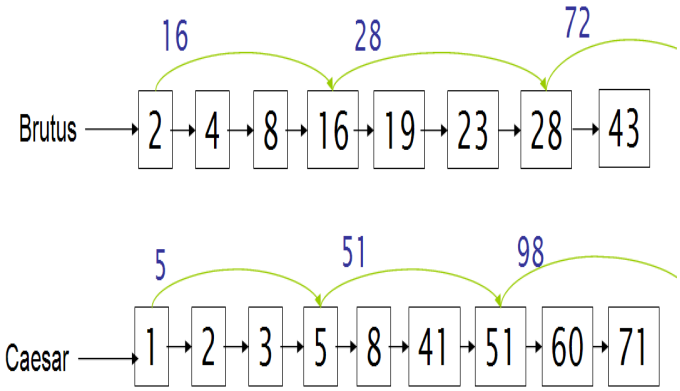
Recall the basic postings list merge operation from Chapter 1, shown in Figure 1.6: we walk through the two postings lists simultaneously, in time linear in the total number of postings entries. If the list lengths are m and n, the merge takes $O(m+n)$ operations. Can we do better than this? That is, empirically, can we usually process postings list merges 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 2.8. 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 that takes into account skip pointers.

Consider first efficient merging, with Figure 2.8 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 avoid stepping to 19 and 23 on the upper list.

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



► **Figure 2.8** Postings lists with skip pointers. The postings merge can use a skip pointer when the end point is still less than the item on the other list.

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.

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).

2.3.2 Phrase queries

PHRASE QUERIES

Many complex or technical concepts and many organization and product names are multiword compounds or other 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. Around 10% of web queries are now 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. An important criterion is that not only should phrase queries be possible, but they should also be efficient. 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 7.3.3 (page 116) in the context of ranked retrieval.

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:

friends romans
romans countrymen

In this model, we treat each of these biwords as a dictionary term. The ability to do two-word-phrase query processing 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

One would expect this query 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 4 word phrase.

PHRASE INDEX

The concept of a biword index can be extended to longer sequences of words, and if it includes variable length word sequences, it is generally referred to as a *phrase index*. Maintaining exhaustive phrase indexes for phrases of length greater than two is a daunting prospect, but towards the end of this section we discuss the utility of partial phrase indexes in compound indexing schemes.

For many text indexing purposes, nouns 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 (POST).[6] We can then bucket terms into (say) 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 dictionary. For example:

renegotiation	of	the	constitution
N	X	X	N

6. Part of speech taggers classify words as nouns, verbs, etc. – or, in practice, often as finer-grained classes like "plural proper noun". Many fairly accurate (c. 96% 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).

⟨be, 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367; . . . ⟩

► **Figure 2.9** Positional index example. For example, be has total collection frequency 993,477, and occurs in document 1 at positions 7, 18, 33, etc.

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.

Some issues with using biword or extended biword indexes include:

- Parsing longer queries into Boolean queries. From the algorithm that we have so far, the query

  tangerine trees and marmalade skies

  is parsed into

  tangerine trees AND trees marmalade AND marmalade skies

  Which does not seem a fully desirable outcome.
- False positives. As noted before, index matches will not necessarily contain longer queries.
- Index blowup. Using a biword dictionary has the result of greatly expanding the dictionary size.
- No natural treatment of individual terms. Individual term searches are not naturally handled in a biword index, and so a single word index must be present in addition.

Positional indexes

POSITIONAL INDEX

For the reasons given, biword indexes are not the standard solution. Rather, *positional indexes* are most commonly employed. Here, for each term in the dictionary, we store in the postings lists entries of the form docID: position1, position2, . . . , as shown in Figure 2.9. As discussed in Chapter 5, one can compress position values/offsets substantially. Nevertheless, use of a positional index expands required postings storage significantly.

To process a phrase query such as to be or not to be, you extract inverted index entries for each distinct term: to, be, or, not. As before, you would

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 documents are on a postings list, 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. Suppose the two postings lists for to and be are:

to: 993427; 2:5[1,17,74,222,551]; 4:5[8,16,190,429,433]; 7:3[13,23,191]; . . .
be: 178239; 1:2[17,19]; 4:5[17,191,291,430,434]; 5:3[14,19,101]; . . .

We first look for documents that contain both terms. Then, 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 4 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 Figure 1.9 (page 13):

LIMIT! /3 STATUTE /3 FEDERAL /2 TORT

Here, $/k$ means "within k words of". Clearly, positional indexes can be used for such queries; biword indexes cannot. More is still needed to handle "within the same sentence" or "within the same paragraph" proximity searches. We discuss issues of document zones in Chapter 6.

Positional index size. Even if we compress position values/offsets as we discuss in Chapter 5, adopting a positional index expands postings storage requirements substantially. However, most applications have little choice but to accept this since most users, in either Boolean or free text query interfaces, now expect to have the functionality of phrase and proximity searches.

Let's work out the space implications of having a positional index. We need an entry for each occurrence of a term, not just one per document in which it occurs. The index size thus depends on the average document size. The average web page has less than 1000 terms, but things like SEC stock filings, books, and even some epic poems easily reach 100,000 terms. Consider a term with frequency 1 in 1000 terms on average. The result is that large documents cause a two orders of magnitude increase in the postings list size:

Document size	Postings	Positional postings
1000	1	1
100,000	1	100