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 *a*e*i*o*u*, 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* would seek 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 will develop many variants of inverted indexes in this chapter, we will use sometimes 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 if any 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 non-accented versions of a word like resume), since 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 will require below 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.

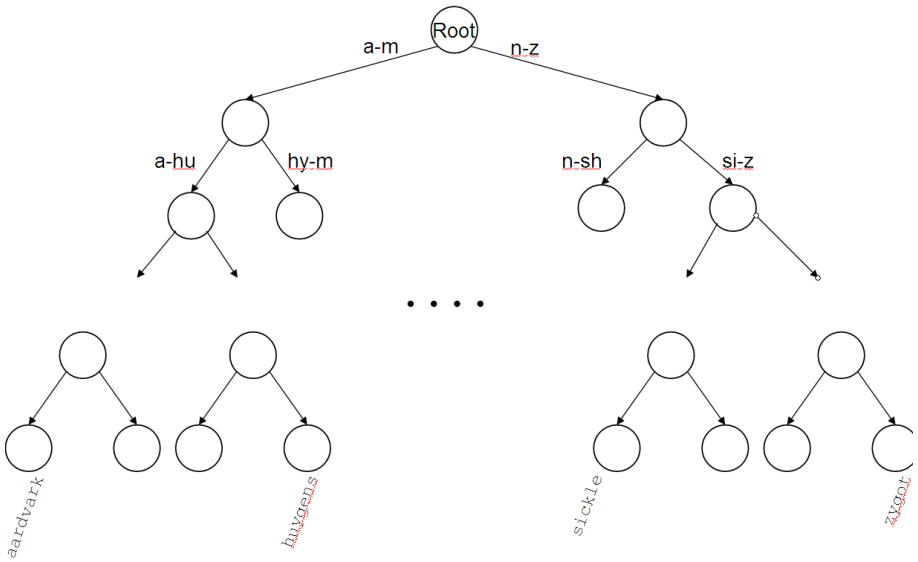
Search trees overcome many of these issues – for instance, they permit us to enumerate all vocabulary terms beginning with automat. The best-known search tree is the *binary tree*, in which each internal node has two children. The search for a term begins at the root of the tree. Each internal node (including the root) represents a binary test, based on whose outcome the search proceeds to one of the two sub-trees 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 sub-trees of any node are either equal or differ by one. 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 sub-trees 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 char-

B-TREE

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



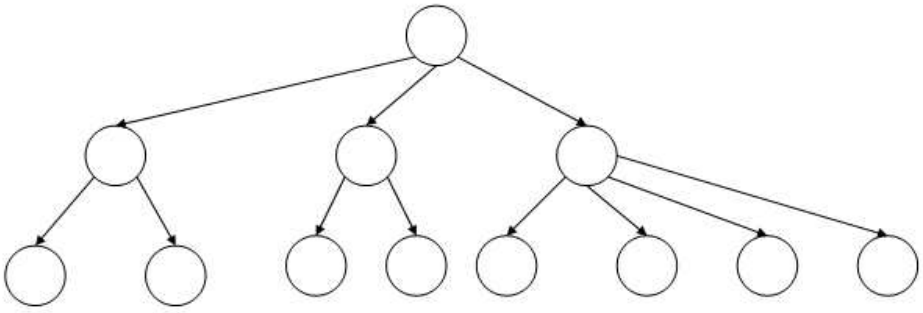
► **Figure 3.1** A binary search tree. In this example the branch at the root partitions vocabulary terms into two subtrees, those whose first letter is between a and m, and the rest.

acter sequences, as in the binary tree example of Figure 3.1. A B-tree may 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 pre-fetching 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 Z. 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.

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



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

leads to the wildcard query S*dney); (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., judicial vs. judiciary, leading to the wildcard query judicia*); (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 se*mon. To do this, we use the regular B-tree to enumerate the set W of dictionary terms beginning with the prefix se, 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 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

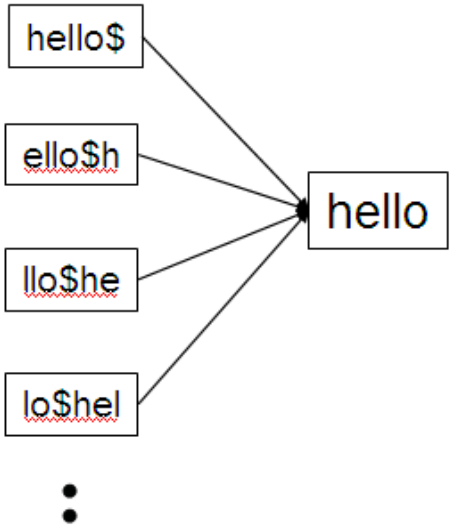
PERMUTERM INDEX

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 m*n. 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 n$m*. Next, we look up this string in the permuterm index, where seeking n$m* (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 fi*mo*er? In this case we first enumerate the terms in the dictionary that are in the permuterm index of er$fi*. Not all such dictionary terms will have the string 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 fishmonger would survive this filtering but filibuster would not. We then



► **Figure 3.3** A portion of a permuterm index.

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 ten-fold space increase. We now present a second technique, known as the *k*-gram index, for processing wildcard queries. We will 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

► **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.

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 re*ve. 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 above for the query red*. Following the process described above, 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 *post-filtering* 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, re*d AND fe*ri. What is the appropriate semantics for such a query? Since 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 re*d *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 s*ng 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 fi*mo*er 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 mon*h 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 (http://www.google.com/jobs/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 mis-spelled query, choose the "nearest" one. This demands that we have a notion of nearness or proximity between a pair of queries. We will 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 would be 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 would fail to detect, for instance, that the query flew form Heathrow contains a mis-spelling 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 3. 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 will 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 will hold (after the algorithm is executed) the edit distance between 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 $s_1[i] =$

```
EDITDISTANCE(s1, s2)
 1  int m[i, j] = 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.

		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.

$s_2[j]$, $m[i-1, j]+1$ and $m[i, j-1]+1$. The cells with numbers in italics depict the path by which we determine the Levenshtein distance.

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 would be 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 that 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.

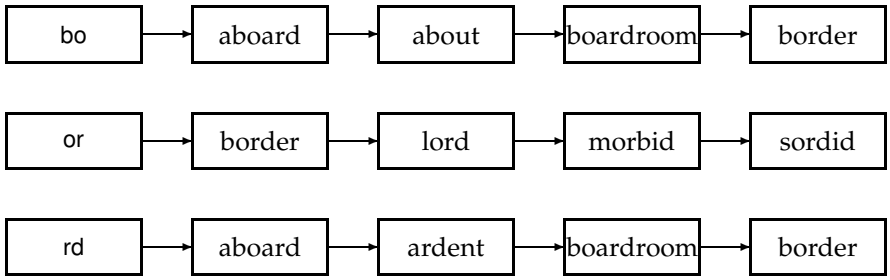
3.3.4 *k*-gram indexes for spelling correction

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 54) 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 will use the k-gram index to retrieve vocabulary terms that have many k-grams in common with the query. We will 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 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. Conse-



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

JACCARD COEFFICIENT

quently, 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.

Since 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 since, 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 (pre-computed) 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) while 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 flew 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 flew 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 flew form Heathrow, we enumerate such phrases as fled form Heathrow and flew 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, since we could encounter a large number of combinations of alternatives. Several heuristics are used to trim this space. In the example above, as we expand the alternatives for flew 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 flew from as an alternative to try and extend to a three-term corrected query, but perhaps not fled fore or flea form. In this example, the biword fled fore is likely to be rare compared to the biword flew from. Then, we only attempt to extend the list of top biwords (such as flew 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 61.

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 10 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 10 most common 2-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 10 commonest substitute prefixes of catched in will lead to one of the 10 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 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 4-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 4-character forms. A commonly used conversion results in a 4-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', '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. While 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. While 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 j 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.