3 *Tolerant retrieval*

In Chapters 1 and 2 we developed the ideas underlying inverted indexes for handling Boolean and proximity queries. Here we begin by studying *wildcard queries*: 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. We then turn to other forms of imprecisely posed queries, focusing on spelling errors. Users make spelling errors either by accident, or because the term they are searching for (e.g., Chebyshev) has no unambiguous spelling in the collection.

3.1 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 S*dney); (2) the user is aware of multiple variants of rendering a term and 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 (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 the inverted index to retrieve all documents containing any term in W. Typically, the search tree data structure most suited for such applications is a *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. For instance when the index is partially disk-resident, the integers a and b

Preliminary draft (c) 2007 Cambridge UP

are determined by the sizes of disk blocks. Section 3.4 contains pointers to further background on search trees and B-trees.

But what about wildcard queries in which the * symbol is not constrained to be at the end of the search string? Before handling the 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 dictionary with a given suffix.

In fact, using a regular together with a reverse B-tree, we can handle an even more general case: wildcard queries in which there's 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 dictionary 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 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.

3.1.1 General wildcard queries

We now study two techniques for handling general wildcard queries. Both techniques share a common strategy: express the given wildcard query w as a Boolean query Q on a specially constructed index, such that the answer to Q is a superset of the set of dictionary terms matching w. Then, we check each term in the answer to Q against w, discarding those dictionary terms in the answer that do not match w. At this point we have the dictionary terms matching 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 represented as hello$. Next, we construct a permuterm index, in which the dictionary consists of all rotations of each term (with the $ terminating symbol appended). The postings for each rotation consist of all dictionary terms containing that rotation. Figure 3.1 gives an example of such a permuterm index entry for a rotation of the term hello.

llo$he	→	hello
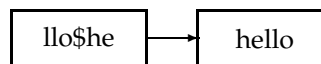
> ▶ **Figure 3.1**   Example of an entry in the permuterm index.

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

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$m*. Next, we look up this string in the permuterm index, where the entry n$m* points to the terms man and men. What of longer terms matching this wildcard query, such as moron? The permuterm index contains six rotations of moron, including n$moro. Now, n$m is a prefix of n$moro. Thus, when we traverse the B-tree into the permuterm dictionary seeking n$m, we find n$moro in the sub-tree, pointing into the original lexicon term moron.

Exercise 3.1

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

Now that the permuterm index enables us to identify the original lexicon 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 lexicon that are in the permuterm index of er$fi*. Not all such lexicon terms will have the string mo in the middle - we filter these out by exhaustive enumeration, checking

Preliminary draft (c) 2007 Cambridge UP
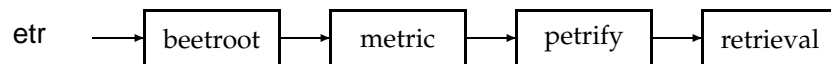
▶ **Figure 3.2** Example of a postings list in a 3-gram index. Here the 3-gram etr is illustrated.

each candidate to see if it contains mo. In this example, the term fishmonger would survive this filtering but filibuster would not. We then run the surviving terms through the regular 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.

3.1.2 *k*-gram indexes

We now present a second technique, known as the *k*-gram index, for processing wildcard queries. 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   A *k-gram index* is an index in which the dictionary consists of all *k*-grams that occur in any term in the lexicon. Each postings list points from a *k*-gram to all lexicon terms containing that *k*-gram. For instance, the 3-gram etr would point to lexicon terms such as metric and retrieval. An example is given in Figure 3.2.

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 inverted index to yield documents matching the query.

There is however a difficulty with the use of *k*-gram indexes, that demands

Preliminary draft (c) 2007 Cambridge UP

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 run against the inverted index as usual.

Exercise 3.4

Consider again the query fi*mo*er from Section 3.1.1. What Boolean query on a bigram index would be generated for this query? Can you think of a term that meets this Boolean query but does not satisfy the permuterm query in Section 3.1.1?

Exercise 3.5

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.

We have seen that a wildcard query can result in multiple terms being enumerated, each of which becomes a single-term query on the 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. 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. Surfacing such functionality often encourages users to invoke it, increasing the processing load on the engine.

3.2 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. Or, Google reports (http://www.google.com/jobs/britney.html) that the following are all observed misspellings of the query britney spears: britian spears, britney's spears, brandy spears and prittany spears. We look at two approaches to this problem: the first based on *edit distance* and the second

Preliminary draft (c) 2007 Cambridge UP

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.

3.2.1   Implementing spelling correction

Search engines implement this feature 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 absent from the dictionary.

3. As in (1) above, but only when the original query (in this case carot) 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 *spell suggestion* to the end user: this suggestion consists of the spell-corrected query term(s).

3.2.2   Forms of spell correction

We focus on two specific forms of spell 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 methods for isolated-term correction: edit distance, and *k*-gram overlap. We then proceed to context-sensitive correction.

3.2.3   Edit distance

EDIT DISTANCE
LEVENSHTEIN
DISTANCE

Given two character strings S_1 and S_2, the *edit distance* (also known as *Levenshtein 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 example, the edit distance between cat and dog is 3. In fact, the notion of edit distance can be generalized to allowing varying 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

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

m[0,0] = 0
m[i,0] = i, i=1..|s1|
m[0,j] = j, j=1..|s2|

m[i,j] = min(m[i-1,j-1]
             + if s1[i]=s2[j] then 0 else 1 fi,
             m[i-1, j] + 1,
             m[i, j-1] + 1 ), i=1..|s1|, j=1..|s2|
```

▶ **Figure 3.3** Dynamic programming algorithm for computing the edit distance between strings s1 and s2.

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 3.3 for consideration of phonetic similarity). However, the remainder of our treatment here will focus on the case in which all edit operations have the same weight.

Exercise 3.6

If $|S|$ denotes the length of string S, show that the edit distance between S_1 and S_2 is never more than $\max\{|S_1|, |S_2|\}$.

It is well-known how to compute the (weighted) edit distance between two strings in time $O(|S_1| * |S_2|)$, where $|S|$ denotes the length of a string S. The idea is to use the dynamic programming algorithm in Figure 3.3.

The spell correction problem however is somewhat different from that of computing edit distance: given a set $\mathcal{S}$ of strings (corresponding to terms in the dictionary) and a query string q, we seek the string(s) in $\mathcal{S}$ of least edit distance from q. We may view this as a decoding problem, but one in which the codewords (the strings in $\mathcal{S}$) are prescribed in advance. The obvious way of doing this is to compute the edit distance from q to each string in $\mathcal{S}$, 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 dictionary terms likely to have low edit distance to the query q.

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 the permuterm index, omitting 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. 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 dictionary terms retrieved 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.

Exercise 3.7

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 3.3.

Exercise 3.8

Show that if the query term q is edit distance 1 from a dictionary term t, then the set R includes t.
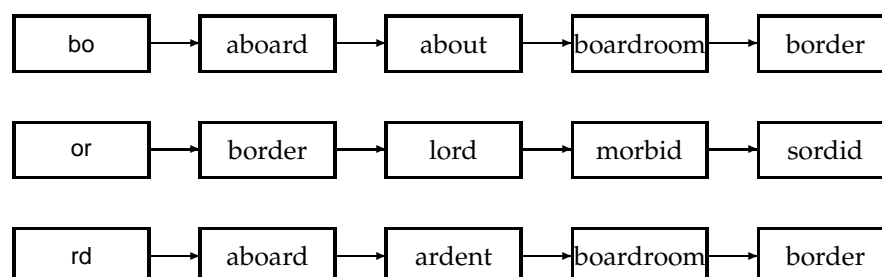
3.2.4   *k*-grom indexes

k-GRAM INDEX

To further limit the set of dictionary terms for which we compute edit distances to the query term, we now shows invoke the *k-gram index* of Section 3.1.2 to assist with retrieving dictionary 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 dictionary 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 3.4 shows (a portion of) the postings for the three bigrams in the query bord. Suppose we wanted to retrieve dictionary 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 3.4 we would enumerate aboard, boardroom and border.

This straightforward application of the linear scan merge of postings immediately reveals the shortcoming of simply requiring matched dictionary 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 dictionary term and q. The linear scan merge can be adapted when the measure

JACCARD COEFFICIENT

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

| bo | → | aboard | → | about | → | boardroom | → | border |

| or | → | border | → | lord | → | morbid | → | sordid |

| rd | → | aboard | → | ardent | → | boardroom | → | border |

► **Figure 3.4** Matching at least two of the three 2-grams in the query bord.

of k-grams in the query q, and the set of k-grams in a dictionary term. As the scan proceeds, we proceed from one dictionary term to the next, computing on the fly the Jaccard coefficient between the query q and a dictionary term 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.

Exercise 3.9
Compute the Jaccard coefficients between the query bord and each of the terms in Figure 3.4 that contain the bigram or.

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 an integer encoding of t. The crucial observation is that we only need the length of the string t, to compute the Jaccard coefficient. To see this, recall the example of Figure 3.4 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. For the Jaccard coefficient is $2/(8 + 3 - 2)$; 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

spell correction? One method that has some empirical support is to first use the *k*-gram index to enumerate a set of candidate dictionary 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*.

3.2.5 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 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 above) 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 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. 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. The choice of these alternatives is governed by the relative frequencies of biwords occurring in the collection: 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) in the first two terms, 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 3.10

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 3.11

For each of the prefixes of the query — thus, 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?

Preliminary draft (c) 2007 Cambridge UP

Exercise 3.12

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?

3.3 Phonetic correction

Our final technique for tolerant retrieval has to do with *phonetic* correction: misspellings that arise because the user types a term 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.

Preliminary draft (c) 2007 Cambridge UP