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 in- stance, the query automat* would seek documents containing any of the terms automatic, automation and automated.

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 BINARY TREE 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.

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 B-TREE 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-

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