7 Computing scores in a complete search system

Chapter 6 developed the theory underlying term weighting in documents for the purposes of scoring, leading up to vector space models and the basic cosine scoring algorithm of Section 6.3.3 (page 114). In this chapter, we begin in Section 7.1 with heuristics for speeding up this computation; many of these heuristics achieve their speed at the risk of not finding quite the top K documents matching the query. Some of these heuristics generalize beyond cosine scoring. With Section 7.1 in place, we have essentially all the components needed for a complete search engine. We therefore take a step back from cosine scoring, to the more general problem of computing scores in a search engine. In Section 7.2, we outline a complete search engine, including indexes and structures to support not only cosine scoring, but also more general ranking factors such as query term proximity. We describe how all of the various pieces fit together in Section 7.2.4. We conclude this chapter with Section 7.3, where we discuss how the vector space model for free text queries interacts with common query operators.

7.1 Efficient scoring and ranking

We begin by recapping the algorithm of Figure 6.14. For a query such as $q =$ jealous gossip, two observations are immediate:

1. The unit vector $\vec{v}(q)$ has only two nonzero components.
2. In the absence of any weighting for query terms, these nonzero components are equal – in this case, both equal 0.707.

For the purpose of ranking the documents matching this query, we are really interested in the relative (rather than absolute) scores of the documents in the collection. To this end, it suffices to compute the cosine similarity from each document unit vector $\vec{v}(d)$ to $\vec{V}(q)$ (in which all nonzero components of the query vector are set to 1), rather than to the unit vector $\vec{v}(q)$. For any two

FastCosineScore(q)
1 float $Scores[N] = 0$
2 **for each** d
3 **do** Initialize $Length[d]$ to the length of doc d
4 **for each** query term t
5 **do** calculate $w_{t,q}$ and fetch postings list for t
6 **for each** pair(d, $tf_{t,d}$) in postings list
7 **do** add $wf_{t,d}$ to $Scores[d]$
8 Read the array $Length[d]$
9 **for each** d
10 **do** Divide $Scores[d]$ by $Length[d]$
11 **return** Top K components of $Scores[]$

Figure 7.1 A faster algorithm for vector space scores.

documents d_1, d_2

$$\vec{V}(q) \cdot \vec{v}(d_1) > \vec{V}(q) \cdot \vec{v}(d_2) \Leftrightarrow \vec{v}(q) \cdot \vec{v}(d_1) > \vec{v}(q) \cdot \vec{v}(d_2). \tag{7.1}$$

For any document d, the cosine similarity $\vec{V}(q) \cdot \vec{v}(d)$ is the weighted sum, over all terms in the query q, of the weights of those terms in d. This in turn can be computed by a postings intersection exactly as in the algorithm of Figure 6.14, with line 8 altered because we take $w_{t,q}$ to be 1 so that the multiply-add in that step becomes just an addition; the result is shown in Figure 7.1. We walk through the postings in the inverted index for the terms in q, accumulating the total score for each document – very much as in processing a Boolean query, except we assign a positive score to each document that appears in any of the postings being traversed. As mentioned in Section 6.3.3, we maintain an idf value for each dictionary term and a tf value for each postings entry. This scheme computes a score for every document in the postings of any of the query terms; the total number of such documents may be considerably smaller than N.

Given these scores, the final step before presenting results to a user is to pick out the K highest-scoring documents. Although one could sort the complete set of scores, a better approach is to use a heap to retrieve only the top K documents in order. Where J is the number of documents with nonzero cosine scores, constructing such a heap can be performed in $2J$ comparison steps, following which each of the K highest scoring documents can be "read off" the heap with $\log J$ comparison steps.

7.1.1 Inexact top K document retrieval

Thus far, we have focused on retrieving precisely the K highest-scoring documents for a query. We now consider schemes by which we produce K documents that are *likely* to be among the K highest scoring documents for a

query. In doing so, we hope to dramatically lower the cost of computing the K documents we output, without materially altering the user's perceived relevance of the top K results. Consequently, in most applications it suffices to retrieve K documents whose scores are very close to those of the K best. In the sections that follow, we detail schemes that retrieve K such documents while potentially avoiding computing scores for most of the N documents in the collection.

Such inexact top K retrieval is not necessarily, from the user's perspective, a bad thing. The top K documents by the cosine measure are in any case not necessarily the K best for the query: cosine similarity is only a proxy for the user's perceived relevance. In Sections 7.1.2 through 7.1.6, we give heuristics, using which we are likely to retrieve K documents with cosine scores close to those of the top K documents. The principal cost in computing the output stems from computing cosine similarities between the query and a large number of documents. Having a large number of documents in contention also increases the selection cost in the final stage of culling the top K documents from a heap. We now consider a series of ideas designed to eliminate a large number of documents without computing their cosine scores. The heuristics have the following two-step scheme:

1. Find a set A of documents that are contenders, where $K < |A| \ll N$. A does not necessarily contain the K top-scoring documents for the query, but is likely to have many documents with scores near those of the top K.
2. Return the K top-scoring documents in A.

From the descriptions of these ideas, it will be clear that many of them require parameters to be tuned to the collection and application at hand; pointers to experience in setting these parameters may be found at the end of this chapter. It should also be noted that most of these heuristics are well-suited to free text queries, but not for Boolean or phrase queries.

7.1.2 Index elimination

For a multiterm query q, it is clear we only consider documents containing at least one of the query terms. We can take this a step further using additional heuristics:

1. We only consider documents containing terms whose idf exceeds a preset threshold. Thus, in the postings traversal, we only traverse the postings for terms with high idf. This has a fairly significant benefit: The postings lists of low-idf terms are generally long; with these removed from contention, the set of documents for which we compute cosines is greatly reduced. One way of viewing this heuristic: Low-idf terms are treated as stop words and do not contribute to scoring. For instance, on the query

catcher in the rye, we only traverse the postings for catcher and rye. The cutoff threshold can of course be adapted in a query-dependent manner.

2. We only consider documents that contain many (and as a special case, all) of the query terms. This can be accomplished during the postings traversal; we only compute scores for documents containing all (or many) of the query terms. A danger of this scheme is that by requiring all (or even many) query terms to be present in a document before considering it for cosine computation, we may end up with fewer than K candidate documents in the output. This issue is discussed further in Section 7.2.1.

7.1.3 Champion lists

The idea of *champion lists* (sometimes also called *fancy lists* or *top docs*) is to precompute, for each term t in the dictionary, the set of the r documents with the highest weights for t; the value of r is chosen in advance. For tf–idf weighting, these are the r documents with the highest tf values for term t. We call this set of r documents the *champion list* for term t.

Now, given a query q we create a set A as follows: We take the union of the champion lists for each of the terms comprising q. We now restrict cosine computation to only the documents in A. A critical parameter in this scheme is the value r, which is highly application dependent. Intuitively, r should be large compared with K, especially if we use any form of the index elimination described in Section 7.1.2. One issue here is that the value r is set at the time of index construction, whereas K is application dependent and may not be available until the query is received; as a result, we may (as in the case of index elimination) find ourselves with a set A that has fewer than K documents. There is no reason to have the same value of r for all terms in the dictionary; it could for instance be set to be higher for rarer terms.

7.1.4 Static quality scores and ordering

STATIC QUALITY SCORES

We now further develop the idea of champion lists, in the somewhat more general setting of *static quality scores*. In many search engines, we have available a measure of quality $g(d)$ for each document d that is query independent and thus *static*. This quality measure may be viewed as a number between 0 and 1. For instance, in the context of news stories on the web, $g(d)$ may be derived from the number of favorable reviews of the story by web surfers. Section 4.6 (page 73) provides further discussion on this topic, as does Chapter 21 in the context of web search.

The net score for a document d is some combination of $g(d)$ together with the query-dependent score induced (say) by (6.12). The precise combination may be determined by the learning methods of Section 6.1.2, to be developed

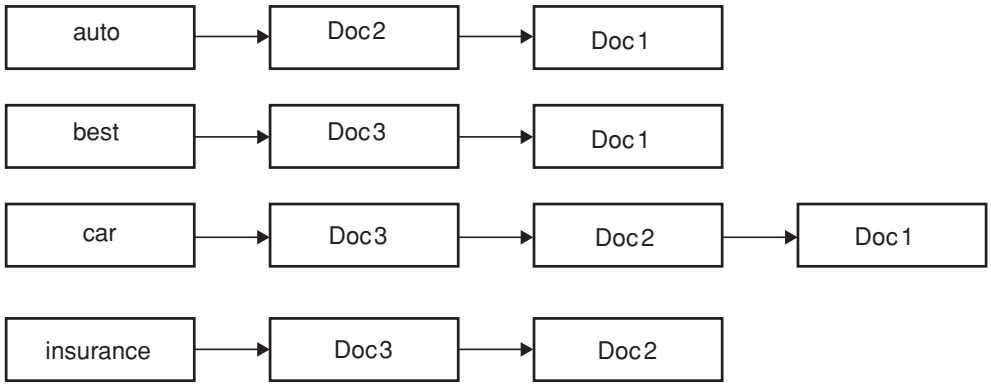

Figure 7.2 A static quality-ordered index. In this example we assume that Doc1, Doc2 and Doc3 respectively have static quality scores $g(1) = 0.25$, $g(2) = 0.5$, $g(3) = 1$.

further in Section 15.4.1; but for the purposes of our exposition here, let us consider a simple sum:

$$\text{net-score}(q, d) = g(d) + \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)||\vec{V}(d)|}. \tag{7.2}$$

In this simple form, the static quality $g(d)$ and the query-dependent score from (6.10) have equal contributions, assuming each is between 0 and 1. Other relative weightings are possible; the effectiveness of our heuristics depends on the specific relative weighting.

First, consider ordering the documents in the postings list for each term by decreasing value of $g(d)$. This allows us to perform the postings intersection algorithm of Figure 1.6. To perform the intersection by a single pass through the postings of each query term, the algorithm of Figure 1.6 relied on the postings being ordered by document IDs. But, in fact, we only required that all postings be ordered by a single common ordering; here, we rely on the $g(d)$ values to provide this common ordering. This is illustrated in Figure 7.2, where the postings are ordered in decreasing order of $g(d)$.

The next idea is a direct extension of champion lists: For a well-chosen value r, we maintain for each term t a *global champion list* of the r documents with the highest values for $g(d) + \text{tf-idf}_{t,d}$. The list itself is, like all the postings lists considered so far, sorted by a common order (either by document IDs or by static quality). Then at query time, we only compute the net scores (7.2) for documents in the union of these global champion lists. Intuitively, this has the effect of focusing on documents likely to have large net scores.

We conclude the discussion of global champion lists with one further idea. We maintain for each term t two postings lists consisting of disjoint sets of documents, each sorted by $g(d)$ values. The first list, which we call *high*, contains the m documents with the highest tf values for t. The second list, which we call *low*, contains all other documents containing t. When processing a query, we first scan only the high lists of the query terms, computing net

scores for any document on the high lists of all (or more than a certain number of) query terms. If we obtain scores for K documents in the process, we terminate. If not, we continue the scanning into the low lists, scoring documents in these postings lists. This idea is developed further in Section 7.2.1.

7.1.5 Impact ordering

In all the postings lists described thus far, we order the documents consistently by some common ordering: typically by document ID but in Section 7.1.4 by static quality scores. As noted at the end of Section 6.3.3, such a common ordering supports the concurrent traversal of all of the query terms' postings lists, computing the score for each document as we encounter it. Computing scores in this manner is sometimes referred to as *document-at-a-time scoring*. We now introduce a technique for inexact top K retrieval in which the postings are not all ordered by a common ordering, thereby precluding such a concurrent traversal. We therefore require scores to be "accumulated" one term at a time as in the scheme of Figure 6.14, so that we have term-at-a-time scoring.

The idea is to order the documents d in the postings list of term t by decreasing order of $\text{tf}_{t,d}$. Thus, the ordering of documents varies from one postings list to another, and we cannot compute scores by a concurrent traversal of the postings lists of all query terms. Given postings lists ordered by decreasing order of $\text{tf}_{t,d}$, two ideas have been found to significantly lower the number of documents for which we accumulate scores: (1) when traversing the postings list for a query term t, we stop after considering a prefix of the postings list – either after a fixed number of documents r have been seen, or after the value of $\text{tf}_{t,d}$ has dropped below a threshold; (2) when accumulating scores in the outer loop of Figure 6.14, we consider the query terms in decreasing order of idf, so that the query terms likely to contribute the most to the final scores are considered first. This latter idea can be adaptive at the time of processing a query: As we get to query terms with lower idf, we can determine whether to proceed based on the changes in document scores from processing the previous query term. If these changes are minimal, we may omit accumulation from the remaining query terms, or alternatively process shorter prefixes of their postings lists.

These ideas form a common generalization of the methods introduced in Sections 7.1.2 through 7.1.4. We may also implement a version of static ordering in which each postings list is ordered by an additive combination of static and query-dependent scores. We again lose the consistency of ordering across postings, and therefore have to process query terms one at time, accumulating scores for all documents as we go along. Depending on the particular scoring function, the postings list for a document may be ordered by other quantities than term frequency; under this more general setting, this idea is known as *impact ordering*.

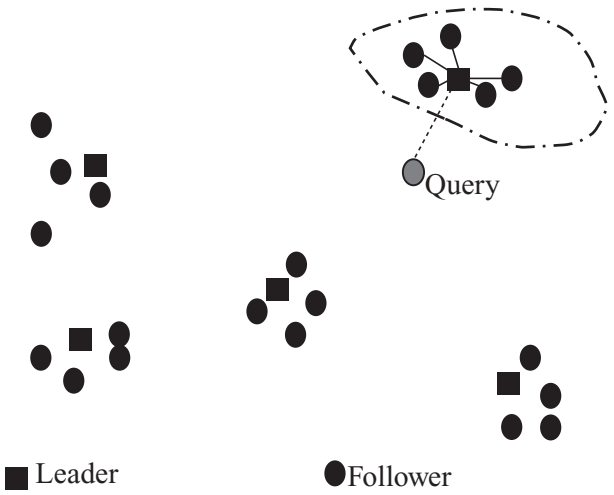

Figure 7.3 Cluster pruning.

7.1.6 Cluster pruning

In *cluster pruning*, we have a preprocessing step during which we cluster the document vectors. Then, at query time, we consider only documents in a small number of clusters as candidates for which we compute cosine scores. Specifically, the preprocessing step is as follows:

1. Pick $\sqrt{N}$ documents at random from the collection. Call these *leaders*.
2. For each document that is not a leader, we compute its nearest leader.

We refer to documents that are not leaders as *followers*. Intuitively, in the partition of the followers induced by the use of $\sqrt{N}$ randomly chosen leaders, the expected number of followers for each leader is $\approx N/\sqrt{N} = \sqrt{N}$. Next, query processing proceeds as follows:

1. Given a query q, find the leader L that is closest to q. This entails computing cosine similarities from q to each of the $\sqrt{N}$ leaders.
2. The candidate set A consists of L together with its followers. We compute the cosine scores for all documents in this candidate set.

The use of randomly chosen leaders for clustering is fast and likely to reflect the distribution of the document vectors in the vector space: A region of the vector space that is dense in documents is likely to produce multiple leaders and thus a finer partition into subregions. This illustrated in Figure 7.3.

Variations of cluster pruning introduce additional parameters b_1 and b_2, both of which are positive integers. In the preprocessing step, we attach each follower to its b_1 closest leaders, rather than a single closest leader. At query time we consider the b_2 leaders closest to the query q. Clearly, this basic scheme corresponds to the case $b_1 = b_2 = 1$. Further, increasing b_1 or b_2 increases the likelihood of finding K documents that are more likely to be in

the set of true top-scoring K documents, at the expense of more computation. We reiterate this approach when describing clustering in Chapter 16 (page 325).

? **Exercise 7.1** We suggested (Figure 7.2) that the postings for static quality ordering be in decreasing order of $g(d)$. Why do we use the decreasing rather than the increasing order?

Exercise 7.2 When discussing champion lists, we simply used the r documents with the largest tf values to create the champion list for t. But, when considering global champion lists, we used idf as well, identifying documents with the largest values of $g(d) + \text{tf-idf}_{t,d}$. Why do we differentiate between these two cases?

Exercise 7.3 If we were to only have one-term queries, explain why the use of global champion lists with $r = K$ suffices for identifying the K highest scoring documents. What is a simple modification to this idea if we were to only have s-term queries for any fixed integer $s > 1$?

Exercise 7.4 Explain how the common global ordering by $g(d)$ values in all high and low lists helps to make the score computation efficient.

Exercise 7.5 Consider again the data of Exercise 6.23 with `nnn.atc` for query-dependent scoring. Suppose that we were given static quality scores of 1 for Doc1 and 2 for Doc2. Determine under Equation (7.2) what ranges of static quality score for Doc3 result in it being the first, second, or third result for the query best car insurance.

Exercise 7.6 Sketch the frequency-ordered postings for the data in Figure 6.9.

Exercise 7.7 Let the static quality scores for Doc1, Doc2, and Doc3 in Figure 6.10 be respectively 0.25, 0.5, and 1. Sketch the postings for impact ordering when each postings list is ordered by the sum of the static quality score and the Euclidean normalized tf values in Figure 6.10.

Exercise 7.8 The nearest neighbor problem in the plane is the following: Given a set of N data points on the plane, we preprocess them into some data structure such that, given a query point Q, we seek the point in N that is closest to Q in Euclidean distance. Clearly, cluster pruning can be used as an approach to the nearest neighbor problem in the plane, if we wished to avoid computing the distance from Q to every one of the query points. Devise a simple example on the plane so that with two leaders, the answer returned by cluster pruning is incorrect (it is not the data point closest to Q).

7.2 Components of an information retrieval system

In this section, we combine the ideas developed so far to describe a rudimentary search system that retrieves and scores documents. We first develop further ideas for scoring, beyond vector spaces. After this, we put together all of these elements to outline a complete system. Because we consider a complete system, we do not restrict ourselves to vector space retrieval in this section. Indeed, our complete system has provisions for vector space as well as other query operators and forms of retrieval. In Section 7.3, we return to how vector space queries interact with other query operators.

7.2.1 Tiered indexes

We mentioned in Section 7.1.2 that, when using heuristics such as index elimination for inexact top-K retrieval, we may occasionally find ourselves with a set A of contenders that has fewer than K documents. A common solution to this issue is the user of *tiered indexes*, which may be viewed as a generalization of champion lists. We illustrate this idea in Figure 7.4, where we represent the documents and terms of Figure 6.9. In this example, we set a tf threshold of 20 for tier 1 and 10 for tier 2, meaning that the tier 1 index only has postings entries with tf values exceeding 20, and the tier 2 index only has postings entries with tf values exceeding 10. In this example, we have chosen to order the postings entries within a tier by document ID.

TIERED INDEXES

7.2.2 Query term proximity

Especially for free text queries on the web (Chapter 19), users prefer a document in which most or all of the query terms appear close to each other, because this is evidence that the document has text focused on their query intent. Consider a query with two or more query terms, $t_1, t_2, \ldots, t_k$. Let ω be the width of the smallest window in a document d that contains all the query terms, measured in the number of words in the window. For instance, if the document were to simply consist of the sentence The quality of mercy is not strained, the smallest window for the query strained mercy is 4. Intuitively, the smaller that ω is, the better that d matches the query. In cases where the document does not contain all of the query terms, we can set ω to be some enormous number. We could also consider variants in which only words that are not stop words are considered in computing ω. Such proximity-weighted scoring functions are a departure from pure cosine similarity and closer to the "soft conjunctive" semantics that Google and other web search engines evidently use.

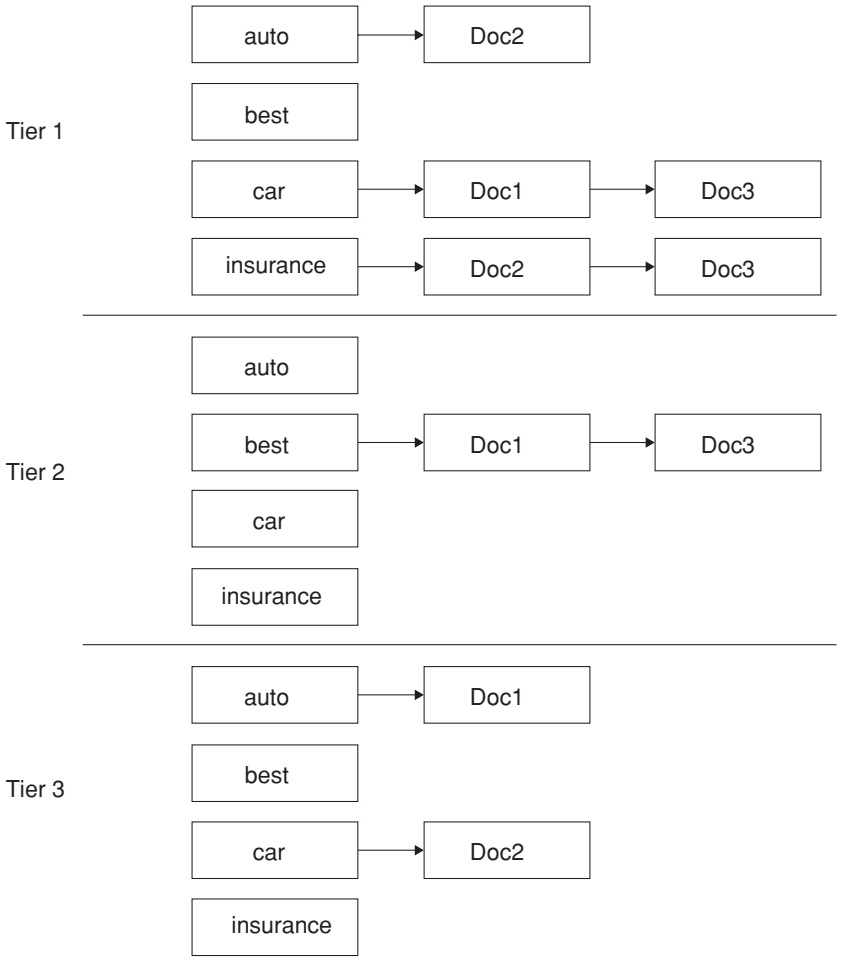

Figure 7.4 Tiered indexes. If we fail to get K results from tier 1, query processing "falls back" to tier 2, and so on. Within each tier, postings are ordered by document ID.

PROXIMITY WEIGHTING

How can we design such a *proximity-weighted* scoring function to depend on ω? The simplest answer relies on a "hand coding" technique we introduce in Section 7.2.3. A more scalable approach goes back to Section 6.1.2 – we treat the integer ω as yet another feature in the scoring function, whose importance is assigned by machine learning, as will be developed further in Section 15.4.1.

7.2.3 Designing parsing and scoring functions

Common search interfaces, particularly for consumer-facing search applications on the web, tend to mask query operators from the end user. The intent is to hide the complexity of these operators from the largely nontechnical audience for such applications, inviting *free text queries*. Given such interfaces,

how should a search equipped with indexes for various retrieval operators treat a query such as rising interest rates? More generally, given the various factors we have studied that could affect the score of a document, how should we combine these features?

The answer of course depends on the user population, the query distribution, and the collection of documents. Typically, a *query parser* is used to translate the user-specified keywords into a query with various operators that is executed against the underlying indexes. Sometimes, this execution can entail multiple queries against the underlying indexes; for example, the query parser may issue a stream of queries:

1. Run the user-generated query string as a phrase query. Rank them by vector space scoring using as query the vector consisting of the three terms rising interest rates.
2. If fewer than ten documents contain the phrase rising interest rates, run the two 2-term phrase queries rising interest and interest rates; rank these using vector space scoring, as well.
3. If we still have fewer than ten results, run the vector space query consisting of the three individual query terms.

Each of these steps (if invoked) may yield a list of scored documents, for each of which we compute a score. This score must combine contributions from vector space scoring, static quality, proximity weighting, and, potentially, other factors – particularly because a document may appear in the lists from multiple steps. This demands an aggregate scoring function that *accumulates evidence* of a document's relevance from multiple sources. How do we devise a query parser and how do we devise the aggregate scoring function?

EVIDENCE ACCUMULATION

The answer depends on the setting. In many enterprise settings, we have application builders who make use of a toolkit of available scoring operators, along with a query parsing layer, with which to manually configure the scoring function as well as the query parser. Such application builders make use of the available zones, metadata, and knowledge of typical documents and queries to tune the parsing and scoring. In collections whose characteristics change infrequently (in an enterprise application, significant changes in collection and query characteristics typically happen with infrequent events such as the introduction of new document formats or document management systems, or a merger with another company). Web search, on the other hand, is faced with a constantly changing document collection with new characteristics being introduced all the time. It is also a setting in which the number of scoring factors can run into the hundreds, making hand-tuned scoring a difficult exercise. To address this, it is becoming increasingly common to use machine-learned scoring, extending the ideas we introduced in Section 6.1.2, as will be discussed further in Section 15.4.1.

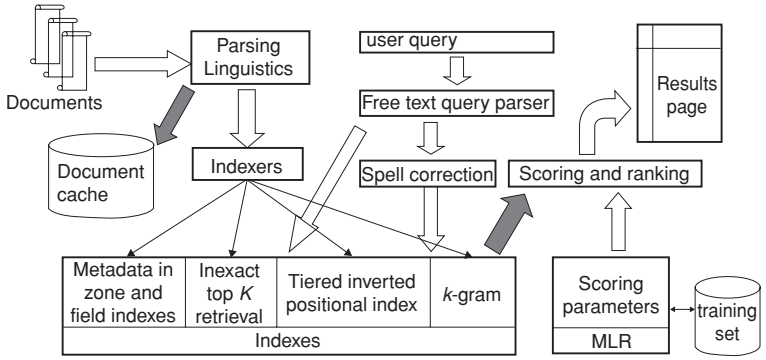

Figure 7.5 A complete search system. Data paths are shown primarily for a free text query.

7.2.4 Putting it all together

We have now studied all the components necessary for a basic search system that supports free text queries as well as Boolean, zone, and field queries. We briefly review how the various pieces fit together into an overall system; this is depicted in Figure 7.5.

In this figure, documents stream in from the left for parsing and linguistic processing (language and format detection, tokenization, and stemming). The resulting stream of tokens feeds into two modules. First, we retain a copy of each parsed document in a document cache. This enables us to generate results snippets: snippets of text accompanying each document in the results list for a query. This snippet tries to give a succinct explanation to the user of why the document matches the query. The automatic generation of such snippets is the subject of Section 8.7. A second copy of the tokens is fed to a bank of indexers that create a bank of indexes, including zone and field indexes that store the metadata for each document, (tiered) positional indexes, indexes for spelling correction and other tolerant retrieval, and structures for accelerating inexact top K retrieval. A free text user query (top center) is sent down to the indexes both directly and through a module for generating spelling-correction candidates. As noted in Chapter 3, the latter may optionally be invoked only when the original query fails to retrieve enough results. Retrieved documents (dark arrow) are passed to a scoring module that computes scores based on machine-learned ranking (MLR), a technique that builds on Section 6.1.2 (to be further developed in Section 15.4.1) for scoring and ranking documents. Finally, these ranked documents are rendered as a results page.

? **Exercise 7.9** Explain how the postings intersection algorithm first introduced in Section 1.3 can be adapted to find the smallest integer ω that contains all query terms.

Exercise 7.10 Adapt this procedure to work when not all query terms are present in a document.

7.3 Vector space scoring and query operator interaction

We introduced the vector space model as a paradigm for free text queries. We conclude this chapter by discussing how the vector space scoring model relates to the query operators we have studied in earlier chapters. The relationship should be viewed at two levels: in terms of the expressiveness of queries that a sophisticated user may pose, and in terms of the index that supports the evaluation of the various retrieval methods. In building a search engine, we may opt to support multiple query operators for an end user. In doing so, we need to understand what components of the index can be shared for executing various query operators, as well as how to handle user queries that mix various query operators.

Vector space scoring supports so-called free text retrieval, in which a query is specified as a set of words without any query operators connecting them. It allows documents matching the query to be scored and thus ranked, unlike the Boolean, wildcard, and phrase queries studied earlier. Classically, the interpretation of such free text queries was that at least one of the query terms be present in any retrieved document. However, more recently, web search engines such as Google have popularized the notion that a set of terms typed into their query boxes (thus on the face of it, a free text query) carries the semantics of a conjunctive query that only retrieves documents containing all or most query terms.

Boolean retrieval

Clearly, a vector space index can be used to answer Boolean queries, as long as the weight of a term t in the document vector for d is nonzero whenever t occurs in d. The reverse is not true; a Boolean index does not by default maintain term weight information. There is no easy way of combining vector space and Boolean queries from a user's standpoint: Vector space queries are fundamentally a form of *evidence accumulation*, where the presence of more query terms in a document adds to the score of a document. Boolean retrieval, on the other hand, requires a user to specify a formula for *selecting* documents through the presence (or absence) of specific combinations of keywords, without inducing any relative ordering among them. Mathematically, it is in fact possible to invoke so-called p-norms to combine Boolean and vector space queries, but we know of no system that makes use of this fact.

Wildcard queries

Wildcard and vector space queries require different indexes, except at the basic level, that both can be implemented using postings and a dictionary (e.g., a dictionary of trigrams for wildcard queries). If a search engine allows

a user to specify a wildcard operator as part of a free text query (for instance, the query rom* restaurant), we may interpret the wildcard component of the query as spawning multiple terms in the vector space (in this example, rome and roman are two such terms), all of which are added to the query vector. The vector space query is then executed as usual, with matching documents being scored and ranked; thus, a document containing both rome and roma is likely to be scored higher than another containing only one of them. The exact score ordering, of course, depends on the relative weights of each term in matching documents.

Phrase queries

The representation of documents as vectors is fundamentally lossy: The relative order of terms in a document is lost in the encoding of a document as a vector. Even if we were to try and somehow treat every biword as a term (and thus an axis in the vector space), the weights on different axes are not independent: For instance, the phrase German shepherd gets encoded in the axis german shepherd, but immediately has a nonzero weight on the axes german and shepherd. Further, notions such as idf would have to be extended to such biwords. Thus, an index built for vector space retrieval cannot, in general, be used for phrase queries. Moreover, there is no way of demanding a vector space score for a phrase query – we only know the relative weights of each term in a document.

For the query german shepherd, we could use vector space retrieval to identify documents heavy in these two terms, with no way of prescribing that they occur consecutively. Phrase retrieval, on the other hand, tells us of the existence of the phrase german shepherd in a document, without any indication of the relative frequency or weight of this phrase. Although these two retrieval paradigms (phrase and vector space) consequently have different implementations in terms of indexes and retrieval algorithms, they can in some cases be combined usefully, as in the three-step example of query parsing in Section 7.2.3.

7.4 References and further reading

Heuristics for fast query processing with early termination are described by Anh et al. (2001), Garcia et al. (2004), Anh and Moffat (2006b), Persin et al. (1996). Cluster pruning is investigated by Singitham et al. (2004) and by Chierichetti et al. (2007); see also Section 16.6 (page 343). Champion lists are described in Persin (1994) and (under the name *top docs*) in Brown (1995), and further developed in Brin and Page (1998), Long and Suel (2003). Although these heuristics are well-suited to free text queries

TOP DOCS

that can be viewed as vectors, they complicate phrase queries; see Anh and Moffat (2006c) for an index structure that supports both weighted and Boolean/phrase searches. Carmel et al. (2001), Clarke et al. (2000), and Song et al. (2005) treat the use of query term proximity in assessing relevance. Pioneering work on learning of ranking functions was done by Fuhr (1989), Fuhr and Pfeifer (1994), Cooper et al. (1994), Bartell (1994), Bartell et al. (1998), and Cohen et al. (1998).