2 Basic Techniques

As a foundation for the remainder of the book, this chapter takes a tour through the elements of information retrieval outlined in Chapter 1, covering the basics of indexing, retrieval and evaluation. The material on indexing and retrieval, constituting the first two major sections, is closely linked, presenting a unified view of these topics. The third major section, on evaluation, examines both the efficiency and the effectiveness of the algorithms introduced in the first two sections.

2.1 Inverted Indices

The inverted index (sometimes called *inverted file*) is the central data structure in virtually every information retrieval system. At its simplest, an inverted index provides a mapping between terms and their locations of occurrence in a text collection $\mathcal{C}$. The fundamental components of an inverted index are illustrated in Figure 2.1, which presents an index for the text of Shakespeare's plays (Figures 1.2 and 1.3). The *dictionary* lists the terms contained in the vocabulary $\mathcal{V}$ of the collection. Each term has associated with it a *postings list* of the positions in which it appears, consistent with the positional numbering in Figure 1.4 (page 14).

If you have encountered inverted indices before, you might be surprised that the index shown in Figure 2.1 contains not document identifiers but "flat" word positions of the individual term occurrences. This type of index is called a *schema-independent* index because it makes no assumptions about the structure (usually referred to as *schema* in the database community) of the underlying text. We chose the schema-independent variant for most of the examples in this chapter because it is the simplest. An overview of alternative index types appears in Section 2.1.3.

Regardless of the specific type of index that is used, its components — the dictionary and the postings lists — may be stored in memory, on disk, or a combination of both. For now, we keep the precise data structures deliberately vague. We define an inverted index as an abstract data type (ADT) with four methods:

- **first**(t) returns the first position at which the term t occurs in the collection
- **last**(t) returns the last position at which t occurs in the collection
- **next**(t, *current*) returns the position of t's first occurrence after the *current* position
- **prev**(t, *current*) returns the position of t's last occurrence before the *current* position.

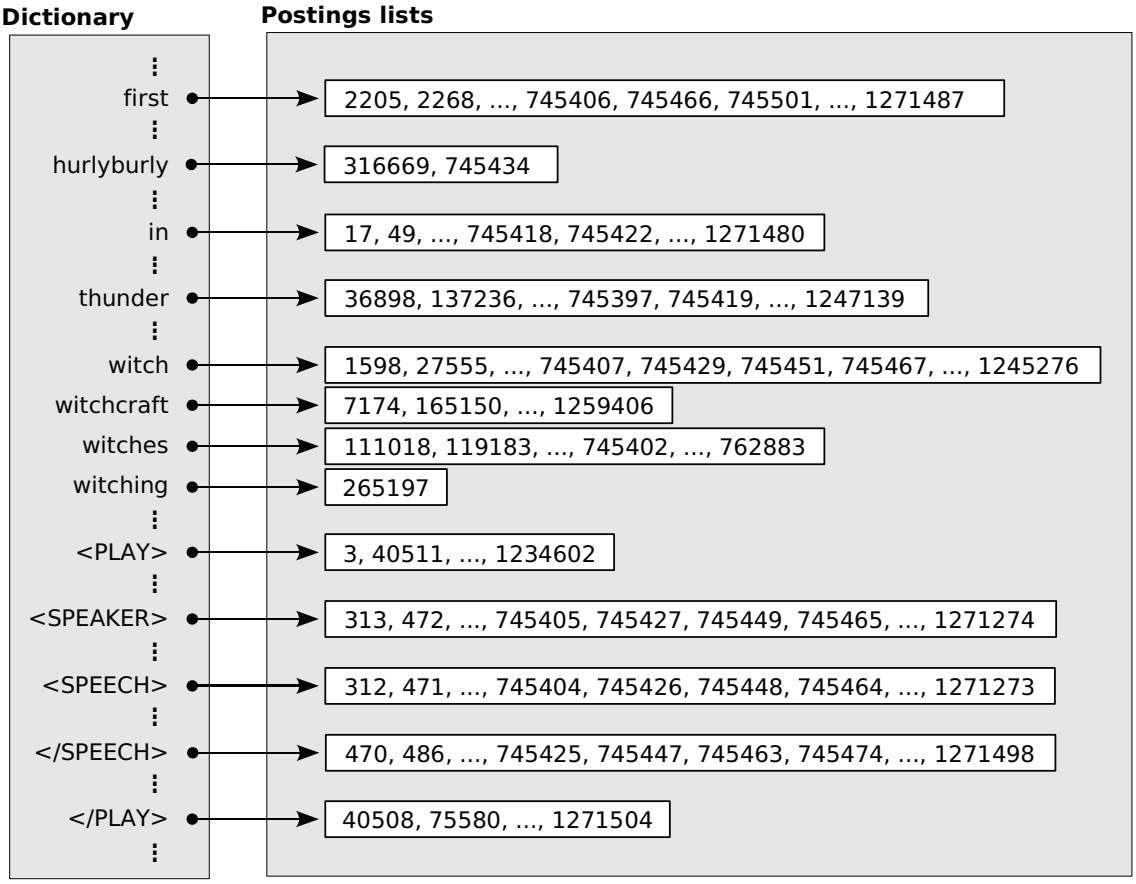


Figure 2.1 A schema-independent inverted index for Shakespeare's plays. The dictionary provides a mapping from terms to their positions of occurrence.

In addition, we define l_t to represent the total number of times the term t appears in the collection (i.e., the length of its postings list). We define $l_{\mathcal{C}}$ to be the length of the collection, so that $\sum_{t \in \mathcal{V}} l_t = l_{\mathcal{C}}$ (where $\mathcal{V}$ is the collection's vocabulary).

For the inverted index in Figure 2.1, we have:

$$\textbf{first}(\text{“hurlyburly”}) = 316669 \qquad \textbf{last}(\text{“thunder”}) = 1247139$$
$$\textbf{first}(\text{“witching”}) = 265197 \qquad \textbf{last}(\text{“witching”}) = 265197$$

$\mathbf{next}(\text{“witch”}, 745429) = 745451$ $\mathbf{prev}(\text{“witch”}, 745451) = 745429$
$\mathbf{next}(\text{“hurlyburly”}, 345678) = 745434$ $\mathbf{prev}(\text{“hurlyburly”}, 456789) = 316669$
$\mathbf{next}(\text{“witch”}, 1245276) = \infty$ $\mathbf{prev}(\text{“witch”}, 1598) = -\infty$
$l_{\text{<PLAY>}} = 37$ $l_{\mathcal{C}} = 1271504$
$l_{\text{witching}} = 1$

The symbols ∞ and $-\infty$ act as beginning-of-file and end-of-file markers, representing positions beyond the beginning and the end of the term sequence. As a practical convention we define:

$$\mathbf{next}(t, -\infty) = \mathbf{first}(t) \qquad \mathbf{next}(t, \infty) = \infty$$
$$\mathbf{prev}(t, \infty) = \mathbf{last}(t) \qquad \mathbf{prev}(t, -\infty) = -\infty$$

The methods of our inverted index permit both sequential and random access into postings lists, with a sequential scan of a postings list being a simple loop:

```
current ← −∞
while current < ∞ do
    current ← next(t, current)
    do something with the current value
```

However, many algorithms require random access into postings lists, including the phrase search algorithm we present next. Often, these algorithms take the result of a method call for one term and apply it as an argument to a method call for another term, skipping through the postings lists nonsequentially.

2.1.1 Extended Example: Phrase Search

Most commercial Web search engines, as well as many other IR systems, treat a list of terms enclosed in double quotes (`"..."`) as a phrase. To process a query that contains a phrase, the IR system must identify the occurrences of the phrase in the collection. This information may then be used during the retrieval process for filtering and ranking — perhaps excluding documents that do not contain an exact phrase match.

Phrase search provides an excellent example of how algorithms over inverted indices operate. Suppose we wish to locate all occurrences of the phrase “first witch” in our collection of Shakespeare’s plays. Perhaps we wish to identify all speeches by this character. By visually scanning the postings lists in Figure 2.1, we can locate one occurrence starting at 745406 and ending at 745407. We might locate other occurrences in a similar fashion by scanning the postings lists for an occurrence of “first” immediately followed by an occurrence of “witch”. In this section we

```
   nextPhrase (t1t2...tn, position) ≡
1      v ← position
2      for i ← 1 to n do
3          v ← next(ti, v)
4      if v = ∞ then
5          return [∞, ∞]
6      u ← v
7      for i ← n − 1 down to 1 do
8          u ← prev(ti, u)
9      if v − u = n − 1 then
10         return [u, v]
11     else
12         return nextPhrase(t1t2...tn, u)
```

Figure 2.2 Function to locate the first occurrence of a phrase after a given position. The function calls the **next** and **prev** methods of the inverted index ADT and returns an interval in the text collection as a result.

present an algorithm that formalizes this process, efficiently locating all occurrences of a given phrase with the aid of our inverted index ADT.

We specify the location of a phrase by an interval [u,v], where u indicates the start of the phrase and v indicates the end of the phrase. In addition to the occurrence at [745406, 745407], the phrase "first witch" may be found at [745466, 745467], at [745501, 745502], and elsewhere. The goal of our phrase searching algorithm is to determine values of u and v for all occurrences of the phrase in the collection.

We use the above interval notation to specify retrieval results throughout the book. In some contexts it is also convenient to think of an interval as a stand-in for the text at that location. For example, the interval [914823, 914829] might represent the text

> O Romeo, Romeo! wherefore art thou Romeo?

Given the phrase "$t_1t_2...t_n$", consisting of a sequence of n terms, our algorithm works through the postings lists for the terms from left to right, making a call to the **next** method for each term, and then from right to left, making a call to the **prev** method for each term. After each pass from left to right and back, it has computed an interval in which the terms appear in the correct order and as close together as possible. It then checks whether the terms are in fact adjacent. If they are, an occurrence of the phrase has been found; if not, the algorithm moves on.

Figure 2.2 presents the core of the algorithm as a function **nextPhrase** that locates the next occurrence of a phrase after a given position. The loop over lines 2–3 calls the methods of the inverted index to locate the terms in order. At the end of the loop, if the phrase occurs in the interval [*position*,v], it ends at v. The loop over lines 7–8 then shrinks the interval to the smallest

size possible while still including all terms in order. Finally, lines 9–12 verify that the terms are adjacent, forming a phrase. If they are not adjacent, the function makes a tail-recursive call. On line 12, note that u (and not v) is passed as the second argument to the recursive call. If the terms in the phrase are all different, then v could be passed. Passing u correctly handles the case in which two terms t_i and t_j are equal ($1 \leq i < j \leq n$).

As an example, suppose we want to find the first occurrences of the phrase "first witch": **nextPhrase**("first witch", $-\infty$). The algorithm starts by identifying the first occurrence of "first":

next("first", $-\infty$) = **first**("first") = 2205.

If this occurrence of "first" is part of the phrase, then the next occurrence of "witch" should immediately follow it. However,

next("witch", 2205) = 27555,

that is, it does not immediately follow it. We now know that the first occurrence of the phrase cannot end before position 27555, and we compute

prev("first", 27555) = 26267.

In jumping from 2205 to 26267 in the postings list for "first", we were able to skip 15 occurrences of "first". Because interval [26267, 27555] has length 1288, and not the required length 2, we move on to consider the next occurrence of "first" at

next("first", 26267) = 27673.

Note that the calls to the **prev** method in line 8 of the algorithm are not strictly necessary (see Exercise 2.2), but they help us to analyze the complexity of the algorithm.

If we want to generate all occurrences of the phrase instead of just a single occurrence, an additional loop is required, calling **nextPhrase** once for each occurrence of the phrase:

```
u ← −∞
while u < ∞ do
    [u,v] ← nextPhrase("t1t2...tn", u)
    if u ≠ ∞ then
        report the interval [u,v]
```

The loop reports each interval as it is generated. Depending on the application, reporting $[u,v]$ might involve returning the document containing the phrase to the user, or it might involve storing the interval in an array or other data structure for further processing. Similar to the code in Figure 2.2, u (and not v) is passed as the second argument to **nextPhrase**. As a result,

the function can correctly locate all six occurrences of the phrase "spam spam spam" in the follow passage from the well-known Monty Python song:

> Spam spam spam spam
> Spam spam spam spam

To determine the time complexity of the algorithm, first observe that each call to **nextPhrase** makes $O(n)$ calls to the **next** and **prev** methods of the inverted index (n calls to **next**, followed by $n-1$ calls to **prev**). After line 8, the interval $[u,v]$ contains all terms in the phrase in order, and there is no smaller interval contained within it that also contains all the terms in order. Next, observe that each occurrence of a term t_i in the collection can be included in no more than one of the intervals computed by lines 1–8. Even if the phrase contains two identical terms, t_i and t_j, a matching token in the collection can be included in only one such interval as a match to t_i, although it might be included in another interval as a match to t_j. The time complexity is therefore determined by the length of the shortest postings list for the terms in the phrase:

$$l = \min_{1 \le i \le n} l_{t_i}. \tag{2.1}$$

Combining these observations, in the worst case the algorithm requires $O(n \cdot l)$ calls to methods of our ADT to locate all occurrences of the phrase. If the phrase includes both common and uncommon terms ("Rosencrantz and Guildenstern are dead"), the number of calls is determined by the least frequent term ("Guildenstern") and not the most frequent one ("and").

We emphasize that $O(n \cdot l)$ represents the number of method calls, not the number of steps taken by the algorithm, and that the time for each method call depends on the details of how it is implemented. For the access patterns generated by the algorithm, there is a surprisingly simple and efficient implementation that gives good performance for phrases containing any mixture of frequent and infrequent terms. We present the details in the next section.

Although the algorithm requires $O(n \cdot l)$ method calls in the worst case, the actual number of calls depends on the relative location of the terms in the collection. For example, suppose we are searching for the phrase "hello world" and the text in the collection is arranged:

> hello ... hello ... hello ... hello world ... world ... world ... world

with all occurrences of "hello" before all occurrences of "world". Then the algorithm makes only four method calls to locate the single occurrence of the phrase, regardless of the size of the text or the number of occurrences of each term. Although this example is extreme and artificial, it illustrates the *adaptive* nature of the algorithm — its actual execution time is determined by characteristics of the data. Other IR problems may be solved with adaptive algorithms, and we exploit this approach whenever possible to improve efficiency.

To make the adaptive nature of the algorithm more explicit, we introduce a measure of the characteristics of the data that determines the actual number of method calls. Consider the interval $[u,v]$ just before the test at line 9 of Figure 2.2. The interval contains all the terms in

the phrase in order, but does not contain any smaller interval containing all the terms in order. We call an interval with this property a *candidate phrase* for the terms. If we define κ to be the number of candidate phrases in a given document collection, then the number of method calls required to locate all occurrences is $O(n \cdot \kappa)$.

2.1.2 Implementing Inverted Indices

> It moves across the blackness that lies between stars, and its mechanical legs move slowly. Each step that it takes, however, crossing from nothing to nothing, carries it twice the distance of the previous step. Each stride also takes the same amount of time as the prior one. Suns flash by, fall behind, wink out. It runs through solid matter, passes through infernos, pierces nebulae, faster and faster moving through the starfall blizzard in the forest of the night. Given a sufficient warm-up run, it is said that it could circumnavigate the universe in a single stride. What would happen if it kept running after that, no one knows.
>
> — Roger Zelazny, *Creatures of Light and Darkness*

When a collection will never change and when it is small enough to be maintained entirely in memory, an inverted index may be implemented with very simple data structures. The dictionary may be stored in a hash table or similar structure, and the postings list for each term t may be stored in a fixed array $P_t[]$ with length l_t. For the term "witch" in the Shakespeare collection, this array may be represented as follows:

1	2		31	32	33	34		92
1598	27555	$\cdots$	745407	745429	745451	745467	$\cdots$	1245276

The **first** and **last** methods of our inverted index ADT may be implemented in constant time by returning $P_t[1]$ and $P_t[l_t]$, respectively. The **next** and **prev** methods may be implemented by a binary search with time complexity $O(\log(l_t))$. Details for the **next** method are provided in Figure 2.3; the details for the **prev** method are similar.

Recall that the phrase searching algorithm of Section 2.1.1 requires $O(n \cdot l)$ calls to the **next** and **prev** methods in the worst case. If we define

$$L = \max_{1 \le i \le n} l_{t_i}\,, \tag{2.2}$$

then the time complexity of the algorithms becomes $O(n \cdot l \cdot \log(L))$ because each call to a method may require up to $O(\log(L))$ time. When expressed in terms of κ, the number of candidate phrases, the time complexity becomes $O(n \cdot \kappa \cdot \log(L))$.

When a phrase contains both frequent and infrequent terms, this implementation can provide excellent performance. For example, the term "tempest" appears only 49 times in the works of Shakespeare. As we saw in Section 1.3.3, the term "the" appears 28,317 times. However, when

```
   next (t, current) ≡
1      if l_t = 0 or P_t[l_t] ≤ current then
2          return ∞
3      if P_t[1] > current then
4          return P_t[1]
5      return P_t[binarySearch (t, 1, l_t, current)]

   binarySearch (t, low, high, current) ≡
6      while high − low > 1 do
7          mid ← ⌊(low + high)/2⌋
8          if P_t[mid] ≤ current then
9              low ← mid
10         else
11             high ← mid
12     return high
```

Figure 2.3 Implementation of the **next** method through a binary search that is implemented by a separate function. The array $P_t[]$ (of length l_t) contains the postings list for term t. The **binarySearch** function assumes that $P_t[low] \leq current$ and $P_t[high] > current$. Lines 1–4 establish this precondition, and the loop at lines 6–11 maintains it as an invariant.

searching for the phrase "the tempest", we access the postings list array for "the" less than two thousand times while conducting at most $2 \cdot 49 = 98$ binary searches.

On the other hand, when a phrase contains terms with similar frequencies, the repeated binary searches may be wasteful. The terms in the phrase "two gentlemen" both appear a few hundred times in Shakespeare (702 and 225 times, to be exact). Identifying all occurrences of this phrase requires more than two thousand accesses to the postings list array for "two". In this case, it would be more efficient if we could scan sequentially through both arrays at the same time, comparing values as we go. By changing the definition of the **next** and **prev** methods, the phrase search algorithm can be adapted to do just that.

To start with, we note that as the phrase search algorithm makes successive calls to the **next** method for a given term t_i, the values passed as the second argument strictly increase across calls to **nextPhrase**, including the recursive calls. During the process of finding all occurrences of a given phrase, the algorithm may make up to l calls to **next** for that term (where l, as before, is the length of the shortest postings list):

$$\mathbf{next}(t_i, v_1),\ \mathbf{next}(t_i, v_2),\ ...,\ \mathbf{next}(t_i, v_l)$$

with

$$v_1 < v_2 < ... < v_l\,.$$

Moreover, the results of these calls also strictly increase:

$$\mathbf{next}(t_i, v_1) < \mathbf{next}(t_i, v_2) < ... < \mathbf{next}(t_i, v_l)\,.$$

```
   next (t, current) ≡
1      if l_t = 0 or P_t[l_t] ≤ current then
2          return ∞
3      if P_t[1] > current then
4          c_t ← 1
5          return P_t[c_t]
6      if c_t > 1 and P_t[c_t − 1] > current then
7          c_t ← 1
8      while P_t[c_t] ≤ current do
9          c_t ← c_t + 1
10     return P_t[c_t]
```

Figure 2.4 Implementation of the **next** method through a linear scan. This implementation updates a cached index offset c_t for each term t, where $P_t[c_t]$ represents the last noninfinite result returned from a call to **next** for this term. If possible, the implementation starts its scan from this cached offset. If not, the cached offset is reset at lines 6–7.

For example, when searching for "first witch" in Shakespeare, the sequence of calls for "first" begins:

$$\textbf{next}(\text{“first”}, -\infty),\ \textbf{next}(\text{“first”}, 26267),\ \textbf{next}(\text{“first”}, 30608),\ ...$$

returning the values

$$2205 < 27673 < 32995 < ...$$

Of course, the exact values for v_1, v_2, ... and the actual number of calls to **next** depend on the locations of the other terms in the phrase. Nonetheless, we know that these values will increase and that there may be l of them in the worst case.

To implement our sequential scan, we remember (or cache) the value returned by a call to **next** for a given term. When the function is called again (for the same term), we continue our scan at this cached location. Figure 2.4 provides the details. The variable c_t caches the array offset of the value returned by the previous call, with a separate value cached for each term in the phrase (e.g., c_{first} and c_{witch}). Because the method may be used in algorithms that do not process postings lists in a strictly increasing order, we are careful to reset c_t if this assumption is violated (lines 6–7).

If we take a similar approach to implementing **prev** and maintain corresponding cached values, the phrase search algorithm scans the postings lists for the terms in the phrase, accessing each element of the postings list arrays a bounded number of times ($O(1)$). Because the algorithm may fully scan the longest postings list (of size L), and all postings lists may be of this length, the overall time complexity of the algorithm is $O(n \cdot L)$. In this case the adaptive nature of the algorithm provides no benefit.

We now have two possible implementations for **next** and **prev** that in effect produce two implementations of **nextPhrase**. The first implementation, with overall time complexity $O(n \cdot l \cdot \log(L))$, is particularly appropriate when the shortest postings list is considerably

```
   next (t, current) ≡
1      if l_t = 0 or P_t[l_t] ≤ current then
2          return ∞
3      if P_t[1] > current then
4          c_t ← 1
5          return P_t[c_t]
6      if c_t > 1 and P_t[c_t − 1] ≤ current then
7          low ← c_t − 1
8      else
9          low ← 1
10     jump ← 1
11     high ← low + jump
12     while high < l_t and P_t[high] ≤ current do
13         low ← high
14         jump ← 2 · jump
15         high ← low + jump
16     if high > l_t then
17         high ← l_t
18     c_t ← binarySearch (t, low, high, current)
19     return P_t[c_t]
```

Figure 2.5 Implementation of the **next** method through a galloping search. Lines 6–9 determine an initial value for *low* such that $P_t[low] \leq current$, using the cached value if possible. Lines 12–17 gallop ahead in exponentially increasing steps until they determine a value for *high* such that $P_t[high] > current$. The final result is determined by a binary search (from Figure 2.3).

shorter than the longest postings list ($l \ll L$). The second implementation, with time complexity $O(n \cdot L)$, is appropriate when all postings lists have approximately the same length ($l \approx L$).

Given this dichotomy, we might imagine choosing between the algorithms at run-time by comparing l with L. However, it is possible to define a third implementation of the methods that combines features of both algorithms, with a time complexity that explicitly depends on the relative sizes of the longest and shortest lists (L/l). This third algorithm is based on a *galloping search*. The idea is to scan forward from a cached position in exponentially increasing steps ("galloping") until the answer is passed. At this point, a binary search is applied to the range formed by the last two steps of the gallop to locate the exact offset of the answer. Figure 2.5 provides the details.

Figure 2.6 illustrates and compares the three approaches for a call to **prev**("witch", 745429) over the Shakespeare collection. Using a binary search (part a), the method would access the array seven times, first at positions 1 and 92 to establish the invariant required by the binary search (not shown), and then at positions 46, 23, 34, 28, and 31 during the binary search itself. Using a sequential scan (part b) starting from an initial cached offset of 1, the method would access the array 34 times, including the accesses required to check boundary conditions (not shown). A galloping search (part c) would access positions 1, 2, 4, 8, 16, and 32 before

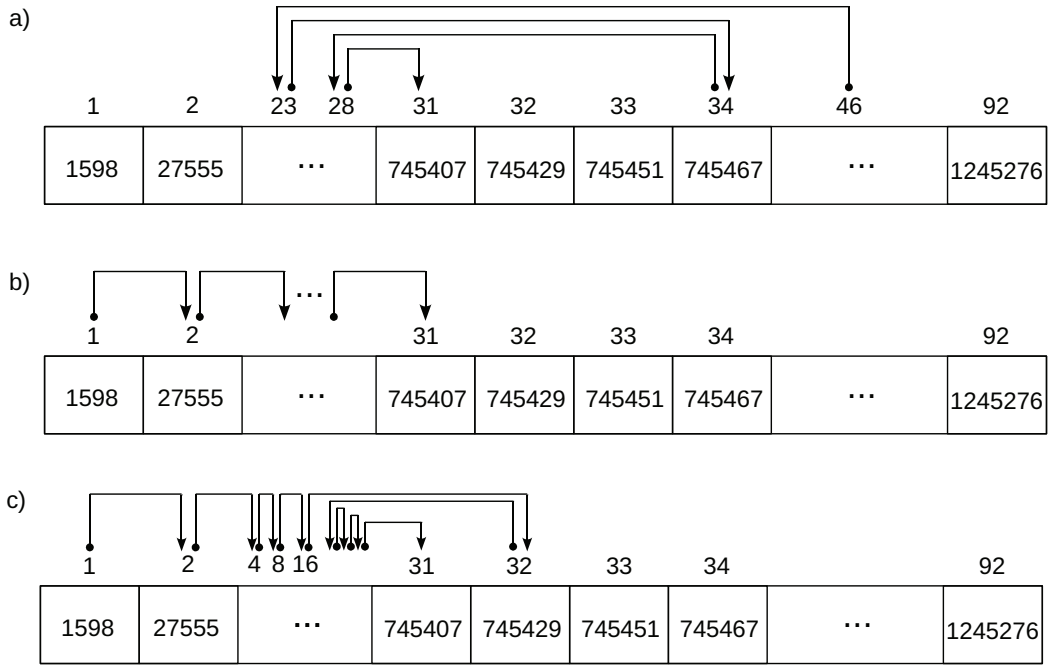


Figure 2.6 Access patterns for three approaches to solving **prev**("witch", 745429) = 745407: (a) binary search, (b) sequential scan, and (c) galloping. For (b) and (c), the algorithms start at an initial cached position of 1.

establishing the conditions for a binary search, which would then access positions 24, 28, 30, and 31, for a total of twelve accesses to the postings list array (including checking the boundary conditions). At the end of both the scanning and the galloping methods, the cached array offset would be updated to 31.

To determine the time complexity of galloping search, we return to consider the sequence of calls to **next** that originally motivated the sequential scanning algorithm. Let c_t^j be the cached value after the jth call to **next** for term t during the processing of a given phrase search.

$$\begin{aligned} P_t[c_t^1] &= \textbf{next}\ (t,\ v_1) \\ P_t[c_t^2] &= \textbf{next}\ (t,\ v_2) \\ &\cdots \\ P_t[c_t^l] &= \textbf{next}\ (t,\ v_l) \end{aligned}$$

For a galloping search the amount of work done by a particular call to **next** depends on the change in the cached value from call to call. If the cached value changes by Δc, then the amount of work done by a call is $O(\log(\Delta c))$. Thus, if we define

$$\begin{aligned} \Delta c_1 &= c_t^1 \\ \Delta c_2 &= c_t^2 - c_t^1 \\ &\cdots \\ \Delta c_l &= c_t^l - c_t^{l-1}, \end{aligned}$$

then the total work done by calls to **next** for term t is

$$\sum_{j=1}^{l} O(\log(\Delta c_j)) \;=\; O\left(\log\left(\prod_{j=1}^{l} \Delta c_j\right)\right). \tag{2.3}$$

We know that the arithmetic mean of a list of nonnegative numbers is always greater than its geometric mean

$$\frac{\sum_{j=1}^{l} \Delta c_j}{l} \;\geq\; \sqrt[l]{\prod_{j=1}^{l} \Delta c_j}\,, \tag{2.4}$$

and since $\sum_{j=1}^{l} \Delta c_j \leq L$, we have

$$\prod_{j=1}^{l} \Delta c_j \;\leq\; (L/l)^l. \tag{2.5}$$

Therefore, the total work done by calls to **next** (or **prev**) for the term t is

$$O\left(\log\left(\prod_{j=1}^{l} \Delta c_j\right)\right) \;\subseteq\; O\left(\log\left((L/l)^l\right)\right) \tag{2.6}$$

$$= \; O\left(l \cdot \log\left(L/l\right)\right). \tag{2.7}$$

The overall time complexity for a phrase with n terms is $O\left(n \cdot l \cdot \log\left(L/l\right)\right)$. When $l \ll L$, this performance is similar to that of binary search; when $l \approx L$, it is similar to scanning. Taking the adaptive nature of the algorithm into account, a similar line of reasoning gives a time complexity of $O(n \cdot \kappa \cdot \log(L/\kappa))$.

Although we have focused on phrase searching in our discussion of the implementation of inverted indices, we shall see that galloping search is an appropriate technique for other problems, too. Part II of the book extends these ideas to data structures stored on disk.

2.1.3 Documents and Other Elements

Most IR systems and algorithms operate over a standard unit of retrieval: the document. As was discussed in Chapter 1, requirements of the specific application environment determine exactly what constitutes a document. Depending on these requirements, a document might be an e-mail message, a Web page, a newspaper article, or similar element.

In many application environments, the definition of a document is fairly natural. However, in a few environments, such as a collection of books, the natural unit (an entire book) may sometimes be too large to return as a reasonable result, particularly when the relevant material is limited to a small part of the text. Instead, it may be desirable to return a chapter, a section, a subsection, or even a range of pages.

In the case of our collection of Shakespeare's plays, the most natural course is probably to treat each play as a document, but acts, scenes, speeches, and lines might all be appropriate units of retrieval in some circumstances. For the purposes of a simple example, assume we are interested in speeches and wish to locate those spoken by the "first witch".

The phrase "first witch" first occurs at $[745406, 745407]$. Computing the speech that contains this phrase is reasonably straightforward. Using the methods of our inverted index ADT, we determine that the start of a speech immediately preceding this phrase is located at

$$\textbf{prev}(\text{“<SPEECH>”}, 745406) = 745404.$$

The end of this speech is located at

$$\textbf{next}(\text{“</SPEECH>”}, 754404) = 745425.$$

Once we confirm that the interval $[745406, 745407]$ is contained in the interval $[745404, 745425]$, we know we have located a speech that contains the phrase. This check to confirm the containment is necessary because the phrase may not always occur as part of a speech. If we wish to locate all speeches by the "first witch", we can repeat this process with the next occurrence of the phrase.

A minor problem remains. Although we know that the phrase occurs in the speech, we do not know that the "first witch" is the speaker. The phrase may actually appear in the lines spoken. Fortunately, confirming that the witch is the speaker requires only two additional calls to methods of the inverted index (Exercise 2.4). In fact, simple calls to these methods are sufficient to compute a broad range of structural relationships, such as the following:

1. Lines spoken by any witch.
2. The speaker who says, "To be or not to be".
3. Titles of plays mentioning witches and thunder.

In a broader context, this flexible approach to specifying retrieval units and filtering them with simple containment criteria has many applications. In Web search systems, simple filtering

can restrict retrieval results to a single domain. In enterprise search, applying constraints to the sender field allows us to select messages sent by a particular person. In file system search, structural constraints may be used to determine if file permissions and security restrictions allow a user to search a directory.

Because a requirement for "lightweight" structure occurs frequently in IR applications, we adopt a simple and uniform approach to supporting this structure by incorporating it directly into the inverted index, making it part of the basic facilities an inverted index provides. The examples above illustrate our approach. Complete details will be presented in Chapter 5, in which the approach forms the basis for implementing the advanced search operators, which are widely used in domain-specific IR applications, such as legal search. The approach may be used to implement the differential field weighting described in Section 8.7, which recognizes that the presence of a query term in a document's title may be a stronger indicator of relevance than its presence in the body. The approach may also be used to provide a foundation for the implementation of the more complex index structures required to fully support XML retrieval (see Chapter 16).

Notwithstanding those circumstances in which lightweight structure is required, most IR research assumes that the text collection naturally divides into documents, which are considered to be atomic units for retrieval. In a system for searching e-mail, messages form this basic retrieval unit. In a file system, files do; on the Web, Web pages. In addition to providing a natural unit of retrieval, documents also provide natural divisions in the text, allowing a collection to be partitioned into multiple subcollections for parallel retrieval and allowing documents to be reordered to improve efficiency, perhaps by grouping all documents from a single source or Web site.

Document-Oriented Indices

Because document retrieval represents such an important special case, indices are usually optimized around it. To accommodate this optimization, the numbering of positions in a document collection may be split into two components: a document number and an offset within the document.

We use the notation $n{:}m$ to indicate positions within a document-oriented index in which n is a document identifier (or *docid*) and m is an *offset*. Figure 2.7 presents an inverted index for Shakespeare's plays that treats plays as documents. The methods of our inverted index ADT continue to operate as before, but they accept *docid*:*offset* pairs as arguments and return them

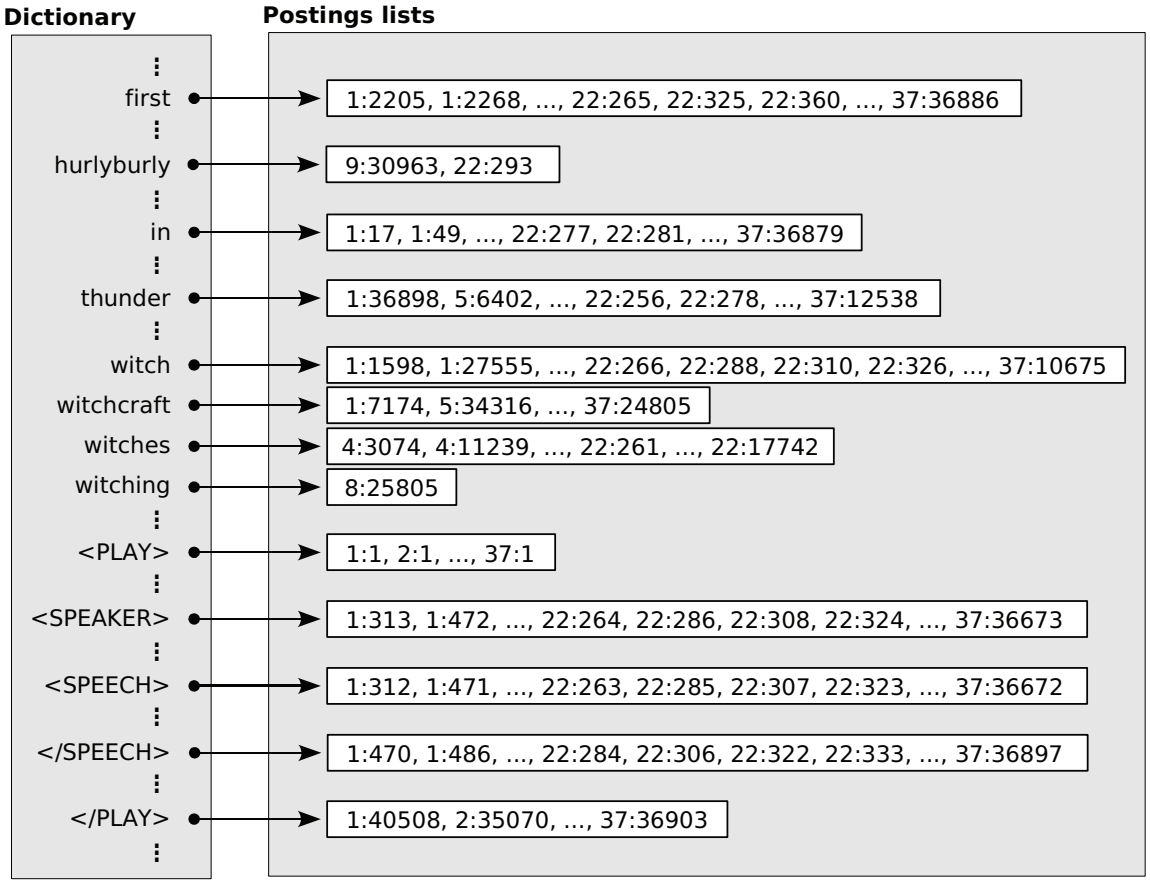


Figure 2.7 A document-centric index for Shakespeare's plays equivalent to the one shown in Figure 2.1 (page 34). Each posting is of the form *docid*:*within-document-position*.

as results. For example,

first("hurlyburly") = 9:30963
first("witching") = 8:25805

last("thunder") = 37:12538
last("witching") = 8:25805

next("witch", 22:288) = 22:310
next("hurlyburly", 9:30963) = 22:293
next("witch", 37:10675) = ∞

prev("witch", 22:310) = 22:288
prev("hurlyburly", 22:293) = 9:30963
prev("witch", 1:1598) = $-\infty$

Offsets within a document start at 1 and range up to the length of the document. We continue to use $-\infty$ and ∞ as beginning-of-file and end-of-file markers despite the slight notational inconsistency, with $-\infty$ read as $-\infty$:$-\infty$ and ∞ read as ∞:∞. When term positions are expressed in this form, their values are compared by treating the document number as the primary key:

$$n{:}m < n'{:}m' \text{ if and only if } (n < n' \text{ or } (n = n' \text{ and } m < m')).$$

We refer to an index whose structure has been optimized to support document-oriented retrieval as a *schema-dependent* inverted index, because the division of the text into retrieval units (its schema) is determined at the time its inverted index is constructed.

An index without these optimizations is a *schema-independent* inverted index. A schema-independent index allows the definition of a document to be specified at query time, with a possible cost in execution time in comparison with its schema-dependent equivalent.

To support ranking algorithms, a schema-dependent inverted index usually maintains various document-oriented statistics. We adopt the following notation for these statistics:

N_t	*document frequency*	the number of documents in the collection containing the term t
$f_{t,d}$	*term frequency*	the number of times term t appears in document d
l_d	*document length*	measured in tokens
l_{avg}	*average length*	average document length across the collection
N	*document count*	total number of documents in the collection

Note that $\sum_{d \in \mathcal{C}} l_d = \sum_{t \in \mathcal{V}} l_t = l_{\mathcal{C}}$, and $l_{avg} = l_{\mathcal{C}}/N$.

Over the collection of Shakespeare's plays, $l_{avg} = 34363$. If $t =$ "witch" and $d = 22$ (i.e., $d =$ *Macbeth*), then $N_t = 18$, $f_{t,d} = 52$, and $l_d = 26805$. In a schema-dependent index, these statistics are usually maintained as an integral part of the index data structures. With a schema-independent index, they have to be computed at query time with the methods of our inverted index ADT (see Exercise 2.5).

To help us write algorithms that operate at the document level, we define additional methods of our inverted index ADT. The first two break down positions into separate docids and offsets:

docid(*position*)	returns the docid associated with a *position*
offset(*position*)	returns the within-document offset associated with a *position*

When a posting takes the form $[u{:}v]$, these methods simply return u and v, respectively. They may also be implemented on top of a schema-independent index, albeit slightly less efficiently.

We also define document-oriented versions of our basic inverted index methods that will prove useful in later parts of this chapter:

firstDoc(t)	returns the docid of the first document containing the term t
lastDoc(t)	returns the docid of the last document containing the term t
nextDoc(t, *current*)	returns the docid of the first document after *current* that contains the term t
prevDoc(t, *current*)	returns the docid of the last document before *current* that contains the term t

In a schema-dependent index, many postings may now share a common prefix in their docids. We can separate out these docids to produce postings of the form

$$(d, f_{t,d}, \langle p_0, ..., p_{f_{t,d}} \rangle)$$

where $\langle p_0, ..., p_{f_{t,d}} \rangle$ is the list of the offsets of all $f_{t,d}$ occurrences of the term t within document d. Besides eliminating unnecessary repetition, this notation better reflects how the postings are actually represented in an implementation of a schema-dependent index. Using this notation, we would write the postings list for the term "witch" as

$$(1, 3, \langle 1598, 27555, 31463 \rangle),\ ...,\ (22, 52, \langle 266, 288, ... \rangle),\ ...,\ (37, 1, \langle 10675 \rangle)$$

In specialized circumstances it may not be necessary for positional offsets to be maintained by an inverted index. Basic keyword search is sufficient for some applications, and effective ranking can be supported with document-level statistics. For the simplest ranking and filtering techniques, these document-level statistics are not even required.

Based on the type of information found in each postings list, we can distinguish four types of inverted indices, the first three of which are schema-dependent:

- A *docid index* is the simplest index type. For each term, it contains the document identifiers of all documents in which the term appears. Despite its simplicity, this index type is sufficient to support filtering with basic Boolean queries (Section 2.2.3) and a simple form of relevance ranking known as coordination level ranking (Exercise 2.7).
- In a *frequency index*, each entry in an inverted list consists of two components: a document ID and a term frequency value. Each posting in a frequency index is of the form $(d, f_{t,d})$. Frequency indices are sufficient to support many effective ranking methods (Section 2.2.1), but are insufficient for phrase searching and advanced filtering.
- A *positional index* consists of postings of the form $(d, f_{t,d}, \langle p_1, ..., p_{f_{t,d}} \rangle)$. Positional indices support all search operations supported by a frequency index. In addition, they can be used to realize phrase queries, proximity ranking (Section 2.2.2), and other query types that take the exact position of a query term within a document into account, including all types of structural queries.
- A *schema-independent index* does not include the document-oriented optimizations found in a positional index, but otherwise the two may be used interchangeably.

Table 2.1 Text fragment from Shakespeare's *Romeo and Juliet*, act I, scene 1.

Document ID	Document Content
1	Do you quarrel, sir?
2	Quarrel sir! no, sir!
3	If you do, sir, I am for you: I serve as good a man as you.
4	No better.
5	Well, sir.

Table 2.2 Postings lists for the terms shown in Table 2.1. In each case the length of the list is appended to the start of the actual list.

Term	Docid list	Positional list	Schema-independent
a	1; 3	1; $(3, 1, \langle 13 \rangle)$	1; 21
am	1; 3	1; $(3, 1, \langle 6 \rangle)$	1; 14
as	1; 3	1; $(3, 2, \langle 11, 15 \rangle)$	2; 19, 23
better	1; 4	1; $(4, 1, \langle 2 \rangle)$	1; 26
do	2; 1, 3	2; $(1, 1, \langle 1 \rangle), (3, 1, \langle 3 \rangle)$	2; 1, 11
for	1; 3	1; $(3, 1, \langle 7 \rangle)$	1; 15
good	1; 3	1; $(3, 1, \langle 12 \rangle)$	1; 20
i	1; 3	1; $(3, 2, \langle 5, 9 \rangle)$	2; 13, 17
if	1; 3	1; $(3, 1, \langle 1 \rangle)$	1; 9
man	1; 3	1; $(3, 1, \langle 14 \rangle)$	1; 22
no	2; 2, 4	2; $(2, 1, \langle 3 \rangle), (4, 1, \langle 1 \rangle)$	2; 7, 25
quarrel	2; 1, 2	2; $(1, 1, \langle 3 \rangle), (2, 1, \langle 1 \rangle)$	2; 3, 5
serve	1; 3	1; $(3, 1, \langle 10 \rangle)$	1; 18
sir	4; 1, 2, 3, 5	4; $(1, 1, \langle 4 \rangle), (2, 2, \langle 2, 4 \rangle), (3, 1, \langle 4 \rangle), (5, 1, \langle 2 \rangle)$	5; 4, 6, 8, 12, 28
well	1; 5	1; $(5, 1, \langle 1 \rangle)$	1; 27
you	2; 1, 3	2; $(1, 1, \langle 2 \rangle), (3, 3, \langle 2, 8, 16 \rangle)$	4; 2, 10, 16, 24

Table 2.1 shows an excerpt from Shakespeare's *Romeo and Juliet*. Here, each line is treated as a document — we have omitted the tags to help shorten the example to a reasonable length. Table 2.2 shows the corresponding postings lists for all terms that appear in the excerpt, giving examples of docid lists, positional postings lists, and schema-independent postings lists.

Of the four different index types, the docid index is always the smallest one because it contains the least information. The positional and the schema-independent indices consume the greatest space, between two times and five times as much space as a frequency index, and between three times and seven times as much as a docid index, for typical text collections. The exact ratio depends on the lengths of the documents in the collection, the skewedness of the term distribution, and the impact of compression. Index sizes for the four different index types and

Table 2.3 Index sizes for various index types and three test collections, with and without applying index compression techniques. In each case the first number refers to an index in which each component is stored as a simple 32-bit integer, and the second number refers to an index in which each entry is compressed using a byte-aligned encoding method.

	Shakespeare	**TREC**	**GOV2**
Docid index	n/a	578 MB/200 MB	37751 MB/12412 MB
Frequency index	n/a	1110 MB/333 MB	73593 MB/21406 MB
Positional index	n/a	2255 MB/739 MB	245538 MB/78819 MB
Schema-independent index	5.7 MB/2.7 MB	1190 MB/533 MB	173854 MB/65960 MB

our three example collections are shown in Table 2.3. Index compression has a substantial effect on index size, and it is discussed in detail in Chapter 6.

With the introduction of document-oriented indices, we have greatly expanded the notation associated with inverted indices from the four basic methods introduced at the beginning of the chapter. Table 2.4 provides a summary of this notation for easy reference throughout the remainder of the book.

2.2 Retrieval and Ranking

Building on the data structures of the previous section, this section presents three simple retrieval methods. The first two methods produce ranked results, ordering the documents in the collection according to their expected relevance to the query. Our third retrieval method allows Boolean filters to be applied to the collection, identifying those documents that match a predicate.

Queries for ranked retrieval are often expressed as *term vectors*. When you type a query into an IR system, you express the components of this vector by entering the terms with white space between them. For example, the query

william shakespeare marriage

might be entered into a commercial Web search engine with the intention of retrieving a ranked list of Web pages concerning Shakespeare's marriage to Anne Hathaway. To make the nature of these queries more obvious, we write term vectors explicitly using the notation $\langle t_1, t_2, ..., t_n \rangle$. The query above would then be written

⟨"william", "shakespeare", "marriage"⟩.

You may wonder why we represent these queries as vectors rather than sets. Representation as a vector (or, rather, a *list*, since we do not assume a fixed-dimensional vector space) is useful when terms are repeated in a query and when the ordering of terms is significant. In ranking formulae, we use the notation q_t to indicate the number of times term t appears in the query.

Table 2.4 Summary of notation for inverted indices.

Basic inverted index methods	
first(*term*)	returns the first position at which the *term* occurs
last(*term*)	returns the last position at which the *term* occurs
next(*term*, *current*)	returns the next position at which the *term* occurs after the *current* position
prev(*term*, *current*)	returns the previous position at which the *term* occurs before the *current* position
Document-oriented equivalents of the basic methods	
firstDoc(*term*), **lastDoc**(*term*), **nextDoc**(*term*, *current*), **lastDoc**(*term*, *current*)	
Schema-dependent index positions	
n:*m*	n = *docid* and m = *offset*
docid(*position*)	returns the docid associated with a *position*
offset(*position*)	returns the within-document offset associated with a *position*
Symbols for document and term statistics	
l_t	the length of t's postings list
N_t	the number of documents containing t
$f_{t,d}$	the number of occurrences of t within the document d
l_d	length of the document d, in tokens
l_{avg}	the average document length in the collection
N	the total number of documents in the collection
The structure of postings lists	
docid index	$d_1, d_2, \ldots, d_{N_t}$
frequency index	$(d_1, f_{t,d_1}), (d_2, f_{t,d_2}), \ldots$
positional index	$(d_1, f_{t,d_1}, \langle p_1, \ldots, p_{f_{t,d_1}} \rangle), \ldots$
schema-independent	$p_1, p_2, \ldots, p_{l_t}$

Boolean predicates are composed with the standard Boolean operators (AND, OR, NOT). The result of a Boolean query is a set of documents matching the predicate. For example, the Boolean query

"william" AND "shakespeare" AND NOT ("marlowe" OR "bacon")

specifies those documents containing the terms "william" and "shakespeare" but not containing either "marlowe" or "bacon". In later chapters we will extend this standard set of Boolean operators, which will allow us to specify additional constraints on the result set.

There is a key difference in the conventional interpretations of term vectors for ranked retrieval and predicates for Boolean retrieval. Boolean predicates are usually interpreted as strict filters — if a document does not match the predicate, it is not returned in the result set. Term vectors, on the other hand, are often interpreted as summarizing an information need. Not all the terms in the vector need to appear in a document for it to be returned. For example, if we are interested in the life and works of Shakespeare, we might attempt to create an exhaustive (and exhausting) query to describe our information need by listing as many related terms as we can:

> william shakespeare stratford avon london plays sonnets poems tragedy comedy poet playwright players actor anne hathaway susanna hamnet judith folio othello hamlet macbeth king lear tempest romeo juliet julius caesar twelfth night antony cleopatra venus adonis willie hughe wriothesley henry ...

Although many relevant Web pages will contain some of these terms, few will contain all of them. It is the role of a ranked retrieval method to determine the impact that any missing terms will have on the final document ordering.

Boolean retrieval combines naturally with ranked retrieval into a two-step retrieval process. A Boolean predicate is first applied to restrict retrieval to a subset of the document collection. The documents contained in the resulting subcollection are then ranked with respect to a given topic. Commercial Web search engines follow this two-step retrieval process. Until recently, most of these systems would interpret the query

> william shakespeare marriage

as both a Boolean conjunction of the terms — "william" AND "shakespeare" AND "marriage" — and as a term vector for ranking — ⟨"william", "shakespeare", "marriage"⟩. For a page to be returned as a result, each of the terms was required to appear in the page itself or in the anchor text linking to the page.

Filtering out relevant pages that are missing one or more terms may have the paradoxical effect of harming performance when extra terms are added to a query. In principle, adding extra terms should improve performance by serving to better define the information need. Although some commercial Web search engines now apply less restrictive filters, allowing additional documents to appear in the ranked results, this two-step retrieval process still takes place. These systems may handle longer queries poorly, returning few or no results in some cases.

In determining an appropriate document ranking, basic ranked retrieval methods compare simple features of the documents. One of the most important of these features is term frequency, $f_{t,d}$, the number of times query term t appears in document d. Given two documents d_1 and d_2, if a query term appears many more times in d_1 than in d_2, this may suggest that d_1 should be ranked higher than d_2, other factors being equal. For the query ⟨"william", "shakespeare", "marriage"⟩, the repeated occurrence of the term "marriage" throughout a document may suggest that it should be ranked above one containing the term only once.

Another important feature is term proximity. If query terms appear closer together in document d_1 than in document d_2, this may suggest that d_1 should be ranked higher than d_2, other factors being equal. In some cases, terms form a phrase ("william shakespeare") or other collocation, but the importance of proximity is not merely a matter of phrase matching. The co-occurrence of "william", "shakespeare", and "marriage" together in a fragment such as

> ... while no direct evidence of the **marriage** of Anne Hathaway and **William Shakespeare** exists, the wedding is believed to have taken place in November of 1582, while she was pregnant with his child ...

suggests a relationship between the terms that might not exist if they appeared farther apart.

Other features help us make trade-offs between competing factors. For example, should a thousand-word document containing four occurrences of "william", five of "shakespeare", and two of "marriage" be ranked before or after a five-hundred-word document containing three occurrences of "william", two of "shakespeare", and seven of "marriage"? These features include the lengths of the documents (l_d) relative to the average document length (l_{avg}), as well as the number of documents in which a term appears (N_t) relative to the total number of documents in the collection (N).

Although the basic features listed above form the core of many retrieval models and ranking methods, including those discussed in this chapter, additional features may contribute as well. In some application areas, such as Web search, the exploitation of these additional features is critical to the success of a search engine.

One important feature is document structure. For example, a query term may be treated differently if it appears in the title of a document rather than in its body. Often the relationship between documents is important, such as the links between Web documents. In the context of Web search, the analysis of the links between Web pages may allow us to assign them a query-independent ordering or *static rank*, which can then be a factor in retrieval. Finally, when a large group of people make regular use of an IR system within an enterprise or on the Web, their behavior can be monitored to improve performance. For example, if results from one Web site are clicked more than results from another, this behavior may indicate a user preference for one site over the other — other factors being equal — that can be exploited to improve ranking. In later chapters these and other additional features will be covered in detail.

2.2.1 The Vector Space Model

The *vector space model* is one of the oldest and best known of the information retrieval models we examine in this book. Starting in the 1960s and continuing into 1990s, the method was developed and promulgated by Gerald Salton, who was perhaps the most influential of the early IR researchers. As a result, the vector space model is intimately associated with the field as a whole and has been adapted to many IR problems beyond ranked retrieval, including document clustering and classification, in which it continues to play an important role. In recent years, the

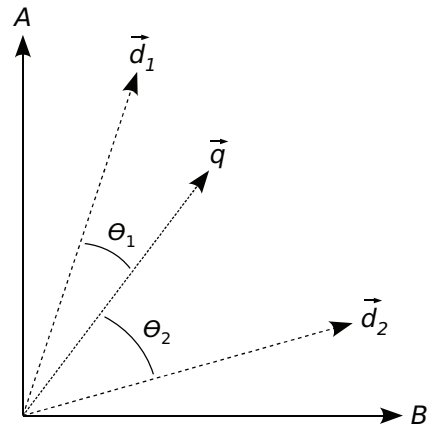


Figure 2.8 Document similarity under the vector space model. Angles are computed between a query vector $\vec{q}$ and two document vectors $\vec{d}_1$ and $\vec{d}_2$. Because $\theta_1 < \theta_2$, d_1 should be ranked higher than d_2.

vector space model has been largely overshadowed by probabilistic models, language models, and machine learning approaches (see Part III). Nonetheless, the simple intuition underlying it, as well as its long history, makes the vector space model an ideal vehicle for introducing ranked retrieval.

The basic idea is simple. Queries as well as documents are represented as vectors in a high-dimensional space in which each vector component corresponds to a term in the vocabulary of the collection. This query vector representation stands in contrast to the term vector representation of the previous section, which included only the terms appearing in the query. Given a query vector and a set of document vectors, one for each document in the collection, we rank the documents by computing a similarity measure between the query vector and each document vector, comparing the angle between them. The smaller the angle, the more similar the vectors. Figure 2.8 illustrates the basic idea, using vectors with only two components (A and B).

Linear algebra provides us with a handy formula to determine the angle θ between two vectors. Given two $|\mathcal{V}|$-dimensional vectors $\vec{x} = \langle x_1, x_2, ..., x_{|\mathcal{V}|} \rangle$ and $\vec{y} = \langle y_1, y_2, ..., y_{|\mathcal{V}|} \rangle$, we have

$$\vec{x} \cdot \vec{y} \;=\; |\vec{x}| \cdot |\vec{y}| \cdot \cos(\theta). \tag{2.8}$$

where $\vec{x} \cdot \vec{y}$ represents the *dot product* (also called the *inner product* or *scalar product*) between the vectors; $|\vec{x}|$ and $|\vec{y}|$ represent the lengths of the vectors. The dot product is defined as

$$\vec{x} \cdot \vec{y} \;=\; \sum_{i=1}^{|\mathcal{V}|} x_i \cdot y_i \tag{2.9}$$

and the length of a vector may be computed from the Euclidean distance formula

$$|\vec{x}| \;=\; \sqrt{\sum_{i=1}^{|\mathcal{V}|} {x_i}^2}\,. \tag{2.10}$$

Substituting and rearranging these equations gives

$$\cos(\theta) \;=\; \frac{\vec{x}}{|\vec{x}|} \cdot \frac{\vec{y}}{|\vec{y}|} \;=\; \frac{\sum_{i=1}^{|\mathcal{V}|} x_i\, y_i}{\left(\sqrt{\sum_{i=1}^{|\mathcal{V}|} {x_i}^2}\right)\left(\sqrt{\sum_{i=1}^{|\mathcal{V}|} {y_i}^2}\right)}\,. \tag{2.11}$$

We could now apply arccos to determine θ, but because the cosine is monotonically decreasing with respect to the angle θ, it is convenient to stop at this point and retain the cosine itself as our measure of similarity. If $\theta = 0°$, then the vectors are colinear, as similar as possible, with $\cos(\theta) = 1$. If $\theta = 90°$, then the vectors are orthogonal, as dissimilar as possible, with $\cos(\theta) = 0$.

To summarize, given a document vector $\vec{d}$ and a query vector $\vec{q}$, the cosine similarity $sim(\vec{d}, \vec{q})$ is computed as

$$sim(\vec{d}, \vec{q}) \;=\; \frac{\vec{d}}{|\vec{d}|} \cdot \frac{\vec{q}}{|\vec{q}|}\,, \tag{2.12}$$

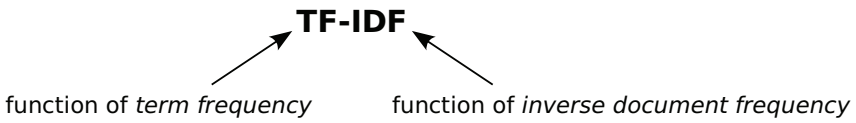
the dot product of the document and query vectors normalized to unit length. Provided all components of the vectors are nonnegative, the value of this *cosine similarity measure* ranges from 0 to 1, with its value increasing with increasing similarity.

Naturally, for a collection of even modest size, this vector space model produces vectors with millions of dimensions. This high-dimensionality might appear inefficient at first glance, but in many circumstances the query vector is sparse, with all but a few components being zero. For example, the vector corresponding to the query ⟨"william", "shakespeare", "marriage"⟩ has only three nonzero components. To compute the length of this vector, or its dot product with a document vector, we need only consider the components corresponding to these three terms. On the other hand, a document vector typically has a nonzero component for each unique term contained in the document, which may consist of thousands of terms. However, the length of a document vector is independent of the query. It may be precomputed and stored in a frequency or positional index along with other document-specific information, or it may be applied to normalize the document vector in advance, with the components of the normalized vector taking the place of term frequencies in the postings lists.

Although queries are usually short, the symmetry between how documents and queries are treated in the vector space model allows entire documents to be used as queries. Equation 2.12 may then be viewed as a formula determining the similarity between two documents. Treating a document as a query is one possible method for implementing the "Similar pages" and "More like this" features seen in some commercial search engines.

As a ranking method the cosine similarity measure has intuitive appeal and natural simplicity. If we can appropriately represent queries and documents as vectors, cosine similarity may be used to rank the documents with respect to the queries. In representing a document or query as a vector, a *weight* must be assigned to each term that represents the value of the corresponding component of the vector. Throughout the long history of the vector space model, many formulae for assigning these weights have been proposed and evaluated. With few exceptions, these formulae may be characterized as belonging to a general family known as *TF-IDF weights*.

When assigning a weight in a document vector, the TF-IDF weights are computed by taking the product of a function of term frequency ($f_{t,d}$) and a function of the inverse of document frequency ($1/N_t$). When assigning a weight to a query vector, the within-query term frequency (q_t) may be substituted for $f_{t,d}$, in essence treating the query as a tiny document. It is also possible (and not at all unusual) to use different TF and IDF functions to determine weights for document vectors and query vectors.



We emphasize that a TF-IDF weight is a product of *functions* of term frequency and inverse document frequency. A common error is to use the raw $f_{t,d}$ value for the term frequency component, which may lead to poor performance.

Over the years a number of variants for both the TF and the IDF functions have been proposed and evaluated. The IDF functions typically relate the document frequency to the total number of documents in the collection (N). The basic intuition behind the IDF functions is that a term appearing in many documents should be assigned a lower weight than a term appearing in few documents. Of the two functions, IDF comes closer to having a "standard form",

$$\text{IDF} \;=\; \log(N/N_t), \tag{2.13}$$

with most IDF variants structured as a logarithm of a fraction involving N_t and N.

The basic intuition behind the various TF functions is that a term appearing many times in a document should be assigned a higher weight for that document than for a document in which it appears fewer times. Another important consideration behind the definition of a TF function is that its value should not necessarily increase linearly with $f_{t,d}$. Although two occurrences of a term should be given more weight than one occurrence, they shouldn't necessarily be given twice the weight. The following function meets these requirements and appears in much of Salton's later work:

$$\text{TF} \;=\; \begin{cases} \log(f_{t,d}) + 1 & \text{if } f_{t,d} > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.14}$$

When this equation is used with a query vector, $f_{t,d}$ is replaced by q_t the query term frequency of t in q. We use this equation, along with Equation 2.13, to compute both document and query weights in the example that follows.

Consider the *Romeo and Juliet* document collection in Table 2.1 (page 50) and the corresponding postings lists given in Table 2.2. Because there are five documents in the collection and "sir" appears in four of them, the IDF value for "sir" is

$$\log(N/f_{\text{sir}}) \;=\; \log(5/4) \;\approx\; 0.32.$$

In this formula and in other TF-IDF formulae involving logarithms, the base of the logarithm is usually unimportant. As necessary, for purposes of this example and others throughout the book, we assume a base of 2.

Because "sir" appears twice in document 2, the TF-IDF value for the corresponding component of its vector is

$$(\log(f_{\text{sir},2}) + 1) \cdot (\log(N/f_{\text{sir}})) \;=\; (\log(2) + 1) \cdot (\log(5/4)) \;\approx\; 0.64.$$

Computing TF-IDF values for the remaining components and the remaining documents, gives the following set of vectors:

$$\begin{aligned}
\vec{d_1} &\approx \langle 0.00, 0.00, 0.00, 0.00, 1.32, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 1.32, 0.00, 0.32, 0.00, 1.32\rangle \\
\vec{d_2} &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 1.32, 1.32, 0.00, 0.64, 0.00, 0.00\rangle \\
\vec{d_3} &\approx \langle 2.32, 2.32, 4.64, 0.00, 1.32, 2.32, 2.32, 4.64, 2.32, 2.32, 0.00, 0.00, 2.32, 0.32, 0.00, 3.42\rangle \\
\vec{d_4} &\approx \langle 0.00, 0.00, 0.00, 2.32, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 1.32, 0.00, 0.00, 0.00, 0.00, 0.00\rangle \\
\vec{d_5} &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.32, 2.32, 0.00\rangle
\end{aligned}$$

where the components are sorted alphabetically according to their corresponding terms. Normalizing these vectors, dividing by their lengths, produces:

$$\begin{aligned}
\vec{d_1}/|\vec{d_1}| &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.57, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.57, 0.00, 0.14, 0.00, 0.57\rangle \\
\vec{d_2}/|\vec{d_2}| &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.67, 0.67, 0.00, 0.33, 0.00, 0.00\rangle \\
\vec{d_3}/|\vec{d_3}| &\approx \langle 0.24, 0.24, 0.48, 0.00, 0.14, 0.24, 0.24, 0.48, 0.24, 0.24, 0.00, 0.00, 0.24, 0.03, 0.00, 0.35\rangle \\
\vec{d_4}/|\vec{d_4}| &\approx \langle 0.00, 0.00, 0.00, 0.87, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.49, 0.00, 0.00, 0.00, 0.00, 0.00\rangle \\
\vec{d_5}/|\vec{d_5}| &\approx \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.14, 0.99, 0.00\rangle
\end{aligned}$$

If we wish to rank these five documents with respect to the query ⟨"quarrel", "sir"⟩, we first construct the query vector, normalized by length:

$$\vec{q}/|\vec{q}| \;\approx\; \langle 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.97, 0.00, 0.24, 0.00, 0.00\rangle$$

```
rankCosine (⟨t_1, ..., t_n⟩, k) ≡
1   j ← 1
2   d ← min_{1≤i≤n} nextDoc (t_i, −∞)
3   while d < ∞ do
4       Result[j].docid ← d
5       Result[j].score ← (d⃗ / |d|) · (q⃗ / |q|)
6       j ← j + 1
7       d ← min_{1≤i≤n} nextDoc (t_i, d)
8   sort Result by score
9   return Result[1..k]
```

Figure 2.9 Query processing for ranked retrieval under the vector space model. Given the term vector $\langle t_1, ..., t_n \rangle$ (with corresponding query vector $\vec{q}$), the function identifies the top k documents.

Computing the dot product between this vector and each document vector gives the following cosine similarity values:

Document ID	1	2	3	4	5
Similarity	0.59	0.73	0.01	0.00	0.03

The final document ranking is 2, 1, 5, 3, 4.

Query processing for the vector space model is straightforward (Figure 2.9), essentially performing a merge of the postings lists for the query terms. Docids and corresponding scores are accumulated in an array of records as the scores are computed. The function operates on one document at a time. During each iteration of the **while** loop, the algorithm computes the score for document d (with corresponding document vector $\vec{d}$), stores the docid and score in the array of records *Result*, and determines the next docid for processing. The algorithm does not explicitly compute a score for documents that do not contain any of the query terms, which are implicitly assigned a score of zero. At the end of the function, *Result* is sorted by *score* and the top k documents are returned.

For many retrieval applications, the entire ranked list of documents is not required. Instead we return at most k documents, where the value of k is determined by the needs of the application environment. For example, a Web search engine might return only the first $k = 10$ or 20 results on its first page. It then may seem inefficient to compute the score for every document containing any of the terms, even a single term with low weight, when only the top k documents are required. This apparent inefficiency has led to proposals for improved query processing methods that are applicable to other IR models as well as to the vector space model. These query processing methods will be discussed in Chapter 5.

Of the document features listed at the start of this section — term frequency, term proximity, document frequency, and document length — the vector space model makes explicit use of only term frequency and document frequency. Document length is handled implicitly when the

vectors are normalized to unit length. If one document is twice as long as another but contains the same terms in the same proportions, their normalized vectors are identical. Term proximity is not considered by the model at all. This property has led to the colorful description of the vector space model (and other IR models with the same property) as a "bag of words" model.

We based the version of the vector space model presented in this section on the introductory descriptions given in Salton's later works. In practice, implementations of the vector space model often eliminate both length normalization and the IDF factor in document vectors, in part to improve efficiency. Moreover, the Euclidean length normalization inherent in the cosine similarity measure has proved inadequate to handle collections containing mixtures of long and short documents, and substantial adjustments are required to support these collections. These efficiency and effectiveness issues are examined in Section 2.3.

The vector space model may be criticized for its entirely heuristic nature. Beyond intuition, its simple mathematics, and the experiments of Section 2.3, we do not provide further justification for it. IR models introduced in later chapters (Chapters 8 and 9) are more solidly grounded in theoretical frameworks. Perhaps as a result, these models are more adaptable, and are more readily extended to accommodate additional document features.

2.2.2 Proximity Ranking

The vector space ranking method from the previous section explicitly depends only on TF and IDF. In contrast, the method detailed in this section explicitly depends only on term proximity. Term frequency is handled implicitly; document frequency, document length, and other features play no role at all.

When the components of a term vector $\langle t_1, t_2, \ldots, t_n \rangle$ appear in close proximity within a document, it suggests that the document is more likely to be relevant than one in which the terms appear farther apart. Given a term vector $\langle t_1, t_2, ..., t_n \rangle$, we define a *cover* for the vector as an interval in the collection $[u, v]$ that contains a match to all the terms without containing a smaller interval $[u', v']$, $u \leq u' \leq v' \leq v$, that also contains a match to all the terms. The candidate phrases defined on page 39 are a special case of a cover in which all the terms appear in order.

In the collection of Table 2.1 (page 50), the intervals [1:2, 1:4], [3:2, 3:4], and [3:4, 3:8] are covers for the term vector ⟨"you", "sir"⟩. The interval [3:4, 3:16] is not a cover, even though both terms are contained within it, because it contains the cover [3:4, 3:8]. Similarly, there are two covers for the term vector ⟨"quarrel", "sir"⟩: [1:3, 1:4] and [2:1, 2:2].

Note that covers may overlap. However, a token matching a term t_i appears in at most $n \cdot l$ covers, where l is the length of the shortest postings list for the terms in the vector. To see that there may be as many as $n \cdot l$ covers for the term vector $\langle t_1, t_2, ..., t_n \rangle$, consider a collection in which all the terms occur in the same order the same number of times:

$$... \; t_1 \; ... \; t_2 \; ... \; t_3 \; ... \; t_n \; ... \; t_1 \; ... \; t_2 \; ... \; t_3 \; ... \; t_n \; ... \; t_1 \; ...$$

```
   nextCover (⟨t_1, ..., t_n⟩, position) ≡
1      v ← max_{1≤i≤n}(next(t_i, position))
2      if v = ∞ then
3          return [∞, ∞]
4      u ← min_{1≤i≤n}(prev(t_i, v + 1))
5      if docid(u) = docid(v) then
6          return [u, v]
7      else
8          return nextCover(⟨t_1, ..., t_n⟩, u)
```

Figure 2.10 Function to locate the next occurrence of a cover for the term vector $\langle t_1, ..., t_n \rangle$ after a given position.

We leave the demonstration that there may be no more than $n \cdot l$ covers to Exercise 2.8. A new cover starts at each occurrence of a term from the vector. Thus, the total number of covers for a term vector is constrained by $n \cdot l$ and does not depend on the length of the longest postings list L. With respect to proximity ranking, we define κ to be the number of covers for a term vector occurring in a document collection where $\kappa \leq n \cdot l$.

Perhaps not surprisingly, our algorithm to compute covers is a close cousin of the phrase searching algorithm in Figure 2.2 (page 36). The function in Figure 2.10 locates the next occurrence of a cover after a given position. On line 1, the algorithm determines the smallest position v such that the interval $[position, v]$ contains all the terms in the vector. A cover starting after u cannot end before this position. On line 4, the algorithm shrinks the interval ending at v, adjusting u so that no smaller interval ending at v contains all the terms. The check at line 5 determines whether u and v are contained in the same document. If not, **nextCover** is called recursively.

This last check is required only because the cover will ultimately contribute to a document's score for ranking. Technically, the interval [1:4, 2:1] is a perfectly acceptable cover for the term vector ⟨"quarrel","sir"⟩. However, in a schema-dependent index, a cover that crosses document boundaries is unlikely to be meaningful.

Ranking is based on two assumptions: (1) the shorter the cover, the more likely that the text containing the cover is relevant, and (2) the more covers contained in a document, the more likely that the document is relevant. These assumptions are consistent with intuition. The first assumption suggests that a score for an individual cover may be based on its length. The second assumption suggests that a document may be assigned a score by summing the individual scores of the covers contained within it. Combining these ideas, we score a document d containing covers $[u_1,v_1]$, $[u_2,v_2]$, $[u_3,v_3]$, ... using the formula

$$score(d) \;=\; \sum \left(\frac{1}{v_i - u_i + 1} \right). \tag{2.15}$$

```
    rankProximity (⟨t_1, ..., t_n⟩, k) ≡
1       [u, v] ← nextCover(⟨t_0, t_1, ..., t_n⟩, −∞)
2       d ← docid(u)
3       score ← 0
4       j ← 0
5       while u < ∞ do
6           if d < docid(u) then
7               j ← j + 1
8               Result[j].docid ← d
9               Result[j].score ← score
10              d ← docid(u)
11              score ← 0
12          score ← score + 1/(v − u + 1)
13          [u, v] ← nextCover(⟨t_1, ..., t_n⟩, u)
14      if d < ∞ then
15          j ← j + 1
16          Result[j].docid ← d
17          Result[j].score ← score
18      sort Result[1..j] by score
19      return Result[1..k]
```

Figure 2.11 Query processing for proximity ranking. The **nextCover** function from Figure 2.10 is called to generate each cover.

Query processing for proximity ranking is presented in Figure 2.11. Covers are generated by calls to the **nextCover** function and processed one by one in the while loop of lines 5–13. The number of covers κ in the collection is exactly equal to the number of calls to **nextCover** at line 13. When a document boundary is crossed (line 6), the score and docid are stored in an array of records *Result* (lines 8–9). After all covers are processed, information on the last document is recorded in the array (lines 14–17), the array is sorted by score (line 18), and the top k documents are returned (line 19).

As the **rankProximity** function makes calls to the **nextCover** function, the position passed as its second argument strictly increases. In turn, as the **nextCover** function makes successive calls to the **next** and **prev** methods, the values of their second arguments also strictly increase. As we did for the phrase searching algorithm of Section 2.1.1, we may exploit this property by implementing **next** and **prev** using galloping search. Following a similar argument, when galloping search is used, the overall time complexity of the **rankProximity** algorithm is $O\left(n^2\, l \cdot \log\left(L/l\right)\right)$.

Note that the time complexity is quadratic in n, the size of the term vector, because there may be $O(n \cdot l)$ covers in the worst case. Fortunately, the adaptive nature of the algorithm comes to our assistance again, giving a time complexity of $O\left(n \cdot \kappa \cdot \log\left(L/\kappa\right)\right)$.

For a document to receive a nonzero score, all terms must be present in it. In this respect, proximity ranking shares the behavior exhibited until recently by many commercial search

engines. When applied to the document collection of Table 2.1 to rank the collection with respect to the query ⟨"you", "sir"⟩, proximity ranking assigns a score of 0.33 to document 1, a score of 0.53 to document 3, and a score of 0 to the remaining documents.

When applied to rank the same collection with respect to the query ⟨"quarrel","sir"⟩, the method assigns scores of 0.50 to documents 1 and 2, and a score of 0.00 to documents 3 to 5. Unlike cosine similarity, the second occurrence of "sir" in document 2 does not contribute to the document's score. The frequency of individual terms is not a factor in proximity ranking; rather, the frequency and proximity of their co-occurrence are factors. It is conceivable that a document could include many matches to all the terms but contain only a single cover, with the query terms clustered into discrete groups.

2.2.3 Boolean Retrieval

Apart from the implicit Boolean filters applied by Web search engines, explicit support for Boolean queries is important in specific application areas such as digital libraries and the legal domain. In contrast to ranked retrieval, Boolean retrieval returns sets of documents rather than ranked lists. Under the Boolean retrieval model, a term t is considered to specify the set of documents containing it. The standard Boolean operators (AND, OR, and NOT) are used to construct Boolean queries, which are interpreted as operations over these sets, as follows:

A AND B	intersection of A and B ($A \cap B$)
A OR B	union of A and B ($A \cup B$)
NOT A	complement of A with respect to the document collection ($\bar{A}$)

where A and B are terms or other Boolean queries. For example, over the collection in Table 2.1, the query

("quarrel" OR "sir") AND "you"

specifies the set $\{1, 3\}$, whereas the query

("quarrel" OR "sir") AND NOT "you"

specifies the set $\{2, 5\}$.

Our algorithm for solving Boolean queries is another variant of the phrase searching algorithm of Figure 2.2 and the cover finding algorithm of Figure 2.10. The algorithm locates *candidate solutions* to a Boolean query where each candidate solution represents a *range* of documents that together satisfy the Boolean query, such that no smaller range of documents contained within it also satisfies the query. When the range represented by a candidate solution has a length of 1, this single document satisfies the query and should be included in the result set.

The same overall method of operation appears in both of the previous algorithms. In the phrase search algorithm, lines 1–6 identify a range containing all the terms in order, such that no smaller range contained within it also contains all the terms in order. In the cover finding algorithm, lines 1–4 similarly locate all the terms as close together as possible. In both algorithms an additional constraint is then applied.

To simplify our definition of our Boolean search algorithm, we define two functions that operate over Boolean queries, extending the **nextDoc** and **prevDoc** methods of schema-dependent inverted indices.

$\mathbf{docRight}(Q, u)$ —	end point of the first candidate solution to Q starting after document u
$\mathbf{docLeft}(Q, v)$ —	start point of the last candidate solution to Q ending before document v

For terms we define:

$$\begin{aligned} \mathbf{docRight}(t, u) &\equiv \mathbf{nextDoc}(t, u) \\ \mathbf{docLeft}(t, v) &\equiv \mathbf{prevDoc}(t, v) \end{aligned}$$

and for the AND and OR operators we define:

$$\begin{aligned} \mathbf{docRight}(A\ \mathbf{AND}\ B, u) &\equiv \max(\mathbf{docRight}(A, u), \mathbf{docRight}(B, u)) \\ \mathbf{docLeft}(A\ \mathbf{AND}\ B, v) &\equiv \min(\mathbf{docLeft}(A, v), \mathbf{docLeft}(B, v)) \\ \mathbf{docRight}(A\ \mathbf{OR}\ B, u) &\equiv \min(\mathbf{docRight}(A, u), \mathbf{docRight}(B, u)) \\ \mathbf{docLeft}(A\ \mathbf{OR}\ B, v) &\equiv \max(\mathbf{docLeft}(A, v), \mathbf{docLeft}(B, v)) \end{aligned}$$

To determine the result for a given query, these definitions are applied recursively. For example:

$$\begin{aligned} &\mathbf{docRight}((\text{“quarrel” OR “sir”})\ \text{AND “you”}, 1) \\ &\quad \equiv \max(\mathbf{docRight}(\text{“quarrel” OR “sir”}, 1), \mathbf{docRight}(\text{“you”}, 1)) \\ &\quad \equiv \max(\min(\mathbf{docRight}(\text{“quarrel”}, 1), \mathbf{docRight}(\text{“sir”}, 1)), \mathbf{nextDoc}(\text{“you”}, 1)) \\ &\quad \equiv \max(\min(\mathbf{nextDoc}(\text{“quarrel”}, 1), \mathbf{nextDoc}(\text{“sir”}, 1)), 3) \\ &\quad \equiv \max(\min(2, 2), 3) \\ &\quad \equiv 3 \end{aligned}$$

$$\begin{aligned} &\mathbf{docLeft}((\text{“quarrel” OR “sir”})\ \text{AND “you”}, 4) \\ &\quad \equiv \min(\mathbf{docLeft}(\text{“quarrel” OR “sir”}, 4), \mathbf{docLeft}(\text{“you”}, 4)) \\ &\quad \equiv \min(\max(\mathbf{docLeft}(\text{“quarrel”}, 4), \mathbf{docLeft}(\text{“sir”}, 4)), \mathbf{prevDoc}(\text{“you”}, 4)) \\ &\quad \equiv \min(\max(\mathbf{prevDoc}(\text{“quarrel”}, 4), \mathbf{prevDoc}(\text{“sir”}, 4)), 3) \\ &\quad \equiv \min(\max(2, 3), 3) \\ &\quad \equiv 3 \end{aligned}$$

```
   nextSolution (Q, position) ≡
1     v ← docRight(Q, position)
2     if v = ∞ then
3        return ∞
4     u ← docLeft(Q, v + 1)
5     if u = v then
6        return u
7     else
8        return nextSolution (Q, v)
```

Figure 2.12 Function to locate the next solution to the Boolean query Q after a given position. The function **nextSolution** calls **docRight** and **docLeft** to generate a candidate solution. These functions make recursive calls that depend on the structure of the query.

Definitions for the NOT operator are more problematic, and we ignore the operator until after we present the main algorithm.

Figure 2.12 presents the **nextSolution** function, which locates the next solution to a Boolean query after a given position. The function calls **docRight** and **docLeft** to generate a candidate solution. Just after line 4, the interval $[u,v]$ contains this candidate solution. If the candidate solution consists of a single document, it is returned. Otherwise, the function makes a recursive call. Given this function, all solutions to Boolean query Q may be generated by the following:

```
u ← −∞
while u < ∞ do
   u ← nextSolution(Q, u)
   if u < ∞ then
      report docid(u)
```

Using a galloping search implementation of nextDoc and prevDoc, the time complexity of this algorithm is $O(n \cdot l \cdot \log(L/l))$, where n is the number of terms in the query. If a docid or frequency index is used, and positional information is not recorded in the index, l and L represent the lengths of the shortest and longest postings lists of the terms in the query as measured by the number of documents. The reasoning required to demonstrate this time complexity is similar to that of our phrase search algorithm and proximity ranking algorithm. When considered in terms of the number of candidate solutions κ, which reflects the adaptive nature of the algorithm, the time complexity becomes $O(n \cdot \kappa \cdot \log(L/\kappa))$. Note that the call to the **docLeft** method in line 4 of the algorithm can be avoided (see Exercise 2.9), but it helps us to analyze the complexity of the algorithm, by providing a clear definition of a candidate solution.

We ignored the NOT operator in our definitions of **docRight** and **docLeft**. Indeed, it is not necessary to implement general versions of these functions in order to implement the NOT operator. Instead, De Morgan's laws may be used to transform a query, moving any NOT

operators inward until they are directly associated with query terms:

$$\begin{aligned} \textbf{NOT}\ (A\ \ \textbf{AND}\ \ B) &\equiv \textbf{NOT}\ A\ \ \textbf{OR}\ \ \textbf{NOT}\ B \\ \textbf{NOT}\ (A\ \ \textbf{OR}\ \ B) &\equiv \textbf{NOT}\ A\ \ \textbf{AND}\ \ \textbf{NOT}\ B \end{aligned}$$

For example, the query

"william" AND "shakespeare" AND NOT ("marlowe" OR "bacon")

would be transformed into

"william" AND "shakespeare" AND (NOT "marlowe" AND NOT "bacon").

This transformation does not change the number of AND and OR operators appearing in the query, and hence does not change the number of terms appearing in the query (n). After appropriate application of De Morgan's laws, we are left with a query containing expressions of the form **NOT** t, where t is a term. In order to process queries containing expressions of this form, we require corresponding definitions of **docRight** and **docLeft**. It is possible to write these definitions in terms of **nextDoc** and **prevDoc**.

```
docRight(NOT t, u) ≡
    u' ← nextDoc(t, u)
    while u' = u + 1 do
        u ← u'
        u' ← nextDoc(t, u)
    return u + 1
```

Unfortunately, this approach introduces potential inefficiencies. Although this definition will exhibit acceptable performance when few documents contain the term t, it may exhibit unacceptable performance when most documents contain t, essentially reverting to the linear scan of the postings list that we avoided by introducing galloping search. Moreover, the equivalent implementation of **docLeft**(**NOT** t, v) requires a scan *backward* through the postings list, violating the requirement necessary to realize the benefits of galloping search.

Instead, we may implement the NOT operator directly over the data structures described in Section 2.1.2, extending the methods supported by our inverted index with explicit methods for **nextDoc**(**NOT** t, u) and **prevDoc**(**NOT** t, v). We leave the details for Exercise 2.5.

2.3 Evaluation

Our presentation of both cosine similarity and proximity ranking relies heavily on intuition. We appeal to intuition to justify the representation of documents and queries as vectors, to justify the determination of similarity by comparing angles, and to justify the assignment of higher

weights when terms appear more frequently or closer together. This reliance on intuition can be accepted only when the methods are effective in practice. Moreover, an implementation of a retrieval method must be efficient enough to compute the results of a typical query in adequate time to satisfy the user, and possible trade-offs between efficiency and effectiveness must be considered. A user may not wish to wait for a longer period of time — additional seconds or even minutes — in order to receive a result that is only slightly better than a result she could have received immediately.

2.3.1 Recall and Precision

Measuring the effectiveness of a retrieval method depends on human assessments of relevance. In some cases, it might be possible to infer these assessments implicitly from user behavior. For example, if a user clicks on a result and then quickly backtracks to the result page, we might infer a negative assessment. Nonetheless, most published information retrieval experiments are based on manual assessments created explicitly for experimental purposes, such as the assessments for the TREC experiments described in Chapter 1. These assessments are often binary — an assessor reads the document and judges it *relevant* or *not relevant* with respect to a topic. TREC experiments generally use these binary judgments, with a document being judged relevant if any part of it is relevant.

For example, given the information need described by TREC topic 426 (Figure 1.8 on page 25), a user might formulate the Boolean query

(("law" AND "enforcement") OR "police") AND ("dog" OR "dogs").

Running this query over the TREC45 collection produces a set of 881 documents, representing 0.17% of the half-million documents in the collection.

In order to determine the effectiveness of a Boolean query such as this one, we compare two sets: (1) the set of documents returned by the query, *Res*, and (2) the set of relevant documents for the topic contained in the collection, *Rel*. From these two sets we may then compute two standard effectiveness measures: *recall* and *precision*.

$$\text{recall} = \frac{|Rel \cap Res|}{|Rel|} \tag{2.16}$$

$$\text{precision} = \frac{|Rel \cap Res|}{|Res|} \tag{2.17}$$

In a nutshell, recall indicates the fraction of relevant documents that appears in the result set, whereas precision indicates the fraction of the result set that is relevant.

According to official NIST judgments, there are 202 relevant documents in the TREC45 test collection for topic 426. Our query returns 167 of these documents, giving a precision of 0.190 and a recall of 0.827. A user may find this result acceptable. Just 35 relevant documents are

outside the result set. However, in order to find a relevant document, the user must read an average of 4.28 documents that are not relevant.

You will sometimes see recall and precision combined into a single value known as an *F-measure*. The simplest F-measure, F_1, is the harmonic mean of recall and precision:

$$F_1 = \frac{2}{\frac{1}{R} + \frac{1}{P}} = \frac{2 \cdot R \cdot P}{R + P}, \tag{2.18}$$

where R represents recall and P represents precision. In comparison with the arithmetic mean $((R + P)/2)$, the harmonic mean enforces a balance between recall and precision. For example, if we return the entire collection as the result of a query, recall will be 1 but precision will almost always be close to 0. The arithmetic mean gives a value greater than 0.5 for such a result. In contrast, the harmonic mean gives a value of $2P/(1 + P)$, which will be close to 0 if P is close to 0.

This formula may be generalized through a *weighted harmonic mean* to allow greater emphasis to be placed on either precision or recall,

$$\frac{1}{\alpha \frac{1}{R} + (1 - \alpha) \frac{1}{P}}. \tag{2.19}$$

where $0 \leq \alpha \leq 1$. For $\alpha = 0$, the measure is equivalent to precision. For $\alpha = 1$, it is equivalent to recall. For $\alpha = 0.5$ it is equivalent to Equation 2.18. Following standard practice (van Rijsbergen, 1979, Chapter 7), we set $\alpha = 1/(\beta^2 + 1)$ and define the F-measure as

$$F_\beta = \frac{(\beta^2 + 1) \cdot R \cdot P}{\beta^2 \cdot R + P}, \tag{2.20}$$

where β may be any real number. Thus, F_0 is recall and F_∞ is precision. Values of $|\beta| < 1$ place emphasis on recall; values of $|\beta| > 1$ place emphasis on precision.

2.3.2 Effectiveness Measures for Ranked Retrieval

If the user is interested in reading only one or two relevant documents, ranked retrieval may provide a more useful result than Boolean retrieval. To extend our notions of recall and precision to the ordered lists returned by ranked retrieval algorithms, we consider the top k documents returned by a query, $Res[1..k]$, and define:

$$\text{recall@}k = \frac{|Rel \cap Res[1..k]|}{|Rel|} \tag{2.21}$$

$$\text{precision@}k = \frac{|Rel \cap Res[1..k]|}{|Res[1..k]|}, \tag{2.22}$$

where precision@k is often written as P@k. If we treat the title of topic 426 as a term vector for ranked retrieval, ⟨"law", "enforcement", "dogs"⟩, proximity ranking gives P@10 = 0.400 and recall@10 = 0.0198. If the user starts reading from the top of the list, she will find four relevant documents in the top ten.

By definition, recall@k increases monotonically with respect to k. Conversely, if a ranked retrieval method adheres to the Probability Ranking Principle defined in Chapter 1 (i.e., ranking documents in order of decreasing probability of relevance), then P@k will tend to decrease as k increases. For topic 426, proximity ranking gives

k	10	20	50	100	200	1000
P@k	0.400	0.450	0.380	0.230	0.115	0.023
recall@k	0.020	0.045	0.094	0.114	0.114	0.114

Since only 82 documents contain all of the terms in the query, proximity ranking cannot return 200 or 1,000 documents with scores greater than 0. In order to allow comparison with other ranking methods, we compute precision and recall at these values by assuming that the empty lower ranks contain documents that are not relevant. All things considered, a user may be happier with the results of the Boolean query. But of course the comparison is not really fair. The Boolean query contains terms not found in the term vector, and perhaps would require more effort to craft. Using the same term vector, cosine similarity gives

k	10	20	50	100	200	1000
P@k	0.000	0.000	0.000	0.060	0.070	0.051
recall@k	0.000	0.000	0.000	0.030	0.069	0.253

Cosine ranking performs surprisingly poorly on this query, and it is unlikely that a user would be happy with these results.

By varying the value of k, we may trade precision for recall, accepting a lower precision in order to identify more relevant documents and vice versa. An IR experiment might consider k over a range of values, with lower values corresponding to a situation in which the user is interested in a quick answer and is willing to examine only a few results. Higher values correspond to a situation in which the user is interested in discovering as much information as possible about a topic and is willing to explore deep into the result list. The first situation is common in Web search, where the user may examine only the top one or two results before trying something else. The second situation may occur in legal domains, where a case may turn on a single precedent or piece of evidence and a thorough search is required.

As we can see from the example above, even at $k = 1000$ recall may be considerably lower than 100%. We may have to go very deep into the results to increase recall substantially beyond this point. Moreover, because many relevant documents may not contain any of the query terms at

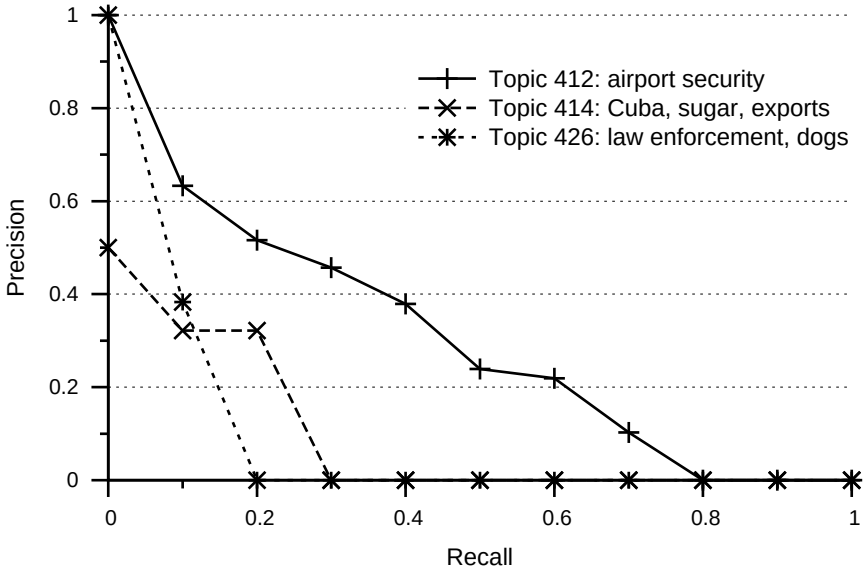


Figure 2.13 Eleven-point interpolated recall-precision curves for three TREC topics over the TREC45 collection. Results were generated with proximity ranking.

all, it is usually not possible to achieve 100% recall without including documents with 0 scores. For simplicity and consistency, information retrieval experiments generally consider only a fixed number of documents, often the top $k = 1000$. At higher levels we simply treat precision as being equal to 0. When conducting an experiment, we pass this value for k as a parameter to the retrieval function, as shown in Figures 2.9 and 2.11.

In order to examine the trade-off between recall and precision, we may plot a *recall-precision curve*. Figure 2.13 shows three examples for proximity ranking. The figure plots curves for topic 426 and two other topics taken from the 1998 TREC adhoc task: topic 412 ("airport security") and topic 414 ("Cuba, sugar, exports"). For 11 recall points, from 0% to 100% by 10% increments, the curve plots the maximum precision achieved at that recall level or higher. The value plotted at 0% recall represents the highest precision achieved at any recall level. Thus, the highest precision achieved for topic 412 is 80%, for topic 414 is 50%, and for topic 426 is 100%. At 20% or higher recall, proximity ranking achieves a precision of up to 57% for topic 412, 32% for topic 414, but 0% for topic 426. This technique of taking the maximum precision achieved at or above a given recall level is called *interpolated precision*. Interpolation has the pleasant property of producing monotonically decreasing curves, which may better illustrate the trade-off between recall and precision.

As an indication of effectiveness across the full range of recall values, we may compute an *average precision* value, which we define as follows:

$$\frac{1}{|Rel|} \cdot \sum_{i=1}^{k} \text{relevant}(i) \times \text{P@}i \tag{2.23}$$

where relevant(i) = 1 if the document at rank i is relevant (i.e., if $Res[i] \in Rel$) and 0 if it is not. Average precision represents an approximation of the area under a (noninterpolated) recall-precision curve. Over the top one thousand documents returned for topic 426, proximity ranking achieves an average precision of 0.058; cosine similarity achieves an average precision of 0.016.

So far, we have considered effectiveness measures for a single topic only. Naturally, performance on a single topic tells us very little, and a typical IR experiment will involve fifty or more topics. The standard procedure for computing effectiveness measures over a set of topics is to compute the measure on individual topics and then take the arithmetic mean of these values. In the IR research literature, values stated for P@k, recall@k, and other measures, as well as recall-precision curves, generally represent averages over a set of topics. You will rarely see values or plots for individual topics unless authors wish to discuss specific characteristics of these topics. In the case of average precision, its arithmetic mean over a set of topics is explicitly referred to as *mean average precision* or MAP, thus avoiding possible confusion with averaged P@k values.

Partly because it encapsulates system performance over the full range of recall values, and partly because of its prevalence at TREC and other evaluation forums, the reporting of MAP values for retrieval experiments was nearly ubiquitous in the IR literature until a few years ago. Recently, various limitations of MAP have become apparent and other measures have become more widespread, with MAP gradually assuming a secondary role.

Unfortunately, because it is an average of averages, it is difficult to interpret MAP in a way that provides any clear intuition regarding the actual performance that might be experienced by the user. Although a measure such as P@10 provides less information on the overall performance of a system, it does provide a more understandable number. As a result, we report both P@10 and MAP in experiments throughout the book. In Part III, we explore alternative effectiveness measures, comparing them with precision, recall, and MAP.

For simplicity, and to help guarantee consistency with published results, we suggest you do not write your own code to compute effectiveness measures. NIST provides a program, `trec_eval`[1] that computes a vast array of standard measures, including P@k and MAP. The program is the standard tool for computing results reported at TREC. Chris Buckley, the creator and maintainer of `trec_eval`, updates the program regularly, often including new measures as they appear in the literature.

[1] `trec.nist.gov/trec_eval`

Table 2.5 Effectiveness measures for selected retrieval methods discussed in this book.

	TREC45				GOV2			
	1998		1999		2004		2005	
Method	**P@10**	**MAP**	**P@10**	**MAP**	**P@10**	**MAP**	**P@10**	**MAP**
Cosine (2.2.1)	0.264	0.126	0.252	0.135	0.120	0.060	0.194	0.092
Proximity (2.2.2)	0.396	0.124	0.370	0.146	0.425	0.173	0.562	0.230
Cosine (raw TF)	0.266	0.106	0.240	0.120	0.298	0.093	0.282	0.097
Cosine (TF docs)	0.342	0.132	0.328	0.154	0.400	0.144	0.466	0.151
BM25 (Ch. 8)	0.424	0.178	0.440	0.205	0.471	0.243	0.534	0.277
LMD (Ch. 9)	0.450	0.193	0.428	0.226	0.484	0.244	0.580	0.293
DFR (Ch. 9)	0.426	0.183	0.446	0.216	0.465	0.248	0.550	0.269

Effectiveness results

Table 2.5 presents MAP and P@10 values for various retrieval methods over our four test collections. The first row provides values for the cosine similarity ranking described in Section 2.2.1. The second row provides values for the proximity ranking method described in Section 2.2.2.

As we indicated in Section 2.2.1, a large number of variants of cosine similarity have been explored over the years. The next two lines of the table provide values for two of them. The first of these replaces Equation 2.14 (page 57) with raw TF values, $f_{t,d}$, the number of occurrences of each term. In the case of the TREC45 collection, this change harms performance but substantially improves performance on the GOV2 collection. For the second variant (the fourth row) we omitted both document length normalization and document IDF values (but kept IDF in the query vector). Under this variant we compute a score for a document simply by taking the inner product of this unnormalized document vector and the query vector:

$$score(q,d) \;=\; \sum_{t \in (q \cap d)} q_t \cdot \log\left(\frac{N}{N_t}\right) \cdot (\log(f_{t,d}) + 1)\,. \tag{2.24}$$

Perhaps surprisingly, this change substantially improves performance to roughly the level of proximity ranking.

How can we explain this improvement? The vector space model was introduced and developed at a time when documents were of similar length, generally being short abstracts of books or scientific articles. The idea of representing a document as a vector represents the fundamental inspiration underlying the model. Once we think of a document as a vector, it is not difficult to take the next step and imagine applying standard mathematical operations to these vectors, including addition, normalization, inner product, and cosine similarity. Unfortunately, when it is applied to collections containing documents of different lengths, vector normalization does

Table 2.6 Selected ranking formulae discussed in later chapters. In these formulae the value q_t represents query term frequency, the number of times term t appears in the query. b and k_1, for BM25, and μ, for LMD, are free parameters set to $b = 0.75$, $k_1 = 1.2$, and $\mu = 1000$ in our experiments.

Method	Formula
BM25 (Ch. 8)	$\sum_{t \in q} q_t \cdot (f_{t,d} \cdot (k_1 + 1))/(k_1 \cdot ((1 - b) + b \cdot (l_d/l_{avg})) + f_{t,d}) \cdot \log(N/N_t)$
LMD (Ch. 9)	$\sum_{t \in q} q_t \cdot (\log(f_{t,d} + \mu \cdot l_t/\mathcal{C}) - \log(l_d + \mu))$
DFR (Ch. 9)	$\sum_{t \in q} q_t \cdot (\log(1 + l_t/N) + {f'}_{t,d} \cdot \log(1 + N/l_t))/({f'}_{t,d} + 1)$, where ${f'}_{t,d} = f_{t,d} \cdot \log(1 + l_{avg}/l_d)$

not successfully adjust for these varying lengths. For this reason, by the early 1990s this last variant of the vector space model had become standard.

The inclusion of the vector space model in this chapter is due more to history than anything else. Given its long-standing influence, no textbook on information retrieval would be complete without it. In later chapters we will present ranked retrieval methods with different theoretical underpinnings. The final three rows of Table 2.5 provide results for three of these methods: a method based on probabilistic models (BM25), a method based on language modeling (LMD), and a method based on divergence from randomness (DFR). The formulae for these methods are given in Table 2.6. As can be seen from the table, all three of these methods depend only on the simple features discussed at the start of Section 2.2. All outperform both cosine similarity and proximity ranking.

Figure 2.14 plots 11-point interpolated recall-precision curves for the 1998 TREC adhoc task, corresponding to the fourth and fifth columns of Table 2.5. The LM and DFR methods appear to slightly outperform BM25, which in turn outperforms proximity ranking and the TF docs version of cosine similarity.

2.3.3 Building a Test Collection

Measuring the effectiveness of a retrieval method depends on human assessments of relevance. Given a topic and a set of documents, an assessor reads each document and makes a decision: Is it relevant or not? Once a topic is well understood, assessment can proceed surprisingly quickly. Once up to speed, an assessor might be able to make a judgment in 10 seconds or less. Unfortunately, even at a rate of one judgment every ten seconds, an assessor working eight-hour days with breaks and holidays would spend nearly a year judging a single topic over the half-million documents in TREC45, and her entire career judging a topic over the 25 million documents in GOV2.

Given the difficulty of judging a topic over the entire collection, TREC and other retrieval efforts depend on a technique known as *pooling* to limit the number of judgments required. The standard TREC experimental procedure is to accept one or more experimental *runs* from each group participating in an adhoc task. Each of these runs consists of the top 1,000 or 10,000

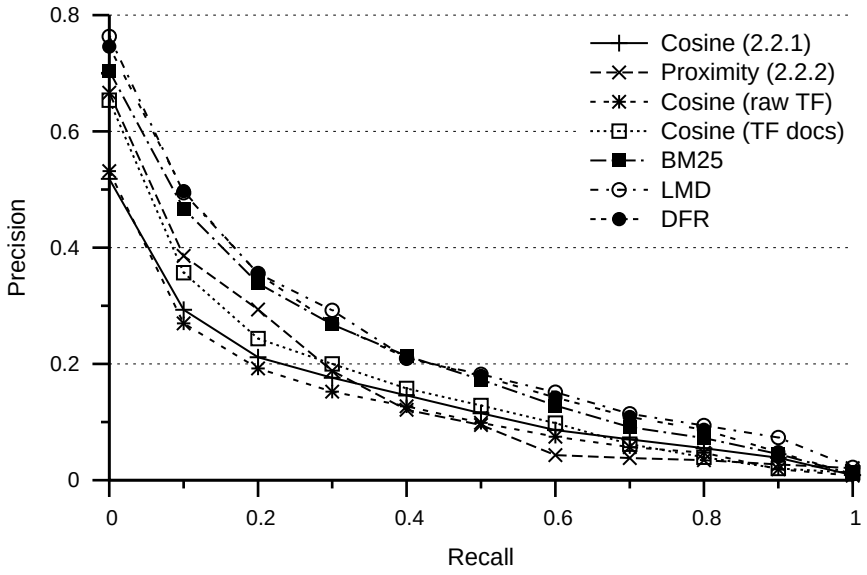


Figure 2.14 11-point interpolated recall-precision curves for various basic retrieval methods on the 1999 TREC adhoc task (topics 401–450, TREC45 document collection).

documents for each topic. Under the pooling method, the top 100 or so documents from each run are then *pooled* into a single set and presented to assessors in random order for judging. Even with dozens of participants the size of this pool may be less than a thousand documents per topic, because the same document may be returned by multiple systems. Even if assessment is done very carefully, with each judgment taking several minutes on average, it is possible for an assessor to work through this pool in less than a week. Often, less time is required.

The goal of the pooling method is to reduce the amount of assessment effort while retaining the ability to make reasonable estimates of precision and recall. For runs contributing to a pool, the values for P@k and recall@k are accurate at least down to the pool depth. Beyond this depth many documents will still be judged, because other runs will have ranked these documents above the pool depth. MAP is still calculated over the entire 1,000 or 10,000 documents in each run, with *unjudged documents treated as not relevant*. Because the top documents have the greatest impact on MAP, estimates made in this way are still acceptably accurate.

Armed with the pooling method, the creation of a test collection at TREC and similar evaluation efforts generally proceed as follows:

1. Obtain an appropriate document set either from public sources or through negotiation with its owners.
2. Develop at least 50 topics, the minimum generally considered acceptable for a meaningful evaluation.

3. Release the topics to the track participants and receive their experimental runs.
4. Create the pools, judge the topics, and return the results to the participants.

When developing a topic, a preliminary test of the topic on a standard IR system should reveal a reasonable mixture of relevant and non-relevant documents in the top ranks. With many relevant documents it may be too easy for systems to achieve a MAP value close to 1.0, making it difficult to distinguish one system from another. With few relevant documents, many systems will fail to achieve a MAP value above 0.0, again making it difficult to distinguish between them. Borrowing our criteria from Goldilocks in *The Story of the Three Bears*, we want not too many, not too few, but just right.

Ideally, a test collection would be reusable. A researcher developing a new retrieval method could use the documents, topics, and judgments to determine how the new method compares with previous methods. Although the pooling method allows us to create a test collection in less than a lifetime, you may have some legitimate concern regarding its ability to create such reusable collections. A truly novel retrieval method might surface many relevant documents that did not form part of the original pool and are not judged. If unjudged documents are treated as nonrelevant for the purposes of computing evaluation measures, this novel method might receive an unduly harsh score. Tragically, it might even be discarded by the researcher and lost to science forever. Alternative effectiveness measures to address this and related issues are covered in Chapter 12.

2.3.4 Efficiency Measures

In addition to evaluating a system's retrieval effectiveness, it is often necessary to evaluate its efficiency because it affects the cost of running the system and the happiness of its users. From a user's perspective the only efficiency measure of interest is *response time*, the time between entering a query and receiving the results. *Throughput*, the average number of queries processed in a given period of time, is primarily of interest to search engine operators, particularly when the search engine is shared by many users and must cope with thousands of queries per second. Adequate resources must be available to cope with the query load generated by these users without compromising the response time experienced by each. Unacceptably long response times may lead to unhappy users and, as a consequence, fewer users.

A realistic measurement of throughput and its trade-off against response time requires a detailed simulation of query load. For the purposes of making simple efficiency measurements, we may focus on response time and consider the performance seen by a single user issuing one query at a time on an otherwise unburdened search engine. Chapter 13 provides a more detailed discussion of throughput along with a broader discussion of efficiency.

A simple but reasonable procedure for measuring response time is to execute a full query set, capturing the start time from the operating system at a well-defined point just before the first query is issued, and the end time just after the last result is generated. The time to execute the full set is then divided by the number of queries to give an *average response time* per query.

Table 2.7 Average time per query for the Wumpus implementation of Okapi BM25 (Chapter 8), using two different index types and four different query sets.

	TREC45		GOV2	
Index type	**1998**	**1999**	**2004**	**2005**
Schema-independent index	61 ms	57 ms	1686 ms	4763 ms
Frequency index	41 ms	41 ms	204 ms	202 ms

Results may be discarded as they are generated, rather than stored to disk or sent over a network, because the overhead of these activities can dominate the response times, particularly with small collections.

Before executing a query set, the IR system should be restarted or reset to clear any information precomputed and stored in memory from previous queries, and the operating system's I/O cache must be flushed. To increase the accuracy of the measurements, the query set may be executed multiple times, with the system reset each time and an average of the measured execution times used to compute the average response time.

An an example, Table 2.7 compares the average response time of a schema-independent index versus a frequency index, using the Wumpus implementation of the Okapi BM25 ranking function (shown in Table 2.6). We use Okapi BM25 for this example, because the Wumpus implementation of this ranking function has been explicitly tuned for efficiency.

The efficiency benefits of using the frequency index are obvious, particularly on the larger GOV2 collection. The use of a schema-independent index requires the computation at run-time of document and term statistics that are precomputed in the frequency index. To a user, a 202 ms response time would seem instantaneous, whereas a 4.7 sec response time would be a noticeable lag. However, with a frequency index it is not possible to perform phrase searches or to apply ranking functions to elements other than documents.

The efficiency measurements shown in the table, as well as others throughout the book, were made on a rack-mounted server based on an AMD Opteron processor (2.8 GHz) with 2 GB of RAM. A detailed performance overview of the computer system is in Appendix A.

2.4 Summary

This chapter has covered a broad range of topics, often in considerable detail. The key points include the following:

- We view an inverted index as an abstract data type that may be accessed through the methods and definitions summarized in Table 2.4 (page 52). There are four important variants — docid indices, frequency indices, positional indices, and schema-independent indices — that differ in the type and format of the information they store.

- Many retrieval algorithms — such as the phrase searching, proximity ranking, and Boolean query processing algorithms presented in this chapter — may be efficiently implemented using galloping search. These algorithms are adaptive in the sense that their time complexity depends on characteristics of the data, such as the number of candidate phrases.
- Both ranked retrieval and Boolean filters play important roles in current IR systems. Reasonable methods for ranked retrieval may be based on simple document and term statistics, such as term frequency (TF), inverse document frequency (IDF), and term proximity. The well-known cosine similarity measure represents documents and queries as vectors and ranks documents according to the cosine of the angle between them and the query vector.
- Recall and precision are two widely used effectiveness measures. A trade-off frequently exists between them, such that increasing recall leads to a corresponding decrease in precision. Mean average precision (MAP) represents a standard method for summarizing the effectiveness of an IR system over a broad range of recall levels. MAP and P@10 are the principal effectiveness measures reported throughout the book.
- Response time represents the efficiency of an IR system as experienced by a user. We may make reasonable estimates of minimum response time by processing queries sequentially, reading them one at a time and reporting the results of one query before starting the next query.

2.5 Further Reading

Inverted indices have long been the standard data structure underlying the implementation of IR systems (Faloutsos, 1985; Knuth, 1973, pages 552–554). Other data structures have been proposed, generally with the intention of providing more efficient support for specific retrieval operations. However, none of these data structures provide the flexibility and generality of inverted indices, and they have mostly fallen out of use.

Signature files were long viewed as an important competitor to inverted indices, particularly when disk and memory were at a premium (Faloutsos, 1985; Faloutsos and Christodoulakis, 1984; Zobel et al., 1998). Signature files are intended to provide efficient support for Boolean queries by quickly eliminating documents that do not match the query. They provide one method for implementing the filtering step of the two-step retrieval process described on page 53. Unfortunately, false matches can be reported by signature files, and they cannot be easily extended to support phrase queries and ranked retrieval.

A suffix tree (Weiner, 1973) is a search tree in which every path from the root to a leaf corresponds to a unique suffix in a text collection. A suffix array (Gonnet, 1987; Manber and Myers, 1990) is an array of pointers to unique suffixes in the collection that is sorted in lexicographical

order. Both suffix trees and suffix arrays are intended to support efficient phrase searching, as well as operations such as lexicographical range searching and structural retrieval (Gonnet et al., 1992). Unfortunately, both data structures provide poor support for ranked retrieval.

Galloping search was first described by Bentley and Yao (1976), who called it "unbounded search". The algorithms presented in this chapter for phrase searching, proximity ranking, and Boolean retrieval may all be viewed as variants of algorithms for computing set operations over sorted lists. The term "galloping" was coined by Demaine et al. (2000), who present and analyze adaptive algorithms for set union, intersection and difference. Other algorithms for computing the intersection of sorted lists are described by Barbay and Kenyon (2002) and by Baeza-Yates (2004). Our abstraction of an inverted index into an ADT accessed through a small number of simple methods is based on Clarke et al. (2000). The proximity ranking algorithm presented in this chapter is a simplified version of the algorithm presented in that paper.

The vector space model was developed into the version presented in this chapter through a long process that can be traced back at least to the work of Luhn in the 1950s (Luhn, 1957, 1958). The long-standing success and influence of the model are largely due to the efforts of Salton and his research group at Cornell, who ultimately made the model a key element of their SMART IR system, the first version of which became operational in the fall of 1964 (Salton, 1968, page 422). Until the early 1990s SMART was one of a small number of platforms available for IR research, and during that period it was widely adopted by researchers outside Salton's group as a foundation for their own work.

By the time of the first TREC experiments in the early 1990s, the vector space model had evolved into a form very close to that presented in this chapter (Buckley et al., 1994). A further development, introduced at the fourth TREC in 1995, was the addition of an explicit adjustment for document length, known as *pivoted document length normalization* (Singhal et al., 1996). This last development has been accepted as an essential part of the vector space model for ranked retrieval but not for other applications, such as clustering or classification. After Salton's death in 1995, development of the SMART system continued under the supervision of Buckley, and it remained a competitive research system more than ten years later (Buckley, 2005).

Latent Semantic Analysis (LSA) is an important and well-known extension of the vector space model (Deerwester et al., 1990) that we do not cover in this book. LSA applies singular value decomposition, from linear algebra, to reduce the dimensionality of the term-vector space. The goal is to reduce the negative impact of synonymy — multiple terms with the same meaning — by merging related words into common dimensions. A related technique, Probabilistic Latent Semantic Analysis (PLSA), approaches the same goal by way of a probabilistic model (Hofmann, 1999). Unfortunately, the widespread application of LSA and PLSA to IR systems has been limited by the difficulty of efficient implementation.

The Spam song (page 38) has had perhaps the greatest influence of any song on computer terminology. The song was introduced in the twelfth episode of the second season of *Monty Python's Flying Circus*, which first aired on December 15, 1970, on BBC Television.

2.6 Exercises

Exercise 2.1 Simplify **next**("the", **prev**("the", **next** ("the", $-\infty$))).

Exercise 2.2 Consider the following version of the phrase searching algorithm shown in Figure 2.2 (page 36).

```
nextPhrase2 (t1t2...tn, position) ≡
    u ← next(t1, position)
    v ← u
    for i ← 2 to n do
        v ← next(ti, v)
    if v = ∞ then
        return [∞, ∞]
    if v − u = n − 1 then
        return [u, v]
    else
        return nextPhrase2(t1t2...tn, v − n)
```

It makes the same number of calls to **next** as the version shown in the figure but does not make any calls to **prev**. Explain how the algorithm operates. How does it find the next occurrence of a phrase after a given position? When combined with galloping search to find all occurrences of a phrase, would you expect improved efficiency compared to the original version? Explain.

Exercise 2.3 Using the methods of the inverted index ADT, write an algorithm that locates all intervals corresponding to speeches (<SPEECH>...</SPEECH>). Assume the schema-independent indexing shown in Figure 2.1, as illustrated by Figure 1.3.

Exercise 2.4 Using the methods of the inverted index ADT, write an algorithm that locates all speeches by a witch in Shakespeare's plays. Assume the schema-independent indexing shown in Figure 2.1, as illustrated by Figure 1.3.

Exercise 2.5 Assume we have a schema-independent inverted index for Shakespeare's plays. We decide to treat each PLAY as a separate document.

(a) Write algorithms to compute the following statistics, using only the **first**, **last**, **next**, and **prev** methods: (i) N_t, the number of documents in the collection containing the term t; (ii) $f_{t,d}$, the number of times term t appears in document d; (iii) l_d, the length of document d; (iv) l_{avg}, average document length; and (v) N, total number of documents.

(b) Write algorithms to implement the **docid**, **offset**, **firstDoc**, **lastDoc**, **nextDoc**, and **prevDoc** methods, using only the **first**, **last**, **next**, and **prev** methods.

For simplicity, you may treat the position of each ⟨PLAY⟩ tag as the document ID for the document starting at that position; there's no requirement for document IDs to be assigned sequentially. For example, *Macbeth* would be assigned docid 745142 (with an initial occurrence of "first witch" at [745142:265, 745142:266]).

Exercise 2.6 When postings lists are stored on disk rather than entirely in memory, the time required by an inverted index method call can depend heavily on the overhead required for accessing the disk. When conducting a search for a phrase such as "Winnie the Pooh", which contains a mixture of frequent and infrequent terms, the first method call for "winnie" (**next**("winnie", $-\infty$)) might read the entire postings list from disk into an array. Further calls would return results from this array using galloping search. Similarly, the postings list for "pooh" could be read into memory during its first method call. On the other hand, the postings list for "the" may not fit into memory in its entirety, and only a very small number of these postings may form part of the target phrase.

Ideally, all of the postings for "the" would not be read from disk. Instead, the methods might read a small part of the postings list at a time, generating multiple disk accesses as the list is traversed and skipping portions of the list as appropriate.

Because each method call for "the" may generate a disk access, it may be preferable to search for the phrase "Winnie the Pooh" by immediately checking for an occurrence of "pooh" (two positions away) after locating each occurrence of "winnie". If "pooh" does not occur in the expected location, we may abandon this occurrence of "winnie" and continue to another occurrence without checking for an occurrence of "the" between the two. Only when we have located an occurrence of "winnie" and "pooh" correctly spaced ("winnie ___ pooh") do we need to make a method call for "the".

To generalize, given a phrase "$t_1 t_2 ... t_n$", a search algorithm might work from the least frequent term to the most frequent term, abandoning a potential phrase once a term is found to be missing and thus minimizing the number of calls for the most frequent terms. Using the basic definitions and methods of our inverted index ADT, complete the details of this algorithm. What is the time complexity of the algorithm in terms of calls to the **next** and **prev** methods? What is the overall time complexity if galloping search is used to implement these methods?

Can the algorithm be considered adaptive? (*Hint:* Consider a search in the "hello world" collection on page 38. How many method calls must be made?)

Exercise 2.7 Coordination level is the number of terms from a vector $\langle t_1, ..., t_n \rangle$ that appear in a document. Coordination level may range from 1 to n. Using the methods of our inverted index ADT, write an algorithm that ranks documents according to their coordination level.

Exercise 2.8 Demonstrate that the term vector $\langle t_1, ..., t_n \rangle$ may have at most $n \cdot l$ covers (see Section 2.2.2), where l is the length of the shortest postings list for the terms in the vector.

Exercise 2.9 Taking a hint from Exercise 2.2, design a version of the algorithm shown in Figure 2.10 (page 61) that does not make a call to the **docLeft** method.

Exercise 2.10 If $\alpha = 1/(\beta^2 + 1)$, show that Equations 2.19 and 2.20 are equivalent.

Exercise 2.11 (project exercise) Implement the in-memory array-based version of inverted indices presented in Section 2.1.2, including all three versions of the **next** method: binary search, sequential scan and galloping search. Using your implementation of inverted indices, implement the phrase searching algorithm of Section 2.1.1.

To test your phrase searching implementation, we suggest a corpus 256MB or larger that fits comfortably in the main memory of your computer. The corpus created in Exercise 1.9 (or a subset) would be suitable. Select at least 10,000 phrases of various lengths from your corpus and verify that your implementation successfully locates these phrases. Your selection should include short phrases of length 2–3, longer phrases of length 4–10, and very long phrases with length 100 or greater. Include phrases containing both frequent and infrequent terms. At least half of the phrases should contain a least one very common term, such as an article or a preposition.

Following the guidelines of Section 2.3.4, compare the efficiency of phrase searching, using your three versions of the **next** method. Compute average response times. Plot response time against phrase length for all three versions. For clarity, you may need to plot the results for linear scan separately from those of the other two methods.

Select another set of phrases, all with length 2. Include phrases containing combinations of frequent and infrequent terms. Plot response time against L (the length of the longest postings list) for all three versions. Plot response time against l (the length of the shortest postings list) for all three versions.

Optional extension: Implement the phrase searching algorithm described in Exercise 2.6. Repeat the performance measurements using this new implementation.

Exercise 2.12 (project exercise) Implement cosine similarity ranking, as described in Section 2.2.1. Test your implementation using the test collection developed in Exercise 2.13 or with any other available collection, such as a TREC collection.

Exercise 2.13 (project exercise) As a class project, undertake a TREC-style adhoc retrieval experiment, developing a test collection based on Wikipedia. Continuing from Exercises 1.9 and 1.10, each student should contribute enough topics to make a combined set of 50 topics or more. Each student should implement their own retrieval system, perhaps by starting with an open-source IR system and extending it with techniques from later chapters, or with techniques of their own invention. Each student should then run the titles of the topics as queries to their system. Pool and judge the results, with each student judging the topics they created. The design and implementation of an interactive judging interface might be undertaken as a sub-project by a group of interested students. Use `trec_eval` to compare runs and techniques.

2.7 Bibliography

Baeza-Yates, R. (2004). A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching*, pages 400–408. Istanbul, Turkey.

Barbay, J., and Kenyon, C. (2002). Adaptive intersection and t-threshold problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 390–399. San Francisco, California.

Bentley, J. L., and Yao, A. C. C. (1976). An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87.

Buckley, C. (2005). The SMART project at TREC. In Voorhees, E. M., and Harman, D. K., editors, *TREC — Experiment and Evaluation in Information Retrieval*, chapter 13, pages 301–320. Cambridge, Massachusetts: MIT Press.

Buckley, C., Salton, G., Allan, J., and Singhal, A. (1994). Automatic query expansion using SMART: TREC 3. In *Proceedings of the 3rd Text REtrieval Conference*. Gaithersburg, Maryland.

Clarke, C. L. A., Cormack, G. V., and Tudhope, E. A. (2000). Relevance ranking for one to three term queries. *Information Processing & Management*, 36(2):291–311.

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407.

Demaine, E. D., López-Ortiz, A., and Munro, J. I. (2000). Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 743–752. San Francisco, California.

Faloutsos, C. (1985). Access methods for text. *ACM Computing Surveys*, 17(1):49–74.

Faloutsos, C., and Christodoulakis, S. (1984). Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288.

Gonnet, G. H. (1987). PAT *3.1 — An Efficient Text Searching System — User's Manual*. University of Waterloo, Canada.

Gonnet, G. H., Baeza-Yates, R. A., and Snider, T. (1992). New indices for text — PAT trees and PAT arrays. In Frakes, W. B., and Baeza-Yates, R., editors, *Information Retrieval — Data Structures and Algorithms*, chapter 5, pages 66–82. Englewood Cliffs, New Jersey: Prentice Hall.

Hofmann, T. (1999). Probabilistic latent semantic indexing. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 50–57. Berkeley, California.

Knuth, D. E. (1973). *The Art of Computer Programming*, volume 3. Reading, Massachusetts: Addison-Wesley.

Luhn, H. P. (1957). A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of Research and Development*, 1(4):309–317.

Luhn, H. P. (1958). The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2(2):159–165.

Manber, U., and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327. San Francisco, California.

Salton, G. (1968). *Automatic Information Organziation and Retrieval*. New York: McGraw-Hill.

Singhal, A., Salton, G., Mitra, M., and Buckley, C. (1996). Document length normalization. *Information Processing & Management*, 32(5):619–633.

van Rijsbergen, C. J. (1979). *Information Retrieval* (2nd ed.). London, England: Butterworths.

Weiner, P. (1973). Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11. Iowa City, Iowa.

Zobel, J., Moffat, A., and Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490.