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

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, ⟨13⟩)	1; 21
am	1; 3	1; ⟨3, 1, ⟨6⟩⟩	1; 14
as	1; 3	1; ⟨3, 2, ⟨11, 15⟩⟩	2; 19, 23
better	1; 4	1; ⟨4, 1, ⟨2⟩⟩	1; 26
do	2; 1, 3	2; (1, 1, ⟨1⟩), ⟨3, 1, ⟨3⟩⟩	2; 1, 11
for	1; 3	1; (3, 1, ⟨7⟩)	1; 15
good	1; 3	1; (3, 1, ⟨12⟩)	1; 20
i	1; 3	1; (3, 2, ⟨5, 9⟩)	2; 13, 17
if	1; 3	1; (3, 1, ⟨1⟩)	1; 9
man	1; 3	1; (3, 1, ⟨14⟩)	1; 22
no	2; 2, 4	2; (2, 1, ⟨3⟩), ⟨4, 1, ⟨1⟩⟩	2; 7, 25
quarrel	2; 1, 2	2; ⟨1, 1, ⟨3⟩⟩, (2, 1, ⟨1⟩)	2; 3, 5
serve	1; 3	1; (3, 1, ⟨10⟩)	1; 18
sir	4; 1, 2, 3, 5	4; (1, 1, ⟨4⟩), (2, 2, ⟨2, 4⟩), (3, 1, (4⟩), (5, 1, ⟨2⟩)	5; 4, 6, 8, 12, 28
well	1; 5	1; (5, 1, ⟨1⟩)	1; 27
you	2; 1, 3	2; (1, 1, ⟨2⟩), (3, 3, ⟨2, 8, 16⟩)	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

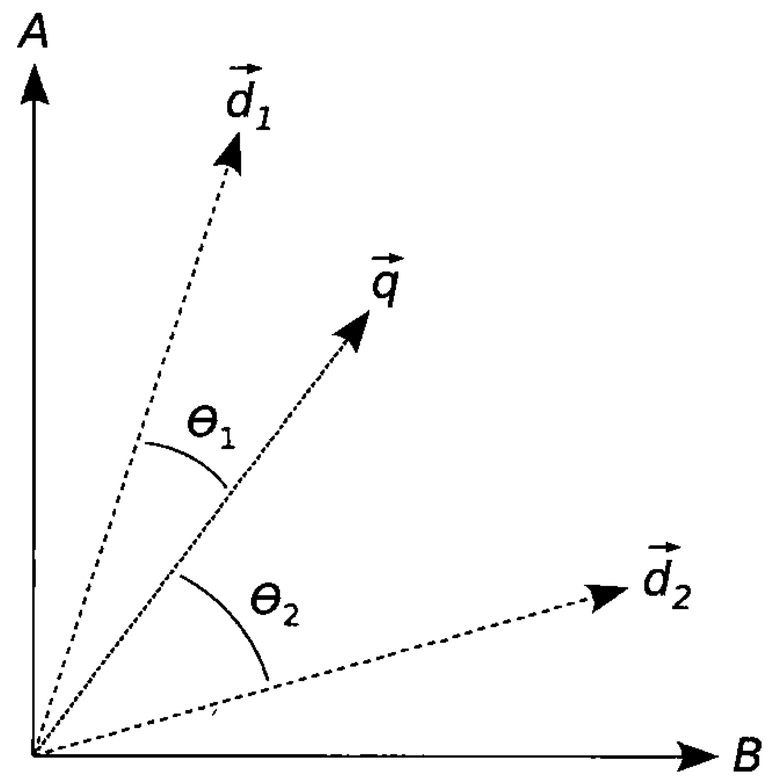


Figure 2.8 Document similarity under the vector space model. Angles are computed between a query vector $\vec{q}$ and two document vectors $\vec{d}_1$ and $\vec{d}_2$. Because $\theta_1 < \theta_2$, d_1 should be ranked higher than d_2.

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

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

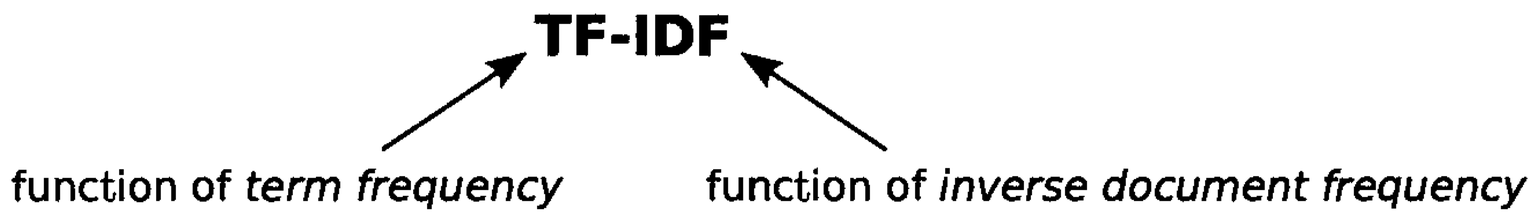
the dot product of the document and query vectors normalized to unit length. Provided all components of the vectors are nonnegative, the value of this *cosine similarity measure* ranges from 0 to 1, with its value increasing with increasing similarity.

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 / |d⃗|) · (q⃗ / |q⃗|)
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:

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