the successive searches, which is probably due to JVM warmup. These results point out that performance testing is tricky business, but it's necessary in many environments. Because of the strong effect your environment has on performance, we urge you to perform your own tests with your own environment. Performance testing is covered in more detail in section 6.5, page 213.

If you choose to expose searching through RMI in this manner, you'll likely want to create a bit of infrastructure to coordinate and manage issues such as closing an index and how the server deals with index updates (remember, the searcher sees a snapshot of the index and must be reopened to see changes).

5.7 Leveraging term vectors

Term vectors are a new feature in Lucene 1.4, but they aren't new as an information retrieval concept. A term vector is a collection of term-frequency pairs. Most of us probably can't envision vectors in hyperdimensional space, so for visualization purposes, let's look at two documents that contain only the terms cat and dog. These words appear various times in each document. Plotting the term frequencies of each document in X, Y coordinates looks something like figure 5.6. What gets interesting with term vectors is the angle between them, as you'll see in more detail in section 5.7.2.

To enable term-vector storage, during indexing you enable the store term vectors attribute on the desired fields. Field.Text and Field.Unstored have additional overloaded methods with a boolean storeTermVector flag in the signature. Setting this value to true turns on the optional term vector support for the field, as we did for the subject field when indexing our book data (see figure 5.7).
Figure 5.7 Enabling term vectors during indexing

Retrieving term vectors for a field in a given document by ID requires a call to an IndexReader method:

```java
TermFreqVector termFreqVector = reader.getTermFreqVector(id, "subject");
```

A TermFreqVector instance has several methods for retrieving the vector information, primarily as matching arrays of Strings and ints (the term value and frequency in the field, respectively). You can use term vectors for some interesting effects, such as finding documents "like" a particular document, which is an example of latent semantic analysis. We built a BooksLikeThis feature as well as a proof-of-concept categorizer that can tell us the most appropriate category for a new book, as you'll see in the following sections.

5.7.1 Books like this

It would be nice to offer other choices to the customers of our bookstore when they're viewing a particular book. The alternatives should be related to the original book, but associating alternatives manually would be labor-intensive and would require ongoing effort to keep up to date. Instead, we use Lucene's boolean query capability and the information from one book to look up other books that are similar. Listing 5.11 demonstrates a basic approach for finding books like each one in our sample data.

Listing 5.11 Books like this

```java
public class BooksLikeThis {
    public static void main(String[] args) throws IOException {
        String indexDir = System.getProperty("index.dir");
        FSDirectory directory = FSDirectory.getDirectory(indexDir, false);
        IndexReader reader = IndexReader.open(directory);
        int numDocs = reader.maxDoc();
        BooksLikeThis blt = new BooksLikeThis(reader);
```
for (int i = 0; i < numDocs; i++) {
    System.out.println();
    Document doc = reader.document(i);
    System.out.println(doc.get("title"));

    Document[] docs = bit.doesLike(i, 10);  
    if (docs.length == 0) {
        System.out.println("None like this");
    }
    for (int j = 0; j < docs.length; j++) {
        Document likeThisDoc = docs[j];
        System.out.println("→ " + likeThisDoc.get("title"));
    }  
}

private IndexReader reader;
private IndexSearcher searcher;

public BooksLikeThis(IndexReader reader) {
    this.reader = reader;
    searcher = new IndexSearcher(reader);
}

public Document[] docsLike(int id, int max) throws IOException {
    Document doc = reader.document(id);

    String[] authors = doc.getValues("author");
    BooleanQuery authorQuery = new BooleanQuery();
    for (int i = 0; i < authors.length; i++) {
        String author = authors[i];
        authorQuery.add(new TermQuery(new Term("author", author)), false, false);
    }
    authorQuery.setBoost(2.0f);

    TermFreqVector vector =
        reader.getTermFreqVector(id, "subject");

    BooleanQuery subjectQuery = new BooleanQuery();
    for (int j = 0; j < vector.size(); j++) {
        TermQuery tq = new TermQuery(
            new Term("subject", vector.getTerms()[j]));
        subjectQuery.add(tq, false, false);
    }

    BooleanQuery likeThisQuery = new BooleanQuery();
    likeThisQuery.add(authorQuery, false, false);
    likeThisQuery.add(subjectQuery, false, false);
As an example, we iterate over every book document in the index and find books like each one.

Here we look up books that are like this one.

Books by the same author are considered alike and are boosted so they will likely appear before books by other authors.

Using the terms from the subject term vectors, we add each to a boolean query.

We combine the author and subject queries into a final boolean query.

We exclude the current book, which would surely be the best match given the other criteria, from consideration.

In 1, we used a different way to get the value of the author field. It was indexed as multiple fields, in the manner (shown in more detail in section 8.4, page 284), where the original author string is a comma-separated list of author(s) of a book:

```
String[] authors = author.split(",");
for (int i = 0; i < authors.length; i++) {
    doc.add(Field.Keyword("author", authors[i]));
}
```

The output is interesting, showing how our books are connected through author and subject:

```
A Modern Art of Education
   -> Mindstorms

Imperial Secrets of Health and Longevity
None like this
```

5.7.2 What do we do next?

Each book is categorized by a field containing a comma-separated list of possible categories. This example shows the decision making process.
If you’d like to see the actual query used for each, uncomment the output lines toward the end of the doc!

The books-like this example could have been done without term vectors, and we aren’t really using them as vectors in this case. We’ve only used the convenience of getting the terms for a given field. Without term vectors, the subject field could have been reanalyzed or indexed such that individual subject terms were added separately in order to get the list of terms for that field (see section 8.4 for discussion of how the sample data was indexed). Our next example also uses the frequency component to a term vector in a much more sophisticated manner.

5.7.2 What category?

Each book in our index is given a single primary category: For example, this book is categorized as "/technology/computers/programming". The best category placement for a new book may be relatively obvious, or (more likely) several possible categories may seem reasonable. You can use term vectors to automate the decision. We’ve written a bit of code that builds a representative subject vector for
each existing category. This representative, archetypical, vector is the sum of all vectors for each document's subject field vector.

With these representative vectors precomputed, our end goal is a calculation that can, given some subject keywords for a new book, tell us what category is the best fit. Our test case uses two example subject strings:

```java
public void testCategorization() throws Exception {
    assertEquals("/technology/computers/programming/methodology",
                getCategory("extreme agile methodology");
    assertEquals("/education/pedagogy",
                getCategory("montessori education philosophy");
}
```

The first assertion says that, based on our sample data, if a new book has "extreme agile methodology" keywords in its subject, the best category fit is "/technology/computers/programming/methodology". The best category is determined by finding the closest category angle-wise in vector space to the new book's subject.

The test `setUp()` builds vectors for each category:

```java
public class CategorizerTest extends LiaTestCase {
    Map categoryMap;

    protected void setUp() throws Exception {
        super.setUp();

        categoryMap = new TreeMap();

        buildCategoryVectors();
        //dumpCategoryVectors();
    }
```

Our code builds category vectors by walking every document in the index and aggregating book subject vectors into a single vector for the book's associated category. Category vectors are stored in a Map, keyed by category name. The value of each item in the category map is another map keyed by term, with the value an Integer for its frequency:

```java
private void buildCategoryVectors() throws IOException {
    IndexReader reader = IndexReader.open(directory);
    int maxDoc = reader.maxDoc();

    for (int i = 0; i < maxDoc; i++) {
```
if (!reader.isDeleted(i)) {
    Document doc = reader.document(i);
    String category = doc.get("category");
    Map vectorMap = (Map) categoryMap.get(category);
    if (vectorMap == null) {
        vectorMap = new TreeMap();
        categoryMap.put(category, vectorMap);
    }
    TermFreqVector termFreqVector =
        reader.getTermFreqVector(i, "subject");
    addTermFreqToMap(vectorMap, termFreqVector);
}

A book’s term frequency vector is added to its category vector in addTermFreqToMap. The arrays returned by getTerms() and getTermFrequencies() align with one another such that the same position in each refers to the same term:

private void addTermFreqToMap(Map vectorMap,
                               TermFreqVector termFreqVector) {
    String[] terms = termFreqVector.getTerms();
    int[] freqs = termFreqVector.getTermFrequencies();
    for (int i = 0; i < terms.length; i++) {
        String term = terms[i];
        if (vectorMap.containsKey(term)) {
            Integer value = (Integer) vectorMap.get(term);
            vectorMap.put(term,
                           new Integer(value.intValue() + freqs[i]));
        } else {
            vectorMap.put(term, new Integer(freqs[i]));
        }
    }
}

That was the easy part—building the category vector maps—because it only involved addition. Computing angles between vectors, however, is more involved mathematically. In the simplest two-dimensional case, as shown earlier in figure 5.6, two categories (A and B) have unique term vectors based on aggregation (as we’ve just done). The closest category, angle-wise, to a new book’s subjects is the match we’ll choose. Figure 5.8 shows the equation for computing an angle between two vectors.
\[ \cos \theta = \frac{A \cdot B}{||A|| \cdot ||B||} \]

Figure 5.8
Formula for computing the angle between two vectors.

Our getCategory method loops through all categories, computing the angle between each category and the new book. The smallest angle is the closest match, and the category name is returned:

```java
private String getCategory(String subject) {
    String[] words = subject.split(" ");

    Iterator categoryIterator = categoryMap.keySet().iterator();
    double bestAngle = Double.MAX_VALUE;
    String bestCategory = null;

    while (categoryIterator.hasNext()) {
        String category = (String) categoryIterator.next();

        double angle = computeAngle(words, category);
        if (angle < bestAngle) {
            bestAngle = angle;
            bestCategory = category;
        }
    }

    return bestCategory;
}
```

We assume that the subject string is in a whitespace-separated form and that each word occurs only once. The angle computation takes these assumptions into account to simplify a part of the computation. Finally, computing the angle between an array of words and a specific category is done in `computeAngle`, shown in listing 5.12.

**Listing 5.12: Computing term vector angles for a new book against a given category**

```java
private double computeAngle(String[] words, String category) {
    Map vectorMap = (Map) categoryMap.get(category);

    int dotProduct = 0;
    int sumOfSquares = 0;
    for (int i = 0; i < words.length; i++) {
        String word = words[i];
        int categoryWordFreq = 0;
        if (vectorMap.containsKey(word)) {
            categoryWordFreq = ((Integer) vectorMap.get(word)).intValue();
        }
        sumOfSquares += categoryWordFreq * categoryWordFreq;
        dotProduct += categoryWordFreq * words[i].toLowerCase().length();
    }
    return dotProduct / Math.sqrt(sumOfSquares); // Compute the angle
}
```

1. The `categoryMap` contains the category names as keys and the `Map` of term frequencies as values.
2. We iterate through each category to find the one with the smallest angle.

You can see that each method or, in fact, each piece of code is an important part of the building process.

### 5.8 Summary

This chapter explains how to build search engines. Contrary to popular belief, building a search engine is not a straightforward task, and there are many ways to approach the problem. Some of the key steps include:

- Indexing documents
- Creating query expansions
- Handling user behavior
- Incorporating user feedback

It is important to understand the limitations of each method and to be aware of the trade-offs involved in choosing one approach over another.
dctProduct += categoryWordFreq;
sumOfSquares += categoryWordFreq * categoryWordFreq;

```java
double denominator;
if (sumOfSquares == words.length) {
    denominator = sumOfSquares;
} else {
    denominator = Math.sqrt(sumOfSquares) * Math.sqrt(words.length);
}

double ratio = dotProduct / denominator;
return Math.acos(ratio);
```

1. The calculation is optimized with the assumption that each word in the words array has a frequency of 1.

2. We multiply the square root of N by the square root of N is N. This shortcut prevents a precision issue where the ratio could be greater than 1 (which is an illegal value for the inverse cosine function).

You should be aware that computing term vector angles between two documents or, in this case, between a document and an archetypical category, is computationally-intensive. It requires square-root and inverse cosine calculations and may be prohibitive in high-volume indexes.

5.8 **Summary**

This chapter has covered some diverse ground, highlighting Lucene's additional built-in search features. Sorting is a dramatic new enhancement that gives you control over the ordering of search results. The new SpanQuery family leverages term-position information for greater searching precision. Filters constrain document search space, regardless of the query. Lucene includes support for multiple (including parallel) and remote index searching, giving developers a head start on distributed and scalable architectures. And finally, the new term vector feature enables interesting effects, such as "like this" term vector angle calculations.

Is this the end of the searching story? Not quite. Lucene also includes several ways to extend its searching behavior, such as custom sorting, filtering, and query expression parsing, which we cover in the following chapter.