Covers Apache Lucene 3.0

Lucene IN ACTION

SECOND EDITION

Michael McCandless
Erik Hatcher
Otis Gospodnetić

Foreword by Doug Cutting

SAMPLE CHAPTER

MANNING

Lucene in Action, Second Edition

by Michael McCandless,
Erik Hatcher, and Otis Gospodnetić

Chapter 3

brief contents

3 Adding search to your application

This chapter covers

- Querying a Lucene index
- Using Lucene's diverse built-in queries
- Working with search results
- Understanding Lucene scoring
- Parsing human-entered query expressions

The previous chapter showed you in great detail how to build a search index in preparation for searching. As fun as indexing is, it's just a means to an end, a necessary evil, and its value only becomes clear once you enable searching on top of it. In this chapter, we'll show you how to capitalize on all your indexing efforts. For example, consider this scenario:

> *Give me a list of all books published in the last 12 months on the subject of "Java" where "open source" or "Jakarta" is mentioned in the contents. Restrict the results to only books that are on special. Oh, and under the covers, also ensure that books mentioning "Apache" are picked up, because we explicitly specified "Jakarta." And make it snappy, on the order of milliseconds for response time.*

Such scenarios are easily handled with Lucene, even when your content source consists of millions of documents, but it'll take us three search chapters to see the necessary functionality in Lucene to achieve this example in full. We start with the frequently used search APIs, described in this chapter. Indeed, the majority of applications using Lucene can provide excellent search functionality using only what's covered in this chapter. But a search engine is only as good as its search capabilities, and it's here where Lucene shines. After visiting analysis in chapter 4—important because it's used during both indexing and searching—we'll return to search in chapter 5, delving into Lucene's more advanced search capabilities, as well as in chapter 6, elaborating on ways to extend Lucene's classes for even greater, customized searching power.

In this chapter we begin with a simple example showing that the code you write to implement search is generally no more than a few lines long. Next we illustrate the scoring formula, providing a deep look into one of Lucene's most special attributes. With this example and a high-level understanding of how Lucene ranks search results, we'll spend most of the chapter exploring the diversity of Lucene's built-in search queries, including searching by specific term, by range (numeric or textual), by prefix or wildcard, by phrase, or by fuzzy term matching. We show how the powerful `BooleanQuery` can join any number of clauses together, including arbitrarily nested clauses, using Boolean constraints. Finally we show how simple it is to create a complex search query from a text search expression entered by the end user using Lucene's built-in `QueryParser`.

This is our first of three chapters about Lucene's search APIs, so we'll limit our discussion for now to the primary classes that you'll typically use for search integration, shown in table 3.1.

When you're *querying* a Lucene index, a `TopDocs` instance, containing an ordered array of `ScoreDoc`, is returned. The array is ordered by *score* by default. Lucene computes a score (a numeric value of relevance) for each document, given a query. The `ScoreDoc`s themselves aren't the actual matching documents, but rather references, via an integer *document ID*, to the documents matched. In most applications that display

Table 3.1 Lucene's primary searching API

Class	Purpose
`IndexSearcher`	Gateway to searching an index. All searches come through an `IndexSearcher` instance using any of the several overloaded `search` methods.
`Query` (and subclasses)	Concrete subclasses encapsulate logic for a particular query type. Instances of `Query` are passed to an `IndexSearcher`'s `search` method.
`QueryParser`	Processes a human-entered (and readable) expression into a concrete `Query` object.
`TopDocs`	Holds the top scoring documents, returned by `IndexSearcher.search`.
`ScoreDoc`	Provides access to each search result in `TopDocs`.

search results, users access only the first few documents, so it isn't necessary to retrieve the full document for all results; you need to retrieve for the current page only the documents that will be presented to the user. In fact, for very large indexes, it often wouldn't even be possible, or would take far too long, to collect all matching documents into available physical computer memory.

Let's see how easy it is to search with Lucene.

3.1 Implementing a simple search feature

Suppose you're tasked with adding search to an application. You've tackled getting the data indexed, using the APIs we covered in the last chapter, but now it's time to expose the full-text searching to the end users. It's hard to imagine that adding search could be any simpler than it is with Lucene. Obtaining search results requires only a few lines of code—literally. Lucene provides easy and highly efficient access to those search results, too, freeing you to focus on your application logic and UI around those results.

When you search with Lucene, you'll have a choice of either programmatically constructing your query or using Lucene's `QueryParser` to translate text entered by the user into the equivalent `Query`. The first approach gives you ultimate power, in that your application can expose whatever UI it wants, and your logic translates interactions from that UI into a `Query`. But the second approach is wonderfully easy to use, and offers a standard search syntax that all users are familiar with. In this section we'll show you how to make the simplest programmatic query, searching for a single term, and then we'll see how to use `QueryParser` to accept textual queries. In the sections that follow, we'll take this simple example further by detailing all the query types built into Lucene. We begin with the simplest search of all: searching for all documents that contain a single term.

3.1.1 Searching for a specific term

`IndexSearcher` is the central class used to search for documents in an index. It has several overloaded search methods. You can search for a specific term using the most commonly used search method. A term is a `String` value that's paired with its containing field name—in our example, subject.

NOTE The original text may have been normalized into terms by the analyzer, which may eliminate terms (such as stop words), convert terms to lowercase, convert terms to base word forms (*stemming*), or insert additional terms (*synonym processing*). It's crucial that the terms passed to `IndexSearcher` be consistent with the terms produced by analysis of the source documents during indexing. Chapter 4 discusses the analysis process in detail.

Using our example book data index, which is stored in the build/index subdirectory with the book's source code, we'll query for the words *ant* and *junit*, which are words we know were indexed. Listing 3.1 creates the term query, performs the search and asserts that the single expected document is found. Lucene provides several built-in `Query` types (see section 3.4), `TermQuery` being the most basic.

Listing 3.1 Simple searching with `TermQuery`

```
public class BasicSearchingTest extends TestCase {
  public void testTerm() throws Exception {
    Directory dir = TestUtil.getBookIndexDirectory();
    IndexSearcher searcher = new IndexSearcher(dir);

    Term t = new Term("subject", "ant");
    Query query = new TermQuery(t);
    TopDocs docs = searcher.search(query, 10);
    assertEquals("Ant in Action",
                 1, docs.totalHits);
    t = new Term("subject", "junit");
    docs = searcher.search(new TermQuery(t), 10);
    assertEquals("Ant in Action, " +
                 "JUnit in Action, Second Edition",
                 2, docs.totalHits);
    searcher.close();
    dir.close();
  }
}
```

Obtain directory from TestUtil

Create IndexSearcher

Confirm one hit for "ant"

Confirm two hits for "junit"

This is our first time seeing the `TestUtil.getBookIndexDirectory` method; it's quite simple:

```
public static Directory getBookIndexDirectory() throws IOException {
  return FSDirectory.open(new File(System.getProperty("index.dir")));
}
```

The `index.dir` property defaults to "build/index" in the build.xml ant script, so that when you run the tests using Ant from the command line, the index directory is set correctly. That index is built from the books under the data directory, using the `CreateTestIndex` tool (under the src/lia/common subdirectory). We use this method in many tests to retrieve the directory containing the index built from our test book data.

A `TopDocs` object is returned from our search. In a real application we'd step through the individual `ScoreDocs` representing the hits, but for this test we were only interested in checking that the proper number of documents were found.

Note that we close the searcher, and then the directory, after we are done. In general it's best to keep these open and share a single searcher for all queries that need to run. Opening a new searcher can be a costly operation because it must load and populate internal data structures from the index.

This example created a simple query (a single term). Next, we discuss how to transform a user-entered query expression into a `Query` object.

3.1.2 Parsing a user-entered query expression: QueryParser

Lucene's search methods require a `Query` object. *Parsing* a query expression is the act of turning a user-entered textual query such as "mock OR junit" into an appropriate `Query` object instance; in this case, the `Query` object would be an instance of `BooleanQuery` with two optional clauses, one for each term. The process is illustrated in figure 3.1. The code in listing 3.2 parses two query expressions and asserts that they

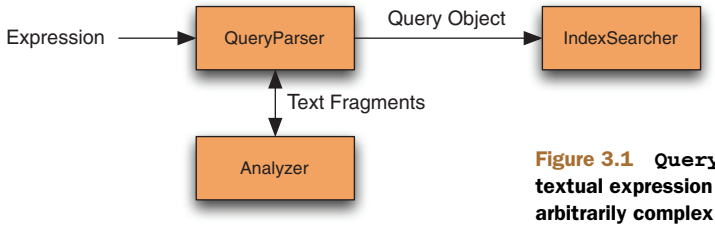

Figure 3.1 `QueryParser` translates a textual expression from the end user into an arbitrarily complex query for searching.

worked as expected. After returning the hits, we retrieve the title from the first document found.

NOTE Query expressions are similar to SQL expressions used to query a database in that the expression must be parsed into something at a lower level that the database server can understand directly.

Listing 3.2 `QueryParser`, which makes it trivial to translate search text into a `Query`

```
public void testQueryParser() throws Exception {
  Directory dir = TestUtil.getBookIndexDirectory();
  IndexSearcher searcher = new IndexSearcher(dir);

  QueryParser parser = new QueryParser(Version.LUCENE_30,
                                       "contents",
                                       new SimpleAnalyzer());

  Query query = parser.parse("+JUNIT +ANT -MOCK");
  TopDocs docs = searcher.search(query, 10);
  assertEquals(1, docs.totalHits);
  Document d = searcher.doc(docs.scoreDocs[0].doc);
  assertEquals("Ant in Action", d.get("title"));

  query = parser.parse("mock OR junit");
  docs = searcher.search(query, 10);
  assertEquals("Ant in Action, " +
               "JUnit in Action, Second Edition",
               2, docs.totalHits);

  searcher.close();
  dir.close();
}
```

Create QueryParser

Parse user's text

Lucene includes an interesting built-in feature that parses query expressions, available through the `QueryParser` class. It parses rich expressions such as the two shown (`"+JUNIT +ANT -MOCK"` and `"mock OR junit"`) into one of the `Query` implementations. The resulting `Query` instances can be very rich and complex! Dealing with human-entered queries is the primary purpose of the `QueryParser`. Once you have the `Query` object returned by `QueryParser`, the rest of the searching is identical to how you'd search programmatically.

As you can see in figure 3.1, `QueryParser` requires an *analyzer* to break pieces of the query text into terms. In the first expression, the query was entirely uppercased. The terms of the contents field, however, were lowercased when indexed. `QueryParser`, in

this example, uses `SimpleAnalyzer`, which lowercases the terms before constructing a `Query` object. (Analysis is covered in great detail in the next chapter, but it's intimately intertwined with indexing text and searching with `QueryParser`.) The main point regarding analysis to consider in this chapter is that you need to be sure to query on the actual terms indexed. `QueryParser` is the only searching piece that uses an analyzer. Querying through the API using `TermQuery` and the others discussed in section 3.4 doesn't require an analyzer but does rely on matching terms to what was indexed. Therefore, if you construct queries entirely programmatically you must ensure the terms included in all of your queries match the tokens produced by the analyzer used during indexing. In section 4.1.2, we talk more about the interactions of `QueryParser` and the analysis process.

Equipped with the examples shown thus far, you're more than ready to begin searching your indexes. There are, of course, many more details to know about searching. In particular, `QueryParser` requires additional explanation. Next is an overview of how to use `QueryParser`, which we return to in greater detail later in this chapter.

USING QUERYPARSER

Before diving into the details of `QueryParser` (which we do in section 3.5), let's first look at how it's used in a general sense. `QueryParser` is instantiated with `matchVersion` (`Version`), a field name (`String`), and an analyzer, which it uses to break the incoming search text into `Terms`:

```
QueryParser parser = new QueryParser(Version matchVersion,
                                     String field,
                                     Analyzer analyzer)
```

The `matchVersion` parameter instructs Lucene which release it should use for matching defaults and settings, in order to preserve backward compatibility. Note that in some cases, Lucene will emulate bugs from past releases. Section 1.4.1 describes `Version` in more detail.

The field name is the default field against which all terms will be searched, unless the search text explicitly requests matches against a different field name using the syntax "field:text" (more on this in section 3.5.11). Then, the `QueryParser` instance has a `parse()` method to allow for the simplest use:

```
public Query parse(String query) throws ParseException
```

The query String is the expression to be parsed, such as +cat +dog.

If the expression fails to parse, a `ParseException` is thrown, a condition that your application should handle in a graceful manner. `ParseException`'s message gives a reasonable indication of why the parsing failed; however, this description may be too technical for end users.

The `parse()` method is quick and convenient to use, but it may not be sufficient. There are various settings that can be controlled on a `QueryParser` instance, such as the default operator when multiple terms are used (which defaults to OR). These settings also include locale (for date parsing), default phrase slop (described in section 3.4.6), the minimum similarity and prefix length for fuzzy queries, the date resolution, whether to lowercase wildcard queries, and various other advanced settings.

HANDLING BASIC QUERY EXPRESSIONS WITH QUERYPARSER

`QueryParser` translates query expressions into one of Lucene's built-in query types. We'll cover each query type in section 3.4; for now, take in the bigger picture provided by table 3.2, which shows some examples of expressions and their translation.

Table 3.2 Expression examples that `QueryParser` handles

Query expression	Matches documents that…
`java`	Contain the term *java* in the default field
`java junit` `java OR junit`	Contain the term *java* or *junit*, or both, in the default field[a]
`+java +junit` `java AND junit`	Contain both *java* and *junit* in the default field
`title:ant`	Contain the term *ant* in the title field
`title:extreme -subject:sports` `title:extreme AND NOT subject:sports`	Contain *extreme* in the title field and don't have *sports* in the subject field
`(agile OR extreme) AND methodology`	Contain *methodology* and must also contain *agile* and/or *extreme*, all in the default field
`title:"junit in action"`	Contain the exact phrase "junit in action" in the title field
`title:"junit action"~5`	Contain the terms junit and *action* within five positions of one another, in the title field
`java*`	Contain terms that begin with *java*, like *javaspaces*, *javaserver*, *java.net*, and the exact tem *java* itself.
`java~`	Contain terms that are close to the word *java*, such as *lava*
`lastmodified: [1/1/09 TO 12/31/09]`	Have `lastmodified` field values between the dates January 1, 2009 and December 31, 2009

a. The default operator is OR. It can be set to AND (see section 3.5.6).

With this broad picture of Lucene's search capabilities, you're ready to dive into details. We'll revisit `QueryParser` in section 3.5, after we cover the more foundational pieces. Let's take a closer look at Lucene's `IndexSearcher` class.

3.2 Using IndexSearcher

Searching with Lucene is a surprisingly simple affair. You first create an instance of `IndexSearcher`, which opens the search index, and then use the `search` methods on that class to perform all searching. The returned `TopDocs` class represents the top

results, and you use that to present results to the user. Next we discuss how to handle pagination, and finally we show how to use Lucene's new (as of version 2.9) near-real-time search capability for fast turnaround on recently indexed documents. Let's begin with the creation of an `IndexSearcher`.

3.2.1 Creating an IndexSearcher

Like the rest of Lucene's primary API, `IndexSearcher` is simple to use. The classes involved are shown in figure 3.2. First, as with indexing, we'll need a directory. Most often you're searching an index in the file system:

```
Directory dir = FSDirectory.open(new File("/path/to/index"));
```

Section 2.10 describes alternate `Directory` implementations. Next we create an `IndexReader`:

```
IndexReader reader = IndexReader.open(dir);
```

Finally, we create the `IndexSearcher`:

```
IndexSearcher searcher = new IndexSearcher(reader);
```

`Directory`, which we've already seen in the context of indexing, provides the abstract file-like API. `IndexReader` uses that API to interact with the index files stored during indexing, and exposes the low-level API that `IndexSearcher` uses for searching. `IndexSearcher`'s APIs accept `Query` objects, for searching, and return `TopDocs` objects representing the results, as we discussed in section 3.2.3.

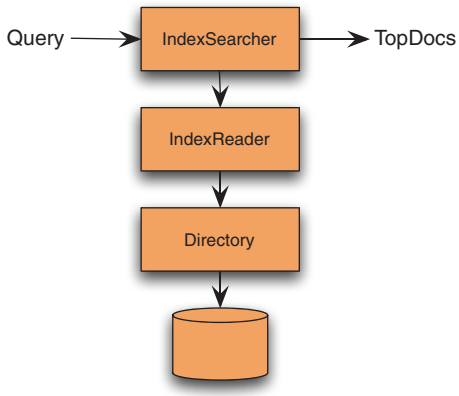

Figure 3.2 The relationship between the common classes used for searching

Note that it's `IndexReader` that does all the heavy lifting to open all index files and expose a low-level reader API, while `IndexSearcher` is a rather thin veneer. Because it's costly to open an `IndexReader`, it's best to reuse a single instance for all of your searches, and open a new one only when necessary.

NOTE Opening an `IndexReader` is costly, so you should reuse a single instance for all of your searching when possible, and limit how often you open a new one.

It's also possible to directly create the `IndexSearcher` from a directory, which creates its own private `IndexReader` under the hood, as we saw in chapter 1. If you go this route, you can retrieve the underlying `IndexReader` by calling `IndexSearcher`'s `getIndexReader` method, but remember that if you close the searcher it will also close this `IndexReader` because it had opened it.

`IndexReader` always searches a point-in-time snapshot of the index as it existed when the `IndexReader` was created. If you need to search changes to the index, you'll have to open a new reader. Fortunately, the `IndexReader.reopen` method is a resource-efficient means of obtaining a new `IndexReader` that covers all changes to the index but shares resources with the current reader when possible. Use it like this:

```
IndexReader newReader = reader.reopen();
if (reader != newReader) {
  reader.close();
  reader = newReader;
  searcher = new IndexSearcher(reader);
}
```

The reopen method only returns a new reader if there were changes in the index, in which case it's your responsibility to close the old reader and create a new `IndexSearcher`. In a real application, where multiple threads may still be searching using the old reader, you'll have to protect this code to make it thread safe. Section 11.2.2 provides a useful drop-in class that does this for you. Section 3.2.5 shows how to obtain a near-real-time `IndexReader` from an `IndexWriter`, which is even more resource efficient in cases where you have access to the `IndexWriter` making changes to the index. Now that we have an `IndexSearcher`, let's see how to search!

NOTE An `IndexSearcher` instance searches only the index as it existed at the time the `IndexSearcher` was instantiated. If indexing is occurring concurrently with searching, newer documents indexed won't be visible to searches. In order to see the new documents, you should open a new reader.

3.2.2 Performing searches

Once you have an `IndexSearcher`, simply call one of its `search` methods to perform a search. Under the hood, the search method does a tremendous amount of work, very quickly. It visits every single document that's a candidate for matching the search, only accepting the ones that pass every constraint on the query. Finally, it gathers the top results and returns them to you.

The main search methods available to an `IndexSearcher` instance are shown in table 3.3. In this chapter we only make use of the `search(Query, int)` method because many applications won't need to use the more advanced methods. The other search method signatures, including the filtering and sorting variants, are covered in chapter 5. Chapter 6 covers the customizable search methods that accept a `Collector` for gathering results.

Most of `IndexSearcher`'s search methods return `TopDocs`, which we cover next, to represent the returned results.

3.2.3 Working with TopDocs

Now that we've called `search`, we have a `TopDocs` object at our disposal that we can use for efficient access to the search results. Typically, you'll use one of the `search`

Table 3.3 Primary `IndexSearcher` search methods

`IndexSearcher.search` method signature	When to use
`TopDocs search(Query query, int n)`	Straightforward searches. The `int n` parameter specifies how many top-scoring documents to return.
`TopDocs search(Query query, Filter filter, int n)`	Searches constrained to a subset of available documents, based on filter criteria.
`TopFieldDocs search(Query query, Filter filter, int n, Sort sort)`	Searches constrained to a subset of available documents based on filter criteria, and sorted by a custom `Sort` object
`void search(Query query, Collector results)`	Used when you have custom logic to implement for each document visited, or you'd like to collect a different subset of documents than the top N by the sort criteria.
`void search(Query query, Filter filter, Collector results)`	Same as previous, except documents are only accepted if they pass the filter criteria.

methods that return a `TopDocs` object, as shown in table 3.3. Results are ordered by relevance—in other words, by how well each document matches the query (sorting results in other ways is discussed in section 5.2).

The `TopDocs` class exposes a small number of methods and attributes for retrieving the search results; they're listed in table 3.4. The attribute `TopDocs.totalHits` returns the number of matching documents. The matches, by default, are sorted in decreasing score order. The `TopDocs.scoreDocs` attribute is an array containing the requested number of top matches. Each `ScoreDoc` instance has a float score, which is the relevance score, and an int doc, which is the document ID that can be used to retrieve the stored fields for that document by calling `IndexSearcher.document(doc)`. Finally, `TopDocs.getMaxScore()` returns the best score across all matches; when you sort by relevance (the default), that will always be the score of the first result. But if you sort by other criteria and enable scoring for the search, as described in section 5.2, it will be the maximum score of all matching documents even when the best scoring document isn't in the top results by your sort criteria.

Table 3.4 `TopDocs` methods for efficiently accessing search results03_Ch03.fm

`TopDocs` method or attribute	Return value
`totalHits`	Number of documents that matched the search
`scoreDocs`	Array of `ScoreDoc` instances that contains the results
`getMaxScore()`	Returns best score of all matches, if scoring was done while searching (when sorting by field, you separately control whether scores are computed)

3.2.4 Paging through results

Presenting search results to end users most often involves displaying only the first 10 to 20 most relevant documents. Paging through `ScoreDocs` is a common requirement, although if you find users are frequently doing a lot of paging you should revisit your design: ideally the user almost always finds the result on the first page. That said, pagination is still typically needed. You can choose from a couple of implementation approaches:

- Gather multiple pages' worth of results on the initial search and keep the resulting `ScoreDocs` and `IndexSearcher` instances available while the user is navigating the search results.
- Requery each time the user navigates to a new page.

Requerying is most often the better solution. Requerying eliminates the need to store per-user state, which in a web application can be costly, especially with a large number of users. Requerying at first glance seems a waste, but Lucene's blazing speed more than compensates. Also, thanks to the I/O caching in modern operating systems, requerying will typically be fast because the necessary bits from disk will already be cached in RAM. Frequently users don't click past the first page of results anyway.

In order to requery, the original search is reexecuted, with a larger number of requested matches, and the results are displayed beginning on the desired page. How the original query is kept depends on your application architecture. In a web application where the user types in an expression that's parsed with `QueryParser`, the original expression could be made part of the links for navigating the pages and reparsed for each request, or the expression could be kept in a hidden HTML field or as a cookie.

Don't prematurely optimize your paging implementations with caching or persistence. First implement your paging feature with a straightforward requery approach; chances are you'll find this sufficient for your needs. Let's see an example of near-real-time search next.

3.2.5 Near-real-time search

One of the new features in Lucene's 2.9 release is near-real-time search, which enables you to rapidly search changes made to the index with an open `IndexWriter`, without having to first close or commit changes to that writer. Many applications make ongoing changes with an always open `IndexWriter` and require that subsequent searches quickly reflect these changes. If that `IndexWriter` instance is in the same JVM that's doing searching, you can use near-real-time search, as shown in listing 3.3.

This capability is referred to as *near*-real-time search, and not simply real-time search, because it's not possible to make strict guarantees about the turnaround time, in the same sense as a "hard" real-time OS is able to do. Lucene's near-real-time search is more like a "soft" real-time OS. For example, if Java decides to run a major garbage collection cycle, or if a large segment merge has just completed, or if your machine is struggling because there's not enough RAM, the turnaround time of the near-real-

time reader can be much longer. But in practice the turnaround time can be very fast (tens of milliseconds or less), depending on your indexing and searching throughput, and how frequently you obtain a new near-real-time reader.

In the past, without this feature, you'd have to call `commit` on the writer, and then reopen on your reader, but this can be time consuming since `commit` must sync all new files in the index, an operation that's often costly on certain operating systems and file systems because it usually means the underlying I/O device must physically write all buffered bytes to stable storage. Near-real-time search enables you to search segments that are newly created but not yet committed. Section 11.1.3 gives some tips for further reducing the index-to-search turnaround time.

Listing 3.3 Near-real-time search

```
public class NearRealTimeTest extends TestCase {
  public void testNearRealTime() throws Exception {
    Directory dir = new RAMDirectory();
    IndexWriter writer = new IndexWriter(dir, new
     StandardAnalyzer(Version.LUCENE_30),
     IndexWriter.MaxFieldLength.UNLIMITED);
    for(int i=0;i<10;i++) {
      Document doc = new Document();
      doc.add(new Field("id", ""+i, Field.Store.NO,
     Field.Index.NOT_ANALYZED_NO_NORMS));
      doc.add(new Field("text", "aaa", Field.Store.NO,
     Field.Index.ANALYZED));
      writer.addDocument(doc);
    }
    IndexReader reader = writer.getReader();
    IndexSearcher searcher = new IndexSearcher(reader);

    Query query = new TermQuery(new Term("text", "aaa"));
    TopDocs docs = searcher.search(query, 1);
    assertEquals(10, docs.totalHits);

    writer.deleteDocuments(new Term("id", "7"));

    Document doc = new Document();
    doc.add(new Field("id",
                      "11",
                      Field.Store.NO,
                      Field.Index.NOT_ANALYZED_NO_NORMS));
    doc.add(new Field("text",
                      "bbb",
                      Field.Store.NO,
                      Field.Index.ANALYZED));
    writer.addDocument(doc);

    IndexReader newReader = reader.reopen();
    assertFalse(reader == newReader);
    reader.close();
    searcher = new IndexSearcher(newReader);

    TopDocs hits = searcher.search(query, 10);
    assertEquals(9, hits.totalHits);
```

1 Create near-real-time reader

Wrap reader in IndexSearcher

Search returns 10 hits

2 Delete 1 document

3 Add 1 document

4 Reopen reader

5 Confirm reader is new

6 Close old reader

7 Verify 9 hits now

```
    query = new TermQuery(new Term("text", "bbb"));
    hits = searcher.search(query, 1);
    assertEquals(1, hits.totalHits);

    newReader.close();
    writer.close();
  }
}
```

8 **Confirm new document matched**

1 `IndexWriter` returns a reader that's able to search all previously committed changes to the index, plus any uncommitted changes. The returned reader is always read-only.

2 3 We make changes to the index but don't commit them.

4 5 6 We ask the reader to reopen. Note that this simply calls `writer.getReader` again under the hood. Because we made changes, the `newReader` will be different from the old one so we must close the old one.

7 8 The changes made with the writer are reflected in new searches.

The important method is `IndexWriter.getReader`. This method flushes any buffered changes to the directory, and then creates a new `IndexReader` that includes the changes. If further changes are made through the `IndexWriter`, you use the reopen method in the `IndexReader` to get a new reader. If there are changes, a new reader is returned, and you should then close the old reader. The reopen method is very efficient: for any unchanged parts of the index, it shares the open files and caches with the previous reader. Only newly created files since the last open or reopen will be opened. This results in very fast, often subsecond, turnaround. Section 11.2.2 provides further examples of how to use the reopen method with a near-real-time reader.

Next we look at how Lucene scores each document that matches the search.

3.3 Understanding Lucene scoring

Every time a document matches during search, it's assigned a score that reflects how good the match is. This score computes how similar the document is to the query, with higher scores reflecting stronger similarity and thus stronger matches. We chose to discuss this complex topic early in this chapter so you'll have a general sense of the various factors that go into Lucene scoring as you continue to read. We'll start with details on Lucene's scoring formula, and then show how you can see the full explanation of how a certain document arrived at its score.

3.3.1 How Lucene scores

Without further ado, meet Lucene's similarity scoring formula, shown in figure 3.3. It's called the similarity scoring formula because its purpose is to measure the similarity between a query and each document that matches the query. The score is computed for each document (`d`) matching each term (`t`) in a query (`q`).

$$\Sigma_{t \text{ in } q} (tf(t \text{ in } d) \times idf(t)^2 \times boost(t.field \text{ in } d) \times lengthNorm(t.field \text{ in } d)) \times coord(q,d) \times queryNorm(q)$$

Figure 3.3 Lucene uses this formula to determine a document score based on a query.

NOTE If this equation or the thought of mathematical computations scares you, you may safely skip this section. Lucene's scoring is top-notch as is, and a detailed understanding of what makes it tick isn't necessary to take advantage of Lucene's capabilities.

This score is the raw score, which is a floating-point number >= 0.0. Typically, if an application presents the score to the end user, it's best to first normalize the scores by dividing all scores by the maximum score for the query. The larger the similarity score, the better the match of the document to the query. By default Lucene returns documents reverse-sorted by this score, meaning the top documents are the best matching ones. Table 3.5 describes each of the factors in the scoring formula.

Boost factors are built into the equation to let you affect a query or field's influence on score. Field boosts come in explicitly in the equation as the `boost(t.field in d)` factor, set at indexing time. The default value of field boosts, logically, is 1.0. During indexing, a document can be assigned a boost, too. A document boost factor implicitly sets the starting field boost of all fields to the specified value. Field-specific boosts are multiplied by the starting value, giving the final value of the field boost factor. It's possible to add the same named field to a document multiple times, and in such situations the field boost is computed as all the boosts specified for that field and document multiplied together. Section 2.5 discusses index-time boosting in more detail.

In addition to the explicit factors in this equation, other factors can be computed on a per-query basis as part of the `queryNorm` factor. Queries themselves can have an

Table 3.5 Factors in the scoring formula

Factor	Description
`tf(t in d)`	Term frequency factor for the term (`t`) in the document (`d`)—how many times the term `t` occurs in the document.
`idf(t)`	Inverse document frequency of the term: a measure of how "unique" the term is. Very common terms have a low `idf`; very rare terms have a high `idf`.
`boost(t.field in d)`	Field and document boost, as set during indexing (see section 2.5). You may use this to statically boost certain fields and certain documents over others.
`lengthNorm(t.field in d)`	Normalization value of a field, given the number of terms within the field. This value is computed during indexing and stored in the index norms. Shorter fields (fewer tokens) get a bigger boost from this factor.
`coord(q, d)`	Coordination factor, based on the number of query terms the document contains. The coordination factor gives an AND-like boost to documents that contain more of the search terms than other documents.
`queryNorm(q)`	Normalization value for a query, given the sum of the squared weights of each of the query terms.

impact on the document score. Boosting a `Query` instance is sensible only in a multiple-clause query; if only a single term is used for searching, changing its boost would impact all matched documents equally. In a multiple-clause Boolean query, some documents may match one clause but not another, enabling the boost factor to discriminate between matching documents. Queries also default to a 1.0 boost factor.

Most of these scoring formula factors are controlled and implemented as a subclass of the abstract `Similarity` class. `DefaultSimilarity` is the implementation used unless otherwise specified. More computations are performed under the covers of `DefaultSimilarity`; for example, the term *frequency factor* is the square root of the actual frequency. Because this is an "in action" book, it's beyond the book's scope to delve into the inner workings of these calculations. In practice, it's extremely rare to need a change in these factors. Should you need to change them, please refer to `Similarity`'s Javadocs, and be prepared with a solid understanding of these factors and the effect your changes will have.

It's important to note that a change in index-time boosts or the `Similarity` methods used during indexing, such as `lengthNorm`, require that the index be rebuilt for all factors to be in sync.

Let's say you're baffled as to why a certain document got a good score to your `Query`. Lucene offers a nice feature to help provide the answer.

3.3.2 Using explain() to understand hit scoring

Whew! The scoring formula seems daunting—and it is. We're talking about factors that rank one document higher than another based on a query; that in and of itself deserves the sophistication going on. If you want to see how all these factors play out, Lucene provides a helpful feature called `Explanation`. `IndexSearcher` has an `explain` method, which requires a `Query` and a document ID and returns an `Explanation` object.

The `Explanation` object internally contains all the gory details that factor into the score calculation. Each detail can be accessed individually if you like; but generally, dumping out the explanation in its entirety is desired. The `.toString()` method dumps a nicely formatted text representation of the `Explanation`. We wrote a simple program to dump `Explanations`, shown in listing 3.4.

Listing 3.4 The `explain()` method

```
public class Explainer {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: Explainer <index dir> <query>");
      System.exit(1);
    }

    String indexDir = args[0];
    String queryExpression = args[1];

    Directory directory = FSDirectory.open(new File(indexDir));
```

```
    QueryParser parser = new QueryParser(Version.LUCENE_30,
                                         "contents", new SimpleAnalyzer());
    Query query = parser.parse(queryExpression);

    System.out.println("Query: " + queryExpression);

    IndexSearcher searcher = new IndexSearcher(directory);
    TopDocs topDocs = searcher.search(query, 10);

    for (ScoreDoc match : topDocs.scoreDocs) {
      Explanation explanation
         = searcher.explain(query, match.doc);

      System.out.println("----------");
      Document doc = searcher.doc(match.doc);
      System.out.println(doc.get("title"));
      System.out.println(explanation.toString());
    }
    searcher.close();
    directory.close();
  }
}
```

Generate Explanation

Output Explanation

Using the query `junit` against our sample index produced the following output; notice that the most relevant title scored best:

```
Query: junit
----------
JUnit in Action, Second Edition
0.7629841 = (MATCH) fieldWeight(contents:junit in 11), product of:
  1.4142135 = tf(termFreq(contents:junit)=2)
  2.466337 = idf(docFreq=2, maxDocs=13)
  0.21875 = fieldNorm(field=contents, doc=11)

----------
Ant in Action
0.61658424 = (MATCH) fieldWeight(contents:junit in 9), product of:
  1.0 = tf(termFreq(contents:junit)=1)
  2.466337 = idf(docFreq=2, maxDocs=13)
  0.25 = fieldNorm(field=contents, doc=9)
```

1. *JUnit in Action, Second Edition* has the term *junit* twice in its contents field. The contents field in our index is a catchall field, aggregating all textual fields to allow a single field for searching.
2. *Ant in Action* has the term *junit* only once in its contents field.

There's also a `.toHtml()` method that outputs the same hierarchical structure, except as nested HTML `<ul>` elements suitable for outputting in a web browser. In fact, the `Explanation` feature is a core part of the Nutch project, allowing for transparent ranking.

`Explanations` are handy to see the inner workings of the score calculation, but they expend the same amount of effort as a query. So, be sure not to use extraneous `Explanation` generation.

By now you have a strong foundation for getting your search application off the ground: we showed the most important ways of performing searches with Lucene. Now, it's time to drill down into detail on the numerous types of queries Lucene offers.

3.4 Lucene's diverse queries

As you saw in section 3.2, querying Lucene ultimately requires a call to one of `IndexSearcher`'s `search` methods, using an instance of `Query`. `Query` subclasses can be instantiated directly, or, as we discussed in section 3.1.2, a `Query` can be constructed through the use of `QueryParser`, a front end that converts free text into each of the `Query` types we describe here. In each case we'll show you how to programmatically instantiate each `Query`, and also what `QueryParser` syntax to use to create the query.

Even if you're using `QueryParser`, combining a parsed query expression with an API-created `Query` is a common technique to augment, refine, or constrain a human-entered query. For example, you may want to restrict free-form parsed expressions to a subset of the index, like documents only within a category. Depending on your search's UI, you may have date pickers to select a date range, drop-downs for selecting a category, and a free-form search box. Each of these clauses can be stitched together using a combination of `QueryParser` and programmatically constructed queries.

Yet another way to create `Query` objects is by using the XML Query Parser package, contained in Lucene's contrib modules and described in section 9.5. This package allows you to express arbitrary queries directly as XML strings, which the package then converts into a `Query` instance. The XML could be created in any way, but one simple approach is to apply a transform to name-value pairs provided by an advanced search UI.

This section covers each of Lucene's built-in `Query` types, `TermQuery`, `TermRangeQuery`, `NumericRangeQuery`, `PrefixQuery`, `BooleanQuery`, `PhraseQuery`, `WildcardQuery`, `FuzzyQuery`, and the unusual yet aptly named `MatchAllDocsQuery`. We'll see how these queries match documents, and how to create them programmatically. There are still more queries under Lucene's contrib area, described in section 8.6. In section 3.5 we'll show how you can create each of these query types using `QueryParser` instead. We begin with `TermQuery`.

3.4.1 Searching by term: TermQuery

The most elementary way to search an index is for a specific term. A term is the smallest indexed piece, consisting of a field name and a text-value pair. Listing 3.1 provided an example of searching for a specific term. This code constructs a `Term` object instance:

```
Term t = new Term("contents", "java");
```

A `TermQuery` accepts a single `Term`:

```
Query query = new TermQuery(t);
```

All documents that have the word *java* in a contents field are returned from searches using this `TermQuery`. Note that the value is case sensitive, so be sure to match the case

of terms indexed; this may not be the exact case in the original document text, because an analyzer (see chapter 4) may have indexed things differently.

`TermQuery`s are especially useful for retrieving documents by a key. If documents were indexed using `Field.Index.NOT_ANALYZED`, the same value can be used to retrieve these documents. For example, given our book test data, the following code retrieves the single document matching the ISBN provided:

```
public void testKeyword() throws Exception {
  Directory dir = TestUtil.getBookIndexDirectory();
  IndexSearcher searcher = new IndexSearcher(dir);

  Term t = new Term("isbn", "9781935182023");
  Query query = new TermQuery(t);
  TopDocs docs = searcher.search(query, 10);
  assertEquals("JUnit in Action, Second Edition",
               1, docs.totalHits);

  searcher.close();
  dir.close();
}
```

A `Field.Index.NOT_ANALYZED` field doesn't imply that it's unique, though. It's up to you to ensure uniqueness during indexing. In our sample book data, `isbn` is unique among all documents.

3.4.2 Searching within a term range: TermRangeQuery

`Term`s are ordered lexicographically (according to `String.compareTo`) within the index, allowing for straightforward searching of textual terms within a range as provided by Lucene's `TermRangeQuery`. The beginning and ending terms may either be included or excluded. If either term is `null`, that end is open-ended. For example, a null `lowerTerm` means there is no lower bound, so all terms less than the upper term are included. Only use this query for textual ranges, such as for finding all names that begin with N through Q. `NumericRangeQuery`, covered in the next section, should be used for ranges on numeric fields.

The following code illustrates `TermRangeQuery`, searching for all books whose title begins with any letter from d to j. Our books data set has three such books. Note that the `title2` field in our book index is simply the lowercased title, indexed as a single token using `Field.NOT_ANALYZED_NO_NORMS`:

```
public void testTermRangeQuery() throws Exception {
  Directory dir = TestUtil.getBookIndexDirectory();
  IndexSearcher searcher = new IndexSearcher(dir);
  TermRangeQuery query = new TermRangeQuery("title2", "d", "j",
                                            true, true);
  TopDocs matches = searcher.search(query, 100);
  assertEquals(3, matches.totalHits);
  searcher.close();
  dir.close();
}
```

The last two Booleans to the `TermRangeQuery` state whether the start and end points are inclusive (`true`) or exclusive (`false`). We passed `true`, for an inclusive search, but had we passed `false` instead there would be no change in the results because we have no books with the exact title `d` or `j`.

Because Lucene always stores its terms in lexicographic sort order (using `String.compareTo`, which compares by UTF16 code unit), the range defined by the beginning and ending terms is always according to this lexicographic order. However, `TermRangeQuery` can also accept a custom `Collator`, which is then used for range checking. Unfortunately, this process can be extremely slow for a large index because it requires enumerating every single term in the index to check if it's within bounds. The `CollationKeyAnalyzer`, a contrib module, is one way to gain back the performance.

Next we consider the numeric equivalent of `TermRangeQuery`.

3.4.3 Searching within a numeric range: NumericRangeQuery

If you indexed your field with `NumericField`, you can efficiently search a particular range for that field using `NumericRangeQuery`. Under the hood, Lucene translates the requested range into the equivalent set of brackets in the previously indexed *trie structure*. Each bracket is a distinct term in the index whose documents are OR'd together. The number of brackets required will be relatively small, which is what gives `NumericRangeQuery` such good performance when compared to an equivalent `TermRangeQuery`.

Let's look at an example based on the pubmonth field from our book index. We indexed this field as an integer with month precision, meaning March 2010 is indexed as a `NumericField` with the integer value 201,003. We can then do an inclusive range search like this:

```
public void testInclusive() throws Exception {
  Directory dir = TestUtil.getBookIndexDirectory();
  IndexSearcher searcher = new IndexSearcher(dir);
  // pub date of TTC was September 2006
  NumericRangeQuery query = NumericRangeQuery.newIntRange("pubmonth",
                                                          200605,
                                                          200609,
                                                          true,
                                                          true);

  TopDocs matches = searcher.search(query, 10);
  assertEquals(1, matches.totalHits);
  searcher.close();
  dir.close();
}
```

Just like `TermRangeQuery`, the last two Booleans to the `newIntRange` method state whether the start and end points are inclusive (`true`) or exclusive (`false`). There's only one book published in that range, which was published in September 2006. If we change the range search to be exclusive, the book is no longer found:

```
public void testExclusive() throws Exception {
  Directory dir = TestUtil.getBookIndexDirectory();
  IndexSearcher searcher = new IndexSearcher(dir);

  // pub date of TTC was September 2006
  NumericRangeQuery query = NumericRangeQuery.newIntRange("pubmonth",
                                                          200605,
                                                          200609,
                                                          false,
                                                          false);
  TopDocs matches = searcher.search(query, 10);
  assertEquals(0, matches.totalHits);
  searcher.close();
  dir.close();
}
```

`NumericRangeQuery` also optionally accepts the same `precisionStep` parameter as `NumericField`. If you had changed this value from its default during indexing, it's crucial that you provide an acceptable value (either the same value, or a multiple of the value used during indexing) when searching. Otherwise you'll silently get incorrect results. See the Javadocs for `NumericRangeQuery` for more details.

Now let's move on to another query that matches terms by prefix.

3.4.4 Searching on a string: PrefixQuery

`PrefixQuery` matches documents containing terms beginning with a specified string. It's deceptively handy. The following code demonstrates how you can query a hierarchical structure *recursively* with a simple `PrefixQuery`. The documents contain a `category` field representing a hierarchical structure, which is perfect for matching with a `PrefixQuery`, as shown in listing 3.5.

Listing 3.5 `PrefixQuery`

```
public class PrefixQueryTest extends TestCase {
  public void testPrefix() throws Exception {
    Directory dir = TestUtil.getBookIndexDirectory();
    IndexSearcher searcher = new IndexSearcher(dir);
    Term term = new Term("category",
                         "/technology/computers/programming");
    PrefixQuery query = new PrefixQuery(term);

    TopDocs matches = searcher.search(query, 10);
    int programmingAndBelow = matches.totalHits;

    matches = searcher.search(new TermQuery(term), 10);
    int justProgramming = matches.totalHits;

    assertTrue(programmingAndBelow > justProgramming);
    searcher.close();
    dir.close();
  }
}
```

Search, including subcategories

Search, without subcategories

Our `PrefixQueryTest` demonstrates the difference between a `PrefixQuery` and a `TermQuery`. A `methodology` category exists below the `/technology/computers/programming` category. Books in this subcategory are found with a `PrefixQuery` but not with the `TermQuery` on the parent category.

Our next query, `BooleanQuery`, is an interesting one because it's able to embed and combine other queries.

3.4.5 Combining queries: BooleanQuery

The query types discussed here can be combined in complex ways using `BooleanQuery`, which is a container of Boolean *clauses*. A clause is a subquery that can be required, optional, or prohibited. These attributes allow for logical AND, OR, and NOT combinations. You add a clause to a `BooleanQuery` using this API method:

```
public void add(Query query, BooleanClause.Occur occur)
```

where `occur` can be `BooleanClause.Occur.MUST`, `BooleanClause.Occur.SHOULD`, or `BooleanClause.Occur.MUST_NOT`.

A `BooleanQuery` can be a clause within another `BooleanQuery`, allowing for arbitrary nesting. Let's look at some examples. Listing 3.6 shows an AND query to find the most recent books on one of our favorite subjects, *search*.

Listing 3.6 Using `BooleanQuery` to combine required subqueries

```
public void testAnd() throws Exception {
  TermQuery searchingBooks =
    new TermQuery(new Term("subject", "search"));                    <- 1

  Query books2010 =
    NumericRangeQuery.newIntRange("pubmonth", 201001,                    2
                                  201012,
                                  true, true);

  BooleanQuery searchingBooks2010 = new BooleanQuery();                  3
  searchingBooks2010.add(searchingBooks, BooleanClause.Occur.MUST);
  searchingBooks2010.add(books2010, BooleanClause.Occur.MUST);

  Directory dir = TestUtil.getBookIndexDirectory();
  IndexSearcher searcher = new IndexSearcher(dir);
  TopDocs matches = searcher.search(searchingBooks2010, 10);

  assertTrue(TestUtil.hitsIncludeTitle(searcher, matches,
                               "Lucene in Action, Second Edition"));
  searcher.close();
  dir.close();
}
```

1. This query finds all books containing the subject `"search"`.
2. This query finds all books published in 2010.
3. Here we combine the two queries into a single Boolean query with both clauses required (the second argument is `BooleanClause.Occur.MUST`).

In this test case, we used a new utility method, `TestUtil.hitsIncludeTitle`:

```
public static boolean hitsIncludeTitle(IndexSearcher searcher,
                                       TopDocs hits, String title)
  throws IOException {
  for (ScoreDoc match : hits.scoreDocs) {
    Document doc = searcher.doc(match.doc);
    if (title.equals(doc.get("title"))) {
      return true;
    }
  }
  System.out.println("title '" + title + "' not found");
  return false;
}
```

`BooleanQuery.add` has two overloaded method signatures. One accepts only a `BooleanClause`, and the other accepts a `Query` and a `BooleanClause.Occur` instance. A `BooleanClause` is simply a container of a `Query` and a `BooleanClause.Occur` instance, so we omit coverage of it. `BooleanClause.Occur.MUST` means exactly that: only documents matching that clause are considered. `BooleanClause.Occur.SHOULD` means the term is optional. `BooleanClause.Occur.MUST_NOT` means any documents matching this clause are excluded from the results. Use `BooleanClause.Occur.SHOULD` to perform an OR query, as shown in listing 3.7.

Listing 3.7 Using `BooleanQuery` to combine optional subqueries.

```
public void testOr() throws Exception {
  TermQuery methodologyBooks = new TermQuery(
             new Term("category",
               "/technology/computers/programming/methodology"));

  TermQuery easternPhilosophyBooks = new TermQuery(
      new Term("category",
          "/philosophy/eastern"));

  BooleanQuery enlightenmentBooks = new BooleanQuery();
  enlightenmentBooks.add(methodologyBooks,
                         BooleanClause.Occur.SHOULD);
  enlightenmentBooks.add(easternPhilosophyBooks
                         BooleanClause.Occur.SHOULD);

  Directory dir = TestUtil.getBookIndexDirectory();
  IndexSearcher searcher = new IndexSearcher(dir);
  TopDocs matches = searcher.search(enlightenmentBooks, 10);
  System.out.println("or = " + enlightenmentBooks);

  assertTrue(TestUtil.hitsIncludeTitle(searcher, matches,
                                "Extreme Programming Explained"));
  assertTrue(TestUtil.hitsIncludeTitle(searcher, matches,
                                "Tao Te Ching \u9053\u5FB7\u7D93"));
  searcher.close();
  dir.close();
}
```

Match 1st category

Match 2nd category

Combine both categories

It's fine to mix and match different clauses within a single `BooleanQuery`; simply specify the `BooleanClause.Occur` for each. You can create very powerful queries by doing so. For example, you could construct a query that must match "java" and "programming", must not match "ant", and should match "computers" as well as "flowers." Then, you'll know that every returned document will contain both "java" and "programming," won't contain "ant", and will contain either "computers" or "flowers", or both.

`BooleanQuery`s are restricted to a maximum number of clauses; 1,024 is the default. This limitation is in place to prevent queries from accidentally adversely affecting performance. A `TooManyClauses` exception is thrown if the maximum is exceeded. This had been necessary in past Lucene releases, because certain queries would rewrite themselves under the hood to the equivalent `BooleanQuery`. But as of 2.9, these queries are now executed in a more efficient manner. Should you ever have the unusual need of increasing the number of clauses allowed, there's a `setMaxClauseCount(int)` method on `BooleanQuery`, but be aware of the performance cost of executing such queries.

The next query, `PhraseQuery`, differs from the queries we've covered so far in that it pays attention to the positional details of multiple-term occurrences.

3.4.6 Searching by phrase: PhraseQuery

An index by default contains positional information of terms, as long as you didn't create pure Boolean fields by indexing with the `omitTermFreqAndPositions` option (described in section 2.4.1). `PhraseQuery` uses this information to locate documents where terms are within a certain distance of one another. For example, suppose a field contained the phrase *the quick brown fox jumped over the lazy dog*. Without knowing the exact phrase, you can still find this document by searching for documents with fields having *quick* and *fox* near each other. Sure, a plain `TermQuery` would do the trick to locate this document knowing either of those words, but in this case we only want documents that have phrases where the words are either exactly side by side (*quick fox*) or have one word in between (*quick* [irrelevant] *fox*).

The maximum allowable positional distance between terms to be considered a match is called *slop*. *Distance* is the number of positional moves of terms used to reconstruct the phrase in order. Let's take the phrase just mentioned and see how the slop factor plays out. First we need a little test infrastructure, which includes a `setUp()` method to index a single document, a `tearDown()` method to close the directory and searcher, and a custom `matched (String[], int)` method to construct, execute, and assert a phrase query matched the test document, shown in listing 3.8.

Listing 3.8 `PhraseQuery`

```
public class PhraseQueryTest extends TestCase {
  private Directory dir;
  private IndexSearcher searcher;
```

```
  protected void setUp() throws IOException {
    dir = new RAMDirectory();
    IndexWriter writer = new IndexWriter(dir,
                                new WhitespaceAnalyzer(),
                                IndexWriter.MaxFieldLength.UNLIMITED);
    Document doc = new Document();
    doc.add(new Field("field",
            "the quick brown fox jumped over the lazy dog",
            Field.Store.YES,
            Field.Index.ANALYZED));
    writer.addDocument(doc);
    writer.close();

    searcher = new IndexSearcher(dir);
  }

  protected void tearDown() throws IOException {
    searcher.close();
    dir.close();
  }

  private boolean matched(String[] phrase, int slop)
      throws IOException {
    PhraseQuery query = new PhraseQuery();
    query.setSlop(slop);

    for (String word : phrase) {
      query.add(new Term("field", word));
    }

    TopDocs matches = searcher.search(query, 10);
    return matches.totalHits > 0;
  }
}
```

Add a single test document

Create initial PhraseQuery

Add sequential phrase terms

Because we want to demonstrate several phrase query examples, we wrote the `matched` method to simplify the code. Phrase queries are created by adding terms in the desired order. By default, a `PhraseQuery` has its slop factor set to zero, specifying an exact phrase match. With our `setUp()` and helper `matched` method, our test case succinctly illustrates how `PhraseQuery` behaves. Failing and passing slop factors show the boundaries:

```
public void testSlopComparison() throws Exception {
  String[] phrase = new String[] {"quick", "fox"};

  assertFalse("exact phrase not found", matched(phrase, 0));

  assertTrue("close enough", matched(phrase, 1));
}
```

Terms added to a phrase query don't have to be in the same order found in the field, although order does impact slop-factor considerations. For example, had the terms been reversed in the query (*fox* and then *quick*), the number of moves needed to match the document would be three, not one. To visualize this, consider how many moves it would take to physically move the word *fox* two slots past *quick*; you'll see that

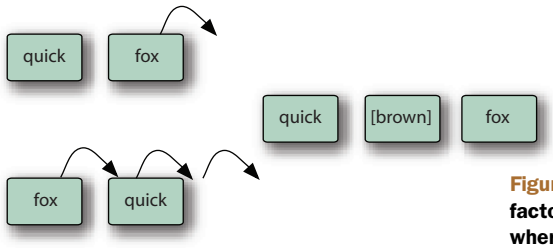

Figure 3.4 Illustrating the `PhraseQuery` slop factor: "quick fox" requires a slop of 1 to match, whereas "fox quick" requires a slop of 3 to match.

it takes one move to move *fox* into the same position as *quick* and then two more to move *fox* beyond *quick* sufficiently to match "quick brown fox."

Figure 3.4 shows how the slop positions work in both of these phrase query scenarios, and this test case shows the match in action:

```
public void testReverse() throws Exception {
  String[] phrase = new String[] {"fox", "quick"};

  assertFalse("hop flop", matched(phrase, 2));
  assertTrue("hop hop slop", matched(phrase, 3));
}
```

Let's now examine how multiple-term phrase queries work.

MULTIPLE-TERM PHRASES

`PhraseQuery` supports multiple-term phrases. Regardless of how many terms are used for a phrase, the slop factor is the maximum total number of moves allowed to put the terms in order. Let's look at an example of a multiple-term phrase query:

```
public void testMultiple() throws Exception {
  assertFalse("not close enough",
      matched(new String[] {"quick", "jumped", "lazy"}, 3));

  assertTrue("just enough",
      matched(new String[] {"quick", "jumped", "lazy"}, 4));

  assertFalse("almost but not quite",
      matched(new String[] {"lazy", "jumped", "quick"}, 7));

  assertTrue("bingo",
      matched(new String[] {"lazy", "jumped", "quick"}, 8));
}
```

Now that you've seen how phrase queries match, let's turn our attention to how phrase queries affect the score.

PHRASE QUERY SCORING

Phrase queries are scored based on the edit distance needed to match the phrase. More exact matches count for more weight than sloppier ones. The phrase query factor is shown in figure 3.5. The inverse relationship with distance ensures that greater distances have lower scores.

Figure 3.5 Sloppy phrase scoring formula

$$\frac{1}{distance + 1}$$

NOTE Terms surrounded by double quotes in `QueryParser`-parsed expressions are translated into a `PhraseQuery`. The slop factor defaults to 0, but you can adjust the slop factor by adding a tilde (~) followed by an integer. For example, the expression `"quick fox"~3` is a `PhraseQuery` with the terms quick and fox and a slop factor of 3. There are additional details about `PhraseQuery` and the slop factor in section 3.4.6. Phrases are analyzed by the analyzer passed to the `QueryParser`, adding another layer of complexity, as discussed in section 4.1.2.

Our next query, `WildcardQuery`, matches terms using wildcard characters.

3.4.7 Searching by wildcard: WildcardQuery

Wildcard queries let you query for terms with missing pieces but still find matches. Two standard wildcard characters are used: `*` for zero or more characters, and `?` for zero or one character. Listing 3.9 demonstrates `WildcardQuery` in action. You can think of `WildcardQuery` as a more general `PrefixQuery` because the wildcard doesn't have to be at the end.

Listing 3.9 `WildcardQuery`

```
private void indexSingleFieldDocs(Field[] fields) throws Exception {
  IndexWriter writer = new IndexWriter(directory,
      new WhitespaceAnalyzer(), IndexWriter.MaxFieldLength.UNLIMITED);
  for (Field f : fields) {
    Document doc = new Document();
    doc.add(f);
    writer.addDocument(doc);
  }
  writer.optimize();
  writer.close();
}

public void testWildcard() throws Exception {
  indexSingleFieldDocs(new Field[]
    { new Field("contents", "wild", Field.Store.YES,
                Field.Index.ANALYZED),
      new Field("contents", "child", Field.Store.YES,
                Field.Index.ANALYZED),
      new Field("contents", "mild", Field.Store.YES,
                Field.Index.ANALYZED),
      new Field("contents", "mildew", Field.Store.YES,
                Field.Index.ANALYZED) });

  IndexSearcher searcher = new IndexSearcher(directory);
  Query query = new WildcardQuery(new Term("contents", "?ild*"));
  TopDocs matches = searcher.search(query, 10);
  assertEquals("child no match", 3, matches.totalHits);

  assertEquals("score the same", matches.scoreDocs[0].score,
                                 matches.scoreDocs[1].score, 0.0);
  assertEquals("score the same", matches.scoreDocs[1].score,
                                 matches.scoreDocs[2].score, 0.0);
  searcher.close();
}
```

Create WildcardQuery

Note how the wildcard pattern is created as a `Term` (the pattern to match) even though it isn't explicitly used as an exact term under the covers. Internally, it's used as a pattern to match terms in the index. A `Term` instance is a convenient placeholder to represent a field name and an arbitrary string.

WARNING Performance degradations can occur when you use `WildcardQuery`. A larger prefix (characters before the first wildcard character) decreases the number of terms enumerated to find matches. Beginning a pattern with a wildcard query forces the term enumeration to search *all* terms in the index for matches.

Oddly, the closeness of a wildcard match has no effect on scoring. The last two assertions in listing 3.9, where *wild* and *mild* are closer matches to the pattern than *mildew*, demonstrate this.

Our next query is `FuzzyQuery`.

3.4.8 Searching for similar terms: FuzzyQuery

Lucene's `FuzzyQuery` matches terms *similar* to a specified term. The *Levenshtein distance* algorithm determines how similar terms in the index are to a specified target term. (See http://en.wikipedia.org/wiki/Levenshtein_Distance for more information about Levenshtein distance.) *Edit distance* is another term for Levenshtein distance; it's a measure of similarity between two strings, where distance is measured as the number of character deletions, insertions, or substitutions required to transform one string to the other string. For example, the edit distance between *three* and *tree* is 1, because only one character deletion is needed.

Levenshtein distance isn't the same as the distance calculation used in `PhraseQuery` and `PrefixQuery`. The phrase query distance is the number of term moves to match, whereas Levenshtein distance is an intraterm computation of character moves. The `FuzzyQuery` test demonstrates its usage and behavior:

```
public void testFuzzy() throws Exception {
  indexSingleFieldDocs(new Field[] { new Field("contents",
                                               "fuzzy",
                                               Field.Store.YES,
                                               Field.Index.ANALYZED),
                                     new Field("contents",
                                               "wuzzy",
                                               Field.Store.YES,
                                               Field.Index.ANALYZED)
                                   });

  IndexSearcher searcher = new IndexSearcher(directory);
  Query query = new FuzzyQuery(new Term("contents", "wuzza"));
  TopDocs matches = searcher.search(query, 10);
  assertEquals("both close enough", 2, matches.totalHits);

  assertTrue("wuzzy closer than fuzzy",
             matches.scoreDocs[0].score != matches.scoreDocs[1].score);

  Document doc = searcher.doc(matches.scoreDocs[0].doc);
```

```
    assertEquals("wuzza bear", "wuzzy", doc.get("contents"));
    searcher.close();
  }
```

This test illustrates a couple of key points. Both documents match; the term searched for (*wuzza*) wasn't indexed but was close enough to match. `FuzzyQuery` uses a *threshold* rather than a pure edit distance. The threshold is a factor of the edit distance divided by the string length. Edit distance affects scoring; terms with less edit distance are scored higher. Other term statistics, such as the inverse document frequency, are also factored in, as described in section 3.3. Distance is computed using the formula shown in figure 3.6.

Figure 3.6 `FuzzyQuery` distance formula

$$1 - \frac{distance}{min(textlen, targetlen)}$$

WARNING `FuzzyQuery` enumerates all terms in an index to find terms within the allowable threshold. Use this type of query sparingly or at least with the knowledge of how it works and the effect it may have on performance.

3.4.9 Matching all documents: MatchAllDocsQuery

`MatchAllDocsQuery`, as the name implies, simply matches every document in your index. By default, it assigns a constant score, the boost of the query (default: 1.0), to all documents that match. If you use this as your top query, it's best to sort by a field other than the default relevance sort.

It's also possible to have `MatchAllDocsQuery` assign as document scores the boosting recorded in the index, for a specified field, like so:

```
Query query = new MatchAllDocsQuery(field);
```

If you do this, documents are scored according to how the specified field was boosted (as described in section 2.5).

We're done reviewing Lucene's basic core `Query` classes. Chapter 5 covers more advanced `Query` classes. Now we'll move on to using `QueryParser` to construct queries from a user's textual query.

3.5 Parsing query expressions: QueryParser

Although API-created queries can be powerful, it isn't reasonable that all queries should be explicitly written in Java code. Using a human-readable textual query representation, Lucene's `QueryParser` constructs one of the previously mentioned `Query` subclasses. Because the `QueryParser` already recognizes the standard search syntax that has become popular thanks to web search engines like Google, using `QueryParser` is also an immediate and simple way for your application to meet that user expectation. `QueryParser` is also easily customized, as we'll see in section 6.3.

The constructed `Query` instance could be a complex entity, consisting of nested `BooleanQuery`s and a combination of almost all the `Query` types mentioned, but an expression entered by the user could be as readable as this:

```
+pubdate:[20100101 TO 20101231] Java AND (Lucene OR Apache)
```

This query searches for all books about Java that also include *Lucene* or *Apache* in their contents and were published in 2010.

NOTE Whenever special characters are used in a query expression, you need to provide an escaping mechanism so that the special characters can be used in a normal fashion. `QueryParser` uses a backslash (\) to escape special characters within terms. The characters that require escaping are as follows:

```
\ + - ! ( ) : ^ ] { } ~ * ?
```

We've already seen a brief introduction to `QueryParser` in section 3.1.2 at the start of the chapter. In this section we'll first delve into the specific syntax for each of Lucene's core `Query` classes that `QueryParser` supports. We'll also describe some of the settings that control the parsing of certain queries. We'll wrap up with further syntax that `QueryParser` accepts for controlling grouping, boosting, and field searching of each query clause. This discussion assumes knowledge of the `Query` types discussed in section 3.4. Note that some of these subsections here are rather short; this is a reflection of just how powerful `QueryParser` is under the hood—it's able to take a simple-to-describe search syntax and easily build rich queries.

We begin with a handy way to glimpse what `QueryParser` does to expressions.

3.5.1 Query.toString

Seemingly strange things can happen to a query expression as it's parsed with `QueryParser`. How can you tell what really happened to your expression? Was it translated properly into what you intended? One way to peek at a resultant `Query` instance is to use its `toString()` method.

All concrete core `Query` classes we've discussed in this chapter have a special `toString()` implementation. The standard `Object.toString()` method is overridden and delegates to a `toString(String field)` method, where `field` is the name of the default field. Calling the no-arg `toString()` method uses an empty default field name, causing the output to explicitly use field selector notation for all terms. Here's an example of using the `toString()` method:

```
public void testToString() throws Exception {
  BooleanQuery query = new BooleanQuery();
  query.add(new FuzzyQuery(new Term("field", "kountry")),
            BooleanClause.Occur.MUST);
  query.add(new TermQuery(new Term("title", "western")),
            BooleanClause.Occur.SHOULD);
  assertEquals("both kinds", "+kountry~0.5 title:western",
               query.toString("field"));
}
```

The `toString()` methods (particularly the `String`-arg one) are handy for visual debugging of complex API queries as well as getting a handle on how `QueryParser` interprets query expressions. Don't rely on the ability to go back and forth accurately between a `Query.toString()` representation and a `QueryParser`-parsed expression,

though. It's generally accurate, but an analyzer is involved and may confuse things; this issue is discussed further in section 4.1.2. Let's begin, again, with the simplest `Query` type, `TermQuery`.

3.5.2 TermQuery

As you might expect, a single word is by default parsed into a `TermQuery` by `QueryParser`, as long as it's not part of a broader expression recognized by the other query types. For example:

```
public void testTermQuery() throws Exception {
  QueryParser parser = new QueryParser(Version.LUCENE_30,
                                       "subject", analyzer);
  Query query = parser.parse("computers");
  System.out.println("term: " + query);
}
```

produces this output:

```
term: subject:computers
```

Note how `QueryParser` built the term query by appending the default field we'd provided when instantiating it, `subject`, to the analyzed term, `computers`. Section 3.5.11 shows how you can specify a field other than the default one. Note that the text for the word is passed through the analysis process, described in the next chapter, before constructing the `TermQuery`. In our `QueryParserTest` we're using an analyzer that simply splits words at whitespace. Had we used a more interesting analyzer, it could have altered the term—for example, by stripping the plural suffix, and perhaps reducing the word to its root form before passing it to the `TermQuery`. It's vital that this analysis done by `QueryParser` match the analysis performed during indexing. Section 4.1.2 delves into this tricky topic.

Let's see how `QueryParser` constructs range searches.

3.5.3 Term range searches

Text or date range queries use bracketed syntax, with `TO` between the beginning term and ending term. Note that `TO` must be all caps. The type of bracket determines whether the range is inclusive (square brackets) or exclusive (curly braces). Note that, unlike with the programmatic construction of `NumericRangeQuery` or `TermRangeQuery`, you can't mix inclusive and exclusive: both the start and end term must be either inclusive or exclusive.

Our `testRangeQuery()` method, in listing 3.10, demonstrates both inclusive and exclusive range queries.

Listing 3.10 Creating a `TermRangeQuery` using `QueryParser`

```
public void testTermRangeQuery() throws Exception {
  Query query = new QueryParser(Version.LUCENE_30,
                                "subject", analyzer)
                        .parse("title2:[Q TO V]");
```

Verify inclusive range

```
  assertTrue(query instanceof TermRangeQuery);

  TopDocs matches = searcher.search(query, 10);
  assertTrue(TestUtil.hitsIncludeTitle(searcher, matches,
              "Tapestry in Action"));

  query = new QueryParser(Version.LUCENE_30,
                "subject",
                analyzer)
         .parse("title2:{Q TO \"Tapestry in Action\" }");
  matches = searcher.search(query, 10);
  assertFalse(TestUtil.hitsIncludeTitle(searcher, matches,
              "Tapestry in Action"));
}
```

Verify exclusive range

Exclude Tapestry in Action

NOTE Nondate range queries lowercase the beginning and ending terms as the user entered them, unless `QueryParser.setLowercaseExpandedTerms(false)` has been called. The text isn't analyzed. If the start or end terms contain whitespace, they must be surrounded with double quotes, or parsing fails.

Let's look next at numeric and date ranges.

3.5.4 Numeric and date range searches

`QueryParser` won't create a `NumericRangeQuery` for you. This is because Lucene currently doesn't keep track of which of your fields were indexed with `NumericField`, though it's possible this limitation has been corrected by the time you read this. `QueryParser` does include certain built-in logic for parsing dates when they appear as part of a range query, but the logic doesn't work when you've indexed your dates using `NumericField`. Fortunately, subclassing `QueryParser` to correctly handle numeric fields is straightforward, as described in sections 6.3.3 and 6.3.4.

Next we'll see how `QueryParser` creates prefix and wildcard queries.

3.5.5 Prefix and wildcard queries

If a term contains an asterisk or a question mark, it's considered a `WildcardQuery`. When the term contains only a trailing asterisk, `QueryParser` optimizes it to a `PrefixQuery` instead. Both prefix and wildcard queries are lowercased by default, but this behavior can be controlled:

```
public void testLowercasing() throws Exception {
  Query q = new QueryParser(Version.LUCENE_30,
          "field", analyzer).parse("PrefixQuery*");
  assertEquals("lowercased",
      "prefixquery*", q.toString("field"));

  QueryParser qp = new QueryParser(Version.LUCENE_30,
                                   "field", analyzer);
  qp.setLowercaseExpandedTerms(false);
  q = qp.parse("PrefixQuery*");
  assertEquals("not lowercased",
      "PrefixQuery*", q.toString("field"));
}
```

Wildcards at the beginning of a term are prohibited using `QueryParser` by default, which you can override at the expense of performance by calling the `setAllowLeadingWildcard` method. Section 3.4.7 discusses more about the performance issue, and section 6.3.2 provides a way to prohibit `WildcardQuerys` entirely from parsed expressions.

Let's look next at `QueryParser`'s ability to create `BooleanQuerys`.

3.5.6 Boolean operators

Constructing Boolean queries textually via `QueryParser` is done using the operators AND, OR, and NOT. Note that these operators must be typed as all caps. Terms listed without an operator specified use an implicit operator, which by default is OR. The query `abc xyz` will be interpreted as either `abc OR xyz` or `abc AND xyz`, based on the implicit operator setting. To switch parsing to use AND:

```
QueryParser parser = new QueryParser(Version.LUCENE_30,
                                     "contents", analyzer);
parser.setDefaultOperator(QueryParser.AND_OPERATOR);
```

Placing a NOT in front of a term excludes documents matching the following term. Negating a term must be combined with at least one non-negated term to return documents; in other words, it isn't possible to use a query like `NOT term` to find all documents that don't contain a term. Each of the uppercase word operators has shortcut syntax; table 3.6 illustrates various syntax equivalents.

Table 3.6 Boolean query operator shortcuts

Verbose syntax	Shortcut syntax
a AND b	+a +b
a OR b	a b
a AND NOT b	+a –b

We'll see how to construct a `PhraseQuery` next.

3.5.7 Phrase queries

Terms enclosed in double quotes create a `PhraseQuery`. The text between the quotes is analyzed; thus the resultant `PhraseQuery` may not be exactly the phrase originally specified. This process has been the subject of some confusion. For example, the query `"This is Some Phrase*"`, when analyzed by the `StandardAnalyzer`, parses to a `PhraseQuery` using the phrase "some phrase." The `StandardAnalyzer` removes the words *this* and *is* because they match the default stop word list and leaves positional holes recording that the words were removed (more in section 4.3.2 on `StandardAnalyzer`). A common question is why the asterisk isn't interpreted as a wildcard query. Keep in mind that surrounding text with double quotes causes the surrounded text to be analyzed and converted into a `PhraseQuery`. Single-term phrases are optimized to a `TermQuery`. The following code demonstrates both the effect of analysis on a phrase query expression and the `TermQuery` optimization:

```
public void testPhraseQuery() throws Exception {
  Query q = new QueryParser(Version.LUCENE_30,
                            "field",
```

```
                                new StandardAnalyzer(
                                  Version.LUCENE_30))
                        .parse("\"This is Some Phrase*\"");

    assertEquals("analyzed",
        "\"? ? some phrase\"", q.toString("field"));

    q = new QueryParser(Version.LUCENE_30,
                            "field", analyzer)
                        .parse("\"term\"");
    assertTrue("reduced to TermQuery", q instanceof TermQuery);
  }
```

You can see that the query represents the positional holes left by the removed stop words, using a `?` character. The default slop factor is 0, but you can change this default by calling `QueryParser.setPhraseSlop`. The slop factor can also be overridden for each phrase by using a trailing tilde (~) and the desired integer slop value:

```
public void testSlop() throws Exception {
  Query q = new QueryParser(Version.LUCENE_30,
       "field", analyzer)
          .parse("\"exact phrase\"");
  assertEquals("zero slop",
      "\"exact phrase\"", q.toString("field"));

  QueryParser qp = new QueryParser(Version.LUCENE_30,
                                   "field", analyzer);
  qp.setPhraseSlop(5);
  q = qp.parse("\"sloppy phrase\"");
  assertEquals("sloppy, implicitly",
      "\"sloppy phrase\"~5", q.toString("field"));
}
```

A sloppy `PhraseQuery`, as noted, doesn't require that the terms match in the same order. But a `SpanNearQuery` (discussed in section 5.5.3) has the ability to guarantee an in-order match. In section 6.3.5, we extend `QueryParser` and substitute a `SpanNearQuery` when phrase queries are parsed, allowing for sloppy in-order phrase matches. The final queries we discuss are `FuzzyQuery` and `MatchAllDocsQuery`.

3.5.8 Fuzzy queries

A trailing tilde (~) creates a fuzzy query on the preceding term. Note that the tilde is also used to specify sloppy phrase queries, but the context is different. Double quotes denote a phrase query and aren't used for fuzzy queries. You can optionally specify a trailing floating point number to specify the minimum required similarity. Here's an example:

```
public void testFuzzyQuery() throws Exception {
  QueryParser parser = new QueryParser(Version.LUCENE_30,
                                       "subject", analyzer);
  Query query = parser.parse("kountry~");
  System.out.println("fuzzy: " + query);

  query = parser.parse("kountry~0.7");
  System.out.println("fuzzy 2: " + query);
}
```

This produces the following output:

```
fuzzy: subject:kountry~0.5
fuzzy 2: subject:kountry~0.7
```

The same performance caveats that apply to `WildcardQuery` also apply to fuzzy queries and can be disabled by customizing, as discussed in section 6.3.2.

3.5.9 MatchAllDocsQuery

`QueryParser` produces the `MatchAllDocsQuery` when you enter `*:*`.

This wraps up our coverage showing how `QueryParser` produces each of Lucene's core query types. But that's not the end of `QueryParser`: it also supports some very useful syntax to group clauses of a `Query`, boost clauses, and restrict clauses to specific fields. Let's begin with grouping.

3.5.10 Grouping

Lucene's `BooleanQuery` lets you construct complex nested clauses; likewise, `QueryParser` enables this same capability with textual query expressions via grouping. Let's find all the methodology books that are about either agile or extreme methodologies. We use parentheses to form subqueries, enabling advanced construction of `BooleanQuery`s:

```
public void testGrouping() throws Exception {
  Query query = new QueryParser(
      Version.LUCENE_30,
      "subject",
      analyzer).parse("(agile OR extreme) AND methodology");
  TopDocs matches = searcher.search(query, 10);

  assertTrue(TestUtil.hitsIncludeTitle(searcher, matches,
                                       "Extreme Programming Explained"));
  assertTrue(TestUtil.hitsIncludeTitle(searcher,
                                       matches,
                                       "The Pragmatic Programmer"));
}
```

You can arbitrarily nest queries within other queries using this code. It's possible to build up some truly amazing queries by doing so. Figure 3.7 shows an example of the recursive structure produced by such rich queries.

Next, we discuss how a specific field can be selected. Notice that field selection can also leverage parentheses.

3.5.11 Field selection

`QueryParser` needs to know the field name to use when constructing queries, but it would generally be unfriendly to require users to identify the field to search (the end user may not need or want to know the field names). As you've seen, the default field name is provided when you create the `QueryParser` instance. Parsed queries aren't restricted, however, to searching only the default field. Using field selector notation, you can specify terms in nondefault fields. For example, if you set query parser to

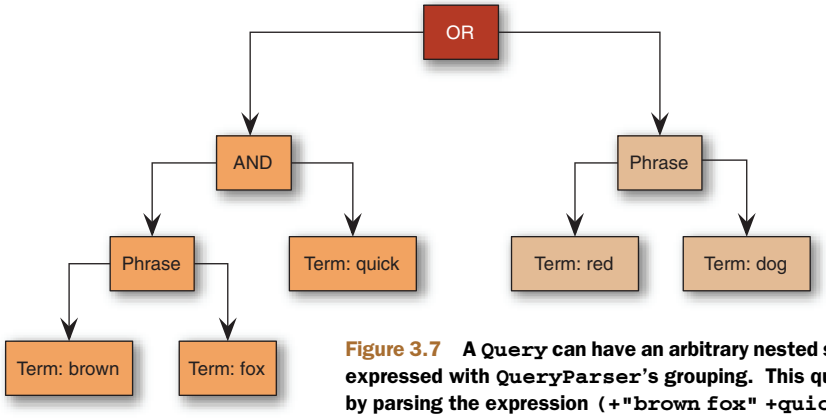

Figure 3.7 A `Query` can have an arbitrary nested structure, easily expressed with `QueryParser`'s grouping. This query is achieved by parsing the expression `(+"brown fox" +quick) "red dog"`.

search a catchall field by default, your users can still restrict the search to the `title` field using `title:lucene`. You can group field selection over multiple clauses. Using `field:(a b c)` will OR together (by default) the three term queries, where each term must appear in the specified field. Let's see how to boost a query clause next.

3.5.12 Setting the boost for a subquery

A caret (`^`) followed by a floating-point number sets the boost factor for the preceding query. For example, the query expression `junit^2.0 testing` sets the `junit TermQuery` to a boost of 2.0 and leaves the `testing TermQuery` at the default boost of 1.0. You can apply a boost to any type of query, including parenthetical groups.

3.5.13 To QueryParse or not to QueryParse?

`QueryParser` is a quick and effortless way to give users powerful query construction, but it isn't right for all scenarios. `QueryParser` can't create every type of query that can be constructed using the API. In chapter 5, we detail a handful of API-only queries that have no `QueryParser` expression capability. You must keep in mind all the possibilities available when exposing free-form query parsing to an end user; some queries have the potential for performance bottlenecks, and the syntax used by the built-in `QueryParser` may not be suitable for your needs. You can exert some limited control by subclassing `QueryParser` (see section 6.3.1).

Should you require different expression syntax or capabilities beyond what `QueryParser` offers, technologies such as ANTLR (http://www.antlr.org) and JFlex (http://jflex.de/) are great options. We don't discuss the creation of a custom query parser, though we do explore extending `QueryParser` in chapter 6. The source code for Lucene's `QueryParser` is freely available for you to borrow from. The contrib area also contains an entirely new `QueryParser` framework, covered in section 9.9, that's designed for more modular extensibility. Another contrib option is the XML query parser, described in section 9.5, that's able to build arbitrary queries described as an XML string.

You can often obtain a happy medium by combining a `QueryParser`-parsed query with API-created queries as clauses in a `BooleanQuery`. For example, if users need to constrain searches to a particular category or narrow them to a date range, you can have the UI separate those selections into a category chooser or separate date-range fields.

3.6 Summary

Lucene provides highly relevant search results to queries—quickly. Most applications need only a few Lucene classes and methods to enable searching. The most fundamental things for you to take away from this chapter are an understanding of the basic query types and how to access search results.

Although it can be a bit daunting, Lucene's scoring formula (coupled with the index format discussed in appendix B and the efficient algorithms) provides the magic of returning the most relevant documents first. Lucene's `QueryParser` parses human-readable query expressions, giving rich full-text search power to end users. `QueryParser` immediately satisfies most application requirements—but it doesn't come without caveats, so be sure you understand the rough edges. Much of the confusion regarding `QueryParser` stems from unexpected analysis interactions; chapter 4 goes into great detail about analysis, including more on the `QueryParser` issues.

And yes, there's more to searching than we've covered in this chapter, but understanding the groundwork is crucial. After analysis in chapter 4, chapter 5 delves into Lucene's more elaborate features, such as constraining (or filtering) the search space of queries and sorting search results by field values; chapter 6 explores the numerous ways you can extend Lucene's searching capabilities for custom sorting and query parsing.

JAVA

Lucene in Action Second Edition

Michael McCandless • Erik Hatcher • Otis Gospodnetić
Foreword by Doug Cutting

When Lucene first appeared, this superfast search engine was nothing short of amazing. Today, Lucene still delivers. Its high-performance, easy-to-use API, features like numeric fields, payloads, near-real-time search, and huge increases in indexing and searching speed make it the leading search tool.

And with clear writing, reusable examples, and unmatched advice, **Lucene in Action, Second Edition** is still the definitive guide to effectively integrating search into your applications. This totally revised book shows you how to index your documents, including formats such as MS Word, PDF, HTML, and XML. It introduces you to searching, sorting, and filtering, and covers the numerous improvements to Lucene since the first edition. Source code is for Lucene 3.0.1.

NEW in the Second Edition

- Performing hot backups
- Using numeric fields
- Tuning for indexing or searching speed
- Boosting matches with payloads
- Creating reusable analyzers
- Adding concurrency with threads
- Four new case studies
- Much more!

Michael McCandless is a Lucene PMC member and committer with more than a decade of experience building search engines. **Erik Hatcher** and **Otis Gospodnetić** are the authors of the first edition of *Lucene in Action* and long-time contributors to Lucene, Solr, Mahout, and other Lucene-based projects.

For online access to the authors and a free ebook for owners of this book, go to manning.com/LuceneinActionSecondEdition

"...brings you up to speed."
—From the Foreword by Doug Cutting, Founder of Lucene, Nutch, and Hadoop.

"This new edition has it all."
—Chad Davis, Blackdog Software
Author of *Struts 2 in Action*

"Very readable, full of expert tips."
—Rick Wagner, Acxiom Corp.

"Elegant, and easy to read—just like Lucene itself."
—Shai Erera
IBM Haifa Research Labs

"For a Lucene developer, it's required reading."
—Stuart Caborn, Thoughtworks

ISBN 13: 978-1-933988-17-7
ISBN 10: 1-933988-17-7
54999
9 781933 988177

MANNING $49.99 / Can $62.99 [INCLUDING eBOOK]