A guide to the Java search engine

SAMPLE CHAPTER

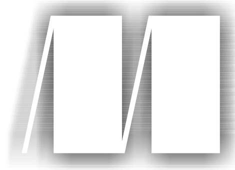
Lucene IN ACTION

Otis Gospodnetić

Erik Hatcher

FOREWORD BY Doug Cutting

MANNING



Lucene in Action
by Erik Hatcher
and Otis Gospodnetić
Sample Chapter 3

Copyright 2004 Manning Publications

brief contents

PART 1 CORE LUCENE 1

1 ▪ Meet Lucene 3
2 ▪ Indexing 28
3 ▪ Adding search to your application 68
4 ▪ Analysis 102
5 ▪ Advanced search techniques 149
6 ▪ Extending search 194

PART 2 APPLIED LUCENE 221

7 ▪ Parsing common document formats 223
8 ▪ Tools and extensions 267
9 ▪ Lucene ports 312
10 ▪ Case studies 325

3 Adding search to your application

This chapter covers

- Querying a Lucene index
- Working with search results
- Understanding Lucene scoring
- Parsing human-entered query expressions

If we can't find it, it effectively doesn't exist. Even if we have indexed documents, our effort is wasted unless it pays off by providing a reliable and fast way to find those documents. 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.[1]

Do you have a repository of hundreds, thousands, or millions of documents that needs similar search capability?

Providing search capability using Lucene's API is straightforward and easy, but lurking under the covers is a sophisticated mechanism that can meet your search requirements such as returning the most relevant documents first and retrieving the results incredibly fast. This chapter covers common ways to search using the Lucene API. The majority of applications using Lucene search can provide a search feature that performs nicely using the techniques shown in this chapter. Chapter 5 delves into more advanced search capabilities, and chapter 6 elaborates on ways to extend Lucene's classes for even greater searching power.

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 then explore the various types of search queries Lucene handles natively.

3.1 Implementing a simple search feature

Suppose you're tasked with adding search to an application. You've tackled getting the data indexed, 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 your application logic and user interface around those results.

---

[1] We cover all the pieces to make this happen with Lucene, including a specials filter in chapter 6, synonym injection in chapter 4, and the Boolean logic in this chapter.

In this chapter, we'll limit our discussion to the primary classes in Lucene's API that you'll typically use for search integration (shown in table 3.1). Sure, there is more to the story, and we go beyond the basics in chapters 5 and 6. In this chapter, we'll cover the details you'll need for the majority of your applications.

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.
`Hits`	Provides access to search results. `Hits` is returned from `IndexSearcher`'s `search` method.

When you're *querying* a Lucene index, an ordered collection of *hits* is returned. The hits collection is ordered by *score* by default.[2] Lucene computes a score (a numeric value of relevance) for each document, given a query. The hits themselves aren't the actual matching documents, but rather are references to the documents matched. In most applications that display search results, users access only the first few documents, so it isn't necessary to retrieve the actual documents for all results; you need to retrieve only the documents that will be presented to the user. For large indexes, it wouldn't even be possible to collect all matching documents into available physical computer memory.

In the next section, we put `IndexSearcher`, `Query`, and `Hits` to work with some basic term searches.

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 value that is paired with its containing field name—in this case, `subject`.

[2] The word *collection* in this sense does *not* refer to `java.util.Collection`.

NOTE Important: 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. Chapter 4 discusses the analysis process in detail.

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

Listing 3.1 `SearchingTest`: Demonstrates the simplicity of searching using a `TermQuery`

```
public class SearchingTest extends LiaTestCase {

  public void testTerm() throws Exception {
    IndexSearcher searcher = new IndexSearcher(directory);
    Term t = new Term("subject", "ant");
    Query query = new TermQuery(t);
    Hits hits = searcher.search(query);
    assertEquals("JDwA", 1, hits.length());

    t = new Term("subject", "junit");
    hits = searcher.search(new TermQuery(t));
    assertEquals(2, hits.length());

    searcher.close();
  }
}
```

A `Hits` object is returned from our search. We'll discuss this object in section 3.2, but for now just note that the `Hits` object encapsulates access to the underlying `Documents`. This encapsulation makes sense for efficient access to documents. Full documents aren't immediately returned; they're fetched on demand. In this example we didn't concern ourselves with the actual documents associated with the hits returned because we were only interested in asserting that the proper number of documents were found.

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

Two more features round out what the majority of searching applications require: sophisticated query expression parsing and access to the documents returned. Lucene's search methods require a `Query` object. *Parsing* a query expression is the act of turning a user-entered query such as "mock OR junit" into an appropriate `Query` object instance;[3] in this case, the `Query` object would be an instance of `BooleanQuery` with two nonrequired clauses, one for each term. The following code parses two query expressions and asserts that they worked as expected. After returning the hits, we retrieve the title from the first document found:

```
public void testQueryParser() throws Exception {
  IndexSearcher searcher = new IndexSearcher(directory);

  Query query = QueryParser.parse("+JUNIT +ANT -MOCK",
                                  "contents",
                                  new SimpleAnalyzer());
  Hits hits = searcher.search(query);
  assertEquals(1, hits.length());
  Document d = hits.doc(0);
  assertEquals("Java Development with Ant", d.get("title"));

  query = QueryParser.parse("mock OR junit",
                            "contents",
                            new SimpleAnalyzer());
  hits = searcher.search(query);
  assertEquals("JDwA and JIA", 2, hits.length());
}
```

Lucene includes an interesting feature that parses query expressions 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. Dealing with human-entered queries is the primary purpose of the `QueryParser`.

`QueryParser` requires an *analyzer* to break pieces of the query 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

[3] 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.

analyzer. Querying through the API using `TermQuery` and the others discussed in section 3.4 doesn't use an analyzer but does rely on matching terms to what was indexed. 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` has a static `parse()` method to allow for the simplest use. Its signature is

```
static public Query
   parse(String query, String field, Analyzer analyzer)
      throws ParseException
```

The `query String` is the expression to be parsed, such as "+cat +dog". The second parameter, `field`, is the name of the default field to associate with terms in the expression (more on this in section 3.5.4). The final argument is an `Analyzer` instance. (We discuss analyzers in detail in the next chapter and then cover the interactions between `QueryParser` and the analyzer in section 4.1.2.) The `testQueryParser()` method shown in section 3.1.2 demonstrates using the static `parse()` method.

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 static `parse()` method is quick and convenient to use, but it may not be sufficient. Under the covers, the static method instantiates an instance of `QueryParser` and invokes the instance `parse()` method. You can do the same thing yourself, which gives you a finer level of control. There are various settings that can be controlled on a `QueryParser` instance, such as the default operator (which defaults to OR). These settings also include locale (for date parsing), default phrase slop, and whether to lowercase wildcard queries. The `QueryParser` constructor takes the default field and analyzer. The instance `parse()` method is passed the expression to parse. See section 3.5.6 for an example.

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`
	Have *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
`java*`	Contain terms that begin with *java*, like *javaspaces*, *javaserver*, and *java.net*
`java~`	Contain terms that are close *to* the word *java*, such as *lava*
`lastmodified:` `[1/1/04 TO 12/31/04]`	Have `lastmodified` field values between the dates January 1, 2004 and December 31, 2004

[a] *The default operator is OR. It can be set to AND (see section 3.5.2).*

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.

3.2 Using IndexSearcher

Let's take a closer look at Lucene's `IndexSearcher` class. Like the rest of Lucene's primary API, it's simple to use. Searches are done using an instance of `IndexSearcher`. Typically, you'll use one of the following approaches to construct an `IndexSearcher`:

- By `Directory`
- By a file system path

We recommend using the `Directory` constructor—it's better to decouple searching from where the index resides, allowing your searching code to be agnostic to whether the index being searched is on the file system, in RAM, or elsewhere. Our base test case, `LiaTestCase`, provides `directory`, a `Directory` implementation. Its actual implementation is an `FSDirectory` loaded from a file system index. Our `setUp()` method opens an index using the static `FSDirectory.getDirectory()` method, with the index path defined from a JVM system property:

```
public abstract class LiaTestCase extends TestCase {
  private String indexDir = System.getProperty("index.dir");
  protected Directory directory;

  protected void setUp() throws Exception {
    directory = FSDirectory.getDirectory(indexDir,false);
  }

// ...
}
```

The last argument to `FSDirectory.getDirectory()` is `false`, indicating that we want to open an existing index, not construct a new one. An `IndexSearcher` is created using a `Directory` instance, as follows:

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

After constructing an `IndexSearcher`, we call one of its `search` methods to perform a search. The three main search method signatures available to an `IndexSearcher` instance are shown in table 3.3. This chapter only deals with `search(Query)` method, and that may be the only one you need to concern yourself with. The other `search` method signatures, including the sorting variants, are covered in chapter 5.

Table 3.3 Primary `IndexSearcher search` methods

`IndexSearcher.search` method signature	When to use
`Hits search(Query query)`	Straightforward searches needing no filtering.
`Hits search(Query query, Filter filter)`	Searches constrained to a subset of available documents, based on filter criteria.
`void search(Query query, HitCollector results)`	Used only when *all* documents found from a search will be needed. Generally, only the top few documents from a search are needed, so using this method could be a performance killer.

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 must instantiate a new `IndexSearcher`.

3.2.1 *Working with Hits*

Now that we've called `search(Query)`, we have a `Hits` object at our disposal. The search results are accessed through `Hits`. Typically, you'll use one of the `search` methods that returns a `Hits` object, as shown in table 3.3. The `Hits` object provides efficient access to search results. 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 5.1).

There are only four methods on a `Hits` instance; they're listed in table 3.4. The method `Hits.length()` returns the number of *matching documents*. A matching document is one with a score greater than zero, as defined by the scoring formula covered in section 3.3. The hits, by default, are in decreasing score order.

Table 3.4 `Hits` methods for efficiently accessing search results

`Hits` method	Return value
`length()`	Number of documents in the `Hits` collection
`doc(n)`	`Document` instance of the *n*th top-scoring document
`id(n)`	Document ID of the *n*th top-scoring document
`score(n)`	Normalized score (based on the score of the topmost document) of the *n*th top-scoring document, guaranteed to be greater than 0 and less than or equal to 1

The `Hits` object caches a limited number of documents and maintains a most-recently-used list. The first 100 documents are automatically retrieved and cached initially. The `Hits` collection lends itself to environments where users are presented with only the top few documents and typically don't need more than those because only the best-scoring hits are the desired documents.

The methods `doc(n)`, `id(n)`, and `score(n)` require documents to be loaded from the index when they aren't already cached. This leads us to recommend only calling these methods for documents you truly need to display or access; defer calling them until needed.

3.2.2 Paging through Hits

Presenting search results to end users most often involves displaying only the first 20 or so most relevant documents. Paging through `Hits` is a common need. There are a couple of implementation approaches:

- Keep the original `Hits` and `IndexSearcher` instances available while the user is navigating the search results.
- Requery each time the user navigates to a new page.

It turns out that requerying is most often the best solution. Requerying eliminates the need to store per-user state. In a web application, staying stateless (no HTTP session) is often desirable. Requerying at first glance seems a waste, but Lucene's blazing speed more than compensates.

In order to requery, the original search is reexecuted 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 is parsed with `QueryParser`, the original expression could be made part of the hyperlinks 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; chances are you'll find this sufficient for your needs.

3.2.3 Reading indexes into memory

Using `RAMDirectory` is suitable for situations requiring only transient indexes, but most applications need to persist their indexes. They will eventually need to use `FSDirectory`, as we've shown in the previous two chapters.

However, in some scenarios, indexes are used in a read-only fashion. Suppose, for instance, that you have a computer whose main memory exceeds the size of a Lucene index stored in the file system. Although it's fine to always search the index stored in the index directory, you could make better use of your hardware resources by loading the index from the slower disk into the faster RAM and then searching that in-memory index. In such cases, `RAMDirectory`'s constructor can be used to read a file system–based index into memory, allowing the application that accesses it to benefit from the superior speed of the RAM:

```
RAMDirectory ramDir = new RAMDirectory(dir);
```

`RAMDirectory` has several overloaded constructors, allowing a `java.io.File`, a path `String`, or another `Directory` to load into RAM. Using an `IndexSearcher` with a `RAMDirectory` is straightforward and no different than using an `FSDirectory`.

3.3 Understanding Lucene scoring

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. Without further ado, meet Lucene's similarity scoring formula, shown in figure 3.1. The score is computed for each document (`d`) matching a specific.

NOTE If this equation or the thought of mathematical computations scares you, you may safely skip this section. Lucene 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*. Scores returned from `Hits` aren't necessarily the raw score, however. If the top-scoring document scores greater than 1.0, all scores are normalized from that score, such that all scores from `Hits` are guaranteed to be 1.0 or less. Table 3.5 describes each of the factors in the scoring formula.

$$\sum_{t\ in\ q} tf(t\ in\ d) \cdot idf(t) \cdot boost(t.field\ in\ d) \cdot lengthNorm(t.field\ in\ d)$$

Figure 3.1 Lucene uses this formula to determine a document score based on a query.

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`).
`idf(t)`	Inverse document frequency of the term.
`boost(t.field in d)`	Field boost, as set during indexing.
`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.
`coord(q, d)`	Coordination factor, based on the number of query terms the document contains.
`queryNorm(q)`	Normalization value for a query, given the sum of the squared weights of each of the query terms.

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 2.3 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 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, boosting it would boost 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 queries. Queries also default to a 1.0 boost factor.

Most of these scoring formula factors are controlled through an implementation of the `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 these factors, 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 require that the index be rebuilt for all factors to be in sync.

3.3.1 Lucene, you got a lot of 'splainin' to do!

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 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 `Explanation`s, shown here:

```
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];

    FSDirectory directory =
        FSDirectory.getDirectory(indexDir, false);

    Query query = QueryParser.parse(queryExpression,
        "contents", new SimpleAnalyzer());

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

    IndexSearcher searcher = new IndexSearcher(directory);
    Hits hits = searcher.search(query);

    for (int i = 0; i < hits.length(); i++) {
```

```
        Explanation explanation =
                                searcher.explain(query, hits.id(i));
        System.out.println("----------");
        Document doc = hits.doc(i);
        System.out.println(doc.get("title"));
        System.out.println(explanation.toString());
      }
    }
  }
```

Generate Explanation of single Document for query

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
0.65311843 = fieldWeight(contents:junit in 2), product of:
  1.4142135 = tf(termFreq(contents:junit)=2)
  1.8472979 = idf(docFreq=2)
  0.25 = fieldNorm(field=contents, doc=2)

----------
Java Development with Ant
0.46182448 = fieldWeight(contents:junit in 1), product of:
  1.0 = tf(termFreq(contents:junit)=1)
  1.8472979 = idf(docFreq=2)
  0.25 = fieldNorm(field=contents, doc=1)
```

❶ "junit" appears twice in contents

❷ "junit" appears once in contents

❶ *JUnit in Action* has the term *junit* twice in its `contents` field. The `contents` field in our index is an aggregation of the `title` and `subject` fields to allow a single field for searching.

❷ *Java Development with Ant* has the term *junit* only once in its `contents` field.

There is 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 (see the case study in section 10.1), 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.

3.4 Creating queries programmatically

As you saw in section 3.2, querying Lucene ultimately requires a call to `IndexSearcher`'s `search` 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 a parser such as QueryParser. If your application will rely solely on QueryParser to construct Query objects, understanding Lucene's direct API capabilities is still important because QueryParser uses them.

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 user interface, 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, BooleanQuery, RangeQuery, and a TermQuery. We demonstrate building a similar aggregate query in section 5.5.4.

This section covers each of Lucene's built-in Query types. The QueryParser expression syntax that maps to each Query type is provided.

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 5) may have indexed things differently.

TermQuerys are especially useful for retrieving documents by a key. If documents were indexed using Field.Keyword(), 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 {
  IndexSearcher searcher = new IndexSearcher(directory);
  Term t = new Term("isbn", "1930110995");
  Query query = new TermQuery(t);
  Hits hits = searcher.search(query);
  assertEquals("JUnit in Action", 1, hits.length());
}
```

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

TermQuery and QueryParser

A single word in a query expression corresponds to a term. A `TermQuery` is returned from `QueryParser` if the expression consists of a single word. The expression `java` creates a `TermQuery`, just as we did with the API in `testKeyword`.

3.4.2 Searching within a range: RangeQuery

Terms are ordered lexicographically within the index, allowing for efficient searching of terms within a range. Lucene's `RangeQuery` facilitates searches from a starting term through an ending term. The beginning and ending terms may either be included or excluded. The following code illustrates range queries inclusive of the begin and end terms:

```
public class RangeQueryTest extends LiaTestCase {
  private Term begin, end;

  protected void setUp() throws Exception {
    begin = new Term("pubmonth","198805");

    // pub date of TTC was October 1988
    end = new Term("pubmonth","198810");

    super.setUp();
  }

  public void testInclusive() throws Exception {
    RangeQuery query = new RangeQuery(begin, end, true);
    IndexSearcher searcher = new IndexSearcher(directory);

    Hits hits = searcher.search(query);
    assertEquals("tao", 1, hits.length());
  }
}
```

Our test data set has only one book, *Tao Te Ching* by Stephen Mitchell, published between May 1988 and October 1988; it was published in October 1988. The third argument to construct a `RangeQuery` is a boolean flag indicating whether the range is inclusive. Using the same data and range, but exclusively, one less book is found:

```
public void testExclusive() throws Exception {
  RangeQuery query = new RangeQuery(begin, end, false);
```

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

    Hits hits = searcher.search(query);
    assertEquals("there is no tao", 0, hits.length());
  }
```

RangeQuery and QueryParser

`QueryParser` constructs `RangeQuery`s from the expression `[begin TO end]` or `{begin TO end}`. Square brackets denote an inclusive range, and curly brackets denote an exclusive range. If the begin and end terms represent dates (and parse successively as such), then ranges over fields created as dates using `DateField` or `Keyword(String, Date)` can be constructed. See section 3.5.5 for more on `RangeQuery` and `QueryParser`.

3.4.3 Searching on a string: PrefixQuery

Searching with a `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` keyword field representing a hierarchical structure:

```
public class PrefixQueryTest extends LiaTestCase {
  public void testPrefix() throws Exception {
    IndexSearcher searcher = new IndexSearcher(directory);

    Term term = new Term("category",
                  "/technology/computers/programming");
    PrefixQuery query = new PrefixQuery(term);

    Hits hits = searcher.search(query);
    int programmingAndBelow = hits.length();

    hits = searcher.search(new TermQuery(term));
    int justProgramming = hits.length();

    assertTrue(programmingAndBelow > justProgramming);
  }
}
```

Search for programming books, including subcategories (annotates the `Term` and `PrefixQuery` lines)

Search only for programming books, not subcategories (annotates the `TermQuery` search line)

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.

PrefixQuery and QueryParser

`QueryParser` creates a `PrefixQuery` for a term when it ends with an asterisk (`*`) in query expressions. For example, `luc*` is converted into a `PrefixQuery` using *luc* as the term. By default, the prefix text is lowercased by `QueryParser`. See section 3.5.7 for details on how to control this setting.

3.4.4 Combining queries: BooleanQuery

The various query types discussed here can be combined in complex ways using `BooleanQuery`. `BooleanQuery` itself is a container of Boolean *clauses*. A clause is a subquery that can be optional, required, 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, boolean required, boolean prohibited)
```

A `BooleanQuery` can be a clause within another `BooleanQuery`, allowing for sophisticated groupings. Let's look at some examples. First, here's an AND query to find the most recent books on one of our favorite subjects, *search*:

```
public void testAnd() throws Exception {
  TermQuery searchingBooks =
              new TermQuery(new Term("subject","search"));      ❶ All books with subject "search"

  RangeQuery currentBooks =
              new RangeQuery(new Term("pubmonth","200401")
                             new Term("pubmonth","200412"),      ❷ All books in 2004
                             true);

  BooleanQuery currentSearchingBooks = new BooleanQuery();
  currentSearchingBooks.add(searchingBook s, true, false);      ❸ Combines two queries
  currentSearchingBooks.add(currentBooks, true, false);

  IndexSearcher searcher = new IndexSearcher(directory);
  Hits hits = searcher.search(currentSearchingBooks);

  assertHitsIncludeTitle(hits, "Lucene in Action");
}

// following method from base LiaTestCase class
protected final void assertHitsIncludeTitle(
                                          Hits hits, String title)
  throws IOException {
  for (int i=0; i < hits.length(); i++) {
    Document doc = hits.doc(i);                                 ❹ Custom convenience assert method
    if (title.equals(doc.get("title"))) {
      assertTrue(true);
```

```
      return;
    }
  }

  fail("title '" + title + "' not found");
}
```

❹ **Custom convenience assert method**

❶ This query finds all books containing the subject `"search"`.

❷ This query find all books published in 2004. (Note that this could also be done with a `"2004"` `PrefixQuery`.)

❸ Here we combine the two queries into a single boolean query with both clauses required (the second argument is true).

❹ This custom convenience assert method allows more readable test cases.

`BooleanQuery.add` has two overloaded method signatures. One accepts a `BooleanClause`, and the other accepts a `Query` and two boolean flags. A `BooleanClause` is a container of a query and the two boolean flags, so we omit coverage of it. The boolean flags are `required` and `prohibited`, respectively. There are four logical combinations of these flags, but the case where both are `true` is an illogical and invalid combination. A required clause means exactly that: Only documents matching that clause are considered. Table 3.6 shows the various combinations and effect of the required and prohibited flags.

Table 3.6 `BooleanQuery` clause attributes

		`required`	
		false	**true**
`prohibited`	**false**	Clause is optional	Clause must match
	true	Clause must `not` match	Invalid

Performing an OR query only requires setting the `required` and `prohibited` flags both to `false`, as in this example:

```
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, false, false);
    enlightenmentBooks.add(easternPhilosophyBooks, false, false);

    IndexSearcher searcher = new IndexSearcher(directory);
    Hits hits = searcher.search(enlightenmentBooks);

    assertHitsIncludeTitle(hits, "Extreme Programming Explained");
    assertHitsIncludeTitle(hits,
                               "Tao Te Ching \u9053\u5FB7\u7D93"4);
  }
```

`BooleanQuery`s are restricted to a maximum number of clauses; 1,024 is the default. This limitation is in place to prevent queries from adversely affecting performance. A `TooManyClauses` exception is thrown if the maximum is exceeded. It may seem that this is an extreme number and that constructing this number of clauses is unlikely, but under the covers Lucene does some of its own query rewriting for queries like `RangeQuery` and turns them into a `BooleanQuery` with nested optional (not required, not prohibited) `TermQuery`s. Should you ever have the unusual need of increasing the number of clauses allowed, there is a `setMaxClauseCount(int)` method on `BooleanQuery`.

BooleanQuery and QueryParser

`QueryParser` handily constructs `BooleanQuery`s when multiple terms are specified. Grouping is done with parentheses, and the `prohibited` and `required` flags are set when the –, +, AND, OR, and NOT operators are specified.

3.4.5 Searching by phrase: PhraseQuery

An index contains positional information of terms. `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 to

---

[4] The `\u` notation is a Unicode escape sequence. In this case, these are the Chinese characters for *Tao Te Ching*. We use this for our search of Asian characters in section 4.8.3.

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 and a custom `matched (String[], int)` method to construct, execute, and assert a phrase query matched the test document:

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

  protected void setUp() throws IOException {
    // set up sample document
    RAMDirectory directory = new RAMDirectory();
    IndexWriter writer = new IndexWriter(directory,
        new WhitespaceAnalyzer(), true);
    Document doc = new Document();
    doc.add(Field.Text("field",
              "the quick brown fox jumped over the lazy dog"));
    writer.addDocument(doc);
    writer.close();

    searcher = new IndexSearcher(directory);
  }

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

    for (int i=0; i < phrase.length; i++) {
      query.add(new Term("field", phrase[i]));
    }

    Hits hits = searcher.search(query);
    return hits.length() > 0;
  }
}
```

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));
}
```

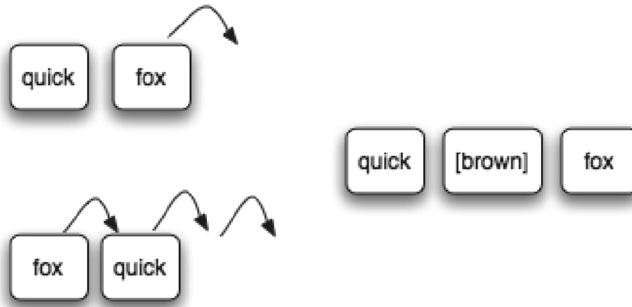


Figure 3.2 Illustrating `PhraseQuery` slop factor: "quick fox" requires a slop of 1 to match, whereas "fox quick" requires a slop of 3 to match.

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 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 3.2 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, we 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 3.3. The inverse relationship with distance ensures that greater distances have lower scores.

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

Figure 3.3 Sloppy phrase scoring

NOTE Terms surrounded by double quotes in `QueryParser` parsed expressions are translated into a `PhraseQuery`. The slop factor defaults to zero, 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 3.5.6. Phrases are analyzed by the analyzer passed to the `QueryParser`, adding another layer of complexity, as discussed in section 4.1.2.

3.4.6 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 3.2 demonstrates `WildcardQuery` in action.

Listing 3.2 Searching on the wild(card) side

```
private void indexSingleFieldDocs(Field[] fields) throws Exception {
  IndexWriter writer = new IndexWriter(directory,
      new WhitespaceAnalyzer(), true);
  for (int i = 0; i < fields.length; i++) {
    Document doc = new Document();
    doc.add(fields[i]);
    writer.addDocument(doc);
  }
```

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

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

    IndexSearcher searcher = new IndexSearcher(directory);
    Query query = new WildcardQuery(
              new Term("contents", "?ild*"));       <- Construct WildcardQuery using Term
    Hits hits = searcher.search(query);
    assertEquals("child no match", 3, hits.length());

    assertEquals("score the same", hits.score(0),
                                   hits.score(1), 0.0);
    assertEquals("score the same", hits.score(1),
                                   hits.score(2), 0.0);
  }
```

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 a string.

WARNING Performance degradations can occur when you use `WildcardQuery`. A larger prefix (characters before the first wildcard character) decreases the 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 affect on scoring. The last two assertions in listing 3.2, where *wild* and *mild* are closer matches to the pattern than *mildew*, demonstrate this.

WildcardQuery and QueryParser

`QueryParser` supports `WildcardQuery` using the same syntax for a term as used by the API. There are a few important differences, though. With `QueryParser`, the first character of a wildcarded term may not be a wildcard character; this restriction prevents users from putting asterisk-prefixed terms into a search expression,

incurring an expensive operation of enumerating all the terms. Also, if the only wildcard character in the term is a trailing asterisk, the query is optimized to a `PrefixQuery`. Wildcard terms are lowercased automatically by default, but this can be changed. See section 3.5.7 for more on wildcard queries and `QueryParser`.

3.4.7 Searching for similar terms: FuzzyQuery

The final built-in query is one of the more interesting. 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.[5] *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 `PhrasePrefixQuery`. 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[] {
    Field.Text("contents", "fuzzy"),
    Field.Text("contents", "wuzzy")
  });

  IndexSearcher searcher = new IndexSearcher(directory);
  Query query = new FuzzyQuery(new Term("contents", "wuzza"));
  Hits hits = searcher.search(query);
  assertEquals("both close enough", 2, hits.length());

  assertTrue("wuzzy closer than fuzzy",
            hits.score(0) !=  hits.score(1));

  assertEquals("wuzza bear",
           "wuzzy", hits.doc(0).get("contents"));
}
```

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.

---

[5] See http://www.merriampark.com/ld.htm for more information about Levenshtein distance.

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

Figure 3.4 **`FuzzyQuery` distance formula.**

Edit distance affects scoring, such that terms with less edit distance are scored higher. Distance is computed using the formula shown in figure 3.4.

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.

FuzzyQuery and QueryParser

`QueryParser` supports `FuzzyQuery` by suffixing a term with a tilde (~). For example, the `FuzzyQuery` from the previous example would be `wuzza~` in a query expression. 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.

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. This 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:[20040101 TO 20041231] Java AND (Jakarta OR Apache)
```

This query searches for all books about Java that also include *Jakarta* or *Apache* in their contents and were published in 2004.

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 escapable characters are as follows:

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

The following sections detail the expression syntax, examples of using `QueryParser`, and customizing `QueryParser`'s behavior. The discussion of `QueryParser` in this section assumes knowledge of the query types previously discussed in section 3.4. 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 the `toString()` method.

All concrete core `Query` classes we've discussed in this chapter have a special `toString()` implementation. They output valid `QueryParser` parsable strings. 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", "kountry6")), true, false);
  query.add(
     new TermQuery(new Term("title", "western")), false, false);

  assertEquals("both kinds",
               "+kountry~ 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.

3.5.2 Boolean operators

Constructing Boolean queries textually via `QueryParser` is done using the operators AND, OR, and NOT. 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, use an instance of `QueryParser` rather than the static `parse` method:

---

[6] Misspelled on purpose to illustrate FuzzyQuery.

```
QueryParser parser = new QueryParser("contents", analyzer);
parser.setOperator(QueryParser.DEFAULT_OPERATOR_AND);
```

Placing a NOT in front of a term excludes documents matching the following term. Negating a term must be combined with at least one nonnegated 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 3.7 illustrates various syntax equivalents.

Table 3.7 Boolean query operator shortcuts

Verbose syntax	Shortcut syntax
`a AND b`	`+a +b`
`a OR b`	`a b`
`a AND NOT b`	`+a -b`

3.5.3 Grouping

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

```
public void testGrouping() throws Exception {
  Query query = QueryParser.parse(
      "(agile OR extreme) AND methodology",
      "subject",
      analyzer);
  Hits hits = searcher.search(query);

  assertHitsIncludeTitle(hits, "Extreme Programming Explained");
  assertHitsIncludeTitle(hits, "The Pragmatic Programmer");
}
```

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

3.5.4 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 to the `parse` method. 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, when HTML documents are indexed with the title and body areas as separate fields, the default field will likely be `body`. Users can search for `title` fields using a query such as `title:lucene`. You can group field selection over several terms using `field:(a b c)`.

3.5.5 Range searches

Text or date range queries use bracketed syntax, with `TO` between the beginning term and ending term. The type of bracket determines whether the range is inclusive (square brackets) or exclusive (curly brackets). Our `testRangeQuery()` method demonstrates both inclusive and exclusive range queries:

```
public void testRangeQuery() throws Exception {
  Query query = QueryParser.parse(
        "pubmonth:[200401 TO 200412]", "subject", analyzer);   ❶ Inclusive range

  assertTrue(query instanceof RangeQuery);

  Hits hits = searcher.search(query);
  assertHitsIncludeTitle(hits, "Lucene in Action");

  query = QueryParser.parse(
       "{200201 TO 200208}", "pubmonth", analyzer);   ❷ Exclusive range

  hits = searcher.search(query);
  assertEquals("JDwA in 200208", 0, hits.length());   ❸ Demonstrates exclusion of pubmonth 200208
}
```

❶ This inclusive range uses a field selector since the default field is `subject`.

❷ This exclusive range uses the default field `pubmonth`.

❸ *Java Development with Ant* was published in August 2002, so we've demonstrated that the `pubmonth` value 200208 is excluded from the range.

NOTE Nondate range queries use the beginning and ending terms as the user entered them, without modification. In other words, the beginning and ending terms are *not* analyzed. Start and end terms must not contain whitespace, or parsing fails. In our example index, the field `pubmonth` isn't a date field; it's text of the format YYYYMM.

Handling date ranges

When a range query is encountered, the parser code first attempts to convert the start and end terms to dates. If the terms are valid dates, according to `DateFormat.SHORT` and lenient parsing within the default or specified locale, then the dates are converted to their internal textual representation (see section 2.4 on `DateField`).

If either of the two terms fails to parse as a valid date, they're both used as is for a textual range.

The `Query`'s `toString()` output is interesting for date-range queries. Let's parse one to see:

```
Query query = QueryParser.parse("modified:[1/1/04 TO 12/31/04]",
                                "subject", analyzer);
System.out.println(query);
```

This outputs something strange:

```
modified:[0dowcq3k0 TO 0e3dwg0w0]
```

Internally, all terms are text to Lucene, and dates are represented in a lexicographically ordered text format. As long as our modified field was indexed properly as a `Date`, all is well despite this odd-looking output.

Controlling the date-parsing locale

To change the locale used for date parsing, construct a `QueryParser` instance and call `setLocale()`. Typically the client's locale would be determined and used, rather than the default locale. For example, in a web application, the `HttpServletRequest` object contains the locale set by the client browser. You can use this locale to control the locale used by date parsing in `QueryParser`, as shown in listing 3.3.

Listing 3.3 Using the client locale in a web application

```
public class SearchServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
      throws ServletException, IOException {

    QueryParser parser = new QueryParser("contents",
        new StandardAnalyzer());

    parser.setLocale(request.getLocale());

    try {
      Query query = parser.parse(request.getParameter("q"));
    } catch (ParseException e) {
      // ... handle exception
    }

    // ... display results ...
  }
}
```

QueryParser's setLocale is one way in which Lucene facilitates internationalization (often abbreviated I18N) concerns. Text analysis is another, more important, place where such concerns are handled. Further I18N issues are discussed in section 4.8.2.

3.5.6 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 (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 = QueryParser.parse("\"This is Some Phrase*\"",
      "field", new StandardAnalyzer());
  assertEquals("analyzed",
      "\"some phrase\"", q.toString("field"));

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

The slop factor is zero unless you specify it using a trailing tilde (~) and the desired integer slop value. Because the implicit analysis of phrases may not match what was indexed, the slop factor can be set to something other than zero automatically if it isn't specified using the tilde notation:

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

  QueryParser qp = new QueryParser("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. However, a SpanNearQuery (discussed in section 5.4.3) has the ability to guarantee an in-order match. In section 6.3.4, we extend QueryParser and substitute a SpanNearQuery when phrase queries are parsed, allowing for sloppy in-order phrase matches.

3.5.7 Wildcard and prefix queries

If a term contains an asterisk or a question mark, it's considered a WildcardQuery. When the term only contains 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 = QueryParser.parse("PrefixQuery*", "field",
                  analyzer);
  assertEquals("lowercased",
      "prefixquery*", q.toString("field"));

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

To turn off the automatic lowercasing, you must construct your own instance of QueryParser rather than use the static parse method.

Wildcards at the beginning of a term are prohibited using QueryParser, but an API-coded WildcardQuery may use leading wildcards (at the expense of performance). Section 3.4.6 discusses more about the performance issue, and section 6.3.1 provides a way to prohibit WildcardQuerys from parsed expressions if you wish.

3.5.8 Fuzzy queries

A trailing tilde (~) creates a fuzzy query on the preceding term. The same performance caveats the apply to WildcardQuery also apply to fuzzy queries and can be disabled with a customization similar to that discussed in section 6.3.1.

3.5.9 Boosting queries

A carat (^) followed by a floating-point number sets the boost factor for the preceding query. Section 3.3 discusses boosting queries in more detail. 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.

3.5.10 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[7] and JavaCC[8] are great options. We don't discuss the creation of a custom query parser; however, the source code for Lucene's `QueryParser` is freely available for you to borrow from.

You can often obtain a happy medium by combining a `QueryParser`-parsed query with API-created queries as clauses in a `BooleanQuery`. This approach is demonstrated in section 5.5.4. For example, if users need to constrain searches to a particular category or narrow them to a date range, you can have the user interface separate those selections into a category chooser or separate date-range fields.

3.6 Summary

Lucene rapidly provides highly relevant search results to queries. Most applications need only a few Lucene classes and methods to enable searching. The most fundamental things for you to take from this chapter are an understanding of the basic query types (of which `TermQuery`, `RangeQuery`, and `BooleanQuery` are the primary ones) 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;

---

[7] http://www.antlr.org.

[8] http://javacc.dev.java.net.

however, 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 is more to searching than we've covered in this chapter, but understanding the groundwork is crucial. 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

Otis Gospodnetić • Erik Hatcher FOREWORD BY Doug Cutting

Lucene is a gem in the open-source world—a highly scalable, fast search engine. It delivers performance and is disarmingly easy to use. *Lucene in Action* is the authoritative guide to Lucene. It describes how to index your data, including types you definitely need to know such as MS Word, PDF, HTML, and XML. It introduces you to searching, sorting, filtering, and highlighting search results.

Lucene powers search in surprising places—in discussion groups at Fortune 100 companies, in commercial issue trackers, in email search from Microsoft, in the Nutch web search engine (that scales to billions of pages). It is used by diverse companies including Akamai, Overture, Technorati, HotJobs, Epiphany, FedEx, Mayo Clinic, MIT, New Scientist Magazine, and many others.

Adding search to your application can be easy. With many reusable examples and good advice on best practices, *Lucene in Action* shows you how.

What's Inside

- How to integrate Lucene into your applications
- Ready-to-use framework for rich document handling
- Case studies including Nutch, TheServerSide, jGuru, etc.
- Lucene ports to Perl, Python, C#/.Net, and C++
- Sorting, filtering, term vectors, multiple, and remote index searching
- The new SpanQuery family, extending query parser, hit collecting
- Performance testing and tuning
- Lucene add-ons (hit highlighting, synonym lookup, and others)

A committer on the Ant, Lucene, and Tapestry open-source projects, **Erik Hatcher** is coauthor of Manning's award-winning *Java Development with Ant*. **Otis Gospodnetić** is a Lucene committer, a member of Apache Jakarta Project Management Committee, and maintainer of the jGuru's Lucene FAQ. Both authors have published numerous technical articles including several on Lucene.

"… packed with examples and advice on how to effectively use this incredibly powerful tool."

—Brian Goetz
Principal Consultant,
Quiotix Corporation

"… it unlocked for me the amazing power of Lucene."

—Reece Wilton, Staff Engineer,
Walt Disney Internet Group

"… the code examples are useful and reusable."

—Scott Ganyo
Jakarta Lucene Committer

"… code samples as JUnit test cases are incredibly helpful."

—Norman Richards, co-author
XDoclet in Action

AUTHOR ONLINE
Ask the Authors

Ebook edition

www.manning.com/hatcher

54495
9 781932 394283
ISBN 1-932394-28-1

MANNING $44.95 US/$60.95 Canada