Quick answers to common problems

Lucene 4 Cookbook

Over 70 hands-on recipes to quickly and effectively integrate Lucene into your search application

Edwood Ng    Vineeth Mohan

PACKT open source community experience distilled
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 'Introducing Lucene'
- A synopsis of the book’s content
- More information on Lucene 4 Cookbook
About the Authors

**Edwood Ng** is a technologist with over a decade of experience in building scalable solutions from proprietary implementations to client-facing web-based applications. Currently, he's the director of DevOps at Wellframe, leading infrastructure and DevOps operations.

His background in search engine began at Endeca Technologies in 2004, where he was a technical consultant helping numerous clients to architect and implement faceted search solutions. After Endeca, he drew on his knowledge and began designing and building Lucene-based solutions. His first Lucene implementation that went to production was the search engine behind [http://UpDown.com](http://UpDown.com). From there on, he continued to create search applications using Lucene extensively to deliver robust and scalable systems for his clients. Edwood is a supporter of an open source software. He has also contributed to the plugin `sfI18NGettextPluralPlugin` to the Symphony project.

**Vineeth Mohan** is a data scientist whose major expertise lies in search analytics and data science. His career started at Yahoo! India, working with a team facilitated by a real-time feed processing platform based on Hadoop across all verticals in Yahoo!. He was also in the team that migrated Right Media technologies to Yahoo! stack. He then moved to Algotree, where he worked across many data science projects such as technical pattern detection in TimeSeries. Later, he designed a real-time news analysis platform based on natural language processing and machine learning approaches. He was also the architect of the search solution for this. Vineeth is passionate about Big Data and data science and has extensive knowledge and experience in Elasticsearch/Solr/Lucene and algorithm design.
Apache Lucene 4 is the backbone of many search engine implementations, including big names such as LinkedIn, Twitter, and IBM. It also serves as a foundation for other open source projects such as Solr and Elasticsearch. We will dive into all the features that Lucene offers and show you easy-to-understand recipes to get you started using Lucene.

**What this book covers**

*Chapter 1, Introducing Lucene*, introduces you to the core components of Lucene and gives you the know-how to set up Lucene on your own.

*Chapter 2, Analyzing Your Text*, explores a key feature of Lucene called Analyzers. We will show you how text analyzing works and how to customize it to suit your needs.

*Chapter 3, Indexing Your Data*, looks into the data injection process in Lucene and reviews the core concepts, such as norms and similarity.

*Chapter 4, Searching Your Indexes*, will cover the core search components such as FieldCache and TermVectors, and will give you all the knowledge you need to build an effective search engine.

*Chapter 5, Near Real-time Searching*, will show you how near real-time search is achieved via various methods and their trade-offs, so you can make educated decisions when designing your search application.

*Chapter 6, Querying and Filtering Data*, gives you a glimpse of the various querying and filtering features that have been proven to build successful search engines.

*Chapter 7, Flexible Scoring*, takes a technical dive into the mechanics of scoring, how scores are determined, how it affects ranking positions and what you can do to customize it.
Chapter 8, *Introducing Elasticsearch*, gives an introduction to a Lucene-based open source search solution that gives you everything you need to build a search application in no time.

Chapter 9, *Extending Lucene with Modules*, explores the additional features, such as spatial search and faceting, that extend Lucene functionalities beyond just text search.
Introducing Lucene

Many applications in the modern era often require the handling of large datasets. Managing and searching these large collections of information can be very challenging, hence the creation of efficient and high performance search applications has become a necessity. For decades, many data scientists' research focused on information retrieval. One can say that the open source community now bears the fruits of this hard work as many open source data management platforms are developed. The Apache Software Foundation's answer to this: The Apache Lucene has gained popularity recently and is considered the go-to text search framework by many.

Let us take a look at the recipes that we are going to cover in this chapter:

- Installing Lucene
- Setting up a simple Java Lucene project
- Obtaining an IndexWriter
- Creating an analyzer
- Creating fields
- Creating and writing documents to an index
- Deleting documents
- Obtaining an IndexSearcher
- Creating queries with the Lucene QueryParser
- Performing a search
- Enumerating results
Introducing Lucene

Getting Lucene and setting up a Lucene Java project serves as a guide for you to get started with Lucene. Instructions to download and set up Lucene are covered in detail in these two recipes. All the recipes that follow introduce basic Lucene functionalities, which do not require in-depth knowledge to understand. We will learn how to create an index and add documents to an index. We will practice deleting documents and searching these documents to locate information. The Creating fields section of this chapter introduces you to Lucene’s way of handling information. Then, we will learn how to formulate search queries. At the end of this chapter, we will show you how to retrieve search results from Lucene. Hopefully, by completing this chapter, you will gain enough knowledge to set up Lucene and have a good grasp of Lucene’s concept of indexing and searching information.

Introduction

Before getting into the intricacies of Lucene, we will show you how a typical search application is created. It will help you better understand the scope of Lucene. The following figure outlines a high level indexing process for a news article search engine. For now, we will focus on the essentials of creating a search engine:

![Basic Process Flow for a News Search Application](image)

1. Fetch News
2. Add News Article to an Index
3. Search News
The preceding diagram has a three stage process flow:

- The first stage is data acquisition where the data we intend to make searchable is fetched. The source of this information can be the web, or your private collection of documents in text, pdf, xml and so on.
- The second stage manages the fetched information where the collected data is indexed and stored.
- Finally, we perform a search on the index, and return the results.

Lucene is the platform where we index information and make it searchable. The first stage is independent of Lucene — you will provide the mechanism to fetch the information. Once you have the information, we can use Lucene-provided facilities for indexing so we can add the news articles into the index. To search, we will use Lucene's searcher to provide search functionality against the index. Now, let's have a quick overview of Lucene's way of managing information.

### How Lucene works

Continuing our news search application, let's assume we fetched some news bits from a custom source. The following shows the two news items that we are going to add to our index:

**News Item – 1**

"Title": "Europe stocks tumble on political fears, PMI data",
"DOP": "30/2/2012 00:00:00",
"Content": "LONDON (MarketWatch) - European stock markets tumbled to a three-month low on Monday, driven by steep losses for banks and resource firms after weak purchasing-managers index readings from China and Europe. At the same time, political tensions in France and the Netherlands fueled fears of further euro-zone turmoil",

**News Item – 2**

"Title": "Dow Rises, Gains 1.5% on Week",
"DOP": "3/3/2012 00:00:00",
"Content": "Solid quarterly results from consumer-oriented stocks including Amazon.com"
AMZN +15.75% overshadowed data on slowing economic growth, pushing benchmarks to their biggest weekly advance since mid-March.

For each news bit, we have a title, publishing date, content, and link, which are the constituents of the typical information in a news article. We will treat each news item as a document and add it to our news data store. The act of adding documents to the data store is called **indexing** and the data store itself is called an index. Once the index is created, you can query it to locate documents by search terms, and this is what's referred to as searching the index.

So, how does Lucene maintain an index, and how's an index being leveraged in terms of search? We can think of a scenario where you look for a certain subject from a book. Let's say you are interested in **Object Oriented Programming (OOP)** and learning more about **inheritance**. You then get a book on OOP and start looking for the relevant information about inheritance. You can start from the beginning of the book; start reading until you land on the inheritance topic. If the relevant topic is at the end of the book, it will certainly take a while to reach. As you may notice, this is not a very efficient way to locate information. To locate information quickly in a book, especially a reference book, you can usually rely on the index where you will find the key value pairs of the keyword, and page numbers sorted alphabetically by the keyword. Here, you can look for the word, inheritance, and go to the related pages immediately without scanning through the entire book. This is a more efficient and standard method to quickly locate the relevant information. This is also how Lucene works behind the scene, though with more sophisticated algorithms that make searching efficient and flexible.

Internally, Lucene assigns a unique **document ID** (called **DocId**) to each document when they are added to an index. DocId is used to quickly return details of a document in search results. The following is an example of how Lucene maintains an index. Assuming we start a new index and add three documents as follows:

- Document id 1: Lucene
- Document id 2: Lucene and Solr
- Document id 3: Solr extends Lucene

Lucene indexes these documents by tokenizing the phrases into keywords and putting them into an inverted index. Lucene's inverted index is a reverse mapping lookup between keyword and DocId. Within the index, keywords are stored in sorted and DocIds are associated with each keyword. Matches in keywords can bring up associated DocIds to return the matching documents. This is a simplistic view of how Lucene maintains an index and how it should give you a basic idea of the schematic of Lucene's architecture.
The following is an example of an inverted index table for our current sample data:

<table>
<thead>
<tr>
<th>Lucene Inverted Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
</tr>
<tr>
<td>extends</td>
</tr>
<tr>
<td>Lucene</td>
</tr>
<tr>
<td>Solr</td>
</tr>
</tbody>
</table>

As you notice, the inverted index is designed to optimally answer such queries: get me all documents with the term *xyz*. This data structure allows for a very fast full-text search to locate the relevant documents. For example, a user searches for the term *Solr*. Lucene can quickly locate *Solr* in the inverted index, because it's sorted, and return DocId 2 and DocId 3 as the result. Then, the search can proceed to quickly retrieve the relevant documents by these DocIds. To a great extent, this architecture contributes to Lucene's speed and efficiency. As you continue to read through this book, you will see Lucene’s many techniques to find the relevant information and how you can customize it to suit your needs.

One of the many Lucene features worth noting is text analysis. It’s an important feature because it provides extensibility and gives you an opportunity to massage data into a standard format before feeding the data into an index. It’s analogous to the transform layer in an Extract Transform Load (ETL) process. An example of its typical use is the removal of stop words. These are common words (for example, is, and, the, and so on) of little or no value in search. For an even more flexible search application, we can also use this analyzing layer to turn all keywords into lowercase, in order to perform a case-insensitive search. There are many more analyses you can do with this framework; we will show you the best practices and pitfalls to help you make a decision when customizing your search application.

**Why is Lucene so popular?**

A quick overview of Lucene’s features is as follows:

- Index at about 150GB of data per hour on modern hardware
- Efficient RAM utilization (only 1 MB heap)
- Customizable ranking models
- Supports numerous query types
- Restrictive search (routed to specific fields)
- Sorting by fields
Introducing Lucene

- Real-time indexing and searching
- Faceting, Grouping, Highlighting, and so on
- Suggestions

Lucene makes the most out of the modern hardware, as it is very fast and efficient. Indexing 20 GB of textual content typically produces an index size in the range of 4-6 GB. Lucene's speed and low RAM requirement is indicative of its efficiency. Its extensibility in text analysis and search will allow you to virtually customize a search engine in any way you want.

It is becoming more apparent that there are quite a few big companies using Lucene for their search applications. The list of Lucene users is growing at a steady pace. You can take a look at the list of companies and websites that use Lucene on Lucene's wiki page. More and more data giants are using Lucene nowadays: Netflix, Twitter, MySpace, LinkedIn, FedEx, Apple, Ticketmaster, www.Salesforce.com, Encyclopedia Britannica CD-ROM/DVD, Eclipse IDE, Mayo Clinic, New Scientist magazine, Atlassian (JIRA), Epiphany, MIT's OpenCourseWare and DSpace, HathiTrust digital library, and Akamai's EdgeComputing platform, all come under this list. This wide range of implementations illustrates that Lucene is a stand-out piece of search technology that's trusted by many.

Lucene's wiki page is available at http://wiki.apache.org/lucene-java/FrontPage

Some Lucene implementations

The popularity of Lucene has driven many ports into other languages and environments. Apache Solr and Elastic search have revolutionized search technology, and both of them are built on top of Lucene.

The following are the various implementations of Lucene in different languages:

- **CLucene**: Lucene implementation in C++ (http://sourceforge.net/projects/clucene/)
- **Lucene.Net**: Lucene implementation in Microsoft.NET (http://incubator.apache.org/lucene.net/)
- **Lucene4c**: Lucene implementation in C (http://incubator.apache.org/lucene4c/)
- **LuceneKit**: Lucene implementation in Objective-C, Cocoa/GNUstep support (https://github.com/tcurdt/lucenekit)
- **Lupy**: Lucene implementation in Python (RETIRED) (http://www.divmod.org/projects/lupy)
Chapter 1

- **NLucene**: This is another Lucene implementation in .NET (out of date) (http://sourceforge.net/projects/nlucene/)
- **Plucene**: Lucene implementation in Perl (http://search.cpan.org/search?query=plucene&mode=all)
- **KinoSearch**: This is a new Lucene implementation in Perl (http://www.rectangular.com/kinosearch/)
- **PyLucene**: This is GCJ-compiled version of Java Lucene integrated with Python (http://pylucene.osafoundation.org/)
- **MUTIS**: Lucene implementation in Delphi (http://mutis.sourceforge.net/)
- **Ferret**: Lucene implementation in Ruby (http://ferret.davebalmain.com/trac/)
- **Montezuma**: Lucene implementation in Common Lisp (http://www.cliki.net/Montezuma)

### Installing Lucene

This section will show you what you need, in order to get started with Lucene.

#### How to do it...

First, let's download Lucene. Apache Lucene can be downloaded from its official download page. As of now, the latest version of Lucene is 4.10. Here is the link to the official page of Lucene: [http://lucene.apache.org/core/](http://lucene.apache.org/core/)

The **Download** button will take you to all available Apache mirrors where you can download Lucene.

Lucene's index contains `{lucene version}.zip` or `{lucene version}.tar.gz`, including the Lucene core library, HTML documentation, and demo application. Meanwhile, `{lucene version-src}.zip` or `{lucene version-src}.tar.gz` contains the source code for that particular version.
Introducing Lucene

The following is a sample of what the download page looks like:

<table>
<thead>
<tr>
<th>Name</th>
<th>Last modified</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Directory</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>changes/</td>
<td>29-Oct-2014</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>lucene-4.10.2-src.zip</td>
<td>29-Oct-2014</td>
<td>2B</td>
<td></td>
</tr>
<tr>
<td>lucene-4.10.2.tar.gz</td>
<td>29-Oct-2014</td>
<td>62M</td>
<td></td>
</tr>
<tr>
<td>lucene-4.10.2.zip</td>
<td>29-Oct-2014</td>
<td>72M</td>
<td></td>
</tr>
</tbody>
</table>

How it works...

Lucene is written entirely in Java. The prerequisite for running Lucene is Java Runtime Environment. Lucene runs on Java 6 or higher. If you use Java 7, make sure you install update 1 as well. Once your download is complete, you can extract the contents to a directory and you are good to go. In case you get some errors, the links to the FAQ and mailing list of Lucene users is as follows:

Mailing List:

http://lucene.apache.org/core/discussion.html

FAQ:

http://wiki.apache.org/lucene-java/LuceneFAQ

Setting up a simple Java Lucene project

Having Lucene downloaded, we can get started working with Lucene. Let’s take a look at how a Lucene project is set up.

Getting ready

Java Runtime is required. If you have not installed Java yet, visit Oracle’s website to download Java. Here is a link to the Java download page: http://www.oracle.com/technetwork/java/javase/downloads/index.html.

You may also want to use an IDE to work on a Lucene project. Many users prefer Eclipse IDE, but you are free to make your own choice, such as NetBeans, or whatever you prefer. If you want to give Eclipse IDE a try, you can refer to the following link: https://www.eclipse.org/downloads/.
Having set up a development environment, let's proceed to create our first Lucene project.

**How to do it...**

Thanks to the well-organized and efficient architecture of Lucene, the Lucene core JAR is only 2.4 MB in size. The core library provides the basic functionality to start a Lucene project. By adding it to your Java classpath, you can begin to build a powerful search engine. We will show you a couple ways to do this in Eclipse.

First, we will set up a normal Java project in Eclipse. Then, we will add Lucene libraries to the project. To do so, follow these steps:

- Click on the **Project** dropdown
- Click on **Properties**
- Select **Java Build Path** in left side of navigation
- Click on the **Libraries** tab
- Click on **Add External JARs...**
- Browse to the folder where lucene-core is located, and then select the core JAR file
- Also, add **lucene-analyzers-common** and **lucene-queryparser**:

![Image showing Eclipse Java Build Path settings](image-url)
Introducing Lucene

Another way to include the Lucene library is via Apache Maven. Maven is a project management and build tool that provides facilities to manage project development lifecycles. A detailed explanation of Maven is beyond the scope of this book. If you want to know more about Maven, you can check out the following link: http://maven.apache.org/what-is-maven.html.

To set up Lucene in Maven, you need to insert its dependency information into your project's pom.xml. You can visit a Maven repository (http://mvnrepository.com/artifact/org.apache.lucene) where you can find Lucene.

```xml
<dependency>
  <groupId>org.apache.lucene</groupId>
  <artifactId>lucene-core</artifactId>
  <version>4.10.2</version>
</dependency>
```

After you have updated pom.xml, the Lucene library will show up in your project after your IDE sync up dependencies.

**How it works...**

Once the JAR files are made available to the classpath, you can go ahead and start writing code. Both methods we described here would provide access to the Lucene library. The first method adds JARs directly. With Maven, when you add dependency to pom.xml, the JARs will be downloaded automatically.
Obtaining an IndexWriter

The IndexWriter class provides functionality to create and manage index. The class can be found in Lucene-core. It handles basic operations where you can add, delete, and update documents. It also handles more complex use cases that we will cover during the course of this book.

An IndexWriter constructor takes two arguments:

```java
IndexWriter(directory, config)
```

Construct a new `IndexWriter` as per the settings given in the `conf` file.

The first argument is a Directory object. Directory is a location where the Lucene index is stored. The second argument is an `IndexWriterConfig` object where it holds the configuration information. Lucene provides a number of directory implementations. For performance or quick prototyping, we can use `RAMDirectory` to store the index entirely in memory. Otherwise, the index is typically stored in `FSDirectory` on a file system. Lucene has several FSDirectory implementations that have different strengths and weaknesses depending on your hardware and environment. In most cases, we should let Lucene decide which implementation to use by calling `FSDirectory.open(File path)`.

How to do it...

We need to first define an analyzer to initialize `IndexWriterConfig`. Then, a Directory should be created to tell Lucene where to store the index. With these two objects defined, we are ready to instantiate an `IndexWriter`.

The following is a code snippet that shows you how to obtain an `IndexWriter`:

```java
Analyzer analyzer = new WhitespaceAnalyzer();
Directory directory = new RAMDirectory();
IndexWriterConfig config = new
   IndexWriterConfig(Version.LATEST, analyzer);
IndexWriter indexWriter = new IndexWriter(directory, config);
```
Introducing Lucene

How it works...

First, we instantiate a WhitespaceAnalyzer to parse input text, and tokenize text into word tokens. Then, we create an in-memory Directory by instantiating RAMDirectory. We configure IndexWriterConfig with the WhitespaceAnalyzer we just created and finally, we pass both Directory and IndexWriterConfig to create an IndexWriter. The IndexWriter is now ready to update the index.

An IndexWriter consists of two major components, directory and analyzer. These are necessary so that Lucene knows where to persist indexing information and what treatment to apply to the documents before they are indexed. Analyzer's treatment is especially important because it maintains data consistency. If an index already exists in the specified directory, Lucene will update the existing index. Otherwise, a new index is created.

Creating an analyzer

Analyzer's job is to analyse text. It enforces configured policies (IndexWriterConfig) on how index terms are extracted and tokenized from a raw text input. The output from Analyzer is a set of indexable tokens ready to be processed by the indexer. This step is necessary to ensure consistency in both the data store and search functionality. Also, note that Lucene only accepts plain text. Whatever your data type might be—be it XML, HTML, or PDF, you need to parse these documents into text before tossing them over to Lucene.

Imagine you have this piece of text: Lucene is an information retrieval library written in Java. An analyzer will tokenize this text, manipulate the data to conform to a certain data formatting policy (for example, turn to lowercase, remove stop words, and so on), and eventually output as a set of tokens. Token is a basic element in Lucene's indexing process. Let's take a look at the tokens generated by an analyzer for the above text:

{Lucene} {is} {an} {Information} {Retrieval} {library} {written} {in} {Java}

Each individual unit enclosed in braces is referred to as a token. In this example, we are leveraging WhitespaceAnalyzer to analyze text. This specific analyzer uses whitespace as a delimiter to separate the text into individual words. Note that the separated words are unaltered and stop words (is, an, in) are included. Essentially, every single word is extracted as a token.

Getting ready

The Lucene-analyzers-common module contains all the major components we discussed in this section. Most commonly-used analyzers can be found in the org.apache.lucene.analysis.core package. For language-specific analysis, you can refer to the org.apache.lucene.analysis{language code} packages.
How to do it...

Many analyzers in Lucene-analyzers-common require little or no configuration, so instantiating them is almost effortless. For our current exercise, we will instantiate the WhitespaceAnalyzer by simply using new object:

Analyzer analyzer = new WhitespaceAnalyzer();

How it works...

An analyzer is a wrapper of three major components:

- Character filter
- Tokenizer
- Token filter

The analysis phase includes pre- and post-tokenization functions, and this is where the character filter and token filter come into play. The character filter preprocesses text before tokenization to clean up data such as stripping out HTML markups, removing user-defined patterns, and converting a special character or specific text. The token filter executes the post tokenization filtering; its operations involve various kinds of manipulations. For instance, stemming, stop word filtering, text normalization, and synonym expansion are all part of token filter. As described earlier, the tokenizer splits up text into tokens. The output of these analysis processes is TokenStream where the indexing process can consume and produce an index.

Lucene provides a number of standard analyzer implementations that should fit most of the search applications. Here are some additional analyzers, which we haven’t talked about yet:

- **StopAnalyzer**: This is built with a LowerCaseTokenizer and StopWordFilter. As the names suggest, this analyzer lowercases text, tokenizes non-letter characters and removes stop words.

- **SimpleAnalyzer**: This is built with a LowerCaseTokenizer so that it simply splits text at non-letter characters, and lowercases the tokens.

- **StandardAnalyzer**: This is slightly more complex than SimpleAnalyzer. It consists of StandardTokenizer, StandardFilter, LowerCaseFilter, and StopWordFilter. StandardTokenizer uses a grammar-based tokenization technique that’s applicable for most European languages. StandardFilter normalizes tokens extracted with StandardTokenizer. Then, we have the familiar LowerCaseFilter and StopWordFilter.

- **SnowballAnalyzer**: This is the most featured of the bunch. It’s made up of StandardTokenizer with StandardFilter, LowerCaseFilter, StopFilter, and SnowballFilter. SnowballFilter stems words, so this analyzer is essentially StandardAnalyzer plus stemming. In simple terms, stemming is a technique to reduce words to their word stem or root form. By reducing words, we can easily find matches of words with the same meaning, but in different forms such as plural and singular forms.
Creating fields

We have learned that indexing information in Lucene requires the creation of document objects. A Lucene document contains one or more field where each one represents a single data point about the document. A field can be a title, description, article ID, and so on. In this section, we will show you the basic structure and how to create a field.

A Lucene field has three attributes:

- Name
- Type
- Value

Name and value are self-explanatory. You can think of a name as a column name in a table, and value as a value in one of the records where record itself is a document. Type determines how the field is treated. You can set FieldType to control whether to store value, to index it or even tokenize text. A Lucene field can hold the following:

- String
- Reader or preanalyzed TokenStream
- Binary(byte[])
- Numeric value

How to do it...

This code snippet shows you how to create a simple TextField:

```java
Document doc = new Document();
String text = "Lucene is an Information Retrieval library written in Java.";
doc.add(new TextField("fieldname", text, Field.Store.YES));
```

How It Works

In this scenario, we create a document object, initialize a text, and add a field by creating a TextField object. We also configure the field to store a value so it can be retrieved during a search.

A Lucene document is a collection of field objects. A field is the name of the value pairs, which you may add to the document. A field is created by simply instantiating one of the Field classes. Field can be inserted into a document via the add method.
Creating and writing documents to an index

This recipe shows you how to index a document. In fact, here we are putting together all that we learned so far from the previous recipes. Let's see how it is done.

How to do it...

The following code sample shows you an example of adding a simple document to an index:

```java
public class LuceneTest {
    public static void main(String[] args) throws IOException {
        Analyzer analyzer = new WhitespaceAnalyzer();
        Directory directory = new RAMDirectory();
        IndexWriterConfig config = new
            IndexWriterConfig(Version.LATEST, analyzer);
        IndexWriter indexWriter = new IndexWriter(directory,
            config);
        Document doc = new Document();
        String text = "Lucene is an Information Retrieval library
            written in Java";
        doc.add(new TextField("fieldname", text, Field.Store.YES));
        indexWriter.addDocument(doc);
        indexWriter.close();
    }
}
```

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

How it works...

Note that the preceding code snippet combined all the sample codes we learned so far. It first initializes an analyzer, directory, IndexWriterConfig, and IndexWriter. Once the IndexWriter is obtained, a new Document is created with a custom TextField. The Document is then added to IndexWriter. Also, note that we call `indexWriter.close()` at the end. Calling this method will commit all changes and close the index.

The `IndexWriter` class exposes an `addDocument(doc)` method that allows you to add documents to an index. `IndexWriter` will write to an index specified by directory.
Deleting documents

We have learned how documents are added to an index. Now, we will see how to delete Documents. Suppose you want to keep your index up to date by deleting documents that are a week old. All of a sudden, the ability to remove documents becomes a very important feature. Let’s see how we can do that.

How to do it...

IndexWriter provides the interface to delete documents from an index. It takes either term or query as argument, and will delete all the documents matching these arguments:

- deleteDocuments(Term)
- deleteDocuments(Term... terms)
- deleteDocuments(Query)
- deleteDocuments(Query... queries)
- deleteAll()

Here is a code snippet on how deleteDocuments is called:

```java
indexWriter.deleteDocuments(new Term("id", "1"));
indexWriter.close();
```

How it works...

Assuming IndexWriter is already instantiated, this code will trigger IndexWriter to delete all the documents that contain the term id where the value equals 1. Then, we call close to commit changes and close the IndexWriting. Note that this is a match to a Field called id; it’s not the same as DocId.

In fact, deletions do not happen at once. They are kept in the memory buffer and later flushed to the directory. The documents are initially marked as deleted on disk so subsequent searches will simply skip the deleted documents; however, to free the memory, you still need to wait. We will see the underlying process in detail in due course.
Obtaining an IndexSearcher

Having reviewed the indexing cycle in Lucene, let's now turn our attention towards search. Keep in mind that indexing is a necessary evil you have to go through to make your text searchable. We take all the pain to customize a search engine now, so we can obtain good search experiences for the users. This will be well worth the effort when users can find information quickly and seamlessly. A well-tuned search engine is the key to every search application.

Consider a simple search scenario where we have an index built already. User is doing research on Lucene and wants to find all Lucene-related documents. Naturally, the term Lucene will be used in a search query. Note that Lucene leverages an inverted index (see the preceding image). Lucene can now locate documents quickly by stepping into the term Lucene in the index, and returning all the related documents by their DocIds. A term in Lucene contains two elements—the value and field in which the term occurs.

How do we specifically perform a search? We create a Query object. In simple terms, a query can be thought of as the communication with an index. This action is also referred to as querying an index. We issue a query to an index and get matched documents back.

The IndexSearcher class is the gateway to search an index as far as Lucene is concerned. An IndexSearcher takes an IndexReader object and performs a search via the reader. IndexReader talks to the index physically and returns the results. IndexSearcher executes a search by accepting a query object. Next, we will learn how to perform a search and create a Query object with a QueryParser. For now, let's take a look at how we can obtain an IndexSearcher.

How to do it...

Here is a code snippet that shows you how to obtain an IndexSearcher:

```java
Directory directory = getDirectory();
IndexReader indexReader = DirectoryReader.open(directory);
IndexSearcher indexSearcher = new IndexSearcher(indexReader);
```

How it works...

The first line assumes we can gain access to a Directory object by calling `getDirectory()`. Then, we obtain an IndexReader by calling `DirectoryReader.open(directory)`. The open method in `DirectoryReader` is a static method that opens an index to read, which is analogous to `IndexWriter` opening a directory to write. With an IndexReader initialized, we can instantiate an IndexSearcher with the reader.
Introducing Lucene

Creating queries with the Lucene QueryParser

Now, we understand that we need to create Query objects to perform a search. We will look at QueryParser and show you how it's done. Lucene supports a powerful query engine that allows for a wide range of query types. You can use search modifier or operator to tell Lucene how matches are done. You can also use fuzzy search and wild card matching.

Internally, Lucene processes Query objects to execute a search. QueryParser is an interpreter that parses a query string into Query objects. It provides the utility to convert textual input into objects. The key method in QueryParser is parse (String). If you want more control over how a search is performed, you can create Query objects directly without using QueryParser, but this would be a much more complicated process. The query string syntax Lucene uses has a few rules. Here is an excerpt from Lucene's Javadoc: https://lucene.apache.org/core/4_10_0/queryparser/org/apache/lucene/queryparser/classic/QueryParser.html.

The syntax for query strings is as follows: a Query is a series of clauses. A clause can be prefixed by:

- A plus (+) or minus (-) sign, indicating that the clause is required or prohibited, respectively.
- Alternatively, a term followed by a colon, indicating the field to be searched. This enables us to construct queries that search multiple fields.

A clause can be:

- A term, indicating all the documents that contain this term.
- Alternatively, a nested query, enclosed in parentheses. Note that this can be used with a+/ prefix to require any of the set of terms.

Thus, in BNF, the query grammar is:

```
Query ::= ( Clause )* 
Clause ::= ["+", "-" ] [TERM] "::" [TERM] | "(" Query ")" 
```

Note that you need to import lucene-queryparser package to use QueryParser. It is not a part of the lucene-core package.

The Backus Normal Form (BNF) is a notation technique to specify syntax for a language, and is often used in computer science.
Chapter 1

How to do it...

Here is a code snippet:

```java
QueryParser parser = new QueryParser("Content", analyzer);
Query query = parser.parse("Lucene");
```

How it works...

Assuming an analyzer is already declared and available as a variable, we pass it into QueryParser to initialize the parser. The second parameter is the name of the field where we will perform a search. In this case, we are searching a field called `Content`. Then, we call `parse(String)` to interpret the search string Lucene into Query object. Note that, at this point, we only return a Query object. We have not actually executed a search yet.

Performing a search

Now that we have a Query object, we are ready to execute a search. We will leverage IndexSearcher from two recipes ago to perform a search.

Note that, by default, Lucene sorts results based on relevance. It has a scoring mechanism assigning a score to every matching document. This score is responsible for the sort order in search results. A score can be affected by the rules defined in the query string (for example, must match, AND operation, and so on). It can also be altered programmatically. We have set aside a chapter to explore the concept of scoring and how we can leverage it to customize a search engine.

How to do it...

Here is what we learned so far and put together into an executable program:

```java
public class LuceneTest {
    public static void main(String[] args) throws IOException,
        ParseException {
        Analyzer analyzer = new StandardAnalyzer();
        Directory directory = new RAMDirectory();
        IndexWriterConfig config = new
            IndexWriterConfig(Version.LATEST, analyzer);
        IndexWriter indexWriter = new IndexWriter(directory,
            config);
        Document doc = new Document();
        String text = "Lucene is an Information Retrieval library
            written in Java";
```
Introducing Lucene

doc.add(new TextField("Content", text, Field.Store.YES));
indexWriter.addDocument(doc);
indexWriter.close();
IndexReader indexReader = DirectoryReader.open(directory);
IndexSearcher indexSearcher = new
IndexSearcher(indexReader);
QueryParser parser = new QueryParser( "Content",
analyzer);
Query query = parser.parse("Lucene");
int hitsPerPage = 10;
TopDocs docs = indexSearcher.search(query, hitsPerPage);
ScoreDoc[] hits = docs.scoreDocs;
int end = Math.min(docs.totalHits, hitsPerPage);
System.out.print("Total Hits: " + docs.totalHits);
System.out.print("Results: ");
for (int i = 0; i < end; i++) {
    Document d = indexSearcher.doc(hits[i].doc);
    System.out.println("Content: " + d.get("Content"));
}

How it works...

The preceding code sets up a StandardAnalyzer to analyze text, uses a RAMDirectory
for index store, configures IndexWriter to put a piece of content into the index, and uses
QueryParser to generate Query object, in order to perform a search. It also has a sample
code that shows how to retrieve search results from TopDocs by displaying total hits,
and shows matching documents by DocId.

Here is a diagram showing how the search portion works between components:
A search string enters into `QueryParser.parse(String)`. `QueryParser` then uses an analyzer to process the search string to produce a set of tokens. The tokens are then mapped to the Query object, and get sent to `IndexSearcher` to execute a search. The search result returned by `IndexSearcher` is a TopDocs object where it contains statistics of total matches and DocIds of the matching documents.

Note that it is preferable to use the same analyzer for both indexing and searching to get the best results.

### Enumerating results

We have already previewed how the results are enumerated from the previous sample code. You might have noticed that the major component in search results is TopDocs. Now, we will show you how to leverage this object to paginate results. Lucene does not provide pagination functionality, but we can still build pagination easily using what’s available in TopDocs.

### How to do it...

Here is a sample implementation on pagination:

```java
public List<Document> getPage(int from, int size) {
    List<Document> documents = new ArrayList<Document>();
    Query query = parser.parse(searchTerm);
    TopDocs hits = searcher.search(query, maxNumberOfResults);
    int end = Math.min(hits.totalHits, size);
    for (int i = from; i < end; i++) {
        int docId = hits.scoreDocs[i].doc;
        // load the document
        Document doc = searcher.doc(docId);
        documents.add(doc);
    }
    return documents;
}
```
Introducing Lucene

How it works...

When we perform search in Lucene, actual results are not preloaded immediately. In TopDocs, we only get back an array of ranked pointers. It's called ranked pointers because they are not actual documents, but a list of references (DocId). By default, results are scored by the scoring mechanism. We will see more about scoring in detail in Introduction section Chapter 7, Flexible Scoring. For paging, we can calculate position offset, apply pagination ourselves, and leverage something like what we showed in the sample code to return results by page. Developers at Lucene actually recommend re-executing a search on every page, instead of storing the initial search results (refer to http://wiki.apache.org/lucene-java/LuceneFAQ#How_do_I IMPLEMENT Paging_i.e._Showing_result_from_1-10_2C_11-20_etc.3F). The reasoning is that people are usually only interested in top results and they are confident in Lucene's performance.

This code assumes that parser (QueryParser), searcher (IndexSearcher), and maxNumberOfResults are already initialized. Note that this sample is for illustrative purpose only and it's not optimized.
**Where to buy this book**

You can buy Lucene 4 Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

[Click here](#) for ordering and shipping details.