Covers Apache Lucene 3.0

Lucene IN ACTION

SECOND EDITION

Michael McCandless
Erik Hatcher
Otis Gospodnetić

FOREWORD BY DOUG CUTTING

SAMPLE CHAPTER

MANNING

Lucene in Action, Second Edition

by Michael McCandless,
Erik Hatcher, and Otis Gospodnetić

Chapter 1

brief contents

1 Meet Lucene

This chapter covers

- Learning about Lucene
- Understanding the typical search application architecture
- Using the basic indexing API
- Working with the search API

Lucene is a powerful Java search library that lets you easily add search to any application. In recent years Lucene has become exceptionally popular and is now the most widely used information retrieval library: it powers the search features behind many websites and desktop applications. Although it's written in Java, thanks to its popularity and the determination of zealous developers you now have at your disposal a number of ports or integrations to other programming languages (C/C++, C#, Ruby, Perl, Python, and PHP, among others).

One of the key factors behind Lucene's popularity is its simplicity, but don't let that fool you: under the hood sophisticated, state-of-the-art information retrieval techniques are quietly at work. The careful exposure of its indexing and searching API is a sign of the well-designed software. You don't need in-depth knowledge about how Lucene's information indexing and retrieval work in order to start using it. Moreover, Lucene's straightforward API requires using only a handful of classes to get started. Finally, for those of you tired of bloatware, Lucene's core JAR is refreshingly tiny—only 1 MB—and it has no dependencies!

In this chapter we cover the overall architecture of a typical search application and where Lucene fits. It's crucial to recognize that Lucene is simply a search library, and you'll need to handle the other components of a search application (crawling, document filtering, runtime server, user interface, administration, etc.) as your application requires. We show you how to perform basic indexing and searching with ready-to-use code examples. We then briefly introduce all the core elements you need to know for both of these processes. We start with the modern problem of information explosion, to understand why we need powerful search functionality in the first place.

NOTE Lucene is an active open source project. By the time you read this, likely Lucene's APIs and features will have changed. This book is based on the 3.0.1 release of Lucene, and thanks to Lucene's backward compatibility policy, all code samples should compile and run fine for future 3.x releases. If you encounter a problem, send an email to java-user@lucene.apache.org and Lucene's large, passionate, and responsive community will surely help.

1.1 Dealing with information explosion

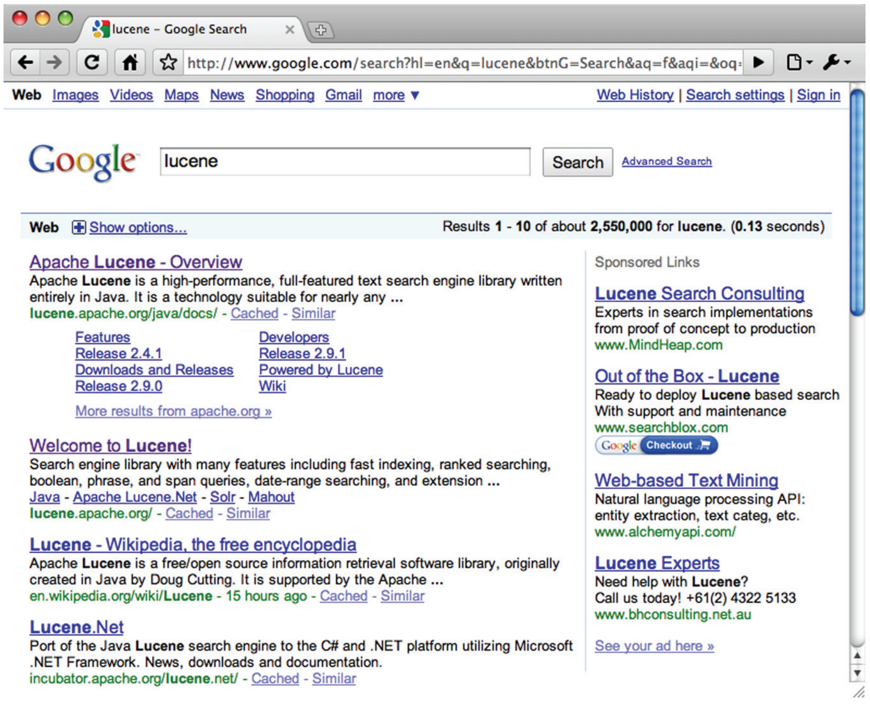
To make sense of the perceived complexity of the world, humans have invented categorizations, classifications, genuses, species, and other types of hierarchical organizational schemes. The Dewey decimal system for categorizing items in a library collection is a classic example of a hierarchical categorization scheme.

The explosion of the internet and digital repositories has brought large amounts of information within our reach. With time, the amount of data available has become so vast that we need alternate, more dynamic ways of finding information (see figure 1.1). Although we can classify data, trawling through hundreds or thousands of categories and subcategories of data is no longer an efficient method for finding information.

The need to quickly locate certain information out of the sea of data isn't limited to the internet realm—desktop computers store increasingly more data on multi-terabyte hard drives. Changing directories and expanding and collapsing hierarchies of folders isn't an effective way to access stored documents. Furthermore, we no longer use computers only for their raw computing abilities: they also serve as communication devices, multimedia players, and media storage devices. Those uses require the ability to quickly find a specific piece of data; what's more, we need to make rich media—such as images, video, and audio files in various formats—easy to locate.

With this abundance of information, and with time one of the most precious commodities for most people, we must be able to make flexible, free-form, ad hoc queries that can quickly cut across rigid category boundaries and find exactly what we're after while requiring the least effort possible.

To illustrate the pervasiveness of searching across the internet and the desktop, figure 1.1 shows a search for *lucene* at Google. Figure 1.2 shows the Apple Mac OS X



lucene – Google Search

http://www.google.com/search?hl=en&q=lucene&btnG=Search&aq=f&aqi=&oq=

Web Images Videos Maps News Shopping Gmail more ▼ Web History | Search settings | Sign in

Google lucene Search Advanced Search

Web Show options... Results **1** - **10** of about **2,550,000** for **lucene**. (**0.13** seconds)

Apache Lucene - Overview
Apache **Lucene** is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any ...
lucene.apache.org/java/docs/ - Cached - Similar

Features	Developers
Release 2.4.1	Release 2.9.1
Downloads and Releases	Powered by Lucene
Release 2.9.0	Wiki

More results from apache.org »

Welcome to Lucene!
Search engine library with many features including fast indexing, ranked searching, boolean, phrase, and span queries, date-range searching, and extension ...
Java - Apache Lucene.Net - Solr - Mahout
lucene.apache.org/ - Cached - Similar

Lucene - Wikipedia, the free encyclopedia
Apache **Lucene** is a free/open source information retrieval software library, originally created in Java by Doug Cutting. It is supported by the Apache ...
en.wikipedia.org/wiki/**Lucene** - 15 hours ago - Cached - Similar

Lucene.Net
Port of the Java **Lucene** search engine to the C# and .NET platform utilizing Microsoft .NET Framework. News, downloads and documentation.
incubator.apache.org/**lucene**.net/ - Cached - Similar

Sponsored Links

Lucene Search Consulting
Experts in search implementations from proof of concept to production
www.MindHeap.com

Out of the Box - Lucene
Ready to deploy **Lucene** based search With support and maintenance
www.searchblox.com
Google Checkout

Web-based Text Mining
Natural language processing API: entity extraction, text categ, etc.
www.alchemyapi.com/

Lucene Experts
Need help with **Lucene**?
Call us today! +61(2) 4322 5133
www.bhconsulting.net.au

See your ad here »

Figure 1.1 Searching the internet with Google

Finder (the counterpart to Microsoft's Explorer on Windows) and the search feature embedded at the upper right. The Mac OS X music player, iTunes, also has embedded search capabilities, as shown in figure 1.3.

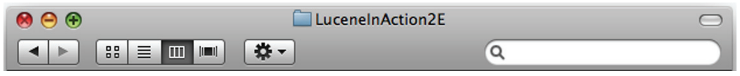


Figure 1.2 Mac OS X Finder with its embedded search capability

Search is needed everywhere! All major operating systems have embedded searching. The Spotlight feature in Mac OS X integrates indexing and searching across all file types, including rich metadata specific to each type of file, such as emails, contacts, and more.[1]

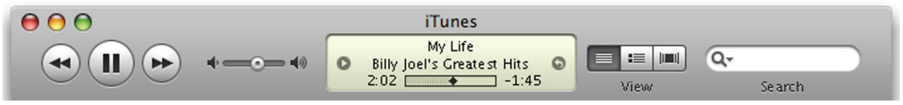


Figure 1.3 Apple's iTunes intuitively embeds search functionality.

[1] Erik and Mike freely admit to fondness of all things Apple.

Different people are fighting the same problem—information overload—using different approaches. Some have been working on novel user interfaces, some on intelligent agents, and others on developing sophisticated search tools and libraries like Lucene. Before we jump into action with code samples, we'll give you a high-level picture of what Lucene is, what it isn't, and how it came to be.

1.2 What is Lucene?

Lucene is a high-performance, scalable information retrieval (IR) library. IR refers to the process of searching for documents, information within documents, or metadata about documents. Lucene lets you add searching capabilities to your applications. It's a mature, free, open source project implemented in Java, and a project in the Apache Software Foundation, licensed under the liberal Apache Software License. As such, Lucene is currently, and has been for quite a few years, the most popular free IR library.

NOTE Throughout the book, we'll use the term *information retrieval* (or its acronym *IR*) to describe search tools like Lucene. People often refer to IR libraries as *search engines*, but you shouldn't confuse IR libraries with web search engines.

As you'll soon discover, Lucene provides a simple yet powerful core API that requires minimal understanding of full-text indexing and searching. You need to learn about only a handful of its classes in order to start integrating Lucene into an application. Because Lucene is a Java library, it doesn't make assumptions about what it indexes and searches, which gives it an advantage over a number of other search applications. Its design is compact and simple, allowing Lucene to be easily embedded into desktop applications.

Beyond Lucene's core JAR are a number of extensions modules that offer useful add-on functionality. Some of these are vital to almost all applications, like the spellchecker and highlighter modules. These modules are housed in a separate area called contrib, and you'll see us referring to such contrib modules throughout the book. There are so many modules that we have two chapters, 8 and 9, to cover them!

Lucene's website, at http://lucene.apache.org/java, is a great place to learn more about the current status of Lucene. There you'll find the tutorial, Javadocs for Lucene's API for all recent releases, an issue-tracking system, links for downloading releases, and Lucene's wiki (http://wiki.apache.org/lucene-java), which contains many community-created and -maintained pages.

You've probably used Lucene without knowing it! Lucene is used in a surprisingly diverse and growing number of places: NetFlix, Digg, MySpace, LinkedIn, Fedex, Apple, Ticketmaster, SalesForce.com, the Encyclopedia Britannica CD-ROM/DVD, the Eclipse IDE, the Mayo Clinic, *New Scientist* magazine, Atlassian (JIRA), Epiphany, MIT's OpenCourseWare and DSpace, the Hathi Trust Digital Library, and Akamai's Edge-Computing platform. Your name may be on this list soon, too! The "powered by" Lucene page on Lucene's wiki has even more examples.

1.2.1 What Lucene can do

People new to Lucene often mistake it for a ready-to-use application like a file-search program, a web crawler, or a website search engine. That isn't what Lucene is: Lucene is a software library, a toolkit if you will, not a full-featured search application. It concerns itself with text indexing and searching, and it does those things very well. Lucene lets your application deal with business rules specific to its problem domain while hiding the complexity of indexing and searching behind a simple-to-use API. Lucene is the core that the application wraps around.

A number of full-featured search applications have been built on top of Lucene. If you're looking for something prebuilt or a framework for crawling, document handling, and searching, the "powered by" page on Lucene's wiki lists some of these options.

Lucene allows you to add search capabilities to your application. Lucene can index and make searchable any data that you can extract text from. Lucene doesn't care about the source of the data, its format, or even its language, as long as you can derive text from it. This means you can index and search data stored in files: web pages on remote web servers, documents stored in local file systems, simple text files, Microsoft Word documents, XML or HTML or PDF files, or any other format from which you can extract textual information.

Similarly, with Lucene's help you can index data stored in your databases, giving your users rich, full-text search capabilities that many databases provide only on a limited basis. Once you integrate Lucene, users of your applications can perform searches by entering queries like `+George +Rice -eat -pudding`, `Apple -pie +Tiger`, `animal:monkey AND food:banana`, and so on. With Lucene, you can index and search email messages, mailing-list archives, instant messenger chats, your wiki pages…the list goes on. Let's recap Lucene's history.

1.2.2 History of Lucene

Lucene was written by Doug Cutting;[2] it was initially available for download from its home at the SourceForge website. It joined the Apache Software Foundation's Jakarta family of high-quality open source Java products in September 2001 and became its own top-level Apache project in February 2005. It now has a number of subprojects, which you can see at http://lucene.apache.org. This book is primarily about the Java subproject, at http://lucene.apache.org/java, though many people refer to it simply as "Lucene."

With each release, the project has enjoyed increased visibility, attracting more users and developers. As of March 2010, the most recent release of Lucene is 3.0.1. Table 1.1 shows Lucene's release history.

[2] *Lucene* is Doug's wife's middle name; it's also her maternal grandmother's first name.

Table 1.1 Lucene's release history

Version	Release date	Milestones
0.01	March 2000	First open source release (SourceForge)
1.0	October 2000	
1.01b	July 2001	Last SourceForge release
1.2	June 2002	First Apache Jakarta release
1.3	December 2003	Compound index format, `QueryParser` enhancements, remote searching, token positioning, extensible scoring API
1.4	July 2004	Sorting, span queries, term vectors
1.4.1	August 2004	Bug fix for sorting performance
1.4.2	October 2004	`IndexSearcher` optimization and miscellaneous fixes
1.4.3	November 2004	Miscellaneous fixes
1.9.0	February 2006	Binary stored fields, `DateTools`, `NumberTools`, `RangeFilter`, `RegexQuery`; requires Java 1.4
1.9.1	March 2006	Bug fix in `BufferedIndexOutput`
2.0	May 2006	Removed deprecated methods
2.1	February 2007	Delete/update document in `IndexWriter`, locking simplifications, `QueryParser` improvements, benchmark contrib module
2.2	June 2007	Performance improvements, function queries, payloads, pre-analyzed fields, custom deletion policies
2.3.0	January 2008	Performance improvements, custom merge policies and merge schedulers, background merges by default, tool to detect index corruption, `IndexReader.reopen`
2.3.1	February 2008	Bug fixes from 2.3.0
2.3.2	May 2008	Bug fixes from 2.3.1
2.4.0	October 2008	Further performance improvements, transactional semantics (rollback, commit), `expungeDeletes` method, delete by query in `IndexWriter`
2.4.1	March 2009	Bug fixes from 2.4.0
2.9	September 2009	New per-segment Collector API, faster search performance, near real-time search, attribute-based analysis
2.9.1	November 2009	Bug fixes from 2.9
2.9.2	February 2010	Bug fixes from 2.9.1
3.0.0	November 2009	Removed deprecated methods, fixed some bugs
3.0.1	February 2010	Bug fixes from 3.0.0

NOTE Lucene's creator, Doug Cutting, has significant theoretical and practical experience in the field of IR. He's published a number of research papers on IR topics and has worked for companies such as Excite, Apple, Grand Central and Yahoo!. In 2004, worried about the decreasing number of web search engines and a potential monopoly in that realm, he created Nutch, the first open source World Wide Web search engine (http://lucene.apache.org/nutch); it's designed to handle crawling, indexing, and searching of several billion frequently updated web pages. Not surprisingly, Lucene is at the core of Nutch. Doug is also actively involved in Hadoop (http://hadoop.apache.org), a project that spun out of Nutch to provide tools for distributed storage and computation using the map/reduce framework.

Doug Cutting remains a strong force behind Lucene, and many more developers have joined the project with time. As of this writing, Lucene's core team includes about half a dozen active developers, three of whom are authors of this book. In addition to the official project developers, Lucene has a fairly large and active technical user community that frequently contributes patches, bug fixes, and new features.

One way to judge the success of open source software is by the number of times it's been ported to other programming languages. Using this metric, Lucene is quite a success! Although Lucene is written entirely in Java, as of this writing there are Lucene ports and bindings in many other programming environments, including Perl, Python, Ruby, C/C++, PHP, and C# (.NET). This is excellent news for developers who need to access Lucene indices from applications written in diverse programming languages. You can learn more about many of these ports in chapter 10.

To understand how Lucene fits into a search application, including what Lucene can and can't do, in the next rather large section we review the architecture of a "typical" modern search application.

1.3 Lucene and the components of a search application

It's important to grasp the big picture so that you have a clear understanding of which parts Lucene can handle and which parts your application must separately handle. A common misconception is that Lucene is an entire search application, when in fact it's simply the core indexing and searching component.

We'll see that a search application starts with an indexing chain, which in turn requires separate steps to retrieve the raw content; create documents from the content, possibly extracting text from binary documents; and index the documents. Once the index is built, the components required for searching are equally diverse, including a user interface, a means for building up a programmatic query, query execution (to retrieve matching documents), and results rendering.

Modern search applications have wonderful diversity. Some run quietly, as a small component deeply embedded inside an existing tool, searching a specific set of content (local files, email messages, calendar entries, etc.). Others run on a remote website, on a dedicated server infrastructure, interacting with many users via a web browser or mobile device, perhaps searching a product catalog or a known and clearly scoped set of documents. Some run inside a company's intranet and search a massive collection of documents visible inside the company. Still others index a large subset of the entire web and must deal with unbelievable scale both in content and in simultaneous search traffic. Yet despite all this variety, search engines generally share a common overall architecture, as shown in figure 1.4.

When designing your application, you clearly have strong opinions on what features are necessary and how they should work. Be forewarned: modern popular web search engines (notably Google) have pretty much set the baseline requirements that all users will expect the first time they interact with your search application. If your search can't meet this baseline, users will be disappointed right from the start. Google's spell correction is amazing, the dynamic summaries with highlighting under each result are accurate, and the response time is well under a second. When in doubt, look to Google for inspiration and guidance on which basic features your search application must provide. Imitation is the sincerest form of flattery!

Let's walk through a search application, one component at a time. As you're reading along, think through what your application requires from each of these components to understand how you could use Lucene to achieve your search goals. We'll also clearly point out which components Lucene can handle (the shaded boxes in figure 1.4) and which will be up to your application or other open source software. We'll then wrap up with a summary of Lucene's role in your search application.

Starting from the bottom of figure 1.4 and working up is the first part of all search engines, a concept called *indexing*: processing the original data into a highly efficient cross-reference lookup in order to facilitate rapid searching.

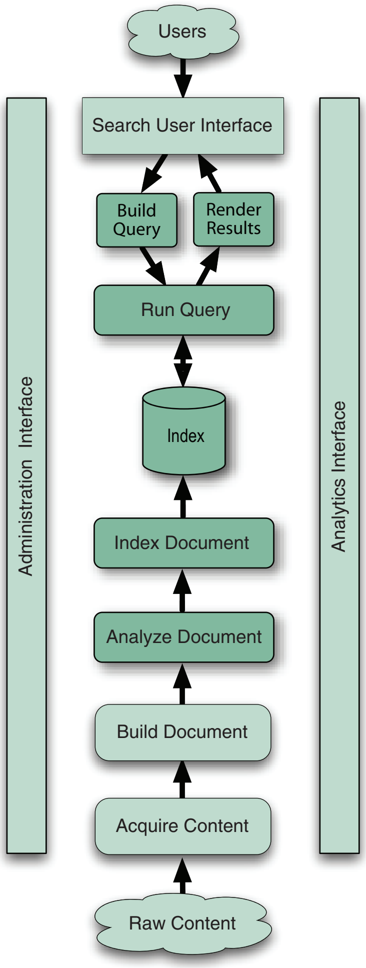


Figure 1.4 Typical components of search application; the shaded components show which parts Lucene handles.

1.3.1 Components for indexing

Suppose you need to search a large number of files, and you want to find files that contain a certain word or a phrase. How would you go about writing a program to do this? A naïve approach would be to sequentially scan each file for the given word or phrase. Although this approach would work, it has a number of flaws, the most obvious of which is that it doesn't scale to larger file sets or cases where files are very large. Here's where indexing comes in: to search large amounts of text quickly, you must first index that text and convert it into a format that will let you search it rapidly, eliminating the slow sequential scanning process. This conversion process is called *indexing*, and its output is called an *index*.

You can think of an index as a data structure that allows fast random access to words stored inside it. The concept behind it is analogous to an index at the end of a book, which lets you quickly locate pages that discuss certain topics. In the case of Lucene, an index is a specially designed data structure, typically stored on the file system as a set of index files. We cover the structure of separate index files in detail in appendix B, but for now think of a Lucene index as a tool that allows quick word lookup.

When you take a closer look, you discover that indexing consists of a sequence of logically distinct steps which we'll explore next. First, you must gain access to the content you need to search.

ACQUIRE CONTENT

The first step, at the bottom of figure 1.4, is to acquire content. This process, which involves using a crawler or spider, gathers and scopes the content that needs to be indexed. That may be trivial, for example, if you're indexing a set of XML files that resides in a specific directory in the file system or if all your content resides in a well-organized database. Alternatively, it may be horribly complex and messy if the content is scattered in all sorts of places (file systems, content management systems, Microsoft Exchange, Lotus Domino, various websites, databases, local XML files, CGI scripts running on intranet servers, and so forth).

Using entitlements (which means allowing only specific authenticated users to see certain documents) can complicate content acquisition, because it may require "superuser" access when acquiring the content. Furthermore, the access rights or access control lists (ACLs) must be acquired along with the document's content, and added to the document as additional fields used during searching to properly enforce the entitlements. We cover security filters during searching in section 5.6.7.

For large content sets, it's important that this component be efficiently incremental, so that it can visit only changed documents since it was last run. It may also be "live," meaning it's a continuously running service, waiting for new or changed content to arrive and loading that content the moment it becomes available.

Lucene, as a core search library, doesn't provide any functionality to support acquiring content. This is entirely up to your application, or a separate piece of software. A number of open source crawlers are available, among them the following:

- Solr (http://lucene.apache.org/solr), a sister project under the Apache Lucene umbrella, has support for natively ingesting relational databases and XML feeds, as well as handling rich documents through Tika integration. (We cover Tika in chapter 7.)
- Nutch (http://lucene.apache.org/nutch), another sister project under the Apache Lucene umbrella, has a high-scale crawler that's suitable for discovering content by crawling websites.
- Grub (http://www.grub.org) is a popular open source web crawler.
- Heritrix is Internet Archive's open source crawler (http://crawler.archive.org).
- Droids, another subproject under the Apache Lucene umbrella, is currently under Apache incubation at http://incubator.apache.org/droids.
- Aperture (http://aperture.sourceforge.net) has support for crawling websites, file systems, and mail boxes and for extracting and indexing text.
- The Google Enterprise Connector Manager project (http://code.google.com/p/google-enterprise-connector-manager) provides connectors for a number of nonweb repositories.

If your application has scattered content, it might make sense to use a preexisting crawling tool. Such tools are typically designed to make it easy to load content stored in various systems, and sometimes provide prebuilt connectors to common content stores, such as websites, databases, popular content management systems, and file systems. If your content source doesn't have a preexisting connector for the crawler, it's likely easy enough to build your own.

The next step is to create bite-sized pieces, called documents, out of your content.

BUILD DOCUMENT

Once you have the raw content that needs to be indexed, you must translate the content into the *units* (usually called *documents*) used by the search engine. The document typically consists of several separately named fields with values, such as *title*, *body*, *abstract*, *author*, and *url*. You'll have to carefully design how to divide the raw content into documents and fields as well as how to compute the value for each of those fields. Often the approach is obvious: one email message becomes one document, or one PDF file or web page is one document. But sometimes it's less clear: how should you handle attachments on an email message? Should you glom together all text extracted from the attachments into a single document, or make separate documents, somehow linked back to the original email message, for each attachment?

Once you've worked out this design, you'll need to extract text from the original raw content for each document. If your content is already textual in nature, with a known standard encoding, your job is simple. But more often these days documents are binary in nature (PDF, Microsoft Office, Open Office, Adobe Flash, streaming video and audio multimedia files) or contain substantial markups that you must remove before indexing (RDF, XML, HTML). You'll need to run document filters to extract text from such content before creating the search engine document.

Interesting business logic may also apply during this step to create additional fields. For example, if you have a large "body text" field, you might run semantic analyzers to pull out proper names, places, dates, times, locations, and so forth into separate fields in the document. Or perhaps you tie in content available in a separate store (such as a database) and merge this for a single document to the search engine.

Another common part of building the document is to inject boosts to individual documents and fields that are deemed more or less important. Perhaps you'd like your press releases to come out ahead of all other documents, all things being equal? Perhaps recently modified documents are more important than older documents? Boosting may be done statically (per document and field) at indexing time, which we cover in detail in section 2.5, or dynamically during searching, which we cover in section 5.7. Nearly all search engines, including Lucene, automatically statically boost fields that are shorter over fields that are longer. Intuitively this makes sense: if you match a word or two in a very long document, it's quite a bit less relevant than matching the same words in a document that's, say, three or four words long.

Lucene provides an API for building fields and documents, but it doesn't provide any logic to build a document because that's entirely application specific. It also doesn't provide any document filters, although Lucene has a sister project at Apache, Tika, which handles document filtering very well (see chapter 7). If your content resides in a database, projects like DBSight, Hibernate Search, LuSQL, Compass, and Oracle/Lucene integration make indexing and searching your tables simple by handling the Acquire Content and Build Document steps seamlessly.

The textual fields in a document can't be indexed by the search engine just yet. In order to do that, the text must first be analyzed.

ANALYZE DOCUMENT

No search engine indexes text directly: rather, the text must be broken into a series of individual atomic elements called *tokens*. This is what happens during the Analyze Document step. Each token corresponds roughly to a "word" in the language, and this step determines how the textual fields in the document are divided into a series of tokens. There are all sorts of interesting questions here: how do you handle compound words? Should you apply spell correction (if your content itself has typos)? Should you inject synonyms inlined with your original tokens, so that a search for "laptop" also returns products mentioning "notebook"? Should you collapse singular and plural forms to the same token? Often a stemmer, such as Dr. Martin Porter's Snowball stemmer (covered in section 8.2.1) is used to derive roots from words (for example, runs, running, and run, all map to the base form *run*). Should you preserve or destroy differences in case? For non-Latin languages, how can you even determine what a "word" is? This component is so important that we have a whole chapter, chapter 4, describing it.

Lucene provides an array of built-in analyzers that give you fine control over this process. It's also straightforward to build your own analyzer, or create arbitrary analyzer chains combining Lucene's tokenizers and token filters, to customize how tokens are created. The final step is to index the document.

INDEX DOCUMENT

During the indexing step, the document is added to the index. Lucene provides everything necessary for this step, and works quite a bit of magic under a surprisingly simple API. Chapter 2 takes you through all the nitty-gritty steps for performing indexing.

We're done reviewing the typical indexing steps for a search application. It's important to remember that indexing is something of a necessary evil that you must undertake in order to provide a good search experience: you should design and customize your indexing process only to the extent that you improve your users' search experience. We'll now visit the steps involved in searching.

1.3.2 Components for searching

Searching is the process of looking up words in an index to find documents where they appear. The quality of a search is typically described using *precision* and *recall* metrics. Recall measures how well the search system finds relevant documents; precision measures how well the system filters out the irrelevant documents. Appendix C describes how to use Lucene's benchmark contrib module to measure precision and recall of your search application.

You must consider a number of other factors when thinking about searching. We already mentioned speed and the ability to quickly search large quantities of text. Support for single and multiterm queries, phrase queries, wildcards, fuzzy queries, result ranking, and sorting are also important, as is a friendly syntax for entering those queries. Lucene offers a number of search features, bells, and whistles—so many that we had to spread our search coverage over three chapters (chapters 3, 5, and 6).

Let's work through the typical components of a search engine, this time working top down in figure 1.4, starting with the search user interface.

SEARCH USER INTERFACE

The user interface is what users actually see, in the web browser, desktop application, or mobile device, when they interact with your search application. The UI is the most important part of your search application! You could have the greatest search engine in the world under the hood, tuned with fabulous state-of-the-art functionality, but with one silly mistake, the UI will lack consumability, thus confusing your precious and fickle users who will then quietly move on to your competitors.

Keep the interface simple: don't present a lot of advanced options on the first page. Provide a ubiquitous, prominent search box, visible everywhere, rather than requiring a two-step process of first clicking a search link and then entering the search text (this is a common mistake).

Don't underestimate the importance of result presentation. Simple details, like failing to highlight matches in the titles and excerpts, or using a small font and cramming too much text into the results, can quickly kill a user's search experience. Be sure the sort order is clearly called out and defaults to an appropriate starting point (usually relevance). Be fully transparent: if your search application is doing something "interesting," such as expanding the search to include synonyms, using boosts to influence sort order, or automatically correcting spelling, say so clearly at the top of the search results and make it easy for the user to turn it off.

NOTE The worst thing that can happen, and it happens quite easily, is to erode the user's trust in the search results. Once this happens, your users will quietly move on and you may never again have the chance to earn back that trust.

Most of all, eat your own dog food: use your own search application extensively. Enjoy what's good about it, but aggressively correct what's bad. Almost certainly your search interface should offer spell correction. Lucene has a contrib module, spellchecker, covered in section 8.5, that you can use. Likewise, providing dynamic excerpts (sometimes called summaries) with hit highlighting under each search result is important, and Lucene's contrib directory offers two such modules, highlighter and fast vector highlighter, covered in sections 8.3 and 8.4, to handle this.

Lucene doesn't provide any default search UI; it's entirely up to your application to build one. Once a user interacts with your search interface, she or he submits a search request, which first must be translated into an appropriate `Query` object for the search engine.

BUILD QUERY

When you manage to entice a user to use your search application, she or he issues a search request, often as the result of an HTML form or Ajax request submitted by a browser to your server. You must then translate the request into the search engine's `Query` object. We call this the Build Query step.

`Query` objects can be simple or complex. Lucene provides a powerful package, called `QueryParser`, to process the user's text into a query object according to a common search syntax. We'll cover `QueryParser` and its syntax in chapter 3, but it's also fully described at http://lucene.apache.org/java/3_0_0/queryparsersyntax.html. The query may contain Boolean operations, phrase queries (in double quotes), or wildcard terms. If your application has further controls on the search UI, or further interesting constraints, you must implement logic to translate this into the equivalent query. For example, if there are entitlement constraints that restrict which set of documents each user is allowed to search, you'll need to set up filters on the query, which we visit in section 5.6.

Many applications will at this point also modify the search query so as to boost or filter for important things, if the boosting wasn't done during indexing. Often an e-commerce site will boost categories of products that are more profitable, or filter out products presently out of stock (so you don't see that they're out of stock and then go elsewhere to buy them). Resist the temptation to heavily boost and filter the search results: users will catch on and lose trust.

Lucene's default `QueryParser` is often sufficient for an application. Sometimes, you'll want to use the output of `QueryParser` but then add your own logic afterward to further refine the query object. Still other times you want to customize the `QueryParser`'s syntax, or customize which `Query` instances it actually creates, which, thanks to Lucene's open source nature, is straightforward. We discuss customizing `QueryParser` in section 6.3. Now, you're ready to execute the search request to retrieve results.

SEARCH QUERY

Search Query is the process of consulting the search index and retrieving the documents matching the `Query`, sorted in the requested sort order. This component covers the complex inner workings of the search engine, and Lucene handles all of it for you. Lucene is also wonderfully extensible at this point, so if you'd like to customize how results are gathered, filtered, sorted, and so forth, it's straightforward. See chapter 6 for details.

There are three common theoretical models of search:

- *Pure Boolean model*—Documents either match or don't match the provided query, and no scoring is done. In this model there are no relevance scores associated with matching documents, and the matching documents are unordered; a query simply identifies a subset of the overall corpus as matching the query.
- *Vector space model*—Both queries and documents are modeled as vectors in a high dimensional space, where each unique term is a dimension. Relevance, or similarity, between a query and a document is computed by a vector distance measure between these vectors.
- *Probabilistic model*—In this model, you compute the probability that a document is a good match to a query using a full probabilistic approach.

Lucene's approach combines the vector space and pure Boolean models, and offers you controls to decide which model you'd like to use on a search-by-search basis. Finally, Lucene returns documents that you next must render in a consumable way for your users.

RENDER RESULTS

Once you have the raw set of documents that match the query, sorted in the right order, you then render them to the user in an intuitive, consumable manner. The UI should also offer a clear path for follow-on searches or actions, such as clicking to the next page, refining the search, or finding documents similar to one of the matches, so that the user never hits a dead end.

We've finished reviewing the components of both the indexing and searching paths in a search application, but we aren't done. Search applications also often require ongoing administration.

1.3.3 The rest of the search application

There's still quite a bit more to a typical fully functional search engine, especially a search engine running on a website. You must include administration, in order to keep track of the application's health, configure the different components, and start and stop servers. You must also include analytics, allowing you to use different views to see how your users are searching, thus giving you the necessary guidance on what's working and what's not. Finally, for large search applications, scaling—so that your application can handle larger and larger content sizes as well as higher and higher numbers of simultaneous search queries—is a very important feature. Spanning the left side of figure 1.4 is the administration interface.

ADMINISTRATION INTERFACE

A modern search engine is a complex piece of software and has numerous controls that need configuration. If you're using a crawler to discover your content, the administration interface should let you set the starting URLs, create rules to scope which sites the crawler should visit or which document types it should load, set how quickly it's allowed to read documents, and so forth. Starting and stopping servers, managing replication (if it's a high-scale search, or if high availability failover is required), culling search logs, checking overall system health, and creating and restoring from backups are all examples of what an administration interface might offer.

Lucene has a number of configuration options that an administration interface would expose. During indexing you may need to tune the size of the RAM buffer, how many segments to merge at once, how often to commit changes, or when to optimize and purge deletes from the index. We'll cover these topics in detail in chapter 2. Searching also has important administration options, such as how often to reopen the reader. You'll probably also want to expose some basic summary information of the index, such as segment and pending deletion counts. If some documents failed to be indexed properly, or queries hit exceptions while searching, your administration API would detail them.

Many search applications, such as desktop search, don't require this component, whereas a full enterprise search application may have a complex administration interface. Often the interface is primarily web based, but it may also consist of additional command-line tools. On the right side of figure 1.4 is the analytics interface.

ANALYTICS INTERFACE

Spanning the right side is the analytics interface, which is often a web-based UI, perhaps running under a separate server hosting a reporting engine. Analytics is important: you can gain a lot of intelligence about your users and why they do or do not buy your widgets through your website, by looking for patterns in the search logs. Some would say this is the most important reason to deploy a good search engine! If you run an e-commerce website, incredibly powerful tools—that let you see how your users run searches, which searches failed to produce satisfactory results, which results users clicked on, and how often a purchase followed or did not follow a search—enable you to optimize the buying experience of your users.

Lucene-specific metrics that could feed the analytics interface include:

- How often which kinds of queries (single term, phrase, Boolean queries, etc.) are run
- Queries that hit low relevance
- Queries where the user didn't click on any results (if your application tracks click-throughs)
- How often users are sorting by specified fields instead of relevance
- The breakdown of Lucene's search time

You may also want to see indexing metrics, such as documents indexed per second or byte size of documents being indexed.

Lucene, since it's a search library, doesn't provide any analytics tools. If your search application is web based, Google Analytics is a fast way to create an analytics interface. If that doesn't fit your needs, you can also build your own custom charts based on Google's visualization API. The final topic we visit is scaling.

SCALING

One particularly tricky area is scaling of your search application. The vast majority of search applications don't have enough content or simultaneous search traffic to require scaling beyond a single computer. Lucene indexing and searching throughput allows for a sizable amount of content on a single modern computer. Still, such applications may want to run two identical computers to ensure there's no single point of failure (no downtime) in the event of hardware failure. This approach also enables you to pull one computer out of production to perform maintenance and upgrades without affecting ongoing searches.

There are two dimensions to scaling: net amount of content, and net query throughput. If you have a tremendous amount of content, you must divide it into shards, so that a separate computer searches each shard. A front-end server sends a single incoming query to all shards, and then coalesces the results into a single result set. If instead you have high search throughput during your peak traffic, you'll have to take the same index and replicate it across multiple computers. A front-end load balancer sends each incoming query to the least loaded back-end computer. If you require both dimensions of scaling, as a web scale search engine will, you combine both of these practices.

A number of complexities are involved in building such an architecture. You'll need a reliable way of replicating the search index across computers. If a computer has some downtime, planned or not, you need a way to bring it up-to-date before putting it back into production. If there are transactional requirements, so that all searchers must "go live" on a new index commit simultaneously, that adds complexity. Error recovery in a distributed setting can be complex. Finally, important functionality like spell correction and highlighting, and even how term weights are computed for scoring, are impacted by such a distributed architecture.

Lucene provides no facilities for scaling. However, both Solr and Nutch, projects under the Apache Lucene umbrella, provide support for index sharding and replication. The Katta open source project, hosted at http://katta.sourceforge.net and based on Lucene, also provides this functionality. Elastic search, at http://www.elasticsearch.com, is another option that's also open source and based on Lucene. Before you build your own approach, it's best to have a solid look at these existing solutions.

We've finished reviewing the components of a modern search application. Now it's time to think about whether Lucene is a fit for your application.

1.3.4 Where Lucene fits into your application

As you've seen, a modern search application can require many components. Yet the needs of a specific application from each of these components vary greatly. Lucene covers many of these components well (the gray shaded ones from figure 1.4), but other components are best covered by complementary open source software or by your own custom application logic. It's possible your application is specialized enough to not require certain components. You should at this point have a good sense of what we mean when we say Lucene is a search library, not a full application.

If Lucene isn't a direct fit, it's likely one of the open source projects that complements or builds upon Lucene does fit. For example, Solr runs within an application server and exposes an administration interface, both dimensions of scaling, the ability to index content from a database, and important end-user functionality like faceted navigation, all built on top of Lucene. Lucene is the search library whereas Solr provides most components of an entire search application.

In addition, some web application frameworks also provide search plug-ins based on Lucene. For example, there's a searchable plug-in for Grails (http://www.grails.org/Searchable+Plugin), based on the Compass Search Engine Framework, which in turn uses Lucene under the hood.

Now let's see a concrete example of using Lucene for indexing and searching.

1.4 Lucene in action: a sample application

It's time to see Lucene in action. To do that, recall the problem of indexing and searching files, which we described in section 1.3. To show you Lucene's indexing and searching capabilities, we'll use a pair of command-line applications: Indexer and Searcher. First we'll index files in a directory; then we'll search the created index.

These example applications will familiarize you with Lucene's API, its ease of use, and its power. The code listings are complete, ready-to-use command-line programs. If file indexing/searching is the problem you need to solve, you can copy the code listings and tweak them to suit your needs. In the chapters that follow, we'll describe each aspect of Lucene's use in much greater detail.

Before we can search with Lucene, we need to build an index, so we start with our Indexer application.

1.4.1 Creating an index

In this section you'll see a simple class called `Indexer`, which indexes all files in a directory ending with the .txt extension. When Indexer completes execution, it leaves behind a Lucene index for its sibling, Searcher (presented next in section 1.4.2).

We don't expect you to be familiar with the few Lucene classes and methods used in this example—we'll explain them shortly. After the annotated code listing, we show you how to use Indexer; if it helps you to learn how Indexer is used before you see how it's coded, go directly to the usage discussion that follows the code.

USING INDEXER TO INDEX TEXT FILES

Listing 1.1 shows the Indexer command-line program, originally written for Erik's introductory Lucene article on java.net. It takes two arguments:

- A path to a directory where we store the Lucene index
- A path to a directory that contains the files we want to index

Listing 1.1 Indexer, which indexes .txt files

```
public class Indexer {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      throw new IllegalArgumentException("Usage: java " +
  ➥Indexer.class.getName()
        + " <index dir> <data dir>");
    }                                              1 Create index in
    String indexDir = args[0];                       this directory
    String dataDir = args[1];
                                                          Index *.txt files
                                                        2 from this directory
    long start = System.currentTimeMillis();
    Indexer indexer = new Indexer(indexDir);
    int numIndexed;
    try {
      numIndexed = indexer.index(dataDir, new TextFilesFilter());
    } finally {
      indexer.close();
    }
    long end = System.currentTimeMillis();

    System.out.println("Indexing " + numIndexed + " files took "
      + (end - start) + " milliseconds");
  }

  private IndexWriter writer;

  public Indexer(String indexDir) throws IOException {
    Directory dir = FSDirectory.open(new File(indexDir));
    writer = new IndexWriter(dir,                              3 Create Lucene
                 new StandardAnalyzer(                           IndexWriter
                     Version.LUCENE_30),
                 true,
                 IndexWriter.MaxFieldLength.UNLIMITED);
  }

  public void close() throws IOException {           4 Close IndexWriter
    writer.close();
  }

  public int index(String dataDir, FileFilter filter)
    throws Exception {

    File[] files = new File(dataDir).listFiles();

    for (File f: files) {
      if (!f.isDirectory() &&
          !f.isHidden() &&
          f.exists() &&
          f.canRead() &&
          (filter == null || filter.accept(f))) {
        indexFile(f);
      }
    }                                         5 Return number of
                                                documents indexed
    return writer.numDocs();
  }
  private static class TextFilesFilter implements FileFilter {
    public boolean accept(File path) {
      return path.getName().toLowerCase()                ❻ Index .txt files only,
             .endsWith(".txt");                            using FileFilter
    }
  }

  protected Document getDocument(File f) throws Exception {
    Document doc = new Document();                                      ❼ Index file
    doc.add(new Field("contents", new FileReader(f)));                     content
    doc.add(new Field("filename", f.getName(),                             ❽ Index
                Field.Store.YES, Field.Index.NOT_ANALYZED));                  filename
    doc.add(new Field("fullpath", f.getCanonicalPath(),
                Field.Store.YES, Field.Index.NOT_ANALYZED));
    return doc;                                                            Index file
  }                                                                      ❾ full path

  private void indexFile(File f) throws Exception {
    System.out.println("Indexing " + f.getCanonicalPath());
    Document doc = getDocument(f);
    writer.addDocument(doc);                             Add document
  }                                                   ❿ to Lucene index
}
```

Indexer is simple. The static main method parses ❶, ❷ the incoming arguments, creates an `Indexer` instance, locates ❻ *.txt in the provided data directory, and prints how many documents were indexed and how much time was required. The code involving the Lucene APIs includes creating ❸ and closing ❹ the `IndexWriter`, creating ❼, ❽, ❾ the document, adding ❿ the document to the index, and returning the number of documents indexed ❺.

This example intentionally focuses on plain text files with .txt extensions to keep things simple, while demonstrating Lucene's usage and power. In chapter 7, we'll show you how to index other common document types, such as Microsoft Word or Adobe PDF, using the Tika framework. Before seeing how to run Indexer, let's talk a bit about the `Version` parameter you see as the first argument to `StandardAnalyzer`.

VERSION PARAMETER

As of version 2.9, a number of classes now accept a parameter of type `Version` (from the `org.apache.lucene.util` package) during construction. This class defines enum constants, such as `LUCENE_24` and `LUCENE_29`, referencing Lucene's minor releases. When you pass one of these values, it instructs Lucene to match the settings and behavior of that particular release. Lucene will also emulate bugs present in that release and fixed in later releases, if the Lucene developers felt that fixing the bug would break backward compatibility of existing indexes. For each class that accepts a `Version` parameter, you'll have to consult the Javadocs to see what settings and bugs are changed across versions. All examples in this book use `LUCENE_30`.

Although some may see the `Version` argument as polluting Lucene's API, it is in fact a demonstration of both Lucene's maturity and how seriously the Lucene developers take backward compatibility. The `Version` parameter gives Lucene the freedom to fix bugs and improve default settings for new users, over time, while still achieving backward compatibility when it's important. It also places the choice—latest and greatest versus strict backward compatibility—in your hands.

Let's use Indexer to build our first Lucene search index!

RUNNING INDEXER

The simplest way to run Indexer is to use Apache Ant. You'll first have to unpack the zip file containing source code with this book, which you can download from Manning's site at http://www.manning.com/hatcher3, and change to the directory lia2e. If you don't see the file build.xml in your working directory, you're not in the right directory. If this is the first time you've run any targets, Ant will compile all the example sources, build the test index, and finally run Indexer, first prompting you for the index and document directory, in case you'd like to change the defaults. It's also fine to run Indexer using Java from the command line; just ensure your classpath includes the JARs under the lib subdirectory as well as the build/classes directory.

By default the index will be placed under the subdirectory indexes/MeetLucene, and the sample documents under the directory src/lia/meetlucene/data will be indexed. This directory contains a sampling of modern open source licenses.

Go ahead and type `ant Indexer`, and you should see output like this:

```
% ant Indexer

Index *.txt files in a directory into a Lucene index.
Use the Searcher target to search this index.

Indexer is covered in the "Meet Lucene" chapter.

Press return to continue...

Directory for new Lucene index: [indexes/MeetLucene]

Directory with .txt files to index: [src/lia/meetlucene/data]

Overwrite indexes/MeetLucene? (y, n) y
Running lia.meetlucene.Indexer...
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/apache1.0.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/apache1.1.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/apache2.0.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/cpl1.0.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/epl1.0.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/freebsd.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/gpl1.0.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/gpl2.0.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/gpl3.0.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/lgpl2.1.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/lgpl3.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/lpgl2.0.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/mit.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/mozilla1.1.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/
➥ mozilla_eula_firefox3.txt
Indexing /Users/mike/lia2e/src/lia/meetlucene/data/
➥ mozilla_eula_thunderbird2.txt
Indexing 16 files took 757 milliseconds

BUILD SUCCESSFUL
```

Indexer prints the names of files it indexes, so you can see that it indexes only files with the .txt extension. When it completes indexing, Indexer prints the number of files it indexed and the time it took to do so. Because the reported time includes both file-directory listing and indexing, you shouldn't consider it an official performance measure. In our example, each of the indexed files was small, but roughly 0.8 seconds to index a handful of text files is reasonably impressive. Indexing throughput is clearly important, and we cover it extensively in chapter 11. But generally, searching is far more important since an index is built once but searched many times.

1.4.2 Searching an index

Searching in Lucene is as fast and simple as indexing; the power of this functionality is astonishing, as chapters 3, 5, and 6 will show you. For now, let's look at Searcher, a command-line program that we'll use to search the index created by Indexer. Keep in mind that our Searcher serves the purpose of demonstrating the use of Lucene's search API. Your search application could also take the form of a web or desktop application with a GUI, a web application, and so on.

In the previous section, we indexed a directory of text files. The index in this example resides in a directory of its own on the file system. We instructed Indexer to create a Lucene index in the indexes/MeetLucene directory, relative to the directory from which we invoked Indexer. As you saw in listing 1.1, this index contains the indexed contents of each file, along with the absolute path. Now we need to use Lucene to search that index in order to find files that contain a specific piece of text. For instance, we may want to find all files that contain the keyword *patent* or *redistribute*, or we may want to find files that include the phrase *modified version*. Let's do some searching now.

USING SEARCHER TO IMPLEMENT A SEARCH

The Searcher program, originally written for Erik's introductory Lucene article on java.net, complements Indexer and provides command-line searching capability. Listing 1.2 shows Searcher in its entirety. It takes two command-line arguments:

- The path to the index created with Indexer
- A query to use to search the index

Listing 1.2 Searcher, which searches a Lucene index

```
public class Searcher {

  public static void main(String[] args) throws IllegalArgumentException,
        IOException, ParseException {
    if (args.length != 2) {
      throw new IllegalArgumentException("Usage: java " +
     Searcher.class.getName()
        + " <index dir> <query>");
    }

    String indexDir = args[0];
    String q = args[1];
```

1 Parse provided index directory

2 Parse provided query string

```
    search(indexDir, q);
  }

  public static void search(String indexDir, String q)
    throws IOException, ParseException {

    Directory dir = FSDirectory.open(new File(indexDir));          3 Open
    IndexSearcher is = new IndexSearcher(dir);                        index

    QueryParser parser = new QueryParser(Version.LUCENE_30,          4 Parse
                     "contents",                                      query
                     new StandardAnalyzer(
                       Version.LUCENE_30));
    Query query = parser.parse(q);
    long start = System.currentTimeMillis();                      5 Search
    TopDocs hits = is.search(query, 10);                              index
    long end = System.currentTimeMillis();

    System.err.println("Found " + hits.totalHits +
      " document(s) (in " + (end - start) +                         6 Write
      " milliseconds) that matched query '" +                         search
      q + "':");                                                      stats

    for(ScoreDoc scoreDoc : hits.scoreDocs) {                      7 Retrieve
      Document doc = is.doc(scoreDoc.doc);                            matching document
      System.out.println(doc.get("fullpath"));                      Display
    }                                                             8 filename

    is.close();                              Close
  }                                        9 IndexSearcher
}
```

Searcher, like its Indexer sibling, is quite simple and has only a few lines of code dealing with Lucene:

1 2 We parse command-line arguments (index directory, query string).

3 We use Lucene's `Directory` and `IndexSearcher` classes to open our index for searching.

4 We use `QueryParser` to parse a human-readable search text into Lucene's `Query` class.

5 Searching returns hits in the form of a `TopDocs` object.

6 Print details on the search (how many hits were found and time taken)

7 8 Note that the `TopDocs` object contains only references to the underlying documents. In other words, instead of being loaded immediately upon search, matches are loaded from the index in a lazy fashion—only when requested with the `IndexSearcher.doc(int)` call. That call returns a `Document` object from which we can then retrieve individual field values.

9 Close the `IndexSearcher` when we're done.

RUNNING SEARCHER

Let's run Searcher and find documents in our index using the query `'patent'`:

```
% ant Searcher

Search an index built using Indexer.
```

```
Searcher is described in the "Meet Lucene" chapter.

Press return to continue...

Directory of existing Lucene index built by
➥ Indexer:  [indexes/MeetLucene]

Query:  [patent]

Running lia.meetlucene.Searcher...
Found 8 document(s) (in 11 milliseconds) that
➥ matched query 'patent':
/Users/mike/lia2e/src/lia/meetlucene/data/cpl1.0.txt
/Users/mike/lia2e/src/lia/meetlucene/data/mozilla1.1.txt
/Users/mike/lia2e/src/lia/meetlucene/data/epl1.0.txt
/Users/mike/lia2e/src/lia/meetlucene/data/gpl3.0.txt
/Users/mike/lia2e/src/lia/meetlucene/data/apache2.0.txt
/Users/mike/lia2e/src/lia/meetlucene/data/lpgl2.0.txt
/Users/mike/lia2e/src/lia/meetlucene/data/gpl2.0.txt
/Users/mike/lia2e/src/lia/meetlucene/data/lgpl2.1.txt

BUILD SUCCESSFUL
Total time: 4 seconds
```

The output shows that 8 of the 16 documents we indexed with Indexer contain the word *patent* and that the search took a meager 11 milliseconds. Because Indexer stores files' absolute paths in the index, Searcher can print them. It's worth noting that storing the file path as a field was our decision and appropriate in this case, but from Lucene's perspective, it's arbitrary metadata included in the indexed documents.

You can use more sophisticated queries, such as `'patent AND freedom'` or `'patent AND NOT apache'` or `'+copyright +developers'`, and so on. Chapters 3, 5, and 6 cover various aspects of searching, including Lucene's query syntax.

Our example indexing and searching applications give you a taste of what Lucene makes possible. Its API usage is simple and unobtrusive. The bulk of the code (and this applies to all applications interacting with Lucene) is plumbing relating to the business purpose—in this case, Indexer's parsing of command-line arguments and directory listings to look for text files and Searcher's code that prints matched filenames based on a query to the standard output. But don't let this fact, or the conciseness of the examples, tempt you into complacence: there's a lot going on under the covers of Lucene.

To effectively leverage Lucene, you must understand how it works and how to extend it when the need arises. The remainder of this book is dedicated to giving you these missing pieces.

Next we'll drill down into the core classes Lucene exposes for indexing and searching.

1.5 Understanding the core indexing classes

As you saw in our `Indexer` class, you need the following classes to perform the simplest indexing procedure:

- `IndexWriter`
- `Directory`
- `Analyzer`
- `Document`
- `Field`

Figure 1.5 shows how these classes each participate in the indexing process. What follows is a brief overview of each of these classes, to give you a rough idea of their role in Lucene. We'll use these classes throughout this book.

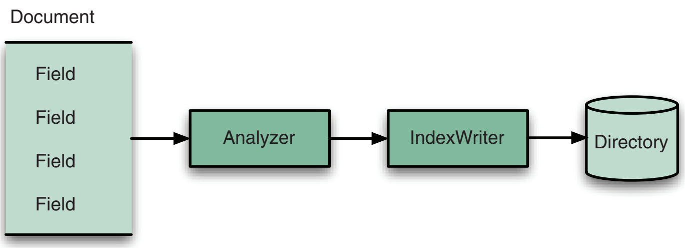


Figure 1.5 Classes used when indexing documents with Lucene

1.5.1 *IndexWriter*

`IndexWriter` is the central component of the indexing process. This class creates a new index or opens an existing one, and adds, removes, or updates documents in the index. Think of `IndexWriter` as an object that gives you write access to the index but doesn't let you read or search it. `IndexWriter` needs somewhere to store its index, and that's what `Directory` is for.

1.5.2 *Directory*

The `Directory` class represents the location of a Lucene index. It's an abstract class that allows its subclasses to store the index as they see fit. In our Indexer example, we used `FSDirectory.open` to get a suitable concrete `FSDirectory` implementation that stores real files in a directory on the file system, and passed that in turn to `IndexWriter`'s constructor.

Lucene includes a number of interesting `Directory` implementations, covered in section 2.10. `IndexWriter` can't index text unless it's first been broken into separate words, using an analyzer.

1.5.3 *Analyzer*

Before text is indexed, it's passed through an analyzer. The analyzer, specified in the `IndexWriter` constructor, is in charge of extracting those tokens out of text that should be indexed and eliminating the rest. If the content to be indexed isn't plain text, you should first extract plain text from it before indexing. Chapter 7 shows how to use Tika to extract text from the most common rich-media document formats. `Analyzer` is an abstract class, but Lucene comes with several implementations of it. Some of them deal with skipping *stop words* (frequently used words that don't help distinguish one document from the other, such as *a*, *an*, *the*, *in*, and *on*); some deal with conversion of tokens to lowercase letters, so that searches aren't case sensitive; and so on. Analyzers are an important part of Lucene and can be used for much more than simple input filtering. For a developer integrating Lucene into an application, the choice of analyzer(s) is a critical element of application design. You'll learn much more about them in chapter 4.

The analysis process requires a document, containing separate fields to be indexed.

1.5.4 Document

The `Document` class represents a collection of fields. Think of it as a virtual document—a chunk of data, such as a web page, an email message, or a text file—that you want to make retrievable at a later time. Fields of a document represent the document or metadata associated with that document. The original source (such as a database record, a Microsoft Word document, a chapter from a book, and so on) of document data is irrelevant to Lucene. It's the text that you extract from such binary documents, and add as a `Field` instance, that Lucene processes. The metadata (such as author, title, subject and date modified) is indexed and stored separately as fields of a document.

NOTE When we refer to a document in this book, we mean a Microsoft Word, RTF, PDF, or other type of a document; we aren't talking about Lucene's `Document` class. Note the distinction in the case and font.

Lucene only deals with text and numbers. Lucene's core doesn't itself handle anything but `java.lang.String`, `java.io.Reader`, and native numeric types (such as int or float). Although various types of documents can be indexed and made searchable, processing them isn't as straightforward as processing purely textual or numeric content. You'll learn more about handling nontext documents in chapter 7.

In our Indexer, we're concerned with indexing text files. So, for each text file we find, we create a new instance of the `Document` class, populate it with fields (described next), and add that document to the index, effectively indexing the file. Similarly, in your application, you must carefully design how a Lucene document and its fields will be constructed to match specific needs of your content sources and application.

A document is simply a container for multiple fields; `Field` is the class that holds the textual content to be indexed.

1.5.5 Field

Each document in an index contains one or more named fields, embodied in a class called `Field`. Each field has a name and corresponding value, and a bunch of options, described in section 2.4, that control precisely how Lucene will index the field's value. A document may have more than one field with the same name. In this case, the values of the fields are appended, during indexing, in the order they were added to the document. When searching, it's exactly as if the text from all the fields were concatenated and treated as a single text field.

You'll apply this handful of classes most often when using Lucene for indexing. To implement basic search functionality, you need to be familiar with an equally small and simple set of Lucene search classes.

1.6 Understanding the core searching classes

The basic search interface that Lucene provides is as straightforward as the one for indexing. Only a few classes are needed to perform the basic search operation:

- `IndexSearcher`
- `Term`
- `Query`
- `TermQuery`
- `TopDocs`

The following sections provide a brief introduction to these classes. We'll expand on these explanations in the chapters that follow, before we dive into more advanced topics.

1.6.1 IndexSearcher

`IndexSearcher` is to searching what `IndexWriter` is to indexing: the central link to the index that exposes several search methods. You can think of `IndexSearcher` as a class that opens an index in a read-only mode. It requires a `Directory` instance, holding the previously created index, and then offers a number of search methods, some of which are implemented in its abstract parent class `Searcher`; the simplest takes a `Query` object and an `int topN` count as parameters and returns a `TopDocs` object. A typical use of this method looks like this:

```
Directory dir = FSDirectory.open(new File("/tmp/index"));
IndexSearcher searcher = new IndexSearcher(dir);
Query q = new TermQuery(new Term("contents", "lucene"));
TopDocs hits = searcher.search(q, 10);
searcher.close();
```

We cover the details of `IndexSearcher` in chapter 3, along with more advanced information in chapters 5 and 6. Now we'll visit the fundamental unit of searching, `Term`.

1.6.2 Term

A `Term` is the basic unit for searching. Similar to the `Field` object, it consists of a pair of string elements: the name of the field and the word (text value) of that field. Note that `Term` objects are also involved in the indexing process. However, they're created by Lucene's internals, so you typically don't need to think about them while indexing. During searching, you may construct `Term` objects and use them together with `TermQuery`:

```
Query q = new TermQuery(new Term("contents", "lucene"));
TopDocs hits = searcher.search(q, 10);
```

This code instructs Lucene to find the top 10 documents that contain the word *lucene* in a field named contents, sorting the documents by descending relevance. Because the `TermQuery` object is derived from the abstract parent class `Query`, you can use the `Query` type on the left side of the statement.

1.6.3 Query

Lucene comes with a number of concrete `Query` subclasses. So far in this chapter we've mentioned only the most basic Lucene `Query`: `TermQuery`. Other `Query` types are `BooleanQuery`, `PhraseQuery`, `PrefixQuery`, `PhrasePrefixQuery`, `TermRangeQuery`, `NumericRangeQuery`, `FilteredQuery`, and `SpanQuery`. All of these are covered in chapters 3 and 5. `Query` is the common, abstract parent class. It contains several utility methods, the most interesting of which is `setBoost(float)`, which enables you to tell Lucene that certain subqueries should have a stronger contribution to the final relevance score than other subqueries. The `setBoost` method is described in section 3.5.12. Next we cover `TermQuery`, which is the building block for most complex queries in Lucene.

1.6.4 TermQuery

`TermQuery` is the most basic type of query supported by Lucene, and it's one of the primitive query types. It's used for matching documents that contain fields with specific values, as you've seen in the last few paragraphs. Finally, wrapping up our brief tour of the core classes used for searching, we touch on `TopDocs`, which represents the result set returned by searching.

1.6.5 TopDocs

The `TopDocs` class is a simple container of pointers to the top N ranked search results—documents that match a given query. For each of the top N results, `TopDocs` records the `int docID` (which you can use to retrieve the document) as well as the float score. Chapter 3 describes `TopDocs` in more detail.

1.7 Summary

In this chapter, you've gained some healthy background knowledge on the architecture of search applications, as well as some initial Lucene knowledge. You now know that Lucene is an information retrieval library, not a ready-to-use standalone product, and that it most certainly doesn't contain a web crawler, document filters, or a search user interface, as people new to Lucene sometimes think. However, as confirmation of Lucene's popularity, there are numerous projects that integrate with or build on Lucene, and that could be a good fit for your application. In addition, you can choose among numerous ways to access Lucene's functionality from programming environments other than Java. You've also learned a bit about how Lucene came to be and about the key people and the organization behind it.

In the spirit of Manning's *in Action* books, we showed you two real, standalone applications, Indexer and Searcher, which are capable of indexing and searching text files stored in a file system. We then briefly described each of the Lucene classes used in these two applications.

Search is everywhere, and chances are that if you're reading this book, you're interested in search becoming an integral part of your applications. Depending on your needs, integrating Lucene may be trivial, or it may involve challenging architectural considerations.

We've organized the next couple of chapters as we did this chapter. The first thing we need to do is index some documents; we discuss this process next in detail in chapter 2.

JAVA

Lucene in Action Second Edition

Michael McCandless • Erik Hatcher • Otis Gospodnetić

Foreword by Doug Cutting

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

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

NEW in the Second Edition

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

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

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

Free ebook
SEE INSERT

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

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

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

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

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

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

MANNING $49.99 / Can $62.99 [INCLUDING eBOOK]