SECOND EDITION

Lucene in Action

Erik Hatcher
Otis Gospodnetić
Michael McCandless

MANNING

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=451

Contents

1

Meet Lucene

This chapter covers

- Understanding Lucene
- General search application architecture
- Using the basic indexing API
- Working with the search API
- Considering alternative products

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 Web sites and desktop tools. While it's written in Java, thanks to its popularity and the determination of zealous developers, there are now a number of ports or integrations to other programming languages (C/C++, C#, Ruby, Perl, Python, PHP, etc.).

One of the key factors behind Lucene's popularity is its simplicity, but don't let that fool you: under the hood there are sophisticated, state of the art Information Retrieval techniques quietly at work. The careful exposure of its indexing and searching API is a sign of the well-designed software. Consequently, 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.

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.) yourself as your application requires. We show you how to perform basic indexing and searching with Lucene with ready-to-use code examples. We then briefly introduce all the core

elements you need to know for both of these processes. We'll start next with the very modern problem of information explosion, to understand why we need powerful search functionality in the first place.

NOTE

Lucene is a very active open-source project. By the time you're reading this, likely Lucene's APIs and features will have changed. This book is based on the 3.0 release of Lucene, and thanks to Lucene's backwards compatibility policy, all code samples should compile and run fine for all future 3.x releases. If you have problems, send an email to java-user@lucene.apache.org and Lucene's large community will surely help.

1.1 Evolution of information organization and access

In order 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 electronic data repositories has brought large amounts of information within our reach. With time, however, the amount of data available has become so vast that we needed 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 specific information you need out of the sea of data isn't limited to the Internet realm—desktop computers store increasingly more data. Changing directories and expanding and collapsing hierarchies of folders isn't an effective way to access stored documents. Furthermore, we no longer use computers just for their raw computing abilities: They also serve as communication devices, multimedia players and media storage devices. Those uses for computers 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 being one of the most precious commodities for most people, we need to 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.

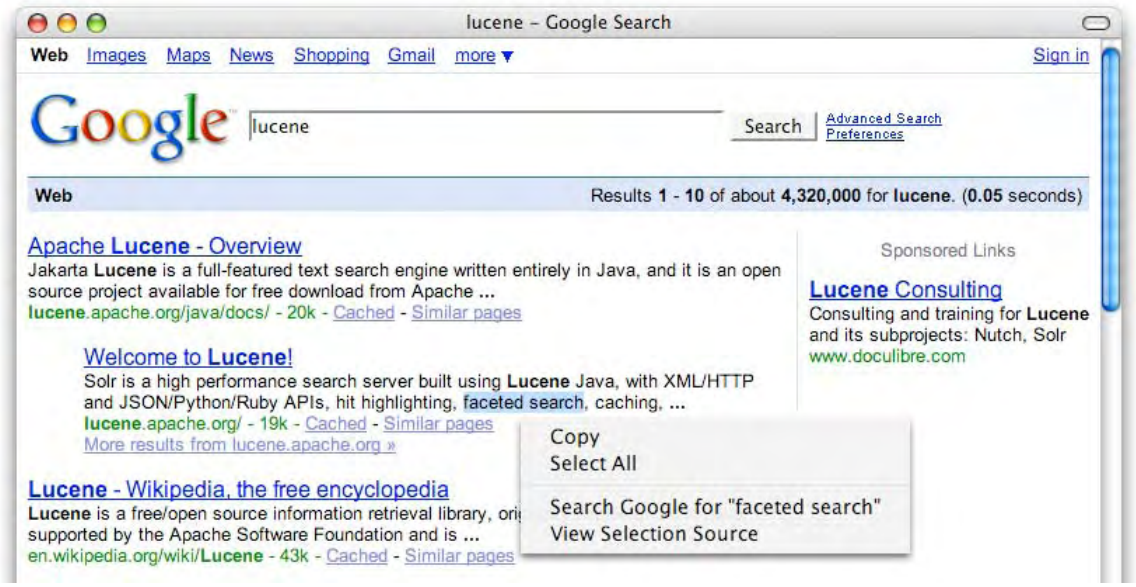


Figure 1.1 Searching the Internet with Google.

To illustrate the pervasiveness of searching across the Internet and the desktop, figure 1.1 shows a search for *lucene* at Google. The figure includes a context menu that lets us use Google to search for the highlighted text. Figure 1.2 shows the Apple Mac OS X Finder (the counterpart to Microsoft's Explorer on Windows) and the search feature embedded at upper right. The Mac OS X music player, iTunes, also has embedded search capabilities, as shown in figure 1.3.

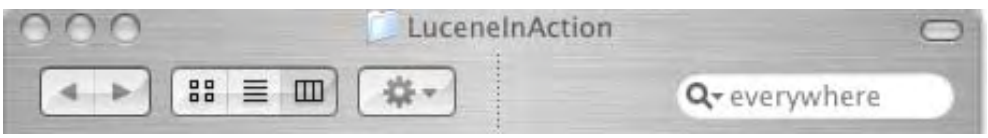


Figure 1.2 Mac OS X Finder with its embedded search capability.

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]

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

Figure 1.3 Apple's iTunes intuitively embeds search functionality.

To understand what role Lucene plays in search, let's start from the basics and learn about what Lucene is and how it can help you with your search needs.

1.2 Understanding Lucene

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 later in this chapter, we'll give you a high-level picture of what Lucene is, what it is not, and how it came to be.

1.2.1 What Lucene is

Lucene is a high performance, scalable Information Retrieval (IR) library. Information retrieval 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 is a mature, free, open-source project implemented in Java; it's 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 (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. You've probably used Lucene already without knowing it!

People new to Lucene often mistake it for a ready-to-use application like a file-search program, a web crawler, or a web site 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 is a single compact JAR file (less than 1 MB!) that has no dependencies. 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 implementation behind a simple-to-use API. You can think of Lucene as the core that the application wraps around, as shown figure 1.4.

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, consult the Lucene Wiki "powered by" page (http://wiki.apache.org/lucene-java/PoweredBy) for many options. We also describe some of these options in section 1.3.4.

1.2.2 What Lucene can do for you

Lucene allows you to add search capabilities to your application. Lucene can index and make searchable any data that you can extract text from. As you can see in figure 1.4, 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 full-text search capabilities that many databases don't provide. 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 now.

1.2.3 History of Lucene

Lucene was originally written by Doug Cutting;[2] it was initially available for download from its home at the SourceForge web site. 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 sub-projects, which you can see at http://lucene.apache.org. With each release, the project has enjoyed increased visibility, attracting more users and developers. As of March 2009, Lucene version 2.4.1 has been released. Table 1.1 shows Lucene's release history.

Table 1.1 Lucene's release history

Version	Release date	Milestones
0.01	March 2000	First open source release (SourceForge)
1.0	October 2000	

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

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 misc. fixes
1.4.3	November 2004	Misc. fixes
1.9.0	February 2006	Binary stored fields, DateTools, NumberTools, RangeFilter, RegexQuery, Require 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, contrib/benchmark
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		
3.0		

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 various 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 n Hadoop (http://hadoop.apache.org/core), 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. At the time 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.

Lucene's popularity can be seen by its diverse usage and numerous ports to other programming languages.

1.2.4 Who uses Lucene

Lucene is in use in a surprisingly diverse and growing number of places. In addition to those organizations mentioned on the Powered by Lucene page on Lucene's Wiki, a number of other large, well-known, multinational organizations are using Lucene. It provides searching capabilities for the Eclipse IDE, the Encyclopedia Britannica CD-ROM/DVD, FedEx, the Mayo Clinic, Netflix, Linked In, Hewlett-Packard, *New Scientist* magazine, Salesforce.com, Atlassian (Jira), Epiphany, MIT's OpenCourseware and DSpace, Akamai's EdgeComputing platform, Digg, and so on. Your name may be on this list soon, too!

1.2.5 Lucene ports: Perl, Python, C++, .NET, Ruby, PHP

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 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 some of these ports in chapter 9.

In order to understand exactly how Lucene fits into a search application, including what Lucene can and cannot do, we will now review the architecture of a "typical" modern search application.

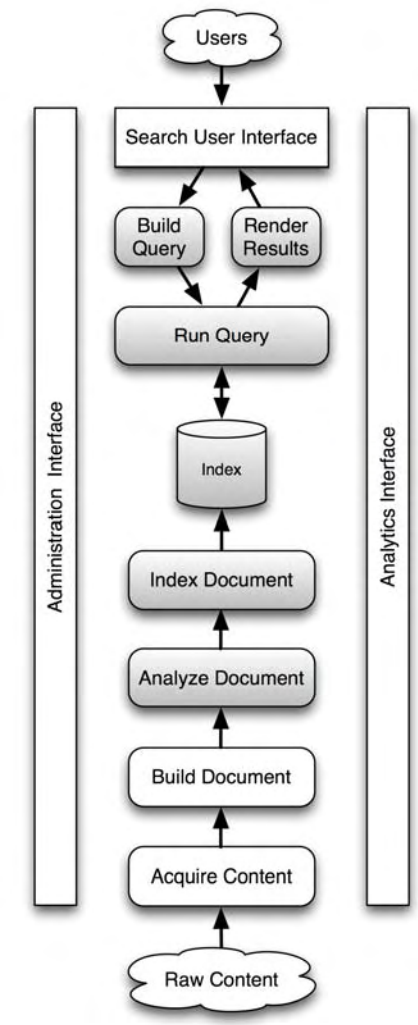


Figure 1.4 Typical components of search application; the shaded components show which parts Lucene handles.

1.3 Indexing and searching

Modern search applications have wonderful diversity. Some run quietly, as a small component deeply embedded inside an existing tool, searching a very specific set of content (local files, email messages, calendar entries, etc). Others run on a remote Web site, on a dedicated server infrastructure, interacting with many users via Web browser or mobile device, perhaps searching a product catalog or known and clearly scoped set of documents. Some run inside a company's intranet and search a massive collection of varied 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.

It's important to grasp the big picture of a search application, so that you have a clear understanding of which parts Lucene can handle and which parts you 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.

When designing your application you clearly have strong opinions on what features are necessary and how they should work. But be forewarned: modern popular web search engines (notably Google) have pretty much set the baseline requirements that all users will expect once they use your application. If your search can't meet this basic baseline users will be disappointed right from the start! Google's spell correction is amazing (we cover spell correction in section 8.XXX), the dynamic summaries with highlighting under each result are very accurate (we cover highlighting in section 8.XXX), and the response time is of course sub-second. When in doubt, look to Google for the basic features your search application must provide. Also look to Google for inspiration when you're struggling with difficult decisions. 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. Starting from the bottom of figure 1.4, and working upwards, 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.

1.3.1 Components for indexing

Suppose you needed to search a large number of files, and you wanted to be able to find files that contained 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. While this would work correctly, this approach 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. This is 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 just think of a Lucene index as a tool that allows quick word lookup.

When you look closer, indexing in fact consists of a sequence of logically distinct steps. Let's take a look at each of these steps. First you must somehow gain access to the content you need to search.

ACQUIRE CONTENT

The very first step, at the bottom of figure 1.4, is to acquire content. This process, which is often referred to as a crawler or spider, gathers and scopes the content that needs to be indexed. That may be trivial, for example if you are indexing a set of XML files that reside 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 web sites, databases, local XML files, CGI scripts running on intranet servers, etc.).

Entitlements, which means only allowing certain authenticated users to see certain documents, can complicate content acquisition, as it may require "superuser" access when acquiring the content. Furthermore, the access rights or ACLs must be acquired along with the document's content, and added to the document as additional fields that are used during searching to properly enforce the entitlements.

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

Lucene, being a core search library, does not provide any functionality to support acquiring content. This is entirely up to your application, or a separate piece of software. There are a number of open-source crawlers, for example:

- Nutch has a high-scale crawler that's suitable for discovering content by crawling Web sites.
- Grub (http://www.grub.org), a popular open-source web crawler
- Heritrix, the Internet Archive's open-source crawler
- Apache Droids project
- Aperture (http://aperture.sourceforge.net) has support for crawling web sites, filesystems and mail boxes and extracting and indexing text

If your application has scattered content, it might makes sense to use a pre-existing crawling tool. Such tools are typically designed to make it easy to load content stored in various systems, and sometimes provide pre-built connectors to common content stores, such as web sites, databases, popular content management systems, filesystems, etc. If your content source doesn't have a pre-existing connector for the crawler, it's likely straightforward 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, for example *title*, *body*, *abstract*, *author*, *url*, etc. 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, for example one email message becomes one document, or one PDF file or web page is one document. But sometimes it's less so: 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 for them, somehow linked back to the original email message, for each attachment?

Once you have 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, this is nearly a no op. But more often these days documents are binary in nature (PDF, Microsoft Office, Open Office, Adobe Flash, streaming video and audio multimedia files, etc.) or contain substantial markups which 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.

Often 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, etc, into separate fields in the document. Or perhaps you tie in content available in a separate store (for example a database) and merge this together 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 2.6, 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 3 or 4 words long.

While Lucene provides an API for building fields and documents, it does not provide any logic to build a document because it's entirely application specific. It also does not provide any document filters, although Lucene has a sister project at Apache, Tika, which handles document filtering very well. We cover Tika in chapter 7.

The textual fields in a document cannot be indexed by the search engine, 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 8.3.1) is used to derive roots

from words (for example runs, running, run all map to “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 Analysis, describing it.

Lucene provides a wide array of built-in analyzers that allow 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. Finally, you’ll be happy to know, your content is finally in a state where Lucene can index it.

INDEX DOCUMENT

During the indexing step, the document is added to the index. Lucene of course 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 how to tune how Lucene indexes your documents.

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 improves 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, whereas precision measures how well the system filters out the irrelevant documents. Appendix D describes how to use Lucene’s benchmark contrib framework to measure precision and recall of your search application.

However, 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’s 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 the user actually sees, in the web browser or desktop tool or mobile device, when she or he interacts with your search application. Believe it or not, this 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 on the user interface it will lack consumability, thus confusing your precious and fickle users who will quietly move on to your competitors.

Keep the interface simple -- don't present a bunch of advanced options on the first page. Provide a ubiquitous, prominent search box, visible everywhere, rather than requiring a 2-step process of first clicking on a search link and then entering the search text, 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 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 you may never again have the chance to earn that trust back, and your users will quietly move on.

Most of all, eat your own dog food: use your own search application extensively yourself. Enjoy what's good about it, but aggressively fix the bad things. Almost certainly your search interface should offer spell correction. Lucene has a sandbox component, spellchecker, covered in section 8.XXX, that you can use. Likewise, providing dynamic excerpts (sometimes called summaries) with hit highlighting is also important, and Lucene has a sandbox component, highlighter, covered in section 8.XXX, to handle this.

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

BUILD QUERY

When you finally manage to entice a user to using your search application, she or he issues a search request, often as the result of an HTML form 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 very simple or very complex. Lucene provides a powerful package, called `QueryParser`, to process the user's text into a query object using a common search syntax, described at http://lucene.apache.org/java/docs/queryparsersyntax.html. The query may contain Boolean operations, phrase queries (in double quotes), wildcard terms, etc. 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 to restrict which set of documents each user is allowed to search, you'll need to set up filters on the query.

Many applications will at this point also modify the search query so as to boost or filter for important things, if the boosting was not done during indexing (see section 1.3.1). Often an ecommerce 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 are 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.

Often, for an application, Lucene's default `QueryParser` is sufficient. Sometimes, you'll want to use the output of QueryParser but then add your own logic afterwards 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. Finally, you're ready to actually execute the search request to retrieve results.

SEARCH QUERY

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

There are three common models of search:

- Pure boolean model -- Documents either match or do not match the provided query, and no scoring is done. In this model there are no relevance scores associated with matching documents; 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 very 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 -- Computes 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. Lucene returns documents which you next must render in a very 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. Importantly, the user interface should also offer a clear path for follow-on searches or actions, such as clicking to the next page, refining the search in some way, finding document similar to one of the matches, etc, so that user never hits a "dead end". Lucene's core doesn't offer any components to fully render results, but the sandbox contains the highlighter package, described in section 8.xxx, for producing dynamic summaries and highlighting hits.

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

1.3.3 The rest of the search application

Believe it or not, there is in fact still quite a bit more to a typical fully functional search engine, especially a search engine running on a web site. This includes administration, in order to keep track of the application's health, configure the different components, start and stop servers, etc. It also includes analytics, allowing you to see different views into how your users are using the search feature, thus giving you the necessary guidance on what's working and what's not. Finally, for large search applications, scaleout in both size of the content you need to search and number of simultaneous search requests, 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 are 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, etc. 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, creating and restoring from backups, etc., are all examples of what an administration interface might offer.

Many search applications, such as desktop search, don't require this component, whereas a full enterprise search application may have a very complex Administration Interface. Often the interface is primarily web-based but 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 web site, 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 are an ecommerce web site, seeing how your users run searches, which searches failed to produce satisfactory results, which results users clicked on, how often a purchase followed or did not follow a search, etc, are all incredibly powerful tools to optimize the buying experience of your users. Lucene does not provide any analytics tools.

If your search application is web-based, Google Analytics is a fast way to create an analytics interface. If that's not right, you can also build your own charts based on Google's visualization API. The final topic we visit is scaleout.

SCALEOUT

One particularly tricky area is scaleout of your search application. The vast majority of search applications do not have enough content nor simultaneous search traffic to require scaleout beyond a single computer. Lucene can handle a sizable amount of content on a single modern computer. Still, such applications may want to run two identical computers to have no single point of failure (no downtime) plus a means of taking one part of the search out of production to do maintenance, upgrades, etc.

There are two dimensions to scaleout: net amount of content, and net query throughput. If you have a tremendous amount of content, you must divide it into shards, such 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 peaks you'll have to take the same index and have multiple computers search it. A front-end load balancer sends each incoming query to the least loaded back-end computer. If you require both dimensions of scaleout, as a Web scale search engine will, you combine both of these practices. Both of these require a reliable means of replicating the search index across computers.

Lucene itself provides no facilities for scaleout. However, both Solr and Nutch and others, which build additional search engine functionality on top of Lucene, do.

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 Is Lucene right for your application?

As you've seen, a modern search application can require many components. Yet, the needs of a specific application for each of these components vary greatly. Lucene covers many of these components (the gray shaded ones from figure 1.4) very well, but other components are best covered by complementary open-source software or by your own custom application logic. Or, 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 is not a direct fit, it's likely one of the open-source projects that complements or builds upon Lucene does fit. For example, Solr, a sister open-source project under the Lucene Apache umbrella, adds a server that exposes an administration interface, scaleout, indexing content from a database, and adds important end-user functionality like faceted navigation, to Lucene. Lucene is the search library while Solr provides most components of an entire search application. Chapter XXX covers Solr, and we also include a case study of how Lucid Imagination uses Solr in chapter XXX. Nutch takes scaleout even further, and is able to build indexes to handle Web-sized content collections. Projects like DBSight, Hibernate Search, LuSQL, Compass and Oracle/Lucene make searching database content very simple by handling the "Acquire Content" and "Build Document" steps seamlessly, as long as your content resides in a database. Many open-source document filters exist, for deriving textual content from binary document types. Most of the raw ingredients are there for you to pull together a powerful, fully open source search application!

Now let's dive down and 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 1.3.1. Furthermore, suppose you need to index and search files stored in a directory tree, not just in a single directory. 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 a directory tree containing text files; 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, then 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. 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: indexes .txt files

```
/**
 * This code was originally written for
 * Erik's Lucene intro java.net article
 */
public class Indexer {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      throw new Exception("Usage: java " + Indexer.class.getName()
        + " <index dir> <data dir>");
    }
    String indexDir = args[0];         //1
    String dataDir = args[1];          //2

    long start = System.currentTimeMillis();
    Indexer indexer = new Indexer(indexDir);
    int numIndexed = indexer.index(dataDir);
    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 = new FSDirectory(new File(indexDir), null);
    writer = new IndexWriter(dir,          //3
                             new StandardAnalyzer(), true,
                             IndexWriter.MaxFieldLength.UNLIMITED);
  }

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

  public int index(String dataDir) throws Exception {

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

```
    for (int i = 0; i < files.length; i++) {
      File f = files[i];

      if (!f.isDirectory() &&
          !f.isHidden() &&
          f.exists() &&
          f.canRead() &&
          acceptFile(f)) {
        indexFile(f);
      }
    }

    return writer.numDocs();                      //5
  }

  protected boolean acceptFile(File f) {                  //6
    return f.getName().endsWith(".txt");
  }

  protected Document getDocument(File f) throws Exception {
    Document doc = new Document();
    doc.add(new Field("contents", new FileReader(f)));      //7
    doc.add(new Field("filename", f.getCanonicalPath(),     //8
              Field.Store.YES, Field.Index.NOT_ANALYZED));
    return doc;
  }

  private void indexFile(File f) throws Exception {
    System.out.println("Indexing " + f.getCanonicalPath());
    Document doc = getDocument(f);
    if (doc != null) {
      writer.addDocument(doc);                            //9
    }
  }
}
```

#1 Create Lucene index in this directory
#2 Index *.txt files in this directory
#3 Create Lucene IndexWriter
#4 Close IndexWriter
#5 Return number of documents indexed
#6 Index .txt files only
#7 Index file content
#8 Index file path
#9 Add document to Lucene index

Indexer is very simple. The static main method parses (#1, #2) the incoming arguments, creates an Indexer instance, locates (#6) *.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 include creating (#3) and closing (#4) the `IndexWriter`, creating (#7, #8) the `Document`, adding (#9) the document to the index, and returning the number of documents indexed (#5) — effectively eight lines of code.

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 using the `Tika` framework.

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

RUNNING INDEXER

The simplest way to run `Indexer` is to use `ant`. You'll first have to unpack the zip file containing source code with this book, and then change to the directory "`lia2e`". If you don't see the file `build.xml` in your working directory, then 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`, 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

     [echo]       Index a directory tree of .txt files into a Lucene filesystem
     [echo]       index.  Use the Searcher target to search this index.
     [echo]
     [echo]       Indexer is covered in the "Meet Lucene" chapter.
     [echo]
    [input] Press return to continue...

    [input] Directory for new Lucene index: [indexes/MeetLucene]

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

    [input] Overwrite indexes/MeetLucene? (y, n) y
     [echo] Running lia.meetlucene.Indexer...
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/apache1.0.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/apache1.1.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/apache2.0.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/cpl1.0.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/epl1.0.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/freebsd.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/gpl1.0.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/gpl2.0.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/gpl3.0.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/lgpl2.1.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/lgpl3.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/lpgl2.0.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/mit.txt
     [java] Indexing /Users/mike/lia2e/src/lia/meetlucene/data/mozilla1.1.txt
     [java] Indexing Users/mike/lia2e/src/lia/meetlucene/data/mozilla_eula_firefox3.txt
     [java] Indexing
Users/mike/lia2e/src/lia/meetlucene/data/mozilla_eula_thunderbird2.txt
```

```
    [java] Indexing 16 files took 757 milliseconds

BUILD SUCCESSFUL
```

`Indexer` prints out 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 out 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 10. 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 a 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 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: searches a Lucene index

```
/**
 * This code was originally written for
 * Erik's Lucene intro java.net article
 */
public class Searcher {

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

```
    String indexDir = args[0];              //1
    String q = args[1];                     //2

    search(indexDir, q);
  }

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

    Directory dir = new FSDirectory(new File(indexDir), null);
    IndexSearcher is = new IndexSearcher(dir);   //3

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

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

    for(int i=0;i<hits.scoreDocs.length;i++) {
      ScoreDoc scoreDoc = hits.scoreDocs[i];
      Document doc = is.doc(scoreDoc.doc);              //7
      System.out.println(doc.get("filename"));  //8
    }

    is.close();                                 //9
  }
}
```

#1 Index directory created by Indexer
#2 Query string
#3 Open index
#4 Parse query
#5 Search index
#6 Write search stats
#7 Retrieve matching document
#8 Display filename
#9 Close index

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

#1, #2 Parse command-line arguments (index directory, query string).

#3 We use Lucene's `IndexSearcher` class 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 are done with it.

RUNNING SEARCHER

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

```
% ant Searcher

Searcher:
     [echo]
     [echo]       Search an index built using Indexer.
     [echo]
     [echo]       Searcher is described in the "Meet Lucene" chapter.
     [echo]
    [input] Press return to continue...

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

    [input] Query:  [patent]

     [echo] Running lia.meetlucene.Searcher...
     [java] Found 8 document(s) (in 11 milliseconds) that matched query 'patent':
     [java] /Users/mike/lia2e/src/lia/meetlucene/data/cpl1.0.txt
     [java] /Users/mike/lia2e/src/lia/meetlucene/data/mozilla1.1.txt
     [java] /Users/mike/lia2e/src/lia/meetlucene/data/epl1.0.txt
     [java] /Users/mike/lia2e/src/lia/meetlucene/data/gpl3.0.txt
     [java] /Users/mike/lia2e/src/lia/meetlucene/data/apache2.0.txt
     [java] /Users/mike/lia2e/src/lia/meetlucene/data/lpgl2.0.txt
     [java] /Users/mike/lia2e/src/lia/meetlucene/data/gpl2.0.txt
     [java] /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 out. 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 meta-data attached to indexed documents.

Of course, 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 all different aspects of searching, including Lucene's query syntax.

Our example indexing and searching applications demonstrate Lucene in a lot of its glory. 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 listing to look for text file 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 is a lot going on under the covers of Lucene.

To effectively leverage Lucene, it's important to understand more about 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.

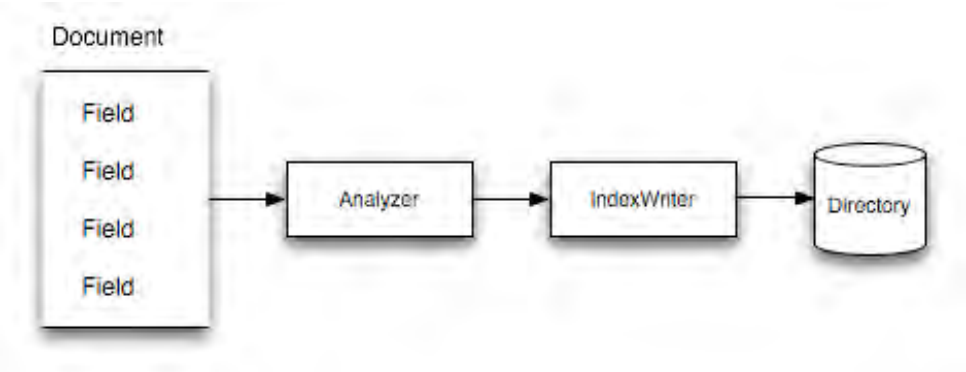


Figure 1.5 Classes used when indexing documents with Lucene.

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 about their role in Lucene. We'll use these classes throughout this book.

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 then adds, removes or updates documents in the index. You can 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 created an `FSDirectory`, which stores real files in a directory in the filesystem, and passed it to `IndexWriter`'s constructor.

The other commonly used implementation of `Directory` is a class called `RAMDirectory`. Although it exposes an interface identical to that of `FSDirectory`, `RAMDirectory` holds all its data in memory. This implementation is therefore useful for smaller indices that can be fully loaded in memory and can be destroyed upon the termination of an application. Because all data is held in the fast-access memory and not on a slower hard disk, `RAMDirectory` is suitable for situations where you need very quick access to the index, whether during indexing or searching. For instance, Lucene's developers make extensive use of `RAMDirectory` in all their unit tests: When a test runs, a fast in-memory index is created or searched; and when a test completes, the index is automatically destroyed, leaving no residuals on the disk. Of course, the performance difference between `RAMDirectory` and `FSDirectory` is less visible when Lucene is used on operating systems that cache files in memory since the index may very well fit entirely into the operating system's IO cache. You'll see both `Directory` implementations used in code snippets in this book.

`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, it should first be converted to it, as depicted in figure 2.1. Chapter 7 shows how 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

A `Document` represents a collection of fields. You can 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 meta-data associated with that document. The original source (such as a database record, a 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, that Lucene processes. The meta-data such as author, title, subject, date modified, and so on, are 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. Lucene's core does not itself handle anything but `java.lang.String` and `java.io.Reader`. Although various types of documents can be indexed and made searchable, processing them isn't as straightforward as processing purely textual content that can easily be converted to a `String` or `Reader` Java type. 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`, which is the class that actually 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 2.2.1, 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 is 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. In order 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 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:

```
IndexSearcher searcher = new IndexSearcher("/tmp/index");
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 will 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", "patent"));
TopDocs hits = searcher.search(q, 10);
```

This code instructs Lucene to find the top 10 documents that contain the word *patent* 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`, `RangeQuery`, `FilteredQuery`, and `SpanQuery`. All of these are covered in chapter 3. `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 sub-queries should have a stronger contribution to the final relevance than other sub-queries. This is described in section 3.5.9. 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 basic Lucene knowledge. You now know that Lucene is an Information Retrieval library, not a ready-to-use standalone product, and that it most certainly does not 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, that could be a good fit for your application. In addition, there are 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 quickly got to the point by showing you two 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 being 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.

2

Indexing

This chapter covers

- Conceptual index model
- Performing basic index operations
- Boosting Documents and Fields during indexing
- Indexing dates, numbers, and Fields for use in sorting search results
- Using parameters that affect Lucene's indexing performance and resource consumption
- Optimizing indexes
- Understanding concurrency, multithreading, and locking issues
- Advanced indexing functions

So you want to search files stored on your hard disk, or perhaps search your email, web pages, or even data stored in a database. Lucene can help you do that. However, before you can search something, you have to index it, and that's what you'll learn to do in this chapter.

In chapter 1, you saw a simple indexing example. This chapter goes further and teaches you about index updates, parameters you can use to tune the indexing process, and more advanced indexing techniques that will help you get the most out of Lucene. Here you'll also find information about the structure of a Lucene index, important issues to keep in mind when accessing a Lucene index with multiple threads and processes, sharing an index over the NFS file system, and the locking mechanism that Lucene employs to prevent concurrent index modification.

Despite the great detail we'll now take you through on how Lucene indexes documents, don't forget the big picture: indexing is simply a means to an end. What really matters is the search experience your applications present to its users; indexing is "merely" the necessary evil you must go through in order to enable a strong user search experience. So while there are great fun details here about indexing, your time is generally better spent working on how to improve the search experience. In nearly every

application, it's the search metrics that are far more important than the indexing metrics. That being said, clearly there are many important search metrics that require you to "do the right thing" during indexing in order to enable the search functionality.

Let's start now with Lucene's conceptual model for documents.

2.1 Conceptual document model

Before we dive into the specifics of Lucene's indexing API, let's first walk through its conceptual approach to modeling content. We'll start with Lucene's fundamental units of indexing and searching, `Documents` and `Fields`, and then move on to some important differences between Lucene and the more structured model of modern databases.

2.1.1 Documents and fields

A `Document` is Lucene's atomic unit of indexing and searching. It's actually just a container that holds one or more `Fields`, which in turn contain the "real" content. Each `Field` has a name to identify it, a text or binary value, and a series of detailed options that describes what Lucene should do with the `Field`'s value when you add the document to the index. In order to index your raw content sources, you must first translate it into Lucene's `Documents` and `Fields`. Then, at search time, the field values are searched; for example users could search for "title:lucene" to find all documents whose title field value contains the terms "lucene".

At a high level, there are three things Lucene can do with each field:

- The value may be indexed or not. A field must be indexed if you intend to search on it. Only text fields may be indexed (binary valued fields may only be stored). When a field is indexed, tokens are first derived from its text value, using a process called analysis, and then those tokens are enrolled into the index. See section XXX for all options on how the field's value is indexed.
- If it is indexed, the field may also optionally store term vectors, which is really a miniature inverted index for that one field, allowing you to retrieve all tokens for that field. This enables certain advanced use cases like searching for documents similar to an existing one (more uses are covered in Section 5.7). See section XXX for all options with how term vectors are indexed.
- Separately, the field's value may be stored, meaning a verbatim copy of the un-analyzed value is written away in the index so that it can later be retrieved. This is useful for fields you'd like to present unchanged to the user, such as the document's title or abstract. See section XXX for options on how fields are stored.

How you factor your raw content sources into Lucene's documents and fields is typically an iterative design process and very application dependent. Lucene couldn't care less which fields you use, what their names are, etc. Documents usually have quite a few fields, for example title, author, date, abstract, body text, URL, keywords, etc., might all be different fields. The common approach is to gather all textual field values into a single "contents" field, and only index that one field, but still separately store all the original fields for presentation at search time. Once you've created your document, you add it to your index. Then, at search time, you can retrieve the documents that match each query and use their stored fields to present results to the end user.

Lucene is often compared to a database, since both can store content and retrieve it later. However, there are important differences. The first once is Lucene's flexible schema.

NOTE

When you retrieve a document from the index, only the stored fields will be present. For example, fields that were indexed but not stored will not be in the document. This is frequently a source of confusion.

2.1.2 Flexible schema

Unlike a database, Lucene has no notion of a fixed global schema. In other words, each document you add to the index is a blank slate and can be completely different from the document before it: it can have whatever fields, with any indexing and storing and term vector options. It need not have the same fields as the previous document added. It can even have the same field, with different options, than in other documents.

This is actually quite powerful: It allows you to take an iterative approach to building your index. You can jump right in and index documents without having to pre-design the schema. If you change your mind about your fields, just start adding additional fields later on and then go back and re-index previously added documents, or just rebuild the index.

This also means a single index can hold `Documents` that represent different entities. For instance, you could have `Documents` that represent retail products with `Fields` such as *name* and *price*, and `Documents` that represent people with `Fields` such as `name`, `age`, and `gender`. You could also include un-searchable "meta" documents, which simply hold metadata about the index or your application, such as what time the index was last updated or which product catalog was indexed, but are never included in search results.

The second major difference between Lucene and databases is that Lucene requires you to flatten, or denormalize, your content when you index it.

2.1.3 Denormalization

One common challenge is resolving any "mismatch" between the structure of your documents versus what Lucene can represent. For example, XML can describe a recursive document structure by nesting tags within one another. A database can have an arbitrary number of joins, via primary and secondary keys, relating tables to one other. Microsoft's OLE documents can reference other documents for embedding. Yet, Lucene documents are flat. Such recursion and joins must be denormalized when creating your documents. Open source projects that build on Lucene, like Hibernate Search, Compass, LuSQL, DBSight, Browse Engine and Oracle/Lucene integration each have different and interesting approaches for handling this denormalization.

Now that you understand how Lucene models documents at a conceptual level, it's time to visit the steps of the indexing process at a high level.

2.2 Understanding the indexing process

As you saw in the chapter 1, only a few methods of Lucene's public API need to be called in order to index a document. As a result, from the outside, indexing with Lucene looks like a deceptively simple and

monolithic operation. However, behind the simple API lies an interesting and relatively complex set of operations that we can break down into three major and functionally distinct groups, as described in the following sections and depicted in Figure 2.1.

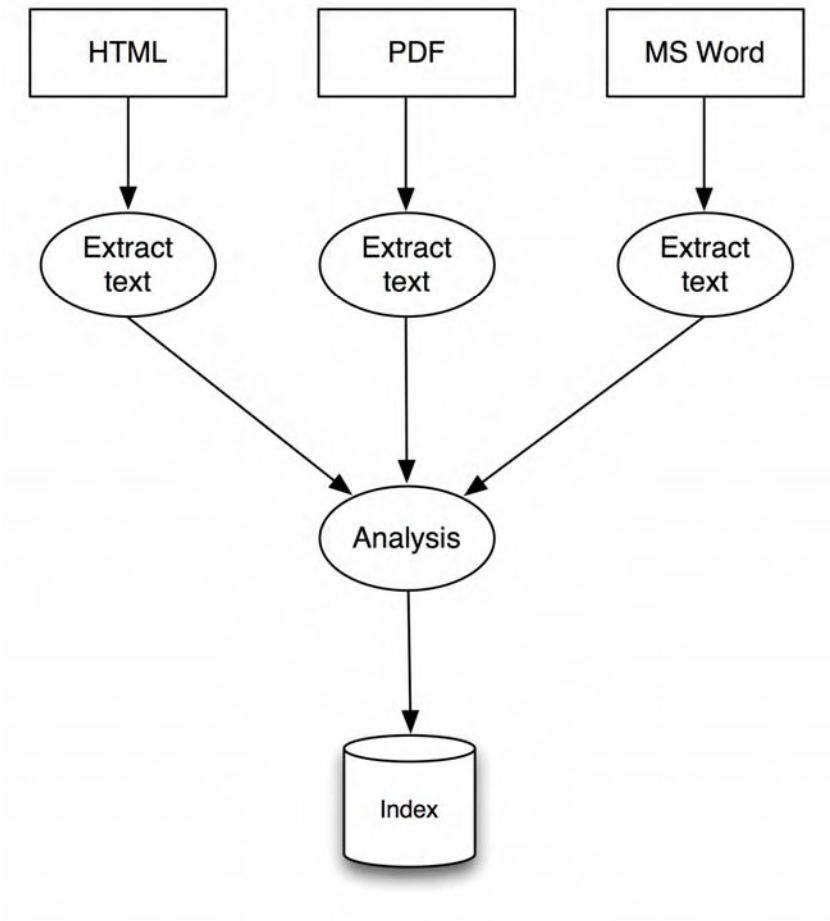


Figure 2.1 Indexing with Lucene breaks down into three main operations: extracting text from source documents, analyzing it and saving it to the index.

During indexing, the text is first extracted from the original content and used to create an instance of `Document`, containing `Field` instances hold the content. The text in the fields is then analyzed, to produce a stream of tokens. Finally, those tokens are added to the index in a segmented architecture. Let's talk about text extraction first.

2.2.1 Extracting text and creating the document

To index data with Lucene, you must extract plain text from it, the format that Lucene can digest, and then create a Lucene Document. In chapter 1, we limited our examples to indexing and searching .txt files, which allowed us to easily slurp their content and use it to populate `Field` instances. However, things aren't always that simple: the "Build Document" step from Figure 1.4 has quite a bit of work hidden behind it.

Suppose you need to index a set of manuals in PDF format. To prepare these manuals for indexing, you must first find a way to extract the textual information from the PDF documents and use that extracted text to create Lucene `Documents` and their `Fields`. No methods would accept a PDF Java type, even if such a type existed. You face the same situation if you want to index Microsoft Word documents or any document format other than plain text. Even when you're dealing with XML or HTML documents, which use plain-text characters, you still need to be smart about preparing the data for indexing, to avoid indexing the XML elements or HTML tags, and index only the real text.

The details of text extraction are in chapter 7 where we describe the Tika framework, which makes it almost too simple to extract text from documents in diverse formats. One you have the text you'd like to index, and you've created a `Document` with all `Fields` you'd like to index, all text must then be analyzed.

2.2.2 Analysis

Once you've created Lucene `Documents` populated with `Fields`, you can call `IndexWriter`'s `addDocument` method and hand your data off to Lucene to index. When you do that, Lucene first analyzes the text, a process that splits the textual data into a stream of *tokens*, and performs a number of optional operations on them. For instance, the tokens could be lowercased before indexing, to make searches case-insensitive., using Lucene's `LowerCaseFilter`. Typically it's also desirable to remove all stop words, which are frequent but meaningless tokens, from the input (for example *a*, *an*, *the*, *in*, *on,* and so on, in English text) using `StopFilter`. Similarly, it's common to process input tokens to reduce them to their roots, for example by using `PorterStemFilter` for English text (similar classes exist in Lucene's contrib packages for other languages). The combination of an original source of tokens followed by the series of filters that modify the tokens produced by that source, together make up the Analyzer. You are also free to build up your own analyzer by chaining together Lucene's token sources and filters, or your own, in customized ways.

This very important step, covered under the "Analyze Document" step in Figure 1.4, is called *analysis*. The input to Lucene can be analyzed in so many interesting and useful ways that we cover this process in detail in chapter 4. The analysis process produces a stream of tokens that are then written into the files in the index.

2.2.3 Index writing and files

After the input has been analyzed, it's ready to be added to the index. Lucene stores the input in a data structure known as an *inverted index*. This data structure makes efficient use of disk space while allowing quick keyword lookups. What makes this structure inverted is that it uses tokens extracted from input documents as lookup keys instead of treating documents as the central entities. In other words, instead of

trying to answer the question "what words are contained in this document?" this structure is optimized for providing quick answers to "which documents contain word X?"

If you think about your favorite web search engine and the format of your typical query, you'll see that this is exactly the query that you want to be as quick as possible. The core of today's web search engines are inverted indexes.

Lucene's index directory has a unique segmented architecture, which we describe next.

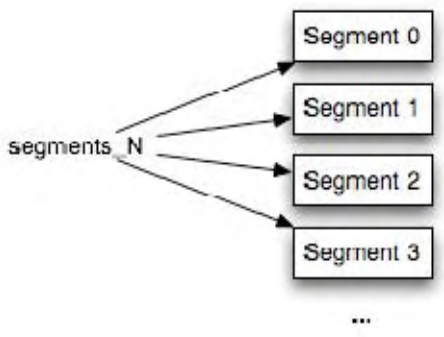


Figure 2.2 Structure of a Lucene inverted index

2.2.4 Index segments

Lucene has a rich and detailed index file format that has been carefully optimized with time. While you really don't need to know the details of this format in order to use Lucene it's still helpful to have some basic understanding a high level. If you find yourself curious about all the details, see Appendix B.

Every Lucene index consists of one or more segments, as depicted in Figure 2.2. Each segment is actually a standalone index itself, holding a subset of all indexed documents. A new segment is created whenever the writer flushes buffered added and deleted documents into the `Directory`. At search time, each segment is visited separately and the results are combined together.

Each segment, in turn, consists of multiple files, of the form `_X.<ext>`, where `X` is the segment's name and `<ext>` is the extension that identifies which part of the index that file corresponds to. There are separate files to hold the different parts of the index (term vectors, stored fields, inverted index, etc.). If you are using the compound file format, which you set with `IndexWriter.setUseCompoundFile` and is enabled by `default`, then most of these index files are collapsed into a single compound file `_X.cfs`. This reduces the number of open file descriptors during searching, at a small cost of searching and indexing performance. Chapter 10 covers this tradeoff in more detail.

There is one special file, often referred to as "the segments file", and named `segments_<N>` that references all live segments. This file is important! Lucene first opens this file, and then opens each segment referenced by it. The value <N>, called "the generation", is an integer that increases by one every time a change is committed to the index.

Naturally, over time the index will accumulate many segments, especially if you open and close your writer frequently. This is fine. Periodically, `IndexWriter` will select segments and coalesce them by merging all of them into a single new segment. The selection of segments to be merged is governed by a separate `MergePolicy`. Once merges are selected, their actual execution is done by the `MergeScheduler`. These classes are advanced topics, covered in section 2.14.5.

Let's now walk through the basic operations (add, update, delete) you do when indexing.

2.3 Basic index operations

We've covered Lucene's conceptual approach to modeling documents, and then we described the logical steps of the indexing process. Now it's time to look at some real code, using Lucene's APIs to add, remove and update documents. We start with adding documents to an index since that is the most frequent operation.

2.3.1 Adding documents to an index

Let's look at how to create a new index and add documents to it. There are two methods for adding documents:

- `addDocument(Document)` – adds the `Document` using the default analyzer, which you specified when creating the `IndexWriter`, for tokenization
- `addDocument(Document, Analyzer)` – adds the `Document` using the provided analyzer for tokenization. But be careful! In order for searches to work correctly you need the analyzer used at search time to "match" the tokens produced by the analyzers at indexing time. See section 3.XXX for more details.

The code in listing 2.1 shows all steps necessary to create a new index, and add two tiny documents. In this simplistic example, the content for the documents is contained entirely in the source code as Strings, but in the real world the content for your documents would typically come from an external source. The `setUp()` method is called by the JUnit framework before every test.

Listing 2.1 Adding documents to an index

```
public class IndexingTest extends TestCase {
  protected String[] ids = {"1", "2"};
  protected String[] unindexed = {"Netherlands", "Italy"};
  protected String[] unstored = {"Amsterdam has lots of bridges",
                                 "Venice has lots of canals"};
  protected String[] text = {"Amsterdam", "Venice"};

  private Directory directory;

  protected void setUp() throws Exception {     //A
    directory = new RAMDirectory();

    IndexWriter writer = getWriter();           //B
    for (int i = 0; i < ids.length; i++) {      //C
      Document doc = new Document();
```

```
    doc.add(new Field("id", ids[i],
                      Field.Store.YES,
                      Field.Index.NOT_ANALYZED));
    doc.add(new Field("country", unindexed[i],
                      Field.Store.YES,
                      Field.Index.NO));
    doc.add(new Field("contents", unstored[i],
                      Field.Store.NO,
                      Field.Index.ANALYZED));
    doc.add(new Field("city", text[i],
                      Field.Store.YES,
                      Field.Index.ANALYZED));
    writer.addDocument(doc);
  }
  writer.close();
}

private IndexWriter getWriter() throws IOException {    // D
  return new IndexWriter(directory, new WhitespaceAnalyzer(),
                         IndexWriter.MaxFieldLength.UNLIMITED);
}

protected int getHitCount(String fieldName, String searchString)
  throws IOException {
  IndexSearcher searcher = new IndexSearcher(directory); //E
  Term t = new Term(fieldName, searchString);
  Query query = new TermQuery(t);                        //F
  int hitCount = TestUtil.hitCount(searcher, query);     //G
  searcher.close();
  return hitCount;
}

public void testIndexWriter() throws IOException {
  IndexWriter writer = getWriter();
  assertEquals(ids.length, writer.numDocs());            //H
  writer.close();
}

public void testIndexReader() throws IOException {
  IndexReader reader = IndexReader.open(directory);
  assertEquals(ids.length, reader.maxDoc());             //I
  assertEquals(ids.length, reader.numDocs());            //I
  reader.close();
}
```

#A Run before every test
#B Create index
#C Add documents
#D Create IndexWriter
#E Create new searcher
#F Build simple single-term query
#G Get number of hits
#H Verify writer document count
#I Verify searcher document count

#A The `setUp()` method first creates a new `RAMDirectory`, to hold the index.

#B, #D Next, it creates an `IndexWriter` on this `Directory`. We created the `getWriter` convenience method since we need to get the `IndexWriter` in many places

#C Finally, `setUp()` iterates over our content, creating `Document` and `Field`s and then adds the `Document` to the index.

#E, #F, #G We create the IndexSearcher and a basic single term query and then verify the hit count is correct.

The index contains two documents, each representing a country and a city in that country, whose text is analyzed with `WhitespaceAnalyzer`. Since `setUp()` is called before each test is executed, each test runs against a freshly created index.

In the `getWriter` method we create the `IndexWriter` with 3 arguments. The first argument is the `Directory` where the index is stored. The second argument is the analyzer to use when indexing tokenized fields (analysis is covered in Chapter 4). The last argument, `MaxFieldLength.UNLIMITED`, is a required argument that tells `IndexWriter` to index all tokens in the document (section 2.7 describes this setting in more detail). `IndexWriter` will detect that there's no prior index in this `Directory` and create a new one. If there were an existing index, `IndexWriter` would simply add to it.

There are many other `IndexWriter` constructors. Some accept a `String` or `File` argument, in place of `Directory`, and will create an `FSDirectory` at that path. Others explicitly take a create argument, allowing you to force a new index to be created over an existing one. More advanced constructors allow you to specify your own `IndexDeletionPolicy` or `IndexCommit`, for expert use cases as described in section 2.9.

Once the index is created, we then construct each document using the for loop. It's actually quite simple: first we create a new empty `Document`, then one by one we add each `Field` we'd like to have on the document. Each document gets 4 fields, each with different options (`Field` options are described in section 2.4). Finally, we call `writer.addDocument` to index the document. After the for loop we close the writer, which commits all changes to the directory. We could also have called `commit()`, which would commit the changes to the directory but leave the writer open for further changes.

Notice how we use the static method `TestUtil.getHitCount` to get the number of hits for a query. TestUtil is a utility class that includes a small number of common methods that we re-use throughout the book's code examples. It's methods are quite self-explanatory, and as we use each for the first time we'll show you the source code. For example, this is the one-line method `hitCount`:

```
public static int hitCount(IndexSearcher searcher, Query query) throws IOException {
  return searcher.search(query, 1).totalHits;
}
```

Next let's look at the opposite of adding documents: deleting them.

2.3.2 Deleting documents from an index

Although most applications are more concerned with getting `Documents` into a Lucene index, some also need to remove them. For instance, a newspaper publisher may want to keep only the last week's worth

of news in its searchable indexes. Other applications may want to remove all `Documents` that contain a certain term or replace an old version of a document with a newer one. `IndexWriter` provides various methods to remove documents from an index:

- `deleteDocuments(Term)` deletes all documents containing the provided term.
- `deleteDocuments(Term[])` deletes all documents containing any of the terms in the provided array.
- `deleteDocuments(Query)` deletes all documents matching the provided query.
- `deleteDocuments(Query[])` deletes all documents matching any of the queries in the provided array.

If you intend to delete a single document by `Term`, you must ensure you've indexed a `Field` on every document, and that all `Field` values are unique so that each document can be singled out for deletion. This is the same concepts as a *primary key* column in a database table. You can of course name this `Field` anything you want ("ID" is common). This field should be indexed as an un-analyzed field (see section 2.4.1) to ensure the analyzer does not break it up into separate tokens. Then, use the field for document deletion like this:

```
writer.deleteDocument(new Term("ID", documentID));
```

Be careful with these methods! If you accidentally specify the wrong `Term`, for example a `Term` from an ordinary indexed text field instead of your unique `ID` field, you could easily and very quickly delete a great many documents from your index. In each case, the deletes are not done immediately. Instead, they are buffered in memory, just like the added documents, and periodically flushed to disk. As with added documents, you must call `commit()` or `close()` on your writer to commit the changes to the index.

When you delete a document, the disk space for that document is not immediately freed. Section 2.14.2 describes this in more detail.

Let's look at Listing 2.2 to see `deleteDocuments` in action. We created two test cases, to show the `deleteDocuments` methods and to illustrate the effect of optimizing after deletion.

Listing 2.2 Deleting documents from an index

```
public void testDeleteBeforeIndexMerge() throws IOException {
  IndexWriter writer = getWriter();
  assertEquals(2, writer.numDocs()); //1
  writer.deleteDocuments(new Term("id", "1"));  //2
  writer.commit();
  assertTrue(writer.hasDeletions());    //3
  assertEquals(2, writer.maxDoc());    //4
  assertEquals(1, writer.numDocs());   //4
  writer.close();
}

public void testDeleteAfterIndexMerge() throws IOException {
```

```
    IndexWriter writer = getWriter();
    assertEquals(2, writer.numDocs());
    writer.deleteDocuments(new Term("id", "1"));
    writer.optimize();                //5
    writer.commit();
    assertFalse(writer.hasDeletions());
    assertEquals(1, writer.maxDoc());  //6
    assertEquals(1, writer.numDocs()); //6
    writer.close();
  }
```

#1 2 docs in the index
#2 Delete first document
#3 Index contains deletions
#4 1 indexed document, 1 deleted document
#5 Optimize compacts deletes
#6 1 indexed document, 0 deleted documents

#4 Shows the difference between two methods that are often mixed up: `maxDoc()` and `numDocs()`. The former returns the total number of deleted or un-deleted documents in the index, while the latter returns the number of un-deleted `Documents` in an index. Because our index contains two `Documents`, one of which is deleted, `numDocs()` returns 1 and `maxDocs()` returns 2.

#3 The unit test also demonstrates the use of the `hasDeletions()` method to check if an index contains any `Documents` marked for deletion.

#5 In the method `testDeleteAfterIndexMerge()` we force Lucene to merge index segments, after deleting one document, by optimizing the index. Then, the `maxDoc()` method returns 1 rather than 2, because after a delete and merge, Lucene truly removes the deleted document. Only one `Document` remains in the index.

NOTE

Users often confuse the `maxDoc()` and `numDocs()` methods in `IndexWriter` and `IndexReader`. The first method, `maxDoc()` returns the total number of deleted or un-deleted documents in the index, whereas `numDocs()` returns only the number of un-deleted documents.

We've finished adding and deleting documents; now we'll visit updating documents.

2.3.3 Updating documents in the index

In many applications, after initially indexing a document there may still be further changes to it and you need to re-index it. For example, if your documents are crawled from a web server, then one way to detect that the content has changed is to look for a changed ETag HTTP header. If it's different from when you last indexed the document, then there have been some changes to the content and you should update the document in the index.

In some cases you may want to only update certain fields of the document. Perhaps the title changed but the body was unchanged. Unfortunately, while this is a frequently requested feature, Lucene cannot do that: instead, it deletes the entire previous document and then adds a new document to the index. This requires that the new document contains all fields, even unchanged ones, from the original document. `IndexWriter` provides two convenience methods to replace a document in the index:

- `updateDocument(Term, Document)` first deletes all documents containing the provided term and then adds the new document using the writer's default analyzer.
- `updateDocument(Term, Document, Analyzer)` does the same, but uses the provided analyzer instead of the writer's default analyzer.

The `updateDocument` methods are probably the most common way to do deletion since they are typically used to replace a single document in the index that has changed. Note that these methods are simply shorthand for first calling `deleteDocument(Term)` and then `addDocument`. Use `updateDocument` like this:

```
writer.updateDocument(new Term("ID", documenteId), newDocument);
```

Since `updateDocument` uses `deleteDocuments` under the hood, the same caveat applies: be sure the `Term` you pass in uniquely identifies the one document you intend to update. See Listing 2.3 for an example.

Listing 2.3 Updating indexed `Documents`

```
public void testUpdate() throws IOException {

    assertEquals(1, getHitCount("city", "Amsterdam"));

    IndexWriter writer = getWriter();

    Document doc = new Document();                    //1
    doc.add(new Field("id", "1",
                      Field.Store.YES,
                      Field.Index.NOT_ANALYZED));     //1
    doc.add(new Field("country", "Netherlands",
                      Field.Store.YES,
                      Field.Index.NO));               //1
    doc.add(new Field("contents",
                      "Amsterdam has lots of bridges",
                      Field.Store.NO,
                      Field.Index.ANALYZED));         //1
    doc.add(new Field("city", "Haag",
                      Field.Store.YES,
                      Field.Index.ANALYZED));         //1

    writer.updateDocument(new Term("id", "1"),
                          doc);                       //2
    writer.close();

    assertEquals(0, getHitCount("city", "Amsterdam"));//3
```

```
    assertEquals(1, getHitCount("city", "Haag"));     //4
  }
```

#1 Create new document with "Haag" in city field
#2 Replace original document with new version
#3 Verify old document is gone
#4 Verify new document is indexed

We create a new document that will replace the original document with id 1. Then we call `updateDocument` to replace it. We have effectively updated one of the `Documents` in the index.

We've covered the basis on how to add, delete and update documents. Now it's time to delve into all the interesting field-specific options available when creating a document.

2.4 Field options

`Field` is perhaps the most important class when indexing documents: it is the actual class that holds each value to be indexed. When you create a `Field`, there are numerous options that you specify to control exactly what Lucene should do with that field once you add the document to the index. We touched on these options at a high level at the start of this chapter, now it's time to revisit this topic and enumerate each in more detail.

The options break down into multiple independent categories, which we cover in each subsection below: indexing, storing and term vectors. After describing those, we enumerate some other values (besides `String`) that you can assign to a `Field`. Finally we enumerate the common combinations of field options, in practice.

Let's start with the options to control how the `Field`'s value is added to the inverted index.

2.4.1 Field options for indexing

The options for indexing (`Field.Index.*`) control how the text in the field will be made searchable via the inverted index. Here are the choices:

- `Index.ANALYZED` – use the analyzer to break the Field's value into a stream of separate tokens and make each token searchable. This is useful for normal text fields (body, title, abstract, etc.).
- `Index.NOT_ANALYZED` – do index the field, but do not analyze the String. Instead, treat the Field's entire value as a single token and make that token searchable. This is useful for fields that you would like to search on, but should not be broken up, such as URLs, file system paths, dates, personal names, Social Security numbers, telephone numbers, and so on. This is especially useful for enabling "exact match" searching. We indexed the file system path in `Indexer` (listing 1.1) using this option.
- `Index.ANALYZED_NO_NORMS` – an advanced variant of `Index.ANALYZED` which does not store norms information in the index. Norms record boost information in the index, but can be memory consuming when searching. Section 2.6.1 describes norms in detail.
- `Index.NOT_ANALYZED_NO_NORMS` – just like , but also do not store Norms.
- `Index.NO` – don't make this field's value available for searching at all.

When Lucene builds the inverted index, by default it stores all necessary information to implement the Vector Space model. This model requires the count of every term that occurred in the document, as well as the positions of each occurrence (needed for phrase searches). However, sometimes you know the field will be used only for pure Boolean searching and need not contribute to the relevance score. Fields that are used only for filtering, such as entitlements or date filtering, is a common example. In this case, you can tell Lucene to skip indexing the term frequency and positions by calling `Field.setOmitTermFreqAndPositions(true)`. This will save some disk space in the index, and may also speed up searching and filtering, but will silently prevent searches that require positional information, such as `PhraseQuery` and `SpanQuery`, from working. Let's move on to controlling how Lucene stores a field's value.

2.4.2 Field options for storing fields

The options for stored fields (`Field.Store.*`) determine whether the field's exact value should be stored away so that you can later retrieve it during searching:

- `Store.YES` — store the value. When the value is stored, the original String in its entirety is recorded in the index and may be retrieved by an `IndexReader`. This is useful for fields that you'd like to use when displaying the search results (such as a URL, title or database primary key). Try not to store very large fields, if index size is a concern, as stored fields consume space in the index.
- `Store.NO` – do not store the value. This is often used along with `Index.ANALYZED` to index a large text field that doesn't need to be retrieved in its original form, such as bodies of web pages, or any other type of text document.

Lucene includes a helpful utility class, `CompressionTools`, that can compress and decompress byte arrays. Under the hood it simply uses Java's builtin `java.util.Zip` classes. You can use this to compress values before storing them in Lucene. Note that while this will save space in your index, depending on how compressible the content is, it will slow down indexing and searching. If the field values are small, compression is rarely worthwhile. Finally, let's visit options for controlling how term vectors are indexed.

2.4.3 Field options for term vectors

Sometimes, when you index a document you'd like to retrieve all of its unique terms at search time. One common use is to speed up highlighting the matched tokens in stored fields. Highlighting is covered more in section 8.7. Another use is to enable a link "Find similar documents" that when clicked runs a new search using the salient terms in an original document. Yet another example is automatic categorization of documents. Section 5.7 show concrete examples of using term vectors, once they are in your index.

But what exactly are term vectors? Term vectors are something a mix of between an indexed field and a stored field. They are similar to a stored field because you can quickly retrieve all term vector fields for a given document: term vectors are keyed first by document ID. But then, they are keyed secondarily by term, meaning they store a miniature inverted index for that one document. Unlike a stored field, where the original `String` content is stored verbatim, term vectors store the actual separate terms that were produced by the `Analyzer`. This allows you to retrieve all terms, and the frequency of their occurrence

within the document and sorted in lexicographic order, for a particular indexed `Field` of a particular `Document`. Since the tokens coming out of an analyzer also have position and offset information (see section 4.2.1), you can choose separately whether these details are also stored in your term vectors by passing these constants as the 4th argument to the `Field` constructor:

- `TermVector.YES` – record the unique terms that occurred, and their counts, in each document, but do not store any positions or offsets information.
- `TermVector.WITH_POSITIONS` – record the unique terms and their counts, and also the positions of each occurrence of every term, but no offsets.
- `TermVector.WITH_OFFSETS` – record the unique terms and their counts, with the offsets (start & end character position) of each occurrence of every term, but no positions.
- `TermVector.WITH_POSITIONS_OFFSETS` – store unique terms and their counts, along with positions and offsets.
- `TermVector.NO` – do not store any term vector information.

Note that you cannot index term vectors unless you've also turned on indexing for the field. Stated more directly: if `Index.NO` is specified for a field, then you must also specify `TermVector.NO`.

We're done with the detailed options to control indexing, storing and term vectors. Now let's see how you can create a `Field` with values other than `String`.

2.4.4 Other Field values

There are a few other constructors for Field that allow you to use values other than `String`:

- `Field(String name, Reader value, TermVector vector)` uses a `Reader` instead of a `String` to represent the value. In this case the value cannot be stored (hardwired to `Store.NO`) and is always analyzed and indexed (`Index.ANALYZED`).
- `Field(String name, TokenStream tokenStream, TermVector TermVector)` allows you to pre-analyze the field value into a `TokenStream`. Likewise, such fields are not stored and are always analyzed and indexed.
- `Field(String name, byte[] value, Store store)` is used to store a binary field. Such fields are never indexed (`Index.NO`), and have no term vectors (`TermVector.NO`). The store argument must be `Store.YES`.

It should be clear by now that Field is quite a rich class and exposes a number of options to express to Lucene precisely how its value should be handled. Let's see how these options are typically combined in practice.

2.4.5 Field option combinations

Table 2.2 provides a summary of different field characteristics, showing you how fields are created, along with common usage examples.

Index	Store	TermVector	Example usage
NOT_ANALYZED	YES	NO	Identifiers (file names, primary keys), Telephone and Social Security numbers, URLs, personal names, Dates
ANALYZED	YES	WITH_POSITIONS_OFFSETS	Document title, document abstract
ANALYZED	NO	WITH_POSITIONS_OFFSETS	Document body
NO	YES	NO	Document type, database primary key (if not used for searching)
NOT_ANALYZED	NO	NO	Hidden keywords

You've now seen all the options for the three categories (indexing, storing and term vectors) you can use to control how Lucene handles a field. These options can nearly be set independently, resulting in a number of possible combinations. Table 2.2 lists some commonly used options and their example usage, but remember you are free to set the options however you'd like!

Now that we're done with the exhaustive indexing options for fields, we'll change gears and talk about how to handle a field that has more than one value.

2.5 Multi-valued Fields

Suppose your documents have an author field, but sometimes there's more than one author for a document. One way to handle this would be to loop through all the authors, appending them into a single `String`, which you could then use to create a Lucene `Field`. Another, perhaps more elegant way is to just keep adding the same `Field` with different value, like this:

```
Document doc = new Document();
for (int i = 0; i < authors.length; i++) {
  doc.add(new Field("author", authors[i],
                    Field.Store.YES,
                    Field.Index.ANALYZED));
}
```

This is perfectly acceptable and encouraged, as it's a natural way to represent a field that legitimately has multiple values. Internally, whenever multiple fields with the same name appear in one document, both the inverted index and term vectors will logically append the tokens of the field to one another, in the order the fields were added. There are some advanced options during analysis that control certain important details of this appending; see section 4.2.5 for details. Unlike indexing, when the fields are

stored they are stored separately in order in the document, so that when you retrieve the Document at search time you'll see multiple Field instances.

Sometimes, certain document or fields are known to be important; Lucene offers a feature called *boosting* to express this.

2.6 Boosting Documents and Fields

Not all Documents and Fields are created equal—or at least you can make sure that's the case by selectively boosting Documents or Fields. Imagine you have to write an application that indexes and searches corporate email. Perhaps the requirement is to give company employees' emails more importance than other email messages when sorting search results. How would you go about doing this?

Document *boosting* is a feature that makes such a requirement simple to implement. By default, all Documents have no boost—or, rather, they all have the same boost factor of 1.0. By changing a Document's boost factor, you can instruct Lucene to consider it more or less important with respect to other Documents in the index, when computing relevance. The API for doing this consists of a single method, setBoost(float), which can be used as follows (note that certain methods below, like getSenderEmail and isImportant, are not defined in this fragment, but are included in the full examples sources available for download on http://manning.com):

```
public static final String COMPANY_DOMAIN = "example.com";
public static final String BAD_DOMAIN = "yucky-domain.com";

Document doc = new Document();
String senderEmail = getSenderEmail();
String senderName = getSenderName();
String subject = getSubject();
String body = getBody();
doc.add(new Field("senderEmail", senderEmail,
                  Field.Store.YES,
                  Field.Index.NOT_ANALYZED));
doc.add(new Field("senderName", senderName,
                  Field.Store.YES,
                  Field.Index.ANALYZED));
doc.add(new Field("subject", subject,
                  Field.Store.YES,
                  Field.Index.ANALYZED));
doc.add(new Field("body", body,
                  Field.Store.NO,
                  Field.Index.ANALYZED));
String lowerDomain = getSenderDomain().toLowerCase();
if (isImportant(lowerDomain)) {
  doc.setBoost(1.5F);     //1
}
else if (isUnimportant(lowerDomain)) {
  doc.setBoost(0.1F);    //2
}
writer.addDocument(doc);
```

#1 Good domain boost factor: 1.5
#2 Bad domain boost factor: 0.1

In this example, we check the domain name of the email message sender to determine whether the sender is a company employee.

#1 When we index messages sent by an important domain name (say, the company's employees), we set their boost factor to 1.5, which is greater than the default factor of 1.0.

#2 When we encounter messages from a sender associated with a fictional bad domain, as checked by `isUnimportant`, we label them as nearly insignificant by lowering their boost factor to 0.1.

Just as you can boost `Documents`, you can also boost individual `Fields`. When you boost a `Document`, Lucene internally uses the same boost factor to boost each of its `Fields`. Imagine that another requirement for the email-indexing application is to consider the subject `Field` more important than the `Field` with a sender's name. In other words, search matches made in the subject `Field` should be more valuable than equivalent matches in the `senderName Field` in our earlier example. To achieve this behavior, we use the `setBoost(float)` method of the `Field` class:

```
Field senderNameField = new Field("senderName", senderName,
                                  Field.Store.YES,
                                  Field.Index.ANALYZED);
Field subjectField = new Field("subject", subject,
                               Field.Store.YES,
                               Field.Index.ANALYZED);
subjectField.setBoost(1.2F);
```

In this example, we arbitrarily picked a boost factor of 1.2, just as we arbitrarily picked `Document` boost factors of 1.5 and 0.1 earlier. The boost factor values you should use depend on what you're trying to achieve; you may need to do a bit of experimentation and tuning to achieve the desired effect, but remember when you want to change the boost on a field or document you will have to fully remove and then re-add the entire document, or use the `updateDocument` method, which does the same thing.

It's worth noting that shorter `Fields` have an implicit boost associated with them, due to the way Lucene's scoring algorithm works. While indexing, IndexWriter consults the Similarity.lengthNorm method to perform this computation. To override this logic, you can implement your own Similarity class, and tell IndexWriter to use it by calling its setSimilarity method. Boosting is, in general, an advanced feature that many applications can work very well without, so tread carefully!

`Document` and `Field` boosting come into play at search time, as you'll learn in section 3.5.9. Lucene's search results are ranked according to how closely each `Document` matches the query, and each matching `Document` is assigned a score. Lucene's scoring formula consists of a number of factors, and the boost factor is one of them.

Alternatively, you may want to boost only at search time, as described in Section 6.1 "custom sorting". The benefit of this approach is it's far more dynamic (every search could choose to boost or not to boost). At search time, you can even expose the choice to the user, such as a checkbox that asks "boost recently modified documents?". Still, you should be careful: too much boosting, especially without corresponding transparency in the search user interface explaining that certain documents were boosted, can quickly and

catastrophically erode the user's trust. But how are these boosts stored in the index? This is what norms are for.

2.6.1 Norms

During indexing, all sources of index-time boosts are combined together into a single floating point number for each indexed field in the document. The document may have its own boost; each field may have a boost; finally, Lucene computes a boost based on the number of tokens in the field (shorter fields have a higher boost). These boosts are combined and then compactly encoded (quantized) into a single byte, which is stored per field per document. During searching, norms for any field being searched are loaded into memory, decoded back into a floating point number and used to compute the relevance score.

Even though norms are initially computed during indexing, it's also possible to change them after the fact using `IndexReader`'s setNorm method. This is a very advanced method, requiring you to recompute your own norm, but it's a potentially powerful way to factor in highly dynamic boost factors such as document recency or click-through popularity.

One problem often encountered with norms is that their high memory usage at search time. This is because the full array of norms, which requires one byte per document per separate field searched, is loaded into RAM. For a large index with many fields per document, this can quickly add up to a lot of RAM. Fortunately, you can easily turn norms off by calling `Field.setOmitNorms(true)` before indexing the document containing that field. This will potentially affect scoring, because no boost information will be used during searching, but it's possible the effect is trivial, especially when the fields tend to be roughly the same length and you're not doing any boosting on your own.

But beware: if you decide partway through indexing to turn norms off, you must rebuild the entire index because if even a single document has that field indexed with norms enabled, then through segment merging this will "spread" such that all documents consume one byte even if they had disabled norms. This is because Lucene does not use sparse storage for norms. Next let's discuss storing fields in Lucene's index.

We'll switch gears now and talk about how to index commonly encountered field values, including dates, times, numbers and fields you plan to sort on.

2.7 Indexing dates & times

To Lucene, every field is a `String`. However, in the real world we encounter many different an interesting types like dates, integers, floating point numbers, etc. Fortunately, there is helpful support to provide type-specific behavior despite the fact that internally Lucene treats all tokens as string. When converting a type to a string, it's necessary to choose a format where a string sort in natural string order "matches" the corresponding sort order of the original type. Let's first look at Dates, which Lucene conveniently provides internal support for.

Email messages include sent and received dates, files have several timestamps associated with them, and HTTP responses have a `Last-Modified` header that includes the date of the requested page's last modification. Chances are, like many other Lucene users, you'll need to index dates. Lucene comes equipped with a `DateTools` class, to facilitate conversion from `Date` to `String` and vice/versa which makes date indexing easy. For example, to index today's date, you can do this:

```
Document doc = new Document();
doc.add(new Field("indexDate",
         DateTools.dateToString(new Date(), DateTools.Resolution.DAY),
         Field.Store.YES,
         Field.Index.NOT_ANALYZED);
```

DateTools simply formats the date and time in the format YYYYMMDDhhmmss, stripping off the suffix when you don't require that much resolution. For example, if you use Resolution.DAY, then June 2 1970 will be converted to the string 19700602. It's easy to see that this format ensures that sorting by string value will also match sorting by the original date value.

Handling dates this way is simple, but you must be careful when using this method: the resolution argument allows you to specify what parts of the date are significant to your application, ranging from `Resolution.MILLISECOND` up to `Resolution.YEAR`. The finer the resolution, the more distinct terms will be indexed, which as you'll see in section 6.5, can cause performance problems for certain types of queries. In practice, you rarely need dates that are precise down to the millisecond, at least to query on. Generally, you can round dates to an hour or even to a day.

NOTE

If you only need the date for searching, and not the timestamp, index as `new Field("date", "YYYYMMDD", Field.Store.YES, Field.Index.NOT_ANALYZED)`. If the full timestamp needs to be preserved, but only for retrieval and presentation (not searching), index a second `Field` using the finer resolution. This will enable far more efficient date-only searching while not losing the time portion of the date for presentation.

If you choose to format dates or times in some other manner, take great care that the `String` representation is lexicographically orderable; doing so allows for sensible date-range queries. A benefit of indexing dates in YYYYMMDD format is the ability to query by year only, by year and month, or by exact year, month, and day. To query by year only, use a `PrefixQuery` for YYYY, for example. We discuss `PrefixQuery` further in section 3.4.3.

Handling dates was wonderfully simple, thanks to Lucene's builtin `DateTools` class. Let's see how to index numeric fields next.

2.8 Indexing numbers

There are two common scenarios in which number indexing is important. In one scenario, numbers are embedded in the text to be indexed, and you want to make sure those numbers are indexed so that you can use them later in searches. For instance, your documents may contain sentences like "Be sure to include Form 1099 in your tax return": You want to be able to search for the number 1099 just like you can search for the phrase "tax return" and retrieve the document that contains the exact number.

In the other scenario, you have `Fields` that contain only numeric values, and you want to be able to index them and use them for searching. Moreover, you may want to perform range queries using such `Fields`. For example, if you're indexing email messages, one of the possible index `Fields` could hold the

message size, and you may want to be able to find all messages of a given size; or, you may want to use range queries to find all messages whose size is in a certain range. You may also have to sort results by size.

Lucene can index numeric values by treating them as strings internally. If you need to index numbers that appear in free-form text, the first thing you should do is pick an `Analyzer` that doesn't discard numbers. As we discuss in section 4.3, `WhitespaceAnalyzer` and `StandardAnalyzer` are two possible candidates. If you feed them a sentence such as "Be sure to include Form 1099 in your tax return," they extract 1099 as a token and pass it on for indexing, allowing you to later search for 1099. On the other hand, `SimpleAnalyzer` and `StopAnalyzer` discard numbers from the token stream, which means the search for 1099 won't match any documents. If in doubt, use Luke, which is a wonderful tool for inspecting all details of a Lucene index, to check whether numbers survived your analyzer and were added to the index. Luke is described in more detail in Section 8.2.2.

`Field`s whose sole value is a number don't need to be analyzed, so they should be indexed with `Field.Index.NOT_ANALYZED`. However, before just adding their raw values to the index, you need to manipulate them a bit, in order for range queries to work as expected. When performing range queries, Lucene uses lexicographical values of `Field`s for ordering. Consider three numeric `Field`s whose values are 7, 71, and 20. Although their natural order is 7, 20, 71, their lexicographical order is 20, 7, 71. A simple and common trick for solving this inconsistency is to prepad numeric `Field`s with zeros, like this: 007, 020, 071. Notice that the natural and the lexicographical order of the numbers is now consistent. For more details about searching numeric `Field`s, see section 6.3.3.

NOTE

When you index `Field`s with numeric values, prefix them with zeros if you want to use them for range queries

Many applications allow their users to sort on certain fields instead of the default score. We now describe how such fields must be indexed.

2.9 Indexing Fields for sorting

When returning documents that match a search, Lucene orders them by their score by default. Sometimes, however, you need to order results using some other criteria. For instance, if you're searching email messages, you may want to order results by sent or received date, or perhaps by message size. If you want to be able to sort results by a `Field` value, you must add it as a `Field` that is indexed but not analyzed, using `Field.Index.NOT_ANALYZED`. `Field`s used for sorting must be convertible to `Integer`s, `Float`s, or `String`s:

```
new Field("size", "4096", Field.Store.YES, Field.Index.NOT_ANALYZED);
new Field("price", "10.99", Field.Store.YES, Field.Index.NOT_ANALYZED);
new Field("author", "Arthur C. Clark", Field.Store.YES, Field.Index.NOT_ANALYZED);
```

Although we've indexed numeric values as `String`s, you can specify the correct `Field` type (such as `Integer` or `Long`) at sort time, as described in section 5.1.7.

NOTE

`Field`s used for sorting have to be indexed and must contain one token per document. Typically this means using `Field.Index.NOT_ANALYZED`, but if your analyzer will always produce only one token, such as `KeywordAnalyzer` (covered in section 4.XXX), that will work as well.

Now we visit one final topic with fields, *truncation*.

2.10 Field truncation

Some applications index documents whose sizes aren't known in advance. As a safety mechanism to control the amount of RAM and hard-disk space used, they may need to limit the amount of input they index. It's also possible that a large binary document is accidentally mis-classified as a text document, or contains binary content embedded in it that your document filter failed to process, which quickly adds many absurd binary terms to your index, much to your horror. Other applications deal with documents of known size but want to index only a portion of each document. For example, you may want to index only the first 200 words of each document.

To support these diverse cases, `IndexWriter` allows you to truncate per-`Field` indexing such that only the first N terms are indexed for an analyzed field. When you instantiate IndexWriter, you must pass in a MaxFieldLength instance expressing this limit. MaxFieldLength provides two convenient default instances: `MaxFieldLength.UNLIMITED`, which means no truncation will take place, and `MaxFieldLength.LIMITED`, which means fields are truncated at 10,000 terms. You can also instantiate MaxFieldLength with your own limit.

After creating IndexWriter, you may alter the limit at any time by calling `setMaxFieldLength`, or retrieve the limit with `getMaxFieldLength`. However, any documents already indexed will have been truncated at the previous value: changes to maxFieldLength are not retroactive. If there are multiple Field instances by the same name, the truncation applies separately to each of them, meaning each field has its first N terms indexed. If you're curious about how often the truncation is kicking in, call `IndexWriter.setInfoStream(System.out)` and search for any lines that saying "maxFieldLength N reach for field X" (NOTE: the infoStream also receives many other diagnostic details).

Think carefully before using any field truncation! It means that only the first N terms are available for searching, and any text beyond the Nth term is completely ignored. Eventually users will notice that your search engine fails to find certain document in certain situations. There have been many times when someone asks the Lucene user's list "why doesn't this search find this document", and the answer is inevitably "you'll have to increase your `maxFieldLength`".

NOTE

Use `maxFieldLength` sparingly! Since truncation means some documents' text will be completely ignored, your users will eventually discover that your search fails to find some documents. This will

very quickly erode their trust in your application ("what else can't it find?"), which can be catastrophic to your user base and perhaps your whole business, if search is its core. User trust is the most important thing to protect in your application.

We're done visiting all the interesting things you can do with `Fields`. Next we describe the optimization process.

2.11 Optimizing an index

When you index documents, especially many documents or using multiple sessions with `IndexWriter`, you'll invariably create an index that has many separate segments. When you search the index, Lucene must search each segment separately and then combine the results. While this works flawlessly, applications that handle large indexes should see search performance improvements by optimizing the index, which merges many segments down to one or a few segments. An optimized index also consumes fewer file descriptors during searching. After describing the optimization process and the available methods, we'll talk about disk space consumed during optimization.

NOTE

Optimizing only improves searching speed, not indexing speed.

It's entirely possible that you get excellent search throughput without ever optimizing, so be sure to first test whether you even need to consider optimizing. `IndexWriter` exposes 4 methods to optimize:

- `optimize()` reduces the index to a single segment, not returning until the operation is finished.
- `optimize(int maxNumSegments)`, also known as "partial optimize", reduces the index to at most. Typically, the final merge down to one segment is the most costly, so optimizing to say 5 segments should be quite a bit faster than optimizing down to 1 segment, allowing you to tradeoff optimize time versus search speed.
- `optimize(boolean doWait)` is just like optimize, except if doWait is false then the call returns immediately while the necessary merges take place in the background.
- `optimize(int maxNumSegments, boolean doWait)` is a partial optimize that runs in the background if doWait is false.

Remember that index optimization involves a lot of disk IO, so use it judiciously. It is a tradeoff of a large one-time cost, for faster searching. If you only update your index rarely, and do lots of searching between updates, this tradeoff is worthwhile. Listings 2.4 and 2.5 show the difference in index files between an unoptimized and an optimized non-compound-file index, respectively. The first index had 2 segments, and after optimize() this was reduced to 1 segment. Let's look at disk space consumed during optimization.

Listing 2.4 Index structure of an unoptimized non-compound-file index with 2 segments

```
-rw-rw-rw- 1 mike users 7743790 Feb 29 05:28 _0.fdt
-rw-rw-rw- 1 mike users    3200 Feb 29 05:28 _0.fdx
```

```
-rw-rw-rw- 1 mike users      33 Feb 29 05:28 _0.fnm
-rw-rw-rw- 1 mike users  602012 Feb 29 05:28 _0.frq
-rw-rw-rw- 1 mike users    1204 Feb 29 05:28 _0.nrm
-rw-rw-rw- 1 mike users 1337462 Feb 29 05:28 _0.prx
-rw-rw-rw- 1 mike users   10094 Feb 29 05:28 _0.tii
-rw-rw-rw- 1 mike users  737331 Feb 29 05:28 _0.tis
-rw-rw-rw- 1 mike users    2949 Feb 29 05:28 _0.tvd
-rw-rw-rw- 1 mike users 6294227 Feb 29 05:28 _0.tvf
-rw-rw-rw- 1 mike users    6404 Feb 29 05:28 _0.tvx
-rw-rw-rw- 1 mike users 4583789 Feb 29 05:28 _1.fdt
-rw-rw-rw- 1 mike users    3200 Feb 29 05:28 _1.fdx
-rw-rw-rw- 1 mike users      33 Feb 29 05:28 _1.fnm
-rw-rw-rw- 1 mike users  405527 Feb 29 05:28 _1.frq
-rw-rw-rw- 1 mike users    1204 Feb 29 05:28 _1.nrm
-rw-rw-rw- 1 mike users  790904 Feb 29 05:28 _1.prx
-rw-rw-rw- 1 mike users    7499 Feb 29 05:28 _1.tii
-rw-rw-rw- 1 mike users  548646 Feb 29 05:28 _1.tis
-rw-rw-rw- 1 mike users    2884 Feb 29 05:28 _1.tvd
-rw-rw-rw- 1 mike users 3933404 Feb 29 05:28 _1.tvf
-rw-rw-rw- 1 mike users    6404 Feb 29 05:28 _1.tvx
-rw-rw-rw- 1 mike users      20 Feb 29 05:28 segments.gen
-rw-rw-rw- 1 mike users      78 Feb 29 05:28 segments_3
```

Listing 2.5 Index structure of a fully optimized non-compound-file index with a single segment

```
-rw-rw-rw- 1 mike users 12327579 Feb 29 05:29 _2.fdt
-rw-rw-rw- 1 mike users     6400 Feb 29 05:29 _2.fdx
-rw-rw-rw- 1 mike users       33 Feb 29 05:29 _2.fnm
-rw-rw-rw- 1 mike users  1036074 Feb 29 05:29 _2.frq
-rw-rw-rw- 1 mike users     2404 Feb 29 05:29 _2.nrm
-rw-rw-rw- 1 mike users  2128366 Feb 29 05:29 _2.prx
-rw-rw-rw- 1 mike users    14055 Feb 29 05:29 _2.tii
-rw-rw-rw- 1 mike users  1034353 Feb 29 05:29 _2.tis
-rw-rw-rw- 1 mike users     5829 Feb 29 05:29 _2.tvd
-rw-rw-rw- 1 mike users 10227627 Feb 29 05:29 _2.tvf
-rw-rw-rw- 1 mike users    12804 Feb 29 05:29 _2.tvx
-rw-rw-rw- 1 mike users       20 Feb 29 05:29 segments.gen
-rw-rw-rw- 1 mike users       53 Feb 29 05:29 segments_3
```

2.9.1 Optimizing disk space requirements

Many users are surprised by how much temporary disk space is required by optimize. Because Lucene must merge segments together, while the merge is running, temporary disk space is used to hold the files for the new segment. But the old segments cannot be removed until the merge is complete. This means, roughly, you should expect the size of your index to double, temporarily, during optimize. Furthermore, if you have a reader open on the index before optimize starts, that reader will tie up another 1X of the index size, so expect your index to grow to 3X its normal size during optimize. Once optimize completes, and you've closed all open readers, disk usage will fall back to a lower level than the starting size of the index. Section 10.3.1 describes overall disk usage of Lucene in more detail.

Let's look at some `Directory` implementations other than `FSDirectory`.

2.12 Other Directory Implementations

Recall from chapter 1 that the purpose of Lucene's abstract `Directory` class is to present a simple file-like storage API. Whenever Lucene needs to write to or read from files in the index, it uses the `Directory` methods to do so.

The most commonly used `Directory` implementation is `FSDirectory`, which simply stores files in a real filesystem directory. Lucene also provides a `Directory` implementation, called `RAMDirectory`, that stores all "files" in memory instead of on disk. This is useful in cases where the index is small enough to fit in available memory and where the index is easily and quickly regenerated from the source documents. For example, Lucene's unit tests make extensive use of `RAMDirectory` to create short-lived indices for testing. To build a new index in `RAMDirectory`, simply instantiate your writer like this:

```
Directory ramDir = new RAMDirectory();
IndexWriter writer = new IndexWriter(ramDir, analyzer,
IndexWriter.MaxFieldLength.UNLIMITED);
```

You can then use the writer as you normally would to add, delete or update documents. Just remember that once the JVM exits, your index is gone!

Alternatively, you can load the contents of another `Directory otherDir` into `RAMDirectory` like this:

```
Directory ramDir = new RAMDirectory(otherDir);
```

This is typically used to speed up searching of an existing on-disk index when it is small enough. We discuss this more in section 3.2.3. If you'd like to do the reverse (copying an index from `RAMDirectory` into another `Directory`), use this static method:

```
Directory.copy(ramDir, otherDir)
```

But beware that this blindly replaces any existing files in `otherDir`, and you must ensure no `IndexWriter` is open on the source directory since the copy method does not do any locking. If the `otherDir` already has an index present, and you'd like to add in all documents from ramDir, keeping all documents already indexed in `otherDir`, use `IndexWriter.addIndexesNoOptimize` instead:

```
IndexWriter writer = new IndexWriter(otherDir, analyzer,
                                     IndexWriter.MaxFieldLength.UNLIMITED);
writer.addIndexesNoOptimize(new Directory[] {ramDir});
```

There are other addIndexes methods in `IndexWriter`, however, each of them does their own optimize which likely you don't need or want.

In past versions of Lucene, it was beneficial to control memory buffering yourself by first batch indexing into a `RAMDirectory`, and then periodically adding the index into an index stored on disk. However, as of Lucene 2.3, `IndexWriter` makes very efficient use of memory for buffering changes to the index and this is no longer a win. See section 10.1.2 for other ways to improve indexing throughput.

There is also `MMapDirectory`, which is similar to `FSDirectory` in that it stores files in the file system. The difference is instead of using normal IO to access the files, it uses memory mapping. Beware that on a 32 bit JVM this requires that your total index size fits into the available address space (ie, less than 4 GB).

Finally, `NIOFSDirectory` is very compelling option. It's similar to `FSDirectory` in that all files are stored in a real filesystem directory. The difference is that it uses java's native io package (`java.nio.*`) when reading from the files, which allows it to avoid locking that the normal `FSDirectory` must do when multiple threads read from the same file. If your application has many threads sharing a single searcher it's likely switching to `NIOFSDirectory` will improve your query throughput. However, because of known problems with the implementation of `java.nio.*` under Sun's JRE on Windows, `NIOFSDirectory` offers no gains on that platform and is likely slower than `FSDirectory`. On all other platforms it's likely faster.

2.13 Concurrency, thread-safety, and locking issues

In this section, we cover three closely related topics: concurrent index access, thread-safety of `IndexReader` and `IndexWriter`, and the locking mechanism that Lucene uses to enforce these rules. These issues are often misunderstood by users new to Lucene. Understanding these topics is important, because it will eliminate surprises that can result when your indexing application starts serving multiple users simultaneously or when it has to deal with a sudden need to scale by parallelizing some of its operations. Lucene's concurrency rules are simple but should be strictly followed:

- Any number of `IndexReaders` may be open at once on a single index. It doesn't matter if these readers are in the same JVM or multiple JVMs, or on the same computer or multiple computers. Remember, though, that within a single JVM it's best for resource utilization and performance reasons to share a single `IndexReader` instance for a given index using multiple threads. For instance, multiple threads or processes may search the same index in parallel.
- Only a single writer may be open on an index at once. Lucene uses a write lock file to enforce this (see section 2.13 below). As soon as an `IndexWriter` is created, a write lock is obtained. Only when that `IndexWriter` is closed is the write lock released. Note that if you use `IndexReader` to make changes to the index, for example to change norms (section 2.6.1) or delete documents (section 2.14.1), then that `IndexReader` acts as a writer: it will obtain the write lock on the first method that makes a change, only releasing it once closed.
- `IndexReaders` may be open even while a single `IndexWriter` is making changes to the index. Each `IndexReader` will always show the index as of the point-in-time that it was opened. It will not see any changes being done by the `IndexWriter`, until the writer commits and the reader is re-opened.
- Any number of threads can share a single instance of `IndexReader` or `IndexWriter`. These classes are not only thread safe but also thread friendly, meaning they generally scale well as you add threads, assuming your hardware has concurrency, because the amount of synchronized code inside these classes is kept to a minimum. Figure 2.8 depicts such a scenario. Sections 10.2.1 and 10.2.2 describe issues around using multiple threads for indexing and searching.

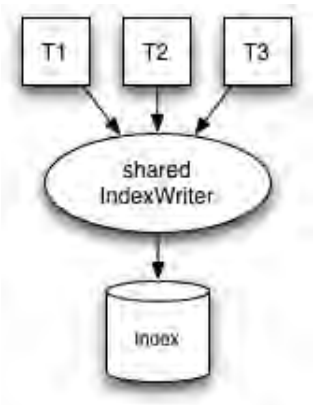


Figure 2.8 A single `IndexWriter` can be shared by multiple threads.

2.13.1 Index locking

In order to enforce a single writer at a time, which means an `IndexWriter` or an `IndexReader` doing deletions or changing norms, Lucene uses a file-based lock: If the file `write.lock` exists in your index directory, a writer currently has the index open. Any attempt to create another writer on the same index will hit a `LockObtainFailedException`.

Lucene allows you to change your locking implementation: any subclass of `LockFactory` can be set as your locking implementation by calling `Directory.setLockFactory`. Be sure to call this before opening an `IndexWriter` on that `Directory` instance. Table 2.2 lists the core locking implementations provided with Lucene.

Table 2.2 Locking implementations provided by Lucene

Locking Class Name	Description
`SimpleFSLockFactory`	This is the default locking for `FSDirectory`, using the `File.createNewFile` API. Beware that if the JVM crashes or IndexWriter is not closed before the JVM exits, this may leave a leftover `write.lock` file which you must manually remove.
`NativeFSLockFactory`	Uses java.nio native OS locking, which will never leave leftover lock files when the JVM exits. However, this locking implementation may not work correctly over certain shared file systems, notably NFS.
`SingleInstanceLockFactory`	Creates a lock entirely in memory. This is the

	default locking implementation for `RAMDirectory`. Use this when you know all `IndexWriters` will be instantiated in a single JVM.
`NoLockFactory`	Disables locking entirely. Be careful! Only use this when you are absolutely certain that Lucene's normal locking safeguard is not necessary, for example when using a private `RAMDirectory` with a single `IndexWriter` instance.

You can also easily implement your own locking implementation, but take care: if you have a bug and accidentally allow two writers access to the same index at once, you will easily corrupt your index. There is a simple tool, `LockStressTest`, which can be used in conjunction with `LockVerifyServer` and `VerifyingLockFactory` to verify that a given locking implementation is functioning properly. These classes are in the `org.apache.lucene.store` package; see their javadocs for how to use them. If you are unsure whether your new lock factory is working properly, use the `LockStressTest` to find out.

You should be aware of two additional methods related to locking:

- `IndexWriter`'s `isLocked(Directory)`—Tells you whether the index specified in its argument is locked. This method can be handy when an application needs to check whether the index is locked before attempting to create an `IndexWriter`.
- `IndexWriter`'s `unlock(Directory)`— Does exactly what its name implies. Although this method gives you power to unlock any Lucene index at any time, using it is dangerous. Lucene creates locks for a good reason, and unlocking an index while it's being modified will quickly result in a corrupt and unusable index.

Although you now know about Lucene's write lock, you should resist touching this file directly. Instead, always rely on Lucene's API to manipulate it. If you don't, your code may break if Lucene starts using a different locking mechanism in the future, or even if it changes the name or location of its lock files.

To demonstrate locking, listing 2.6 shows how the write lock prevents more than one writer from accessing an index simultaneously. In the `testWriteLock()` method, Lucene blocks the second `IndexWriter` from opening an index that has already been opened by another `IndexWriter`. This is an example of `write.lock` in action.

Listing 2.6 Using file-based locks to enforce a single writer at a time

```
public class LockTest extends TestCase {

  private Directory dir;

  protected void setUp() throws IOException {
    String indexDir =
      System.getProperty("java.io.tmpdir", "tmp") +
      System.getProperty("file.separator") + "index";
```

```
    dir = FSDirectory.getDirectory(indexDir);
  }

  public void testWriteLock() throws IOException {
    IndexWriter writer1 = null;
    IndexWriter writer2 = null;

    try {
      writer1 = new IndexWriter(dir, new SimpleAnalyzer(),
                                IndexWriter.MaxFieldLength.UNLIMITED);
      writer2 = new IndexWriter(dir, new SimpleAnalyzer(),
                                IndexWriter.MaxFieldLength.UNLIMITED);
      fail("We should never reach this point");
    }
    catch (LockObtainFailedException e) {
      e.printStackTrace();   //#1
    }
    finally {
      writer1.close();
      assertNull(writer2);
    }
  }
}
```

#1 Expected exception: only one IndexWriter allowed on single index

When we run this code we see an exception stack trace caused by the locked index, which resembles the following stack trace:

```
org.apache.lucene.store.LockObtainFailedException: Lock obtain timed out:
SimpleFSLock@/tmp/index/write.lock
     at org.apache.lucene.store.Lock.obtain(Lock.java:85)
     at org.apache.lucene.index.IndexWriter.init(IndexWriter.java:1094)
```

As we mentioned earlier, new users of Lucene sometimes don't have a good understanding of the concurrency issues described in this section and consequently run into locking issues, such as the one show in the previous stack trace. If you see similar exceptions in your applications, please don't disregard them if the consistency of your indexes is at all important to you! Lock-related exceptions are typically a sign of a misuse of the Lucene API; if they occur in your application, you should scrutinize your code to resolve them promptly.

2.14 Advanced indexing concepts

Up until now, we've covered how Lucene models documents, the steps of the indexing process, and how specifically to use the APIs to create and index `Documents` and `Fields`. We'll switch gears now to cover some more advanced topics, including using `IndexReader` to delete documents and the disk space consumed by deleted documents. Then we'll discuss how `IndexWriter` manages buffering, flushing, committing and merging. Finally we detail how `IndexWriter` provides transactional support. While these are technical details of the internals of `IndexWriter`, and you could happily perform indexing without understanding these concepts, you may someday find yourself wondering exactly when and how changes

made by IndexWriter become visible to readers on the index. We'll start by discussing a different way to delete documents from an index.

2.14.1 Deleting documents with IndexReader

IndexReader also exposes methods to delete documents. Why would you want two ways to do the same thing? Well, there are some interesting differences:

- IndexReader is able to delete by document number. This means you could do a search, step through matching document numbers, perhaps apply some application logic, and then pick and choose which document numbers to delete. IndexWriter cannot expose such a method because document numbers may change suddenly due to merging (see section XXX).
- IndexReader can delete by Term, just like IndexWriter. However, IndexReader returns the number of documents deleted, whereas IndexWriter does not. This is due to a difference in the implementation: IndexReader determines immediately which documents were deleted, and is therefore able to count up the affected documents, whereas IndexWriter simply buffers the deleted Term and applies it later.
- IndexReader's deletions take effect immediately, if you use that same reader for searching. This means you can do deletion then immediately run a search, and the deleted documents will no longer appear in the search results. Whereas with IndexWriter, the deletes must be flushed and committed, and then a new IndexReader must be opened, before the deletions take effect.
- IndexWriter is able to delete by Query, but IndexReader is not (though it's not hard to run your own Query and simply delete every document number that was returned).

If you are tempted to use IndexReader for deletion, remember that Lucene only allows one "writer" to be open at once. Confusingly, an IndexReader that is performing deletions counts as a "writer". This means you are forced to close any open IndexWriter before doing deletions with IndexReader and vice/versa. If you find that you are quickly interleaving added and deleted documents, this will slow down your indexing throughput substantially. It's better to batch up your additions and deletions, to get better performance.

Generally, unless one of the differences above is compelling for your application, it's best to simply use IndexWriter for all deletions. Let's look at the disk space consumed by deleted documents.

2.14.2 Reclaiming disk space used by deleted documents

Lucene uses a black-list approach when recording deleted documents in the index. This means that the document is simply marked as deleted in a bit array, which is a very quick operation, but the data corresponding to that document still consumes disk space in the index. This is necessary because in an inverted index, a given document's terms are scattered all over the place, and it would be impractical to try to reclaim that space when the document is deleted. It's not until segments are merged, either by normal merging over time or by an explicit call to optimize, that these bytes are reclaimed. Section 2.14.5 describes how and when Lucene merges segments. You can also call expungeDeletes to reclaim all disk space consumed by deleted documents. This call merges any segments that have pending deletions, which might be a somewhat lower cost operation than optimize.

2.14.3 Buffering and flushing

As shown in figure 2.2, when new `Documents` are added to a Lucene index, or deletions are pending, they're initially buffered in memory instead of being immediately written to the disk. This is done for performance reasons, to minimize disk IO. Periodically, these changes are flushed to the index `Directory` as a new segment.

`IndexWriter` triggers a flush according to three possible criteria which are set by the application. To flush when the buffer has consumed more than a pre-set amount of RAM, use `setRAMBufferSizeMB`. The RAM buffer size should not be taken as an exact maximum of memory usage since there are many other factors to consider when measuring overall JVM memory usage. Section 10.3.3 has more details. It's also possible to flush after a specific number of documents have been added by calling `setMaxBufferedDocs`. Finally, you can trigger flushing whenever the total number of buffered delete terms and queries exceeds a specified count by calling `setMaxBufferedDeleteTerms`. Flushing happens whenever one of these triggers is hit, whichever comes first. There is a constant `IndexWriter.DISABLE_AUTO_FLUSH` which you can pass to any of these methods to prevent flushing by that criterion. By default, `IndexWriter` flushes only when RAM usage is 16 MB.

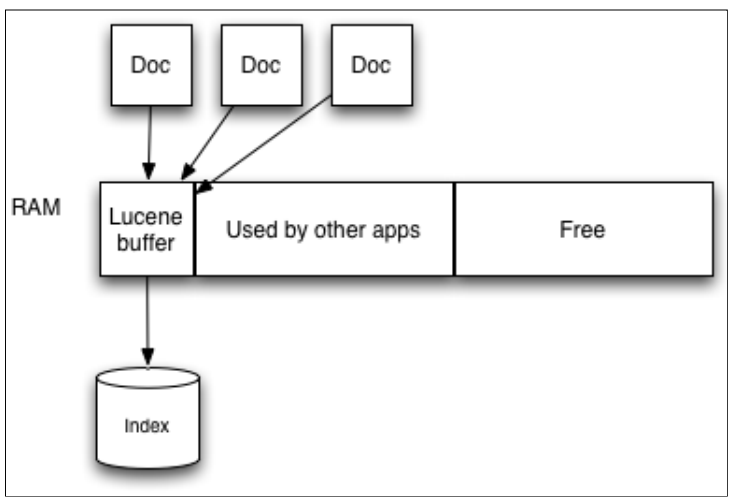


Figure 2.3 An in-memory `Document` buffer helps improve Lucene's indexing performance.

When a flush occurs, the writer creates new segment and deletion files in the `Directory`. However, these files are neither visible nor usable to a newly opened `IndexReader` until the writer commits the changes. It's important to understand this difference. Flushing is done to free up memory consumed by buffered changes to the index, whereas committing is done to make all flushed changes persistent and visible in the index. This means `IndexReader` always sees the starting state of the index (when `IndexWriter` was opened), until the writer commits.

NOTE

While an IndexWriter is making changes to the index, an `IndexReader` will not see any of these changes until `commit()` or `close()` is called.

2.14.4 ACID transactions and index consistency

Lucene's `IndexWriter` exposes a transactional API. Here we'll discuss just what that means, then we'll describe the `IndexDeletionPolicy` that let's you control how many commits are kept in the index. Next we talk through some useful APIs to manage multiple commits. Finally we describe how to involve Lucene in a two-phased commit with other resources. There's no question these are advanced topics and most likely your application won't need to use this functionality; but if you are one of the few applications that do needs this, you'll be glad to know Lucene makes it straightforward.

JLucene implements the ACID transactional model, with the restriction that only one transaction (writer) may be open at once. Here's what ACID stands for, along with details about how Lucene meets it:

- Atomic – all changes done with the writer are either committed to the index, or none are; there is nothing in-between.
- Consistency – the index will also be consistent, for example you will never see a delete without the corresponding addDocument from updateDocument; you will always see all or none of the indexes added from an addIndexes call.
- Isolation -- While you are making changes with IndexWriter, no changes are visible to a newly opened IndexReader, until you successfully commit. The IndexReader only sees the last successful commit.
- Durability -- If your application hits an unhandled exception, or the JVM crashes, or the OS crashes, or the computer loses power, the index will remain consistent and will contain all changes included in the last successful commit. Changes done after that will be lost.

NOTE

If your application, the JVM, the OS or the machine crashes, then the index will not be corrupt and will automatically rollback to the last successful commit. However, Lucene relies on the OS and IO system that holds the index to properly implement the "fsync" system call, by flushing any OS or IO write caches to the actual underlying stable storage. In some cases, it may be necessary to disable write caching on the underlying IO devices.

You can force a commit at any time by calling `IndexWriter.commit()` or `IndexWriter.commit(String commitUserData)`, which records the provided string as opaque metadata into the commit, for later retrieval. Note that commit can be a costly operation, and doing so frequently will slow down your indexing throughput. Closing the writer also commits all changes. If for some reason you decide that you want to discard all changes, you can call `writer.rollback()` to remove all changes in the current `IndexWriter` session since the last commit to the index.

Here are the steps `IndexWriter` takes during commit:

1. Flush any buffered documents and deletions.
2. Sync all newly created files, including newly flushed files and also any files produced by merges that have finished, since commit was last called or since the `IndexWriter` was opened. IndexWriter calls `Directory.sync` to achieve this, which does not return until all pending writes in the specified file have been written to stable storage on the underlying IO system. This is usually a costly operation as it forces the OS to flush any pending writes.
3. Write and sync the next `segments_N` file. Once this completes, `IndexReader`s will suddenly see all changes done since the last commit.
4. Remove old commits by calling on `IndexDeletionPolicy` to remove old commits. You can create your own implementation of this class to customize which commits are deleted, and when.

Let's look at how you can keep multiple commits present in a single index.

INDEXDELETIONPOLICY

`IndexDeletionPolicy` is the class that tells `IndexWriter` when it's safe to remove old commits. The default policy is `KeepOnlyLastCommitDeletionPolicy`, which always removes all prior commits whenever a new commit is complete. Most of the time you should simply use this default. But for some advanced applications, where you'd like to keep an old point-in-time snapshot around even though further changes have been committed to the index, you may implement your own policy.

For example, when sharing an index over NFS, it may be necessary to customize the deletion policy such that a commit is not deleted until all readers using the index have switched to the most recent commit, based on application specific logic (see section 2.13 for indexing and searching over NFS). Another example is a retail company that would like to keep the last N versions of its catalog available for searching. Note that whenever your policy chooses to keep a commit around, that commit will necessarily consume additional disk space in the index.

If you keep multiple commits in your index, there are some useful APIs to help you tell them apart.

MANAGING MULTIPLE INDEX COMMITS

Normally, a Lucene index will have only a single commit present, which is the last commit. However, by implementing a custom deletion policy, you can easily accumulate many commits in the index. You can use the static `IndexReader.listCommits()` method to retrieve all commits present in an index. Then, you can step through each and gather whatever details you need. For example, if you previously called `IndexWriter.commit(String commitUserData)`, then that string is available from each commit by calling its `getUserData()` method. This string may store something meaningful to your application, enabling you to pick out a particular commit of interest.

Once you've found a commit, you can open an `IndexReader` on it: several of the static open methods accept an `IndexCommit`. You could use this to explicitly search a previous version of the index

Using the same logic, you can open an `IndexWriter` on a prior commit, but the use case is very different: this allows you rollback to a previous commit, and start indexing new documents from that point, effectively undoing all changes to the index that had happened after that commit. This is similar to `IndexWriter`'s rollback method, except that method only rolls back changes done within the current

IndexWriter session, whereas opening on a prior commit lets you rollback changes that were already committed to the index, perhaps long ago.

TWO PHASED COMMIT

For applications that need to commit a transaction involving a Lucene index and other external resources, for example a database, Lucene exposes the prepareCommit() and prepareCommit(String commitUserData) methods. Each method does steps 1 and 2 above, as well as most of step 3, but it stops short of making the new segments_N file visible to a reader. After prepareCommit() is called, you should then either call rollback(), to abort the commit, or commit(). Commit() is a very fast call if prepareCommit() was already called. If an error will be hit, for example disk full, most likely prepareCommit() will hit the error, not commit(). The separation of these two steps of committing allows you to build a distributed two-phase commit protocol involving Lucene.

Next we describe how Lucene merges segments, and what you can do to control this process.

2.14.5 Merging

When an index has too many segments, IndexWriter selects some of the segments and merges them into a single, large segment. Merging has several important benefits:

- It reduces the number of segments in the index because once the merge completes, all of the old segments are removed and a single large segment is added in their place. This makes searching faster since there are fewer segments to search, and also prevents hitting the file descriptor limit enforced by the operating system.
- It reduces the size of the index. For example, if there were deletes pending on the merged segments, the merging process frees up the bytes consumed by deleted documents. Even if there are no pending deletions, a single merged segment will generally use fewer bytes to represent exactly the same set of indexed documents.

So when exactly is a merge necessary? What specifically does “too many segments” mean? That is decided by the MergePolicy. But, MergePolicy only decides which merges should be done; it’s up to MergeScheduler to actually carry out these merges. Let’s first drill into MergePolicy.

MERGEPOLICY

IndexWriter relies on a subclass of the abstract MergePolicy base class to decide when a merge should be done. Whenever new segments are flushed, or a previously selected merge has completed, the MergePolicy is consulted to determine if a merge is now necessary, and if so, precisely which segments will be merged. Besides picking “normal” segment merges to do, the MergePolicy also selects merges necessary to optimize the index and to run expungeDeletes.

Lucene provides two core merge policies, both subclassing from LogMergePolicy. The first, which is the default used by IndexWriter, is LogByteSizeMergePolicy. This policy measures the size of a segment as the total size in bytes of all files for that segment. The second one, LogDocMergePolicy, makes the same merging decisions except it measures size of a segment by the document count of the segment. Note that neither merge policy takes deletions into account. If you have mixed document sizes it’s best to use LogByteSizeMergePolicy since it’s a more accurate measure of segment size.

If the core merge policies don't suit your application, you can subclass `MergePolicy` to implement your own. For example, you could implement a time-dependent policy that defers large merges until off-peak hours, to ensure merging doesn't conflict with ongoing searches. Or perhaps you'd like a policy that tries harder to select segments with many pending deletions, so as to reclaim disk space sooner in the index.

Table 2.1 shows the parameters that control how `LogByteSizeMergePolicy` chooses merges. Some of these are also exposed as convenience methods in `IndexWriter`.

Table 2.1 Parameters that control merge selection with the default MergePolicy, `LogByteSizeMergePolicy`

IndexWriter method	LogByteSizeMergePolicy method	Default value	Description
setMergeFactor	setMergeFactor	10	Controls segment merge frequency and size
	setMinMergeMB	1.6 MB	Sets a floor on the smallest segment level
	setMaxMergeMB	Long.MAX_VALUE	Limits the size in bytes of a segment to be merged
setMaxMergeDocs	setMaxMergeDocs	Integer.MAX_VALUE	Limits the number of documents for a segment to be merged

To understand these parameters we first must understand how both of these policies select merges. For each segment, its level is computed using this formula:

```
(int) log(max(minMergeMB, size))/log(mergeFactor)
```

This effectively groups the segments of roughly equal size (in log space) into the same level. Tiny segments, less than `minMergeMB`, are always forced into the lowest level to prevent too many tiny segments in the index. In general, each level contains segments that are up to `mergeFactor` times larger than the previous level.

Once a given level has `mergeFactor` or more segments, they are merged. Thus, `mergeFactor` controls not only when to trigger a merge but also how many segments are merged at once. The larger this is, the more segments will exist in your index and the less frequently merges will be done, for a given number of documents in the index. Larger values generally result in faster indexing throughput, but may result it too many open file descriptors (see Section 10.3.2 for more details on controlling file descriptor usage). It's probably best to leave this at its default value (10) unless you see strong gains when testing different values. When the merge completes, a new segment at the next higher level replaces the merged segments. To prevent merges of very large segments, set `maxMergeMB` or `maxMergeDocs`. If ever a segment is over `maxMergeMB` in byte size, or `maxMergeDocs` in its document count, that segment will never be merged. By setting `maxMergeDocs` you can force extremely large segments to remain separate forever in your index.

Besides selecting merges for normal ongoing maintenance of the index, `MergePolicy` is also responsible for selecting merges when `optimize` or `expungeDeletes` is called. In fact, it's really up to the `MergePolicy` to define what these methods actually mean. For example, maybe during optimize you want to skip segments larger than a certain size. Or perhaps for expungeDeletes you only want to merge a segment if it has more than 10% of its documents deleted. These examples can be easily achieved by creating your own MergePolicy that subclasses `LogByteSizeMergePolicy`.

Over time, `LogByteSizeMergePolicy` produces an index with a logarithmic staircase structure: you have a few very large segments, a few segments `mergeFactor` smaller, etc. The number of segments in your index is proportional to the logarithm of the net size, in bytes or number of documents, of your index. This generally does a good job keeping segment count low while minimizing the net merge cost. However, some of these settings can be tuned to improve indexing throughput, as described in section 10.1.2.

MERGESCHEDULER

Selection of a merge is only the first step. The next step is the actual merging. `IndexWriter` relies on a subclass of `MergeScheduler` to achieve this. By default, `IndexWriter` uses `ConcurrentMergeScheduler`, which merges segments using background threads. There is also `SerialMergeScheduler`, which merges segments using the same thread that's calling `addDocument`, which means you could suddenly see an `addDocument` call take a long time while it executes a merge. You could also implement your own `MergeScheduler`: perhaps you want to defer very large segment merges until after 2 AM but do smaller merges whenever they are needed.

Generally, customizing `MergePolicy` settings, or implementing your own `MergePolicy` or `MergeScheduler`, are extremely advanced use cases. For most applications, Lucene's default settings work very well. If you are curious about when `IndexWriter` is flushing and merging, you can call its `setInfoStream` method, as described in Section 2.16.

We'll switch gears now and talk about how to make Lucene work over a common unix remote filesystem.

2.15 Sharing an index over NFS

The NFS file system, or Networked File System, is ubiquitous and very useful on Unix platforms as a means of sharing files across multiple computers, and sharing a Lucene index is no exception. One common approach is to have one computer create and update the index on a directory which is then shared over NFS to multiple computers that do the searching. While this is convenient, since it avoids having to replicate copies of the index out to multiple computers, NFS presents certain challenges for Lucene that you must work around. First off, be sure you are using Lucene 2.1 or later since 2.1 fixed important issues for NFS.

The core challenge with using Lucene over NFS is how NFS handles deletion of files that are still held open on other computers. Most file systems protect open files from deletion. For example, Windows simply disallows deletion of an open file, whereas most native Unix file systems allow the deletion to proceed, but the actual bytes of the file remain allocated on disk until all open file handles are closed (this is called "delete on last close" semantics). In both approaches, an open file handle can still be used to read all bytes in the file after the file deletion is attempted. NFS does neither of these, and simply removes the file, such that the next IO operation attempted by a computer with an open file handle will encounter the much-dreaded "Stale NFS file handle" IOException.

To prevent this error from hitting your searchers you must create your own `IndexDeletionPolicy` class to control deletion of previous commit points until all searchers on the index have reopened to the newer commit point. For example, a common approach is to remove an index commit only if it's older than say 4 hours, as long as you can ensure that every `IndexReader` reading the index reopens itself less than 4 hours after a commit. Alternatively, on hitting the "Stale NFS file handle" during searching, you could at that moment reopen your searcher and then redo the search. This is a viable approach only if reopening a searcher is not too time consuming. Otherwise, the unlucky query that hit the error will take unacceptably long to get results.

Finally, realize that performance over NFS is not great, because the bytes must cross the wires to get to the computer doing the searching. It's possible mounting the NFS directory as read-only may improve the performance, but likely you'll still be far from the performance of a local native directory.

Our final topic for this already quite long chapter shows you how to gain some insight into the internal operations `IndexWriter` is doing.

2.16 Debugging indexing

Let's discuss one final, fairly unknown Lucene feature (if we may so call it). If you ever need to debug Lucene's index-writing process, remember that you can get Lucene to output information about its indexing operations by setting `IndexWriter`'s `setInfoStream` method, passing in an `OutputStream` such as `System.out`:

```
IndexWriter writer = new IndexWriter(dir, new SimpleAnalyzer(),
                                     true, IndexWriter.MaxFieldLength.UNLIMITED);
writer.setInfoStream(System.out);
```

This reveals very detailed diagnostic information about segment flushes and merges, as shown here, and may help you tune indexing parameters described earlier in the chapter. Likely if you are experiencing an issue, something you may believe to be a bug in Lucene, and you take your issue to the Lucene user's list at Apache, the first request you'll get back is someone asking you to post the output from setting `infoStream`. It will look something like this:

```
flush postings as segment _9 numDocs=1095
  oldRAMSize=16842752 newFlushedSize=5319835 docs/MB=215.832 new/old=31.585%
IFD [main]: now checkpoint "segments_1" [10 segments ; isCommit = false]
IW 0 [main]: LMP: findMerges: 10 segments
IW 0 [main]: LMP:   level 6.2247195 to 6.745619: 10 segments
IW 0 [main]: LMP:     0 to 10: add this merge
IW 0 [main]: add merge to pendingMerges: _0:C1010->_0 _1:C1118->_0 _2:C968->_0
_3:C1201->_0 _4:C947->_0 _5:C1084->_0 _6:C1028->_0 _7:C954->_0 _8:C990->_0 _9:C1095->_0
[total 1 pending]
IW 0 [main]: CMS: now merge
IW 0 [main]: CMS:   index: _0:C1010->_0 _1:C1118->_0 _2:C968->_0 _3:C1201->_0 _4:C947-
>_0 _5:C1084->_0 _6:C1028->_0 _7:C954->_0 _8:C990->_0 _9:C1095->_0
IW 0 [main]: CMS:   consider merge _0:C1010->_0 _1:C1118->_0 _2:C968->_0 _3:C1201->_0
_4:C947->_0 _5:C1084->_0 _6:C1028->_0 _7:C954->_0 _8:C990->_0 _9:C1095->_0 into _a
IW 0 [main]: CMS:     launch new thread [Lucene Merge Thread #0]
IW 0 [main]: CMS:   no more merges pending; now return
IW 0 [Lucene Merge Thread #0]: CMS:   merge thread: start
IW 0 [Lucene Merge Thread #0]: now merge
  merge=_0:C1010->_0 _1:C1118->_0 _2:C968->_0 _3:C1201->_0 _4:C947->_0 _5:C1084->_0
_6:C1028->_0 _7:C954->_0 _8:C990->_0 _9:C1095->_0 into _a
  index=_0:C1010->_0 _1:C1118->_0 _2:C968->_0 _3:C1201->_0 _4:C947->_0 _5:C1084->_0
_6:C1028->_0 _7:C954->_0 _8:C990->_0 _9:C1095->_0
IW 0 [Lucene Merge Thread #0]: merging _0:C1010->_0 _1:C1118->_0 _2:C968->_0 _3:C1201-
>_0 _4:C947->_0 _5:C1084->_0 _6:C1028->_0 _7:C954->_0 _8:C990->_0 _9:C1095->_0 into _a
IW 0 [Lucene Merge Thread #0]: merge: total 10395 docs
```

In addition, if you need to peek inside your index once it's built, you can use Luke, a handy third-party tool that we discuss in section 8.2.

2.17 Summary

We've covered a lot of ground in this chapter! You now have a solid understanding of how to make changes to a Lucene index. You saw Lucene's conceptual model for documents and fields, including a flexible but flat schema (when compared to a database). We saw that the indexing process consists of gathering content, extracting text from it, creating `Documents` and `Fields` from it, analyzing the text into a token stream and then handing it off to `IndexWriter` for addition to an index. We also briefly discussed the interesting segmented structure of an index.

You now know how to add, delete and update documents. We delved into a great many interesting options for controlling how a `Field` is indexed, including how the value is added to the inverted index, stored fields and term vectors, and how a `Field` can hold certain values other than `String`. We described variations like multi-valued fields, field and document boosting, and value truncation. You now know how to index dates, times and numbers, as well as fields for sorting.

We discussed segment-level changes, like optimizing and index and using `expungeDeletes` to reclaim disk space consumed by deleted documents. You now know of all the `Directory` implementations you could use to hold an index, such as `RAMDirectory` and `NIOFSDirectory`. We discussed Lucene's concurrency rules, and the locking it uses to protect an index from more than one writer.

Finally we covered a number of advanced topics: how and why to delete documents using `IndexReader` instead of `IndexWriter`; buffering, flushing and committing; `IndexWriter`'s support for transactions; merging and the classes available for customizing it; using an index over the NFS file system; and turning on `IndexWriter`'s infoStream to see details on the steps its taking internally.

Much of this advanced functionality will not be needed by the vast majority of search applications; in fact a few of `IndexWriter`'s APIs are enough to build a solid search application. By now you should be dying to learn how to search with Lucene, and that's what you'll read about in the next chapter.

3

Adding search to your application

This chapter covers

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

If we can't find it, it effectively doesn't exist. Even if we have indexed documents, our effort is wasted unless it pays off by providing a reliable and fast way to find those documents. For example, consider this scenario:

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

Such scenarios are easily handled with Lucene. We'll cover all the pieces to make this happen, including the search fundamentals and Boolean logic in this chapter, a specials filter in chapter 6, and synonym injection in chapter 4.

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

We begin with a simple example showing that the code you write to implement search is generally no more than a few lines long. Next we illustrate the scoring formula, providing a deep look into one of Lucene's most special attributes. With this example and a high-level understanding of how Lucene ranks search results, we'll then explore the various types of search queries Lucene handles natively. Finally we show how to create a search query from a text search expression entered by the end user.

3.1 Implementing a simple search feature

Suppose you're tasked with adding search to an application. You've tackled getting the data indexed, but now it's time to expose the full-text searching to the end users. It's hard to imagine that adding search could be any simpler than it is with Lucene. Obtaining search results requires only a few lines of code, literally. Lucene provides easy and highly efficient access to those search results, too, freeing you to focus your application logic and user interface around those results. Of course, as described in Chapter 2, you will first have to build up a search index.

In this chapter, we'll limit our discussion to the primary classes in Lucene's API that you'll typically use for search integration (shown in table 3.1).

Table 3.1 Lucene's primary searching API

Class	Purpose
`IndexSearcher`	Gateway to searching an index. All searches come through an `IndexSearcher` instance using any of the several overloaded `search` methods.
`Query` (and subclasses)	Concrete subclasses encapsulate logic for a particular query type. Instances of `Query` are passed to an `IndexSearcher`'s `search` method.
`QueryParser`	Processes a human-entered (and readable) expression into a concrete `Query` object.

`TopDocs`	Holds the top scoring documents, returned by IndexSearcher.search.
`ScoreDoc`	Provides access to each search result in TopDocs.

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

In the next section, we put `IndexSearcher`, `Query`, `TopDocs` and `ScoreDoc` to work with some basic term searches. After that, we show how to use `QueryParser` to create `Query` instances from an end user's textual search query.

3.1.1 Searching for a specific term

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

NOTE

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

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

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

```
public class BasicSearchingTest extends TestCase {
```

```
  public void testTerm() throws Exception {
    IndexSearcher searcher = new IndexSearcher("build/index");
    Term t = new Term("subject", "ant");
    Query query = new TermQuery(t);
    TopDocs docs = searcher.search(query, 10);
    assertEquals("JDwA", 1, docs.totalHits);

    t = new Term("subject", "junit");
    docs = searcher.search(new TermQuery(t), 10);
    assertEquals(2, docs.totalHits);

    searcher.close();
  }
}
```

A `TopDocs` object is returned from our search. We'll discuss this object in section 3.2, but for now just note that it encapsulates the top results, along with their scores and references to the underlying `Documents`. This encapsulation makes sense for efficient access to documents. Full documents aren't immediately returned; instead, you fetch them on demand. In this example we didn't concern ourselves with the actual documents associated with the docs returned because we were only interested in checking that the proper number of documents were found.

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

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

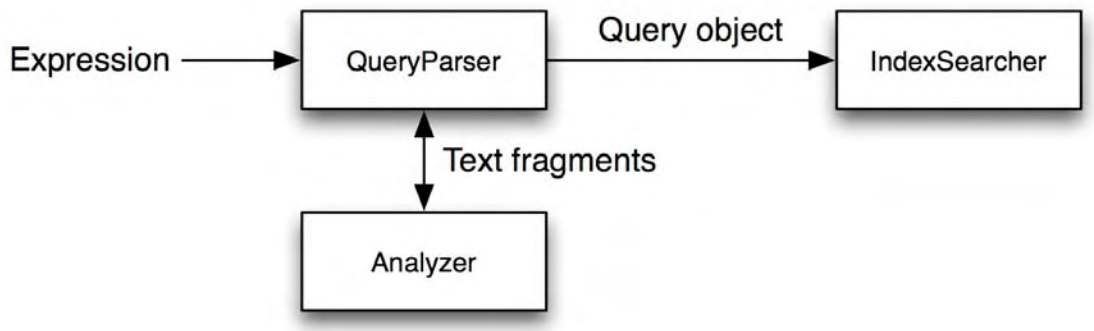


Figure 3.XXX Using `QueryParser` to search using a textual expression

3.1.2 Parsing a user-entered query expression: QueryParser

Two more features round out what the majority of searching applications require: sophisticated query expression parsing and access to the documents returned. Lucene's search methods require a `Query` object. *Parsing* a query expression is the act of turning a user-entered textual query such as "mock OR junit" into an appropriate `Query` object instance[1]; in this case, the `Query` object would be an instance of `BooleanQuery` with two non-required clauses, one for each term. The process is illustrated in Figure 3.XXX. The following code parses two query expressions and asserts that they worked as expected. After returning the hits, we retrieve the title from the first document found:

```
public void testQueryParser() throws Exception {
  IndexSearcher searcher = new IndexSearcher("build/index");
  QueryParser parser = new QueryParser("contents",
                                       new SimpleAnalyzer());

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

  query = new QueryParser("contents",
                          new SimpleAnalyzer()).parse("mock OR junit");
  docs = searcher.search(query, 10);
  assertEquals("JDwA and JIA", 2, docs.totalHits);

  searcher.close();
}
```

Lucene includes an interesting built-in feature that parses query expressions, available through the `QueryParser` class. It parses rich expressions such as the two shown (`"+JUNIT +ANT -MOCK"` and `"mock OR junit"`) into one of the `Query` implementations. The resulting `Query` instances can be very rich and complex! Dealing with human-entered queries is the primary purpose of the `QueryParser`.

As you can see in Figure 3.1, `QueryParser` requires an *analyzer* to break pieces of the query text into terms. In the first expression, the query was entirely uppercased. The terms of the `contents` field, however, were lowercased when indexed. `QueryParser`, in this example, uses `SimpleAnalyzer`, which lowercases the terms before constructing a `Query` object. (Analysis is covered in great detail in the next chapter, but it's intimately intertwined with indexing text and searching with `QueryParser`.) The main point regarding analysis to consider in this chapter is that you need to be sure to query on the actual terms indexed. `QueryParser` is the only searching piece that uses an analyzer. Querying through the API using `TermQuery` and the others discussed in section 3.4 doesn't use an analyzer but does rely on matching terms to what was indexed. Therefore, if you construct queries entirely programmatically you must ensure the `Term`s included in all of your queries match the Tokens produced by the analyzer used

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

during indexing. In section 4.1.2, we talk more about the interactions of QueryParser and the analysis process.

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

USING QUERYPARSER

Before diving into the details of QueryParser (which we do in section 3.5), let's first look at how it's used in a general sense. QueryParser is instantiated with a field name (String) and an Analyzer, which it uses to break the incoming search text into Terms. We discuss analyzers in detail in the next chapter and then cover the interactions between QueryParser and the analyzer in section 4.1.2:

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

The provided field name is the default field against which all terms will be searched, unless the search text explicitly requests matches against a different field name using the syntax "field:text" (more on this in section 3.5.4). Then, the QueryParser instance has a parse() method to allow for the simplest use:

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

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

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

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

HANDLING BASIC QUERY EXPRESSIONS WITH QUERYPARSER

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

Table 3.2 Expression examples that QueryParser handles

Query expression	Matches documents that...

`java`	Contain the term *java* in the default field
`java junit` `java or junit`	Contain the term *java* or *junit*, or both, in the default field[2]
`+java +junit` `java AND junit`	Contain both *java* and *junit* in the default field
`title:ant`	Contain the term *ant* in the `title` field

`title:extreme`
`–subject:sports`
`title:extreme`
`AND NOT subject:sports`
	Have *extreme* in the `title` field and don't have *sports* in the `subject` field

`(agile OR extreme) AND` `methodology`	Contain *methodology* and must also contain *agile* and/or *extreme*, all in the default field
`title:"junit in action"`	Contain the exact phrase "*junit in action*" in the `title` field
`title:"junit action"~5`	Contain the terms *junit* and *action* within five positions of one another
`java*`	Contain terms that begin with *java*, like *javaspaces*, *javaserver*, and *java.net*
`java~`	Contain terms that are close *to* the word *java*, such as *lava*
`lastmodified:` `[1/1/04 TO 12/31/04]`	Have `lastmodified` field values between the dates January 1, 2004 and December 31, 2004

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

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

3.2. Using IndexSearcher

Like the rest of Lucene's primary API, IndexSearcher is simple to use. Searches are done using an instance of `IndexSearcher`. The simplest way to create an `IndexSearcher` is by providing it a `String` file path to your index directory in the filesystem:

```
IndexSearcher searcher = new IndexSearcher("/path/to/index");
```

You can also pass in an instance of `java.io.File`, or your own `Directory` instance, which allows you to use `Directory` implementations other than `FSDirectory`. If you already have an open `IndexReader` instance that you'd like to use for searching, you can create an `IndexSearcher` from that as well.

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

Table 3.3 Primary `IndexSearcher` search methods

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

NOTE

An `IndexSearcher` instance searches only the index as it existed at the time the `IndexSearcher` was instantiated. If indexing is occurring concurrently with searching, newer documents indexed won't be visible to searches. In order to see the new documents, you must commit the changes from `IndexWriter` and then instantiate a new `IndexSearcher`.

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

3.2.1 Working with TopDocs

Now that we've called `search(Query, n)`, we have a `TopDocs` object at our disposal which we use for efficient access to the search results. Typically, you'll use one of the `search` methods that return a `TopDocs` object, as shown in table 3.3. Results are ordered by relevance—in other words, by how well each document matches the query (sorting results in other ways is discussed in section 5.1).

There are only three attributes and methods on a `TopDocs` instance; they're listed in table 3.4. The attribute `TopDocs.totalHits` returns the number of *matching documents*. A matching document is one with a score greater than zero, as defined by the scoring formula covered in section 3.3. The matches, by default, are sorted in decreasing score order. The `TopDocs.scoreDocs` attribute is an array containing the top n matches. Each `ScoreDoc` instance has a float score, which is the relevance score, and an `int doc`, which is the document ID that can be used to retrieve the stored fields for that document by calling `IndexSearcher.document(doc)`. Finally, `TopDocs.getMaxScore()` returns the best score across all matches; when you sort by relevance (the default), that will always be the score of the first result. But if you sort by other criteria, as described in section 5.1, it will be the max score of all matching documents even when the best scoring document isn't in the top results by your sort criteria.

Table 3.4 `TopDocs` methods for efficiently accessing search results

`TopDocs` method or attribute	**Return value**
`totalHits`	Number of documents in the `Hits` collection
`scoreDocs`	`Array of ScoreDoc instances that contains the results`
`getMaxScore()`	Returns best score of all matches

3.2.2 Paging through results

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

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

It turns out that requerying is most often the best solution. Requerying eliminates the need to store per-user state. In a web application, staying stateless is often desirable. Requerying at first glance seems a waste, but Lucene's blazing speed more than compensates. Also, thanks to the IO caching in modern operating systems, requerying will typically be fast because the necessary bits from disk will already be cached in RAM. Frequently users don't click past the first page of results anyway.

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

Don't prematurely optimize your paging implementations with caching or persistence. First implement your paging feature with a straightforward requery approach; chances are you'll find this sufficient for your needs.

3.3 Understanding Lucene scoring

We chose to discuss this complex topic early in this chapter so you'll have a general sense of the various factors that go into Lucene scoring as you continue to read. We'll describe how Lucene scores document matches to a query, and then show you how to get a detailed explanation of how a certain document arrived at its score.

Without further ado, meet Lucene's similarity scoring formula, shown in figure 3.1. It's called the similarity scoring formula because its purpose is to measure the similarity between a query and each document that matches the query. The score is computed for each document (d) matching each term (t) in a query (q).

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

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

NOTE

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

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

Table 3.5 Factors in the scoring formula

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

Boost factors are built into the equation to let you affect a query or field's influence on score. Field boosts come in explicitly in the equation as the `boost(t.field in d)` factor, set at indexing time. The default value of field boosts, logically, is 1.0. During indexing, a `Document` can be assigned a boost, too. A `Document` boost factor implicitly sets the starting field boost of all fields to the specified value. Field-specific boosts are multiplied by the starting value, giving the final value of the field boost factor. It's possible to add the same named field to a `Document` multiple times, and in such situations the field boost is computed as all the boosts specified for that field and document multiplied together. Section 2.3 discusses index-time boosting in more detail.

In addition to the explicit factors in this equation, other factors can be computed on a per-query basis as part of the `queryNorm` factor. Queries themselves can have an impact on the document score. Boosting a `Query` instance is sensible only in a multiple-clause query; if only a single term is used for searching, boosting it would boost all matched documents equally. In a multiple-clause boolean query, some documents may match one clause but not another, enabling the boost factor to discriminate between matching documents. Queries also default to a 1.0 boost factor.

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

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

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

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

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

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

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

```
    }

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

    FSDirectory directory = FSDirectory.getDirectory(indexDir);
    QueryParser parser = new QueryParser("contents", new SimpleAnalyzer());
    Query query = parser.parse(queryExpression);

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

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

    for (int i = 0; i < topDocs.totalHits; i++) {
      ScoreDoc match = topDocs.scoreDocs[i];
      Explanation explanation = searcher.explain(query, match.doc);  //#1

      System.out.println("----------");
      Document doc = searcher.doc(match.doc);
      System.out.println(doc.get("title"));
      System.out.println(explanation.toString());    //#2
    }
  }
}
```

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

```
Query: junit
----------
JUnit in Action
0.81291926 = (MATCH) weight(contents:junit in 6), product of:
  0.99999994 = queryWeight(contents:junit), product of:
    2.299283 = idf(docFreq=2, numDocs=11)
    0.43491817 = queryNorm
  0.8129193 = (MATCH) fieldWeight(contents:junit in 6), product of:
    1.4142135 = tf(termFreq(contents:junit)=2)   //#3
    2.299283 = idf(docFreq=2, numDocs=11)
    0.25 = fieldNorm(field=contents, doc=6)

----------
Java Development with Ant
0.5748207 = (MATCH) weight(contents:junit in 5), product of:
  0.99999994 = queryWeight(contents:junit), product of:
    2.299283 = idf(docFreq=2, numDocs=11)
    0.43491817 = queryNorm
  0.57482076 = (MATCH) fieldWeight(contents:junit in 5), product of:
    1.0 = tf(termFreq(contents:junit)=1)      //#4
    2.299283 = idf(docFreq=2, numDocs=11)
    0.25 = fieldNorm(field=contents, doc=5)
```

#1 Generate Explanation of single Document for query
#2 Output Explanation

#3 "junit" appears twice in contents
#4 "junit" appears once in contents

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

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

There is also a `.toHtml()` method that outputs the same hierarchical structure, except as nested HTML `<ul>` elements suitable for outputting in a web browser. In fact, the `Explanation` feature is a core part of the Nutch project (see the case study in section 10.1), allowing for transparent ranking.

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

Now we'll switch gears and show you how to create queries programmatically.

3.4 Creating queries programmatically

Let's recap where we stand. We've covered some good ground so far. By now you have a strong foundation for getting your search application off the ground: we showed the most important ways of performing searches with Lucene. Now, it's time to drill down into detail on creating Lucene's queries programmatically. It turns out Lucene has a number of interesting queries for you to play with! After that we get back to `QueryParser` and visit a number of details and interesting topics that may arise.

As you saw in section 3.2, querying Lucene ultimately requires a call to `IndexSearcher`'s `search` using an instance of `Query`. `Query` subclasses can be instantiated directly; or, as we discussed in section 3.1.2, a `Query` can be constructed through the use of a parser such as `QueryParser`. If your application will rely solely on `QueryParser` to construct `Query` objects, understanding Lucene's direct API capabilities is still important because `QueryParser` uses them.

Even if you're using `QueryParser`, combining a parsed query expression with an API-created `Query` is a common technique to augment, refine, or constrain a human-entered query. For example, you may want to restrict free-form parsed expressions to a subset of the index, like documents only within a category. Depending on your search's user interface, you may have date pickers to select a date range, drop-downs for selecting a category, and a free-form search box. Each of these clauses can be stitched together using a combination of `QueryParser`, `BooleanQuery`, `RangeQuery`, and a `TermQuery`. We demonstrate building a similar aggregate query in section 5.5.4.

Yet another way to create query objects is by using the XML Query Parser package, contained in Lucene's contrib sandbox and described in detail in section XXX. This package allows you to create an XML string describing, in great detail, the specific query you'd like to run; the package then parses that string into a Query instance.

This section covers each of Lucene's built-in `Query` types, including `TermQuery`, `RangeQuery`, `PrefixQuery`, `BooleanQuery`, `PhraseQuery`, `WildcardQuery`, `FuzzyQuery`, and the unusual `MatchAllDocsQuery`. The `QueryParser` expression syntax that maps to each `Query` type is provided. We begin with `TermQuery`.

3.4.1 Searching by term: TermQuery

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

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

A `TermQuery` accepts a single `Term`:

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

All documents that have the word *java* in a `contents` field are returned from searches using this `TermQuery`. Note that the value is case-sensitive, so be sure to match the case of terms indexed; this may not be the exact case in the original document text, because an analyzer (see chapter 5) may have indexed things differently.

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

```
public void testKeyword() throws Exception {
  IndexSearcher searcher = new IndexSearcher("build/index");

  Term t = new Term("isbn", "1930110995");
  Query query = new TermQuery(t);
  TopDocs docs = searcher.search(query, 10);
  assertEquals("JUnit in Action", 1, docs.totalHits);

  searcher.close();
}
```

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

TERMQUERY AND QUERYPARSER

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

3.4.2 Searching within a range: `RangeQuery`

`Term`s are ordered lexicographically within the index, allowing for straightforward searching of terms within a range. Lucene's `RangeQuery` facilitates searches from a starting term through an ending term. The beginning and ending terms may either be included or excluded.

When the RangeQuery search is executed, there are two supported approaches under the hood. The first approach, which is the default, is to expand to an OR query of all terms within the range. This option has two serious downsides: first, it can be very slow if the number of terms in the range is large. This can easily happen if the values in the field are indexed with fine granularity. Section XXX drills into this problem. Second, the relevance scores assigned to the matching documents are counterintuitive and not generally useful.

The second option is to use constant scoring mode, enabled by calling setConstantScoreRewrite(true). With this option, the matching documents are first enumerated into an internal bit set and then that bit set is used to match documents. Each document is assigned a constant score equal to the Query's boost value. Generally this gives faster performance and is the recommended usage of RangeQuery. If you still have performance problems, you may want to switch to TrieRangeQuery, available in Lucene's sandbox and covered in Section 8.12.5.

The following code illustrates range queries inclusive of the begin (198805) and end (198810) terms:

```
public class RangeQueryTest extends TestCase {
  public void testInclusive() throws Exception {
    // pub date of TTC was October 1988
    RangeQuery query = new RangeQuery("pubmonth", "198805", "198810",
                                      true, true);
    IndexSearcher searcher = new IndexSearcher("build/index");

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

Our test data set has only one book, *Tao Te Ching* by Stephen Mitchell, published between May 1988 and October 1988; it was published in October 1988. The third and forth arguments to construct a RangeQuery are boolean flags, indicating whether the begin and end of the range are inclusive, respectively. Using the same dates and range, but exclusively, no book is found:

```
  public void testExclusive() throws Exception {
    // pub date of TTC was October 1988
    RangeQuery query = new RangeQuery("pubmonth", "198805", "198810",
                                      false, false);
    IndexSearcher searcher = new IndexSearcher("build/index");

    TopDocs matches = searcher.search(query, 10);
    assertEquals("there is no tao", 0, matches.totalHits);
    searcher.close();
  }
```

If you are searching numeric fields (int, float, etc), don't forget to zero-pad the numbers during indexing to ensure the lexicographic sort order of the terms matches the numeric sort order. Section 2.9 describes this in more detail.

RANGEQUERY AND QUERYPARSER

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

Now we move to another query that can match many terms from the index, `PrefixQuery`.

3.4.3 Searching on a string: PrefixQuery

Searching with a `PrefixQuery` matches documents containing terms beginning with a specified string. It's deceptively handy. The following code demonstrates how you can query a hierarchical structure *recursively* with a simple `PrefixQuery`. The documents contain a `category` keyword field representing a hierarchical structure:

```
public class PrefixQueryTest extends TestCase {
  public void testPrefix() throws Exception {
    IndexSearcher searcher = new IndexSearcher("build/index");

    // search for programming books, including subcategories
    Term term = new Term("category",                              //#1
                         "/technology/computers/programming");    //#1
    PrefixQuery query = new PrefixQuery(term);                    //#1

    TopDocs matches = searcher.search(query, 10);                 //#1
    int programmingAndBelow = matches.totalHits;

    // only programming books, not subcategories
    matches = searcher.search(new TermQuery(term), 10);           //#2
    int justProgramming = matches.totalHits;

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

#1 Search for programming books, including subcategories
#2 Search only for programming books, not subcategories

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

Just like `RangeQuery`, `PrefixQuery` also accepts `setConstantScoreRewrite(true)` to enable more efficient constant scoring.

PREFIXQUERY AND QUERYPARSER

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

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

3.4.4 Combining queries: BooleanQuery

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

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

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

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

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

  RangeQuery books2004 =                          //#2
    new RangeQuery("pubmonth", "200401",          //#2
                   "200412",                      //#2
                   true, true);                   //#2

  BooleanQuery searchingBooks2004 = new BooleanQuery();              //#3
  searchingBooks2004.add(searchingBooks, BooleanClause.Occur.MUST);  //#3
  searchingBooks2004.add(books2004, BooleanClause.Occur.MUST);       //#3

  IndexSearcher searcher = new IndexSearcher("build/index");
  TopDocs matches = searcher.search(searchingBooks2004, 10);

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

#1 All books with subject "search"
#2 All books in 2004
#3 Combines two queries

#1 This query finds all books containing the subject "search".

#2 This query find all books published in 2004.

(Note that this could also be done with a "2004" PrefixQuery.)

#3 Here we combine the two queries into a single boolean query with both clauses required (the second argument is BooleanClause.Occur.MUST).

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

```
   public static boolean hitsIncludeTitle(IndexSearcher searcher, TopDocs hits,
String title)
      throws IOException {
      for (int i=0; i < hits.totalHits; i++) {
        Document doc = searcher.doc(hits.scoreDocs[i].doc);
        if (title.equals(doc.get("title"))) {
          return true;
        }
      }
      System.out.println("title '" + title + "' not found");
      return false;
    }
```

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

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

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

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

      IndexSearcher searcher = new IndexSearcher("build/index");
      TopDocs matches = searcher.search(enlightenmentBooks, 10);
```

```
    System.out.println("or = " + enlightenmentBooks);

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

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

BOOLEANQUERY AND QUERYPARSER

`QueryParser` handily constructs `BooleanQuery`s when multiple terms are specified. Grouping is done with parentheses, and the `BooleanClause.Occur` instances are set when the –, +, AND, OR, and NOT operators are specified by the user.

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

3.4.5 Searching by phrase: PhraseQuery

An index by default contains positional information of terms, as long as you did not create pure Boolean fields by indexing with the `omitTermFreqAndPositions` option (described in section 2.2.1). `PhraseQuery` uses this information to locate documents where terms are within a certain distance of one another. For example, suppose a field contained the phrase “the quick brown fox jumped over the lazy dog”. Without knowing the exact phrase, you can still find this document by searching for documents with fields having *quick* and *fox* near each other. Sure, a plain `TermQuery` would do the trick to locate this document knowing either of those words; but in this case we only want documents that have phrases where the words are either exactly side by side (*quick fox*) or have one word in between (*quick* [irrelevant] *fox*).

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

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

  protected void setUp() throws IOException {
```

```
    // set up sample document
    RAMDirectory directory = new RAMDirectory();
    IndexWriter writer = new IndexWriter(directory,
                                         new WhitespaceAnalyzer(),
                                         IndexWriter.MaxFieldLength.UNLIMITED);
    Document doc = new Document();
    doc.add(new Field("field",
              "the quick brown fox jumped over the lazy dog",
                     Field.Store.YES,
                     Field.Index.ANALYZED));
    writer.addDocument(doc);
    writer.close();

    searcher = new IndexSearcher(directory);
  }

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

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

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

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

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

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

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

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

Figure 3.2 shows how the slop positions work in both of these phrase query scenarios, and this test case shows the match in action:

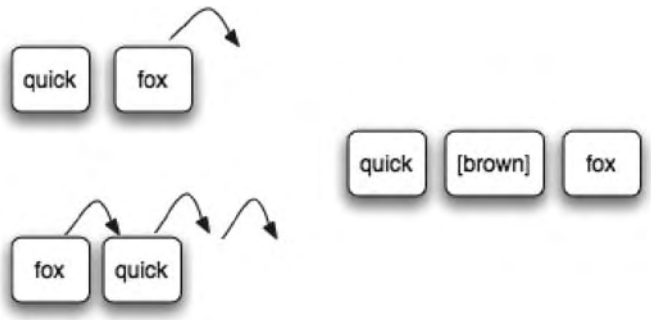


Figure 3.2 Illustrating `PhraseQuery` slop factor: "quick fox" requires a slop of 1 to match, whereas "fox quick" requires a slop of 3 to match.

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

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

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

MULTIPLE-TERM PHRASES

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

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

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

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

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

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

PHRASE QUERY SCORING

Phrase queries are scored based on the edit distance needed to match the phrase. More exact matches count for more weight than sloppier ones. The phrase query factor is shown in figure 3.3. The inverse relationship with distance ensures that greater distances have lower scores.

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

Figure 3.3 Sloppy phrase scoring

NOTE

Terms surrounded by double quotes in `QueryParser` parsed expressions are translated into a `PhraseQuery`. The slop factor defaults to zero, but you can adjust the slop factor by adding a tilde (~) followed by an integer. For example, the expression `"quick fox"~3` is a `PhraseQuery` with the terms *quick* and *fox* and a slop factor of 3. There are additional details about `PhraseQuery` and the slop factor in section 3.5.6. Phrases are analyzed by the analyzer passed to the `QueryParser`, adding another layer of complexity, as discussed in section 4.1.2.

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

3.4.6 Searching by wildcard: WildcardQuery

Wildcard queries let you query for terms with missing pieces but still find matches. Two standard wildcard characters are used: `*` for zero or more characters, and `?` for zero or one character. Listing 3.2 demonstrates `WildcardQuery` in action. You can think of `WildcardQuery` as a more general `PrefixQuery` because the wildcard doesn't have to be at the end.

Listing 3.2 Searching on the wild(card) side

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

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

```
    IndexSearcher searcher = new IndexSearcher(directory);
    Query query = new WildcardQuery(new Term("contents", "?ild*"));  //#1
    TopDocs matches = searcher.search(query, 10);
    assertEquals("child no match", 3, matches.totalHits);

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

#1 Construct WildcardQuery using Term

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

WARNING

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

Oddly, the closeness of a wildcard match has no effect on scoring. The last two assertions in listing 3.2, where *wild* and *mild* are closer matches to the pattern than *mildew*, demonstrate this.

Just like `RangeQuery`, `WildcardQuery` also accepts `setConstantScoreRewrite(true)` to enable constant scoring, which is recommended (see Section 3.4.2).

WILDCARDQUERY AND QUERYPARSER

`QueryParser` supports `WildcardQuery` using the same syntax for a term as used by the API. There are a few important differences, though. With `QueryParser`, the first character of a wildcarded term may not be a wildcard character; this restriction prevents users from putting asterisk-prefixed terms into a search expression, incurring an expensive operation of enumerating all the terms. Also, if the only wildcard character in the term is a trailing asterisk, the query is optimized to a `PrefixQuery`. Wildcard terms are lowercased automatically by default, but this can be changed. See section 3.5.7 for more on wildcard queries and `QueryParser`.

3.4.7 Searching for similar terms: FuzzyQuery

Lucene's `FuzzyQuery` matches terms *similar* to a specified term. The *Levenshtein distance* algorithm determines how similar terms in the index are to a specified target term.[3] *Edit distance* is another term for Levenshtein distance; it's a measure of similarity between two strings, where distance is measured as the

[3] See http://en.wikipedia.org/wiki/Levenshtein_Distance for more information about Levenshtein distance.

number of character deletions, insertions, or substitutions required to transform one string to the other string. For example, the edit distance between *three* and *tree* is 1, because only one character deletion is needed.

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

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

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

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

    Document doc = searcher.doc(matches.scoreDocs[0].doc);
    assertEquals("wuzza bear", "wuzzy", doc.get("contents"));
  }
```

This test illustrates a couple of key points. Both documents match; the term searched for (*wuzza*) wasn't indexed but was close enough to match. `FuzzyQuery` uses a *threshold* rather than a pure edit distance. The threshold is a factor of the edit distance divided by the string length.

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

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

Figure 3.4 `FuzzyQuery` distance formula.

WARNING

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

FUZZYQUERY AND QUERYPARSER

QueryParser supports FuzzyQuery by suffixing a term with a tilde (~). For example, the FuzzyQuery from the previous example would be wuzza~ in a query expression. Note that the tilde is also used to specify sloppy phrase queries, but the context is different. Double quotes denote a phrase query and aren't used for fuzzy queries.

Our final query, before we move onto QueryParser, is MatchAllDocsQuery.

3.4.8 Matching all documents: MatchAllDocsQuery

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

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

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

If you do this, documents are scored according to how the specified field was boosted (as described in section 2.xxx).

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

3.5 Parsing query expressions: QueryParser

Although API-created queries can be powerful, it isn't reasonable that all queries should be explicitly written in Java code. Using a human-readable textual query representation, Lucene's QueryParser constructs one of the previously mentioned Query subclasses. This constructed Query instance could be a complex entity, consisting of nested BooleanQuerys and a combination of almost all the Query types mentioned, but an expression entered by the user could be as readable as this:

```
+pubdate:[20040101 TO 20041231] Java AND (Jakarta OR Apache)
```

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

NOTE

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

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

The following sections detail the expression syntax, examples of using QueryParser, and customizing QueryParser's behavior. The discussion of QueryParser in this section assumes knowledge of the query

types previously discussed in section 3.4. We begin with a handy way to glimpse what QueryParser does to expressions.

3.5.1 Query.toString

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

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

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

The toString() methods (particularly the String-arg one) are handy for visual debugging of complex API queries as well as getting a handle on how QueryParser interprets query expressions. Don't rely on the ability to go back and forth accurately between a Query.toString() representation and a QueryParser-parsed expression, though. It's generally accurate, but an analyzer is involved and may confuse things; this issue is discussed further in section 4.1.2. Let's look next at QueryParser's Boolean operators.

3.5.2 Boolean operators

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

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

Placing a NOT in front of a term excludes documents matching the following term. Negating a term must be combined with at least one nonnegated term to return documents; in other words, it isn't possible to use a query like NOT term to find all documents that don't contain a term. Each of the uppercase word operators has shortcut syntax; table 3.7 illustrates various syntax equivalents.

Table 3.7 Boolean query operator shortcuts

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

One powerful feature of QueryParser is the ability to create nested clauses, using grouping

3.5.3 Grouping

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

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

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

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

3.5.4 Field selection

QueryParser needs to know the field name to use when constructing queries, but it would generally be unfriendly to require users to identify the field to search (the end user may not need or want to know the field names). As you've seen, the default field name is provided when you create the QueryParser instance. Parsed queries aren't restricted, however, to searching only the default field. Using field selector notation, you can specify terms in nondefault fields. For example, when HTML documents are indexed with the title and body areas as separate fields, the default field will likely be body. Users can search for title fields

using a query such as `title:lucene`. You can group field selection over several terms using `field:(a b c)`.

3.5.5 Range searches

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

Our `testRangeQuery()` method demonstrates both inclusive and exclusive range queries:

```
public void testRangeQuery() throws Exception {
  Query query = QueryParser.parse(
        "pubmonth:[200401 TO 200412]", "subject", analyzer);  |#1

  assertTrue(query instanceof RangeQuery);

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

  query = QueryParser.parse(
       "{200201 TO 200208}", "pubmonth", analyzer);   |#2

  hits = searcher.search(query);
  assertEquals("JDwA in 200208", 0, hits.length());   |#3
}
```

#1 Inclusive range, with square brackets
#2 Exclusive range, with curly brackets
#3 Demonstrates exclusion of pubmonth 200208

#1 This inclusive range uses a field selector since the default field is `subject`.
#2 This exclusive range uses the default field `pubmonth`.
#3 *Java Development with Ant* was published in August 2002, so we've demonstrated that the `pubmonth` value 200208 is excluded from the range.

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

HANDLING DATE RANGES

When a range query is encountered, the parser code first attempts to convert the start and end terms to dates. If the terms are valid dates, according to `DateFormat.SHORT` and lenient parsing within the default or specified locale, then the dates are converted to their internal textual representation. By default, this conversion will use the older `DateField.dateToString` method, which renders each date

with millisecond precison; this is likely not what you want. Instead, you should use QueryParser's setDateResolution methods to notify it what DateTools.Resolution your field(s) were indexed with (see Section 2.4 on DateTools). Then QueryParser will use the newer DateTools.dateToString method to translate the dates into strings with the appropriate resolution (see section 2.4 on DateTools). If either of the two terms fails to parse as a valid date, they're both used as is for a textual range.

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

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

This outputs something truly strange:

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

The reason for this is QueryParser by default uses the old DateField class to parse dates, which internally encodes each date as a long that is then rendered in hexadecimal format. If instead we set the proper resolution for our date field we get the expected result:

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

Produces this much nicer output:

```
modified:[20040101 TO 20050101]
```

Internally, all terms are text to Lucene, and dates are represented in a lexicographically ordered text format. As long as our modified field was indexed properly as the output of DateTools.dateToString, then after calling setDateResolution, QueryParser will produce the right range query.

CONTROLLING THE DATE-PARSING LOCALE

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

Listing 3.3 Using the client locale in a web application

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

    QueryParser parser = new QueryParser("contents",
```

```
            new StandardAnalyzer());

        parser.setLocale(request.getLocale());
        parser.setDateResolution(DateTools.Resolution.DAY);

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

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

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

`QueryParser` also creates `PhraseQuery`s, described next.

3.5.6 Phrase queries

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

```
    public void testPhraseQuery() throws Exception {
      Query q = new QueryParser("field", new StandardAnalyzer()).parse("\"This is Some
  Phrase*\"");
      assertEquals("analyzed",
          "\"some phrase\"", q.toString("field"));

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

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

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

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

A sloppy `PhraseQuery`, as noted, doesn't require that the terms match in the same order. However, a `SpanNearQuery` (discussed in section 5.4.3) has the ability to guarantee an in-order match. In section 6.3.4, we extend `QueryParser` and substitute a `SpanNearQuery` when phrase queries are parsed, allowing for sloppy in-order phrase matches. The final queries we discuss are `WildcardQuery`, `PrefixQuery` and `FuzzyQuery`, all of which `QueryParser` can create.

3.5.7 Wildcard and prefix queries

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

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

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

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

3.5.8 Fuzzy queries

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

3.5.9 Boosting queries

A carat (^) followed by a floating-point number sets the boost factor for the preceding query. Section 3.3 discusses boosting queries in more detail. For example, the query expression `junit^2.0 testing` sets the `junit TermQuery` to a boost of 2.0 and leaves the testing `TermQuery` at the default boost of 1.0. You can apply a boost to any type of query, including parenthetical groups.

3.5.10 To QueryParse or not to QueryParse?

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

Should you require different expression syntax or capabilities beyond what `QueryParser` offers, technologies such as ANTLR[4] and JFlex[5] are great options. We don't discuss the creation of a custom query parser; however, the source code for Lucene's `QueryParser` is freely available for you to borrow from.

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

3.6 Summary

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

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

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

---

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

[5] http://jflex.de/

4

Analysis

This chapter covers

- Understanding the analysis process
- Exploring QueryParser issues
- Writing custom analyzers
- Handling foreign languages

Analysis, in Lucene, is the process of converting field text into its most fundamental indexed representation, *terms*. These terms are used to determine what documents match a query during searches. For example, if this sentence were indexed into a field the terms might start with *for* and *example*, and so on, as separate terms in sequence. An *analyzer* is an encapsulation of the analysis process. An analyzer tokenizes text by performing any number of operations on it, which could include extracting words, discarding punctuation, removing accents from characters, lowercasing (also called *normalizing*), removing common words, reducing words to a root form (*stemming*), or changing words into the basic form (*lemmatization*). This process is also called *tokenization*, and the chunks of text pulled from a stream of text are called *tokens*. Tokens, combined with their associated field name, are terms.

Lucene's primary goal is to facilitate information *retrieval*. The emphasis on retrieval is important. You want to throw gobs of text at Lucene and have them be richly searchable by the individual words within that text. In order for Lucene to know what "words" are, it *analyzes* the text during indexing, extracting it into terms. These terms are the primitive building blocks for searching.

Choosing the right analyzer is a crucial development decision with Lucene, and one size definitely doesn't fit all. Language is one factor, because each has its own unique features. Another factor to consider is the domain of the text being analyzed; different industries have different terminology, acronyms, and abbreviations that may deserve attention. Although we present many of the considerations for choosing analyzers, no single analyzer will suffice for all situations. It's possible that none of the built-

in analysis options are adequate for your needs, and you'll need to invest in creating a custom analysis solution; pleasantly, Lucene's building blocks make this quite easy.

In this chapter, we'll cover all aspects of the Lucene analysis process, including how and where to use analyzers, what the built-in analyzers do, and how to write your own custom analyzers using the building blocks provided by the core Lucene API. Let's begin by seeing when and how `Analyzers` are used by Lucene.

4.1 Using analyzers

Before we get into the gory details of what lurks inside an analyzer, let's look at how an analyzer is used within Lucene. Analysis occurs any time text needs to be converted into terms, which in Lucene's core is at two spots: during indexing and when using `QueryParser` for searching. In the following two sections, we detail how an analyzer is used in these scenarios. In the last section we describe an important difference between parsing and analyzing a document.

If you highlight hits in your search results, which is strongly recommended as it gives a better end user experience, you may need to analyze text at that point as well. Highlighting, available in Lucene's sandbox, is covered in detail in section XXX.

Before we begin with any code details, look at listing 4.1 to get a feel for what the analysis process is all about. Two phrases are analyzed, each by four of the built-in analyzers. The phrases are "The quick brown fox jumped over the lazy dogs" and "XY&Z Corporation - xyz@example.com". Each token is shown between square brackets to make the separations apparent. During indexing, the tokens extracted during analysis are the terms indexed. And, most important, it's only the terms that are indexed that are searchable!

NOTE

Only the tokens produced by the analyzer will be searched.

Listing 4.1 Visualizing analyzer effects

```
Analyzing "The quick brown fox jumped over the lazy dogs"
  WhitespaceAnalyzer :
    [The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

  SimpleAnalyzer :
    [the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

  StopAnalyzer :
    [quick] [brown] [fox] [jumped] [over] [lazy] [dogs]

  StandardAnalyzer:
    [quick] [brown] [fox] [jumped] [over] [lazy] [dogs]

Analyzing "XY&Z Corporation - xyz@example.com"
  WhitespaceAnalyzer:
    [XY&Z] [Corporation] [-] [xyz@example.com]
```

```
SimpleAnalyzer:
  [xy] [z] [corporation] [xyz] [example] [com]

StopAnalyzer:
  [xy] [z] [corporation] [xyz] [example] [com]

StandardAnalyzer:
  [xy&z] [corporation] [xyz@example.com]
```

The code that generated this analyzer output is shown later, in listing 4.2. A few interesting things happen in this example. Look at how the word *the* is treated, and likewise the company name XY&Z and the e-mail address xyz@example.com; look at the special hyphen character (-) and the case of each token. Section 4.2.3 explains more of the details of what happened, but here's a quick summary of each of these analyzers:

- `WhitespaceAnalyzer`, as the name implies, simply splits text into tokens on whitespace characters and makes no other effort to normalize the tokens.
- `SimpleAnalyzer` first splits tokens at non-letter characters, then lowercases each token. Be careful! This analyzer quietly discards numeric characters.
- `StopAnalyzer` is the same as `SimpleAnalyzer`, except it removes common words (called stop words, described more in section XXX). By default it removes common words in the English language (*the*, *a*, etc.), though you can pass in your own set.
- `StandardAnalyzer` is Lucene's most sophisticated core analyzer. It has quite a bit of logic to identify certain kinds of tokens, such as company names, email addresses, and host names. It also lowercases each token and removes stop words.

Lucene doesn't make the results of the analysis process visible to the end user. Terms pulled from the original text are immediately added to the index. It is these terms, and only these terms, that are matched during searching. When searching with `QueryParser`, the analysis process happens again, on the textual parts of the search query, in order to ensure the best possible matches.

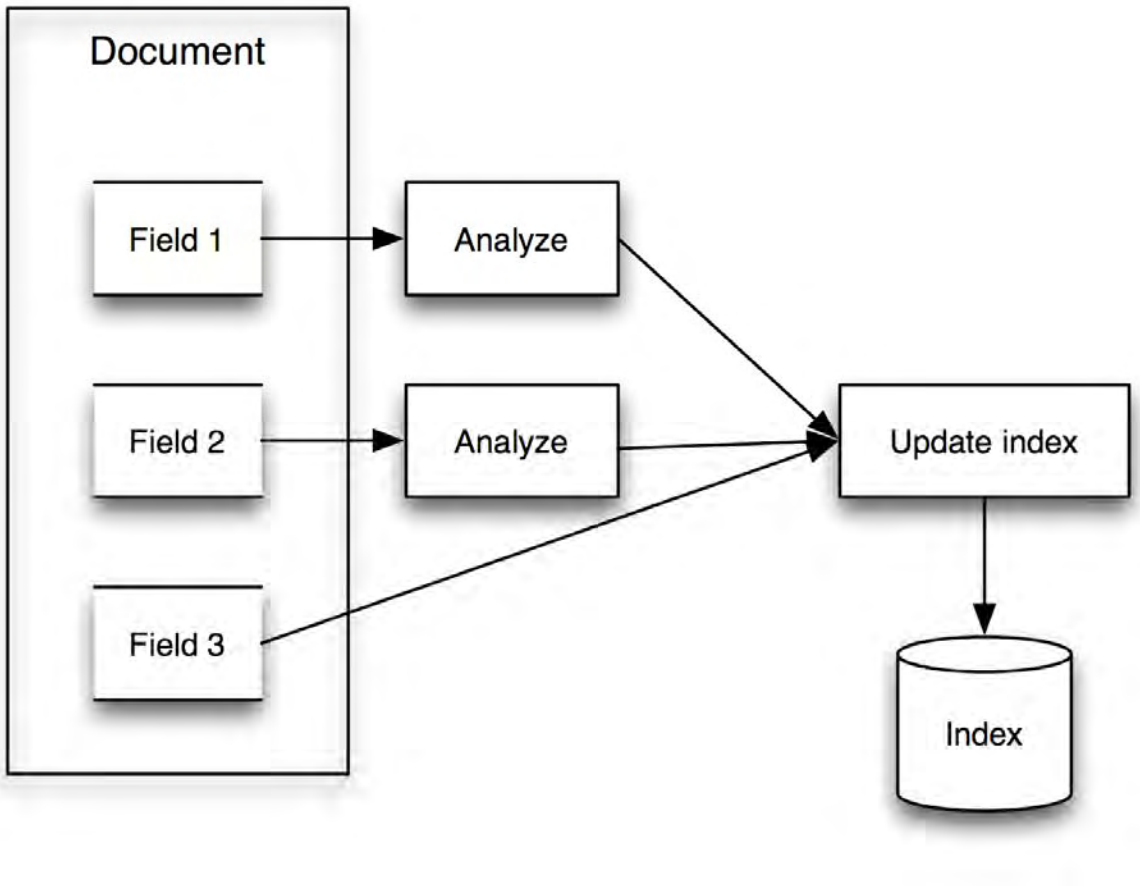


Figure 4.1: Analysis process during indexing. Field 1 and 2 are analyzed; Field 3 is unanalyzed.

Let's look at how the analyzer is used during indexing.

4.1.1 Indexing analysis

During indexing text contained in the document's field values must be converted into tokens, as shown in Figure 4.1. You give `IndexWriter` an `Analyzer` instance up front:

```
Analyzer analyzer = new StandardAnalyzer();
IndexWriter writer = new IndexWriter(directory, analyzer,
IndexWriter.MaxFieldLength.UNLIMITED);
```

In this example, we use the built-in `StandardAnalyzer`, one of the several available within the core Lucene library. Each analyzed field of each document indexed with the `IndexWriter` instance uses the

analyzer specified, by default. To make sure the text is analyzed, pass `Field.Index.ANALYZED` as the 4th argument when creating the field. To index the entire field's value as a single token, like Field 3 in Figure 4.1, pass `Field.Index.NOT_ANALYZED` as the 4th argument.

NOTE

`new Field(String, String, Field.Store.YES, Field.Index.ANALYZED)` creates a tokenized *and* stored field. Rest assured the *original* `String` value is stored. However, the output of the designated `Analyzer` dictates what is indexed.

The following code demonstrates indexing of a document where one field is analyzed and stored, and the second field is analyzed but not stored:

```
Document doc = new Document();
doc.add(new Field("title", "This is the title", Field.Store.YES,
                  Field.Index.ANALYZED));
doc.add(new Field("contents", "...document contents...", Field.Store.NO,
                  Field.Index.ANALYZED));
writer.addDocument(doc);
```

Both `"title"` and `"contents"` are analyzed using the `Analyzer` instance provided to the `IndexWriter`. However, if an individual document has special analysis needs, the analyzer may be specified on a per-document basis, like this:

```
writer.addDocument(doc, analyzer);
```

`QueryParser` must also use an `Analyzer` to parse fragments of the user's textual query.

4.1.2 QueryParser analysis

`QueryParser` is wonderful for presenting the end user with a free-form option of querying. To do its job, of course, `QueryParser` uses an analyzer to break the text it encounters into terms for searching. You must provide an analyzer when you instantiate the `QueryParser`:

```
QueryParser parser = new QueryParser("contents", analyzer);
Query query = parser.parse(expression);
```

The analyzer receives individual contiguous text pieces of the expression, not the expression as a whole, which in general may include operators, parenthesis, and other special expression syntax to denote range, wildcard, and fuzzy searches.

`QueryParser` analyzes all text equally, without knowledge of how it was indexed. This is a particularly thorny issue when you're querying for fields that were indexed without tokenization. We address this situation in section 4.4.

Should you use the same analyzer with `QueryParser` that you used during indexing? The short, most accurate, answer is, "it depends." If you stick with the basic built-in analyzers, then you'll probably be fine

using the same analyzer in both situations. However, when you're using more sophisticated analyzers, quirky cases can come up in which using different analyzers between indexing and `QueryParser` is best. We discuss this issue in more detail in section 4.6. Now we draw the difference between parsing and analyzing a document.

4.1.3 Parsing versus analysis: when an analyzer isn't appropriate

An important point about analyzers is that they're used internally for fields enabled for analysis. Documents such as HTML, Microsoft Word, XML, and others, contain meta-data such as author, title, last modified date, and potentially much more. When you're indexing rich documents, this meta-data should be separated and indexed as separate fields. Analyzers are used to analyze a specific field at a time and break things into tokens only within that field; creating new fields isn't possible within an analyzer.

Analyzers don't help in field separation because their scope is to deal with a single field at a time. Instead, *parsing* these documents prior to analysis is required. For example, it's a common practice to separate at least the `<title>` and `<body>` of HTML documents into separate fields. In these cases, the documents should be parsed, or preprocessed, into separate blocks of text representing each field. Chapter 7 covers this pre-processing step in detail.

Now that we've seen where and how Lucene uses analyzers, it's time to delve into just what an analyzer does and how it works.

4.2 Analyzing the analyzer

In order to do understand the analysis process, we need to open the hood and tinker around a bit. Because it's possible that you'll be constructing your own analyzers, knowing the architecture and building blocks provided is crucial.

The `Analyzer` class is the base class. Quite elegantly, it turns text into a stream of tokens enumerated by the `TokenStream` class. The single required method signature implemented by analyzers is:

```
public TokenStream tokenStream(String fieldName, Reader reader)
```

The returned `TokenStream` is then used to iterate through all tokens.

Let's start "simply" with the `SimpleAnalyzer` and see what makes it tick. The following code is copied directly from Lucene's codebase:

```
public final class SimpleAnalyzer extends Analyzer {
  public TokenStream tokenStream(String fieldName, Reader reader) {
    return new LowerCaseTokenizer(reader);
  }
  public TokenStream reusableTokenStream(String fieldName, Reader reader) throws
IOException {
    Tokenizer tokenizer = (Tokenizer) getPreviousTokenStream();
    if (tokenizer == null) {
      tokenizer = new LowerCaseTokenizer(reader);
      setPreviousTokenStream(tokenizer);
    } else
      tokenizer.reset(reader);
```

```
      return tokenizer;
    }
  }
```

The `LowerCaseTokenizer` divides text at nonletters (determined by `Character.isLetter`), removing nonletter characters and, true to its name, lowercasing each character.

What is the `reusableTokenStream` method? That is an additional, optional method that an `Analyzer` can implement to gain better indexing performance. That method is allowed to re-use the same `TokenStream` that it had previously returned to the same thread. This can save a lot of allocation and garbage collection since every field of every document otherwise needs a new `TokenStream`. It's free to use two utility methods implemented in the `Analyzer` base class, `setPreviousTokenStream` and `getPreviousTokenStream`, to store and retrieve a `TokenStream` in thread local storage. All of the built-in Lucene analyzers implement this method: the first time the method is called from a given thread, a new `TokenStream` instance is created and saved away. Subsequent calls simply return the previous `TokenStream` after resetting it to the new `Reader`.

In the following sections, we take a detailed look at each of the major players used by analyzers, including `Token` and the `TokenStream` family, plus the `Attributes` that represent the components of a `Token`. We'll also show you how to visualize what an analyzer is actually doing, and describe the important of the order of `Tokenizers`. Lets begin with the basic unit of analysis, the `Token`.

4.2.1 What's in a token?

A stream of tokens is the fundamental output of the analysis process. During indexing, fields designated for tokenization are processed with the specified analyzer, and each token is then written into the index.

For example, let's analyze the text "the quick brown fox". Each token represents an individual word of that text. A token carries with it a text value (the word itself) as well as some meta-data: the start and end character offsets in the original text, a token type, and a position increment. Figure 4.1 shows the details of the token stream analyzing this phrase with the `SimpleAnalyzer`.

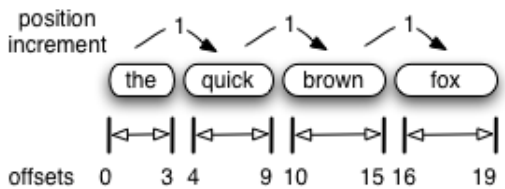


Figure 4.1 Token stream with positional and offset information

The start offset is the character position in the original text where the token text begins, and the end offset is the position just after the last character of the token text. These offsets are useful for highlighting matched tokens in search results, as described in 8.XXX. The token type is a `String`, defaulting to `"word"`, that you can control and use in the token-filtering process if desired. As text is tokenized, the position relative to the previous token is recorded as the *position increment* value. All the built-in

tokenizers leave the position increment at the default value of 1, indicating that all tokens are in successive positions, one after the other.

TOKENS INTO TERMS

After text is analyzed during indexing, each token is posted to the index as a term.. The position increment and start and end offsets are the *only* additional meta-data associated with the token carried through to the index. The token type is discarded—it's only used during the analysis process.

POSITION INCREMENTS

The token position increment value relates the current token's position to the previous token's position. Position increment is usually 1, indicating that each word is in a unique and successive position in the field. Position increments factor directly into performing phrase queries (see section 3.4.5) and span queries (see section 5.4), which rely on knowing how far terms are from one another within a field.

Position increments greater than 1 allow for gaps and can be used to indicate where words have been removed. See section 4.7.1 for an example of stop-word removal that leaves gaps using position increments.

A token with a zero position increment places the token in the same position as the previous token. Analyzers that inject synonyms can use a position increment of zero for the synonyms. The effect is that phrase queries work regardless of which synonym was used in the query. See our `SynonymAnalyzer` in section 4.6 for an example that uses position increments of zero.

4.2.2 TokenStream uncensored

There are two different styles of `TokenStream`s: `Tokenizer` and `TokenFilter`. A good generalization to explain the distinction is that `Tokenizer`s deal with individual characters, and `TokenFilter`s deal with words. `Tokenizer`s produce a new `TokenStream`, while `TokenFilter`s simply filter the tokens from a prior `TokenStream`. Figure 4.2 shows this architecture graphically.

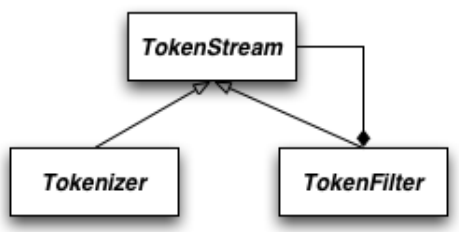


Figure 4.2 `TokenStream` architecture: `TokenFilter`s filter a `TokenStream`.

A `Tokenizer` is a `TokenStream` that tokenizes the input from a `Reader`. When you're indexing a `String`, Lucene wraps the `String` in a `StringReader` for tokenization. The second style of `TokenStream`, `TokenFilter`, lets you chain `TokenStream`s together. This powerful mechanism lives up

to its namesake as a stream filter. A `TokenStream` is fed into a `TokenFilter`, giving the filter a chance to add, remove, or change the stream as it passes through.

Figure 4.3 shows the `Tokenizer` and `TokenFilter` inheritance hierarchy within Lucene. Note the composite pattern used by `TokenFilter` to encapsulate another `TokenStream` (which could, of course, be another `TokenFilter`). Table 4.1 provides detailed descriptions for each of the classes shown in figure 4.3.

Table 4.1 Analyzer building blocks provided in Lucene's core API

Class name	Description
TokenStream	Base class with `next()` and `close()` methods.
Tokenizer	**TokenStream whose input is a Reader.**
CharTokenizer	Parent class of character-based tokenizers, with abstract `isTokenChar()` method. Emits tokens for contiguous blocks when `isTokenChar()` `returns true`. Also provides the capability to normalize (for example, lowercase) characters. `Tokens` are limited to a maximum size of 255 characters.
WhitespaceTokenizer	`CharTokenizer` with `isTokenChar()` true for all nonwhitespace characters.
KeywordTokenizer	`Tokenizes the entire input string as a single token.`
LetterTokenizer	`CharTokenizer` with `isTokenChar()` true when `Character.isLetter` is true.
LowerCaseTokenizer	`LetterTokenizer` that normalizes all characters to lowercase.
SinkTokenizer	A Tokenizer that absorbs tokens, caches them in a private list, and then can later iterate over the tokens it had previously cached. This is used in conjunction with TeeTokenizer to "split" a Token stream.

StandardTokenizer	Sophisticated grammar-based tokenizer, emitting tokens for high-level types like e-mail addresses (see section 4.3.2 for more details). Each emitted token is tagged with a special type, some of which are handled specially by `StandardFilter`.
TokenFilter	**TokenStream whose input is another TokenStream.**
LowerCaseFilter	Lowercases token text.
StopFilter	Removes words that exist in a provided set of words.
PorterStemFilter	Stems each token using the Porter stemming algorithm. For example, *country* and *countries* both stem to *countri*.
TeeTokenFilter	Splits a Token stream, by passing each token it iterates through into a SinkTokenizer, and also returning the Token unmodified to its caller.
ASCIIFoldingFilter	Maps accented unicode characters to their unaccented counterparts.
CachingTokenFilter	Saves all tokens from the input stream and can then replay the stream back over and over.
LengthFilter	Only accepts tokens whose text length falls within a specified range.
StandardFilter	Designed to be fed by a `StandardTokenizer`. Removes dots from acronyms and *'s* (apostrophe followed by *S*) from words with apostrophes.

Taking advantage of the `TokenFilter` chaining pattern, you can build complex analyzers from simple `Tokenizer`/`TokenFilter` building blocks. `Tokenizers` start the analysis process by churning the character input into *tokens* (mostly these correspond to words in the original text). `TokenFilters` then take over the remainder of the analysis, initially wrapping a `Tokenizer` and successively wrapping nested `TokenFilters`. Thus, the purpose of an analyzer is to simply define this analyzer chain (`TokenStream` followed by a series of `TokenFilters`) and implement it in the `tokenStream` method. To illustrate this in code, here is the analyzer chain returned by `StopAnalyzer`:

```
public TokenStream tokenStream(String fieldName, Reader reader) {
```

```
    return new StopFilter(
                new LowerCaseTokenizer(reader),
                stopWords);
  }
```

In `StopAnalyzer`, a `LowerCaseTokenizer` feeds a `StopFilter`. The `LowerCaseTokenizer` emits tokens that are adjacent letters in the original text, lowercasing each of the characters in the process. Nonletter characters form token boundaries aren't included in any emitted token. Following this word tokenizer and lowercasing, `StopFilter` removes words in a stop-word list (see section 4.3.1).

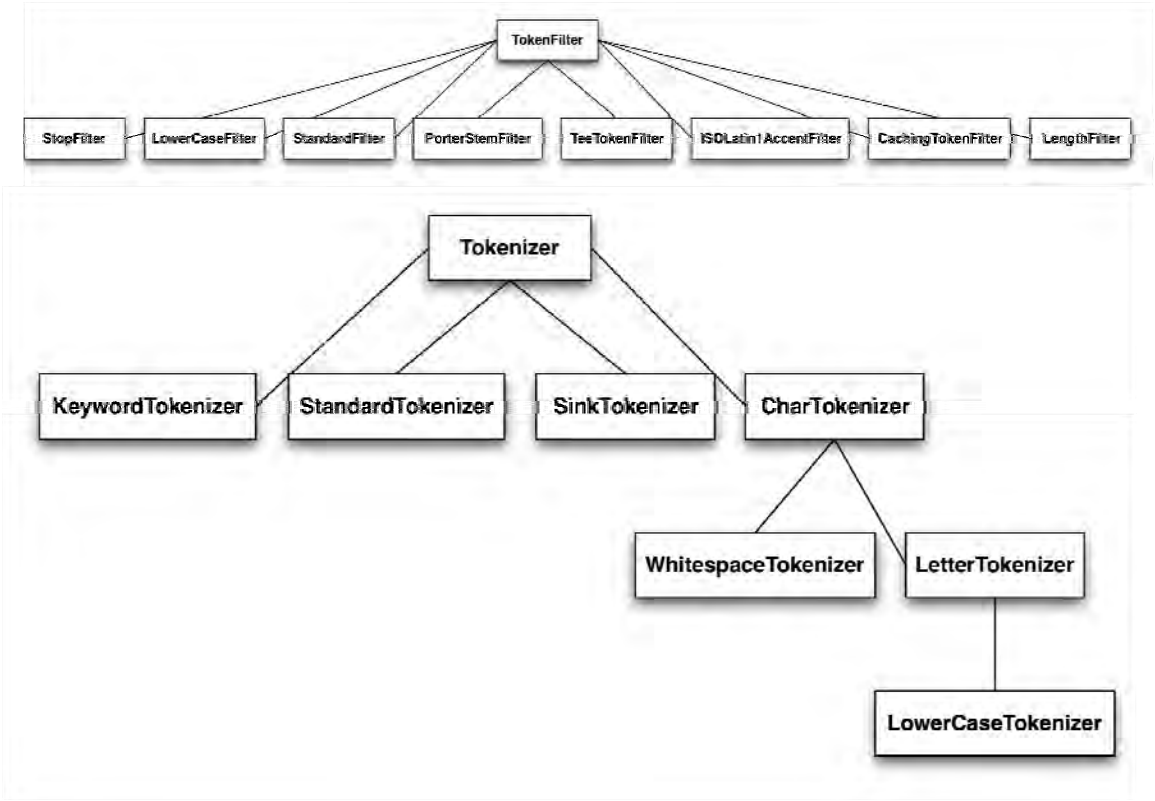


Figure 4.3 `TokenFilter` and `Tokenizer` class hierarchy

Buffering is a feature that's commonly needed in the `TokenStream` implementations. Low-level `Tokenizers` do this to buffer up characters to form tokens at boundaries such as whitespace or nonletter characters. `TokenFilters` that emit additional tokens into the stream they're filtering must queue an

incoming token and the additional ones and emit them one at a time; our `SynonymFilter` in section 4.6 is an example of a queuing filter.

Most of the builtin `TokenFilters` simply alter a single stream of input tokens in some fashion, but two of them are more interesting. `TeeTokenFilter` is a filter that splits the incoming token stream into two output streams, one of which is a `SinkTokenizer`. This is useful when two or more fields would like to share the same basic initial analysis steps, but differ on the final processing of the tokens.

Next we describe how to see the results of the analysis process.

4.2.3 Visualizing analyzers

It's important to understand what various analyzers do with your text. Seeing the effect of an analyzer is a powerful and immediate aid to this understanding. We'll also describe the `Attribute` class, which represents each element of a `Token`, and we'll discuss each of the `Token`'s attributes: `term`, `positionIncrement`, `offset`, `type`, `flags` and `payload`. Listing 4.2 provides a quick and easy way to get visual feedback about the four primary built-in analyzers on a couple of text examples. `AnalyzerDemo` includes two predefined phrases and an array of the four analyzers we're focusing on in this section. Each phrase is analyzed by all the analyzers, with bracketed output to indicate the terms that would be indexed.

Listing 4.2 `AnalyzerDemo`: seeing analysis in action

```
/**
 * Adapted from code which first appeared in a java.net article
 * written by Erik
 */
public class AnalyzerDemo {
  private static final String[] examples = {
    "The quick brown fox jumped over the lazy dogs",
    "XY&Z Corporation - xyz@example.com"
  };

  private static final Analyzer[] analyzers = new Analyzer[] {
    new WhitespaceAnalyzer(),
    new SimpleAnalyzer(),
    new StopAnalyzer(),
    new StandardAnalyzer()
  };
  public static void main(String[] args) throws IOException {
    // Use the embedded example strings, unless
    // command line arguments are specified, then use those.
    String[] strings = examples;
    if (args.length > 0) {
      strings = args;
    }

    for (int i = 0; i < strings.length; i++) {
      analyze(strings[i]);
    }
  }

  private static void analyze(String text) throws IOException {
```

```
      System.out.println("Analyzing \"" + text + "\"");
      for (int i = 0; i < analyzers.length; i++) {
        Analyzer analyzer = analyzers[i];
        String name = analyzer.getClass().getName();
        name = name.substring(name.lastIndexOf(".") + 1);
        System.out.println("  " + name + ":");
        System.out.print("    ");
        AnalyzerUtils.displayTokens(analyzer, text);
        System.out.println("\n");
      }
    }
  }
```

The real fun happens in `AnalyzerUtils` (listing 4.3), where the analyzer is applied to the text and the tokens are extracted. `AnalyzerUtils` passes text to an analyzer without indexing it and pulls the results in a manner similar to what happens during the indexing process under the covers of `IndexWriter`.

Listing 4.3 AnalyzerUtils: delving into an analyzer

```
public class AnalyzerUtils {
  public static AttributeSource[] tokensFromAnalysis(Analyzer analyzer,
                                                     String text) throws IOException {
    TokenStream stream =                                              //1
           analyzer.tokenStream("contents", new StringReader(text));  //1
    ArrayList tokenList = new ArrayList();
    while (true) {
      if (!stream.incrementToken())
        break;

      tokenList.add(stream.captureState());
    }

    return (AttributeSource[]) tokenList.toArray(new AttributeSource[0]);
  }

  public static void displayTokens(Analyzer analyzer,
                                   String text) throws IOException {
    AttributeSource[] tokens = tokensFromAnalysis(analyzer, text);

    for (int i = 0; i < tokens.length; i++) {
      AttributeSource token = tokens[i];
      TermAttribute term = (TermAttribute) token.addAttribute(TermAttribute.class);
      System.out.print("[" + term.term() + "] ");    //2
    }
  }
}
```

#1 Invoke analysis process
#2 Output token text surrounded by brackets

Generally you wouldn't invoke the analyzer's `tokenStream` method explicitly except for this type of diagnostic or informational purpose (and the field name *contents* is arbitrary in the `tokensFromAnalysis()` method).

`AnalyzerDemo` produced the output shown in listing 4.1. Some key points to note are as follows:

- `WhitespaceAnalyzer` didn't lowercase, left in the dash, and did the bare minimum of tokenizing at whitespace boundaries.
- `SimpleAnalyzer` left in what may be considered irrelevant (stop) words, but it did lowercase and tokenize at nonalphabetic character boundaries.
- Both `SimpleAnalyzer` and `StopAnalyzer` mangled the corporation name by splitting `XY&Z` and removing the ampersand.
- `StopAnalyzer` and `StandardAnalyzer` threw away occurrences of the word *the*.
- `StandardAnalyzer` kept the corporation name intact and lowercased it, removed the dash, and kept the e-mail address together. No other built-in analyzer is this thorough.

We recommend keeping a utility like this handy to see what tokens emit from your analyzers of choice. In fact, rather than write this yourself, you can use our `AnalyzerUtils` or the `AnalyzerDemo` code for experimentation. The `AnalyzerDemo` application lets you specify one or more strings from the command line to be analyzed instead of the embedded example ones:

```
% java lia.analysis.AnalyzerDemo "No Fluff, Just Stuff"

Analyzing "No Fluff, Just Stuff"
  org.apache.lucene.analysis.WhitespaceAnalyzer:
    [No] [Fluff,] [Just] [Stuff]

  org.apache.lucene.analysis.SimpleAnalyzer:
    [no] [fluff] [just] [stuff]

  org.apache.lucene.analysis.StopAnalyzer:
    [fluff] [just] [stuff]

  org.apache.lucene.analysis.standard.StandardAnalyzer:
    [fluff] [just] [stuff]
```

Let's now look deeper into what makes up a `Token`.

LOOKING INSIDE TOKENS

`TokenStream`s iterate through tokens and `TokenFilter`s may access and alter their meta-data. To demonstrate accessing token meta-data, we added the `displayTokensWithFullDetails` utility method in `AnalyzerUtils`:

```
public static void displayTokensWithFullDetails(Analyzer analyzer,
                                                 String text) throws IOException {
  AttributeSource[] tokens = tokensFromAnalysis(analyzer, text);

  int position = 0;

  for (int i = 0; i < tokens.length; i++) {
    AttributeSource token = tokens[i];

    TermAttribute term = (TermAttribute) token.addAttribute(TermAttribute.class);
```

```
      PositionIncrementAttribute posIncr =
        (PositionIncrementAttribute)
        token.addAttribute(PositionIncrementAttribute.class);

      OffsetAttribute offset = (OffsetAttribute)
token.addAttribute(OffsetAttribute.class);

      TypeAttribute type = (TypeAttribute) token.addAttribute(TypeAttribute.class);

      int increment = posIncr.getPositionIncrement();

      if (increment > 0) {
        position = position + increment;
        System.out.println();
        System.out.print(position + ": ");
      }

      System.out.print("[" +
                       term.term() + ":" +
                       offset.startOffset() + "->" +
                       offset.endOffset() + ":" +
                       type.type() + "] ");
    }
    System.out.println();
  }
```

We display all token information on the example phrase using `SimpleAnalyzer`:

```
  public static void main(String[] args) throws IOException {
    displayTokensWithFullDetails(new SimpleAnalyzer(),
        "The quick brown fox....");
  }
```

Here's the output:

```
1: [the:0->3:word]
2: [quick:4->9:word]
3: [brown:10->15:word]
4: [fox:16->19:word]
```

Each token is in a successive position relative to the previous one (noted by the incrementing numbers 1, 2, 3, and 4). The word *the* begins at offset 0 and ends just before offset 3 in the original text. Each of the tokens has a type of `word`. We present a similar, but simpler, visualization of token position increments in section 4.6.1, and we provide a visualization of tokens sharing the same position.

Table XXX Lucene's builtin token attributes

Token attribute class	Description
`TermAttribute`	Token's text

`PositionIncrementAttribute`	Position increment (defaults to 1)
`OffsetAttribute`	Start and end character offset
`TypeAttribute`	Token's type (defaults to "word")
`FlagsAttribute`	Bits to encode custom flags
`PayloadAttribute`	Per-token byte[] payload (see section 6.XXX)

ATTRIBUTES

Notice that the `TokenStream` never explicitly creates a Token object. Instead, it extends the `AttributeSource` class, which you use to obtain the different attributes (text, offsets, position increments, etc.) of interest. Table XXX shows the attributes that Lucene's core uses. Past versions of Lucene did use a `Token` object, but in order to be more extensible, and to provide better analysis performance through reuse, Lucene switched to the `AttributeSource` API as of 2.9.

With this API, in order to step through each token you invoke `TokenStream.incrementToken`. This method returns true if there is a new token and false if you've exhausted the stream. You obtain the attributes of interest by calling the `addAttribute` method, which will return a subclass of `Attribute`. That method will add the requested attribute, with default values, if it's not already present, and then return the attribute. You then interact with that subclass to obtain the value for each token. When `incrementToken` returns true, all attributes within it will have altered their state to the next token.

Typically you would obtain the attributes up front, and then iterate through the tokens, asking each attribute for its values. For example, if you're only interested in the position increment, you could simply do this:

```
TokenStream stream = analyzer.tokenStream("contents", new StringReader(text));
PositionIncrementAttribute posIncr = (PositionIncrementAttribute)
  stream.addAttribute(PositionIncrementAttribute.class);
while (stream.incrementToken()) {
  System.out.println("posIncr=" + posIncr.getPositionIncrement());
}
```

Each attribute class is actually bidirectional: you use each to get and to set the value for that attribute. Thus, a `TokenFilter` that would like to alter only the position increment would grab and store the `PositionIncrementAttribute` when it's first instantiated, then implement the `incrementToken` method by first calling `incrementToken` on its input stream and then using `PositionIncrementAttribute.setPositionIncrement` to change the value.

As you can see in our `displayTokensWithFullDetails` method, it's also possible to make a full copy of the attributes for each token, as we do in the `tokensFromAnalysis` method, by calling `TokenStream.captureState`. This returns an `AttributeSource` instance that holds a private copy of all attributes as of when it was called. Generally this is not recommended as it results in much slower performance, but for diagnostic purposes it's fine.

By providing an extensible attribute-based API, Lucene allows you to create your own attributes. Thus, if the existing attributes are not enough, you can simply create your own attribute by subclassing Lucene's `Attribute` class (in `org.apache.lucene.util`). Note that Lucene will do nothing with your attribute during indexing, so this is really only currently useful in cases where one `TokenStream` early in your analysis chain wishes to send information to another `TokenStream` later in the chain.

WHAT GOOD ARE START AND END OFFSETS?

The start and end offset values aren't used in the core of Lucene; they are treated as opaque and you could in fact put any integers you'd like into there. If you index with `TermVectors`, as described in section 2.8, you can store token text, offsets and position information in your index for the fields you specify. Then, at search time, `TermVectors` can be used for highlighting matches in text, as discussed in section 8.7. In this case, the stored offsets hold the start and end character offset in the original text for each token, which the highlighter then uses to make each matched token stand out in the search results. It's also possible to re-analyze the text to do highlighting without storing `TermVectors`, in which case the start and end offsets are used in real-time.

TOKEN-TYPE USEFULNESS

You can use the token-`type` value to denote special lexical types for tokens. Under the covers of `StandardAnalyzer` is a `StandardTokenizer` that parses the incoming text into different types based on a grammar. Analyzing the phrase "I'll e-mail you at xyz@example.com" with `StandardAnalyzer` produces this interesting output:

```
1: [i'll:0->4:<APOSTROPHE>]
2: [e:5->6:<ALPHANUM>]
3: [mail:7->11:<ALPHANUM>]
4: [you:12->15:<ALPHANUM>]
5: [xyz@example.com:19->34:<EMAIL>]
```

Notice the token type of each token. The token `i'll` has an apostrophe, which `StandardTokenizer` notices in order to keep it together as a unit; and likewise for the e-mail address. We cover the other `StandardAnalyzer` effects in section 4.3.2. `StandardAnalyzer` is the only built-in analyzer that leverages the token-type data. Our Metaphone and synonym analyzers, in sections 4.5 and 4.6, provide another example of token type usage.

4.2.4 Filtering order can be important

The order of events can be critically important during analysis. Each step may rely on the work of a previous step. A prime example is that of stop-word removal. `StopFilter` does a case-sensitive look-up of each token in a set of stop words. It relies on being fed lowercased tokens. As an example, we first write a functionally equivalent `StopAnalyzer` variant; we'll follow it with a flawed variant that reverses the order of the steps:

```
public class StopAnalyzer2 extends Analyzer {

  private Set stopWords;
```

```
    public StopAnalyzer2() {
      stopWords = StopFilter.makeStopSet(StopAnalyzer.ENGLISH_STOP_WORDS);
    }

    public StopAnalyzer2(String[] stopWords) {
      this.stopWords = StopFilter.makeStopSet(stopWords);
    }

    public TokenStream tokenStream(String fieldName, Reader reader) {
      return new StopFilter(new LowerCaseFilter(new LetterTokenizer(reader)),
          stopWords);
    }
  }
```

`StopAnalyzer2` uses a `LetterTokenizer` feeding a `LowerCaseFilter`, rather than just a `LowerCaseTokenizer`. A `LowerCaseTokenizer`, however, has a performance advantage since it lowercases as it tokenizes, rather than dividing the process into two steps. This test case proves that our `StopAnalyzer2` works as expected, by using `AnalyzerUtils.tokensFromAnalysis` and asserting that the stop word *the* was removed:

```
    public void testStopAnalyzer2() throws Exception {
      AttributeSource[] tokens =
        AnalyzerUtils.tokensFromAnalysis(
          new StopAnalyzer2(), "The quick brown...");

        AnalyzerUtils.assertTokensEqual(tokens,
                                  new String[] {"quick", "brown"});
    }
```

We've added a utility method to our `AnalyzerUtils` that asserts tokens match an expected list:

```
   public static void assertTokensEqual(
                  AttributeSource[] tokens, String[] strings) {
      Assert.assertEquals(strings.length, tokens.length);

      for (int i = 0; i < tokens.length; i++) {
        AttributeSource token = tokens[i];
        TermAttribute term = (TermAttribute) token.getAttribute(TermAttribute.class);

        Assert.assertEquals("index " + i,
                            strings[i], term.term());
      }
    }
```

To illustrate the importance that the order can make with token filtering, we've written a flawed analyzer that swaps the order of the `StopFilter` and the `LowerCaseFilter`:

```
  /**
   * Stop words actually not necessarily removed due to filtering order
   */
  public class StopAnalyzerFlawed extends Analyzer {
    private Set stopWords;
```

```
    public StopAnalyzerFlawed() {
      stopWords =
          StopFilter.makeStopSet(StopAnalyzer.ENGLISH_STOP_WORDS);
    }

    public StopAnalyzerFlawed(String[] stopWords) {
      this.stopWords = StopFilter.makeStopSet(stopWords);
    }

    /**
     * Ordering mistake here
     */
    public TokenStream tokenStream(String fieldName, Reader reader) {
      return new LowerCaseFilter(
          new StopFilter(new LetterTokenizer(reader),
          stopWords));
    }
  }
```

The `StopFilter` presumes all tokens have already been lowercased and does a case-sensitive lookup. Another test case shows that *The* was not removed (it's the first token of the analyzer output), yet it was lowercased:

```
    public void testStopAnalyzerFlawed() throws Exception {
      AttributeSource[] tokens =
        AnalyzerUtils.tokensFromAnalysis(
          new StopAnalyzerFlawed(), "The quick brown...");

      TermAttribute termAttr = (TermAttribute)
  tokens[0].addAttribute(TermAttribute.class);
      assertEquals("the", termAttr.term());
    }
```

Lowercasing is just one example where order may matter. Filters may assume previous processing was done. For example, the `StandardFilter` is designed to be used in conjunction with `StandardTokenizer` and wouldn't make sense with any other `TokenStream` feeding it. There may also be performance considerations when you order the filtering process. Consider an analyzer that removes stop words and also injects synonyms into the token stream—it would be more efficient to remove the stop words first so that the synonym injection filter would have fewer terms to consider (see section 4.6 for a detailed example).

Now that we've seen in great detail all elements of the `Token` and `Analyzer` classes, let's see which out-of-the-box analyzers Lucene provides.

4.3 Using the built-in analyzers

Lucene includes several built-in analyzers, created by chaining together the built-in `Tokenizers` and `TokenFilters`. The primary ones are shown in table 4.2. We'll leave discussion of the language-specific analyzers in the sandbox, to section 4.8.2 and the special per-field analyzer wrapper, `PerFieldAnalyzerWrapper`, to section 4.4.

Table 4.2 Primary analyzers available in Lucene

Analyzer	Steps taken
`WhitespaceAnalyzer`	Splits tokens at whitespace
`SimpleAnalyzer`	Divides text at nonletter characters and lowercases
`StopAnalyzer`	Divides text at nonletter characters, lowercases, and removes stop words
`KeywordAnalyzer`	Treats entire text as a single token
`StandardAnalyzer`	Tokenizes based on a sophisticated grammar that recognizes e-mail addresses, acronyms, Chinese-Japanese-Korean characters, alphanumerics, and more; lowercases; and removes stop words

The built-in analyzers we discuss in this section — `WhitespaceAnalyzer`, `SimpleAnalyzer`, `StopAnalyzer`, `KeywordAnalyzer` and `StandardAnalyzer` — are designed to work with text in almost any Western (European-based) language. You can see the effect of each of these analyzers in the output in section 4.2.3. `WhitespaceAnalyzer` and `SimpleAnalyzer` are trivial and we don't cover them in more detail here. We cover `KeywordAnalyzer` in section XXX. We explore the `StopAnalyzer` and `StandardAnalyzer` in more depth because they have nontrivial effects.

Remember that an analyzer is simply a chain of an original `Tokenizer` and a series of `TokenFilters`. There's absolutely no reason why you must use one of Lucene's built-in analyzers; you can easily make your own analyzer that defines your own interesting chain. We'll begin with `StopAnalyzer`.

4.3.1 StopAnalyzer

`StopAnalyzer`, beyond doing basic word splitting and lowercasing, also removes special words called *stop words*. Stop words are words that are very common, such as *the*, and thus assumed to carry very little standalone meaning for searching since nearly every document will contain the word.

Embedded in `StopAnalyzer` is the following list of common English stop words; this list is used unless otherwise specified:

```
public static final String[] ENGLISH_STOP_WORDS = {
  "a", "an", "and", "are", "as", "at", "be", "but", "by",
  "for", "if", "in", "into", "is", "it",  "no", "not", "of", "on", "or", "such",
```

```
    "that", "the", "their", "then", "there", "these",
    "they", "this", "to", "was", "will", "with"
  };
```

The StopAnalyzer has a second constructor that allows you to pass your own list as a String[] instead.

Under the hood, StopAnalyzer creates a StopFilter to perform the filtering. Section 4.7.3 describes some of StopFilter's challenges which you should be aware of before just suddenly using StopAnalyzer.

4.3.2 StandardAnalyzer

StandardAnalyzer holds the honor as the most generally useful built-in analyzer. A JFlex-based[1] grammar underlies it, tokenizing with cleverness for the following lexical types: alphanumerics, acronyms, company names, e-mail addresses, computer host names, numbers, words with an interior apostrophe, serial numbers, IP addresses, and CJK (Chinese Japanese Korean) characters. StandardAnalyzer also includes stop-word removal, using the same mechanism as the StopAnalyzer (identical default English list, and an optional String[] constructor to override). StandardAnalyzer makes a great first choice.

Using StandardAnalyzer is no different than using any of the other analyzers, as you can see from its use in section 4.1.1 and AnalyzerDemo (listing 4.2). Its unique effect, though, is apparent in the different treatment of text. For example, look at listing 4.1, and compare the different analyzers on the phrase "XY&Z Corporation - xyz@example.com". StandardAnalyzer is the only one that kept XY&Z together as well as the e-mail address xyz@example.com; both of these showcase the vastly more sophisticated analysis process.

Let's pause now for a bit and recap where we are. You're about halfway through this chapter, and by now you understand where and why Lucene performs analysis of text, and you've seen the internal details of how analysis is actually implemented. You've seen that analysis is a chain of one Tokenizer and any number of TokenFilters, and you know how to create your own analyzer chain. You've seen the nitty gritty details of how a TokenStream produces tokens. We delved into Lucene's builtin analyzers and token filters, and touched on some tricky topics like the importance of filter order.

At this point you have a strong foundational knowledge of Lucene's analysis process. What we'll now do for the second half of the chapter is build on this base knowledge by visiting several real-world topics, use cases and challenges you'll no doubt encounter. First we'll discuss three field-specific variations that impact analysis. Then we'll show you how to use analysis to implement a couple frequently requested features: sounds like querying, and synonyms expansion. Next, we create our own analyzer chain that normalizes tokens by their stems, removing stop words in the process, and discuss some challenges that result. Finally we discuss issues that arise when analyzing different languages, and we'll wrap up with a quick taste of how the Nutch project handles document analysis.

1JFlex is a sophisticated and high performance lexical analyzer. See http://jflex.de.

4.4 Field variations

The fact that a `Document` is composed of multiple fields, with diverse characteristics, introduces some interesting requirements to the analysis process. We'll first consider how analysis is impacted by multi-valued fields. Next we'll discuss how to use different analyzers for different fields. Finally we'll talk about skipping analysis entirely for certain fields.

4.4.1 Analysis of multi-valued fields

Recall from chapter 2 that a `Document` may have more than one `Field` with the same name, and that Lucene logically appends the tokens of these fields sequentially during indexing. Fortunately, your analyzer has some control over what happens at each field boundary. This is important to ensure queries that pay attention to a `Token`'s position, such as phrase or span queries, do not inadvertently match across two separate field instances. The method `Analyzer.setPositionIncrementGap` sets the extra gap that should be added when crossing into a new `Field` instance. By default this is 0 (no gap), which means it acts as if the field values were directly appended to one another. If your documents have multi-valued fields, and you do index them, it's a good idea to increase this to a large enough number (for example 100) so no positional queries, such as `PhraseQuery`, could ever match across the boundary.

Another frequently encountered analysis challenge is how to use a different analyzer for different fields.

4.4.2 Field specific analysis

During indexing, the granularity of analyzer choice is at the `IndexWriter` or per-`Document` level. With `QueryParser`, there is only one analyzer use to analyze all encountered text. Yet for many applications, where the documents have very diverse fields, it would seem that each field may deserve unique analysis.

Internally, analyzers can easily act on the field name being analyzed, since that's passed as an argument to the `tokenStream` method. The built-in analyzers don't leverage this capability because they're designed for general-purpose use and field names are of course application specific, but you can easily create a custom analyzer that does so. Alternatively, Lucene has a helpful builtin utility class, `PerFieldAnalyzerWrapper`, that makes it simple to use different analyzers per field. Use it like this:

```
PerFieldAnalyzerWrapper analyzer = new PerFieldAnalyzerWrapper(
                                       new SimpleAnalyzer());
analyzer.addAnalyzer("body", new StandardAnalyzer());
```

You provide the default analyzer when you create `PerFieldAnalyzerWrapper`. Then, for any field that requires a different analyzer, you call the `addAnalyzer` method. Any field that wasn't assigned a specific analyzer simply falls back to the default one. In the example above, we use `SimpleAnalyzer` for all fields except `body`, which uses `StandardAnalyzer`.

Let's see next how `PerFieldAnalyzerWrapper` can be useful when you need to mix analyzed and unanalyzed fields.

4.4.3 Unanalyzed fields

There are often cases when you'd like to index a field's value without analysis. For example, part numbers, URLs, social-security numbers, etc should all be indexed and searched as a single token. During indexing this is easily done by specifying `Field.Index.NOT_ANALYZED` when you create the field. Of course, you also want users to be able to search on these part numbers. This is simple if your application directly creates a `TermQuery`. A dilemma can arise, however, if you use `QueryParser` and attempt to query on an unanalyzed field, since the fact that the field was not analyzed is only known during indexing. There is nothing special about such a field's terms once indexed; they're just terms.

Let's see the issue exposed with a straightforward test case that indexes a document with an unanalyzed field and then attempts to find that document again:

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

  public void setUp() throws Exception {

    Directory directory = new RAMDirectory();

    IndexWriter writer = new IndexWriter(directory,
                                         new SimpleAnalyzer(),
                                         IndexWriter.MaxFieldLength.UNLIMITED);

    Document doc = new Document();
    doc.add(new Field("partnum",
                      "Q36",
                      Field.Store.NO,
                      Field.Index.NOT_ANALYZED));   //1
    doc.add(new Field("description",
                      "Illidium Space Modulator",
                      Field.Store.YES,
                      Field.Index.ANALYZED));
    writer.addDocument(doc);

    writer.close();

    searcher = new IndexSearcher(directory);
  }

  public void testTermQuery() throws Exception {
    Query query = new TermQuery(new Term("partnum", "Q36"));  //2
    assertEquals(1, searcher.search(query, 1).totalHits); //3
  }
}
```

So far, so good—we've indexed a document and can retrieve it using a `TermQuery`. But what happens if we use `QueryParser` instead?

```
  public void testBasicQueryParser() throws Exception {
    Query query = new QueryParser("description",               //4
                                  new SimpleAnalyzer()).parse("partnum:Q36 AND SPACE");
```

```
    assertEquals("note Q36 -> q",
                 "+partnum:q +space", query.toString("description"));    //5
    assertEquals("doc not found :(", 0, searcher.search(query, 1).totalHits);
  }
```

#1 Field not analyzed
#2 No analysis here
#3 Document found as expected
#4 QueryParser analyzes each term and phrase
#5 toString() method

#4 `QueryParser` analyzes each term and phrase of the query expression. Both *Q36* and *SPACE* are analyzed separately. `SimpleAnalyzer` strips nonletter characters and lowercases, so *Q36* becomes *q*. But at indexing time, *Q36* was left as is. Notice, also, that this is the same analyzer used during indexing, but because the field was indexed with `Field.Index.NOT_ANALYZED`, the analyzer was not used.

#5 Query has a nice `toString()` method (see section 3.5.1) to return the query as a `QueryParser`-like expression. Notice that *Q36* is gone.

This issue of `QueryParser` encountering an unanalyzed field emphasizes a key point: *indexing and analysis are intimately tied to searching*. The `testBasicQueryParser` test shows that searching for terms created using `Index.NOT_ANALYZED` when a query expression is analyzed can be problematic. It's problematic because `QueryParser` analyzed the `partnum` field, but it shouldn't have. There are a few possible solutions:

- Separate your user interface such that a user selects a part number separately from free-form queries. Generally, users don't want to know (and shouldn't need to know) about the field names in the index. It's also very poor practice to present more than one text entry box to the user. They will become confused.
- If part numbers or other textual constructs are common lexical occurrences in the text you're analyzing, consider creating a custom domain-specific analyzer that recognizes part numbers, and so on, and preserves them.
- Subclass `QueryParser` and override one or both of the `getFieldQuery` methods to provide field-specific handling.
- Use `PerFieldAnalyzerWrapper` for field-specific analysis.

Designing a search user interface is very application dependent; `BooleanQuery` (section 3.4.4) and filters (section 5.5) provide the support you need to combine query pieces in sophisticated ways. Section 8.5 covers ways to use JavaScript in a web browser for building queries. Section 8.XXX shows you how to present a forms-based search interface that uses XML to represent the full query. The information in this chapter provides the foundation for building domain-centric analyzers. We cover subclassing `QueryParser` in section 6.3. Of all these solutions, the simplest is to use `PerFieldAnalyzerWrapper`.

We'll use Lucene's KeywordAnalyzer to tokenize the part number as a single token. Note that KeywordAnalyzer and Field.Index.NOT_ANALYZED are completely identical during indexing; it's only with QueryParser that using KeywordAnalyzer is necessary. We only want one field to be "analyzed" in this manner, so we leverage the PerFieldAnalyzerWrapper to apply it only to the partnum field. First let's look at the KeywordAnalyzer in action as it fixes the situation:

```
public void testPerFieldAnalyzer() throws Exception {
  PerFieldAnalyzerWrapper analyzer = new PerFieldAnalyzerWrapper(
                                          new SimpleAnalyzer());
  analyzer.addAnalyzer("partnum", new KeywordAnalyzer());  //1

  Query query = new QueryParser("description", analyzer).parse(
              "partnum:Q36 AND SPACE");

  assertEquals("Q36 kept as-is",
            "+partnum:Q36 +space", query.toString("description"));
  assertEquals("doc found!", 1, searcher.search(query, 1).totalHits);       //2
}
```

#1 Apply KeywordAnalyzer only to partnum
#2 Document is found

#1 We apply the KeywordAnalyzer only to the partnum field, and we use the SimpleAnalyzer for all other fields. This is the same effective result as during indexing.

#2 Note that the query now has the proper term for the partnum field, and the document is found as expected.

Given KeywordAnalyzer, we could streamline our code (in KeywordAnalyzerTest.setUp) and use the same PerFieldAnalyzerWrapper used in testPerFieldAnalyzer during indexing. Using a KeywordAnalyzer on special fields during indexing would eliminate the use of Index.NOT_ANALYZED during indexing and replace it with Index.ANALYZED. Aesthetically, it may be pleasing to see the same analyzer used during indexing and querying, and using PerFieldAnalyzerWrapper makes this possible.

4.5 "Sounds like" querying

Have you ever played the game Charades, cupping your hand to your ear to indicate that your next gestures refer to words that "sound like" the real words you're trying to convey? Neither have we. Suppose, though, that a high-paying client has asked you to implement a search engine accessible by J2ME-enabled devices, such as a cell phone, to help during those tough charade matches. In this section, we'll implement an analyzer to convert words to a phonetic root using an implementation of the Metaphone algorithm from the Jakarta Commons Codec project. We chose the Metaphone algorithm as an example, but other algorithms are available, such as Soundex.

Let's start with a test case showing the high-level goal of our search experience:

```
public void testKoolKat() throws Exception {
  RAMDirectory directory = new RAMDirectory();
  Analyzer analyzer = new MetaphoneReplacementAnalyzer();
```

```
    IndexWriter writer = new IndexWriter(directory, analyzer, true,
                                         IndexWriter.MaxFieldLength.UNLIMITED);

    Document doc = new Document();
    doc.add(new Field("contents", //#1
                      "cool cat",
                      Field.Store.YES,
                      Field.Index.ANALYZED));
    writer.addDocument(doc);
    writer.close();

    IndexSearcher searcher = new IndexSearcher(directory);
    Query query = new QueryParser("contents", analyzer).parse(//#2
                                        "kool kat");

    TopScoreDocCollector collector = new TopScoreDocCollector(1);
    searcher.search(query, collector);
    assertEquals(1, collector.getTotalHits());   //#3
    int docID = collector.topDocs().scoreDocs[0].doc;
    doc = searcher.doc(docID);
    assertEquals("cool cat", doc.get("contents"));   //#4

    searcher.close();
  }
```

#1 Original document
#2 User typed in hip query
#3 Hip query matches!
#4 Original value still available

It seems like magic! The user searched for "kool kat". Neither of those terms was in our original document, yet the search found the desired match. Searches on the original text would also return the expected matches. The trick lies under the `MetaphoneReplacementAnalyzer`:

```
public class MetaphoneReplacementAnalyzer extends Analyzer {
  public TokenStream tokenStream(String fieldName, Reader reader) {
    return new MetaphoneReplacementFilter(
                  new LetterTokenizer(reader));
  }
}
```

Because the Metaphone algorithm expects words that only include letters, the `LetterTokenizer` is used to feed our metaphone filter. The `LetterTokenizer` doesn't lowercase, however. The tokens emitted are *replaced* by their metaphone equivalent, so lowercasing is unnecessary. Let's now dig into the `MetaphoneReplacementFilter`, where the real work is done:

```
public class MetaphoneReplacementFilter extends TokenFilter {
  public static final String METAPHONE = "METAPHONE";

  private Metaphone metaphoner = new Metaphone();    //#1
  private TermAttribute termAttr;
  private TypeAttribute typeAttr;
```

```
  public MetaphoneReplacementFilter(TokenStream input) {
    super(input);
    termAttr = (TermAttribute) addAttribute(TermAttribute.class);
    typeAttr = (TypeAttribute) addAttribute(TypeAttribute.class);
  }

  public boolean incrementToken() throws IOException {
    if (!input.incrementToken())                    //#2
      return false;                                 //#3

    String encoded;
    encoded = metaphoner.encode(termAttr.term());   //#4
    termAttr.setTermBuffer(encoded);                //#5
    typeAttr.setType(METAPHONE);                    //#6
    return true;
  }
}
```

#1 org.apache.commons.codec.language.Metaphone
#2 Pull next token
#3 When null, end has been reached
#4 Convert token to Metaphone encoding
#5 Overwrite characters in token with encoded text
#6 Set token type

The token emitted by our `MetaphoneReplacementFilter`, as its name implies, literally replaces the incoming token. This new token is set with the same position offsets as the original, because it's a replacement in the same position. The last line before returning the token sets the *token type*. Each token can be associated with a `String` indicating its type, giving meta-data to later filtering in the analysis process. The `StandardTokenizer`, as discussed in "Token type usefulness" under section 4.2.3, tags tokens with a type that is later used by the `StandardFilter`. The `METAPHONE` type isn't used in our examples, but it demonstrates that a later filter could be Metaphone-token aware by calling `Token`'s `type()` method.

NOTE

`Token` types, such as the `METAPHONE` type used in `MetaphoneReplacementAnalyzer`, are carried through the analysis phase but aren't encoded into the index. Unless otherwise specified, the type `word` is used for tokens by default. Section 4.2.3 discusses token types further.

As always, it's good to view what an analyzer is doing with text. Using our `AnalyzerUtils`, two phrases that sound similar yet are spelled completely differently are tokenized and displayed:

```
  public static void main(String[] args) throws IOException {
    MetaphoneReplacementAnalyzer analyzer =
                                new MetaphoneReplacementAnalyzer();
    AnalyzerUtils.displayTokens(analyzer,
                  "The quick brown fox jumped over the lazy dogs");

    System.out.println("");
    AnalyzerUtils.displayTokens(analyzer,
                  "Tha quik brown phox jumpd ovvar tha lazi dogz");
```

```
}
```

We get a sample of the Metaphone encoder, shown here:

```
[0] [KK] [BRN] [FKS] [JMPT] [OFR] [0] [LS] [TKS]
[0] [KK] [BRN] [FKS] [JMPT] [OFR] [0] [LS] [TKS]
```

Wow—an exact match!

In practice, it's unlikely you'll want sounds-like matches except in special places; otherwise, far too many undesired matches may be returned.[2] In the "What would Google do?" sense, a sounds-like feature would be great for situations where a user misspelled every word and no documents were found, but alternative words could be suggested. One implementation approach to this idea could be to run all text through a sounds-like analysis and build a cross-reference lookup to consult when a correction is needed.

Now we walk through an `Analyzer` that can handle synonyms during indexing.

4.6 Synonyms, aliases, and words that mean the same

How often have you searched for "spud" and been disappointed that the results did not include "potato"? OK, maybe that precise example doesn't happen often, but you get the idea: natural languages for some reason have evolved many ways to say the same thing. Such synonyms must be handled during searching, or else your users won't find their documents.

Our next custom analyzer injects synonyms of words into the outgoing token stream, but places the synonyms in the *same position* as the original word. By adding synonyms during indexing, searches will find documents that may not contain the original search terms but match the synonyms of those words. We start with the test case showing how we expect our new analyzer to work:

```
public void testJumps() throws Exception {
  AttributeSource[] tokens =
    AnalyzerUtils.tokensFromAnalysis(synonymAnalyzer , "jumps");     //|#1

  AnalyzerUtils.assertTokensEqual(tokens,
                                  new String[] {"jumps", "hops", "leaps"});   //|#2

  // ensure synonyms are in the same position as the original
  assertEquals("jumps", 1, AnalyzerUtils.getPositionIncrement(tokens[0]));
  assertEquals("hops", 0, AnalyzerUtils.getPositionIncrement(tokens[1]));
  assertEquals("leaps", 0, AnalyzerUtils.getPositionIncrement(tokens[2]));
}
```

#1 Analyze one word
#2 Three words come out

[2] While working on this chapter, Erik asked his brilliant 5-year-old son, Jakob, how he would spell *cool cat*. Jakob replied, "c-o-l c-a-t". What a wonderfully confusing language English is. Erik imagines that a "sounds-like" feature in search engines designed for children would be very useful. Metaphone encodes *cool*, *kool*, and *col* all as `KL`.

Notice that our unit test shows not only that synonyms for the word *jumps* are emitted from the `SynonymAnalyzer` but also that the synonyms are placed in the same position (increment of zero) as the original word.

Let's see what the `SynonymAnalyzer` is doing; then we'll explore the implications of position increments. Figure 4.4 graphically shows what our `SynonymAnalyzer` does to text input, and listing 4.5 is the implementation.

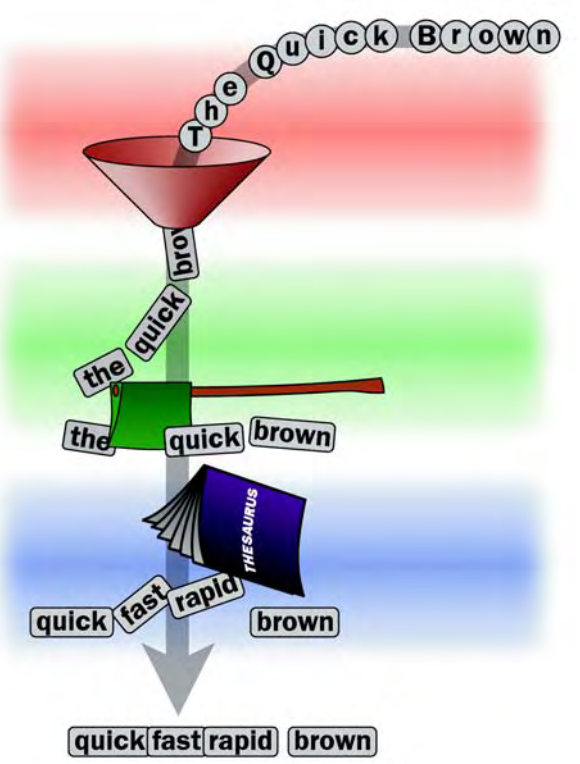


Figure 4.4 `SynonymAnalyzer` visualized as factory automation

Listing 4.5 SynonymAnalyzer implementation

```
public class SynonymAnalyzer extends Analyzer {
  private SynonymEngine engine;

  public SynonymAnalyzer(SynonymEngine engine) {
    this.engine = engine;
  }

  public TokenStream tokenStream(String fieldName, Reader reader) {
```

```
    TokenStream result = new SynonymFilter(
                           new StopFilter(
                             new LowerCaseFilter(
                               new StandardFilter(
                                 new StandardTokenizer(reader))),
                             StandardAnalyzer.STOP_WORDS),
                           engine
                         );
    return result;
  }
}
```

Once again, the analyzer code is minimal and simply chains a `Tokenizer` together with a series of `TokenFilters`; in fact, this is the `StandardAnalyzer` wrapped with an additional filter. (See table 4.1 for more on these basic analyzer building blocks.) The final `TokenFilter` in the chain is the new `SynonymFilter` (listing 4.6), which gets to the heart of the current discussion. When you're injecting terms, buffering is needed. This filter uses a `Stack` as the buffer.

Listing 4.6 SynonymFilter: buffering tokens and emitting one at a time

```
public class SynonymFilter extends TokenFilter {
  public static final String TOKEN_TYPE_SYNONYM = "SYNONYM";

  private Stack synonymStack;
  private SynonymEngine engine;
  private TermAttribute termAttr;

  public SynonymFilter(TokenStream in, SynonymEngine engine) {
    super(in);
    synonymStack = new Stack();  //|#1
    termAttr = (TermAttribute) addAttribute(TermAttribute.class);
    this.engine = engine;
  }

  public boolean incrementToken() throws IOException {
    if (synonymStack.size() > 0) {       //#2
      AttributeSource syn = (AttributeSource) synonymStack.pop(); //#2
      syn.restoreState(this);             //#2
      return true;
    }

    if (!input.incrementToken())   //#3
      return false;

    addAliasesToStack();  //#4

    return true;       //#5
  }

  private void addAliasesToStack() throws IOException {
    String[] synonyms = engine.getSynonyms(termAttr.term());   //#6
    if (synonyms == null) return;

    for (int i = 0; i < synonyms.length; i++) {            //#7
      AttributeSource synToken = captureState();
```

```
        AnalyzerUtils.setTerm(synToken, synonyms[i]);        //#7
        AnalyzerUtils.setType(synToken, TOKEN_TYPE_SYNONYM); //#7
        AnalyzerUtils.setPositionIncrement(synToken, 0);     //#8
        synonymStack.push(synToken);                         //#7
      }
    }
  }
```

#1 Synonym buffer
#2 Pop buffered synonyms
#3 Read next token
#4 Push synonyms of current token onto stack
#5 Return current token
#6 Retrieve synonyms
#7 Push synonyms onto stack
#8 Set position increment to zero

#2 The code successively pops the stack of buffered synonyms from the last streamed-in token until it's empty.

#3 After all previous token synonyms have been emitted, we read the next token.

#4 We push all synonyms of the current token onto the stack.

#5 Now we return the current (and original) token before its associated synonyms.

#6 Synonyms are retrieved from the `SynonymEngine`.

#7 We push each synonym onto the stack.

#8 The position increment is set to zero, allowing synonyms to be logically in the same place as the original term.

The design of `SynonymAnalyzer` allows for pluggable `SynonymEngine` implementations. `SynonymEngine` is a one-method interface:

```
public interface SynonymEngine  {
  String[] getSynonyms(String s) throws IOException;
}
```

Using an interface for this design easily allows test implementations. We leave it as an exercise for you to create production-quality `SynonymEngine` implementations.[3] For our examples, we use a simple test that's hard-coded with a few synonyms:

```
public class TestSynonymEngine implements SynonymEngine {
  private static HashMap map = new HashMap();

  static {
```

[3]It's cruel to leave you hanging with a mock implementation, isn't it? Actually, we've implemented a powerful SynonymEngine using the WordNet database. It's covered in section 8.6.2.

```
    map.put("quick", new String[] {"fast", "speedy"});
    map.put("jumps", new String[] {"leaps", "hops"});
    map.put("over", new String[] {"above"});
    map.put("lazy", new String[] {"apathetic", "sluggish"});
    map.put("dogs", new String[] {"canines", "pooches"});
  }

  public String[] getSynonyms(String s) {
    return (String[]) map.get(s);
  }
}
```

Notice that the synonyms generated by `TestSynonymEngine` are one-way: for example, *quick* has the synonyms *fast* and *speedy*, but *fast* has no synonyms. In a real production environment, you should ensure all synonyms list one another as alternate synonyms, but since we are using this for simple testing, it's fine.

Setting the position increment seems powerful, and indeed it is. You should only modify increments knowing of some odd cases that arise in searching, though. Since synonyms are indexed just like other terms, `TermQuery` works as expected. Also, `PhraseQuery` works as expected when we use a synonym in place of an original word. The `SynonymAnalyzerTest` test case in listing 4.7 demonstrates things working well using API-created queries.

Listing 4.7 SynonymAnalyzerTest: showing that synonym queries work

```
public class SynonymAnalyzerTest extends TestCase {
  private IndexSearcher searcher;
  private static SynonymAnalyzer synonymAnalyzer =
                      new SynonymAnalyzer(new TestSynonymEngine());

  public void setUp() throws Exception {
    RAMDirectory directory = new RAMDirectory();

    IndexWriter writer = new IndexWriter(directory,
                                         synonymAnalyzer,  //#1
                                         IndexWriter.MaxFieldLength.UNLIMITED);
    Document doc = new Document();
    doc.add(new Field("content",
                      "The quick brown fox jumps over the lazy dogs",
                      Field.Store.YES,
                      Field.Index.ANALYZED));  //#2
    writer.addDocument(doc);

    writer.close();

    searcher = new IndexSearcher(directory);
  }

  public void tearDown() throws Exception {
    searcher.close();
  }

  public void testJumps() throws Exception {
```

```
    AttributeSource[] tokens =
      AnalyzerUtils.tokensFromAnalysis(synonymAnalyzer , "jumps");     //|#1

    AnalyzerUtils.assertTokensEqual(tokens,
                                    new String[] {"jumps", "hops", "leaps"});   //|#2

    // ensure synonyms are in the same position as the original
    assertEquals("jumps", 1, AnalyzerUtils.getPositionIncrement(tokens[0]));
    assertEquals("hops", 0, AnalyzerUtils.getPositionIncrement(tokens[1]));
    assertEquals("leaps", 0, AnalyzerUtils.getPositionIncrement(tokens[2]));
  }


  public void testSearchByAPI() throws Exception {

    TermQuery tq = new TermQuery(new Term("content", "hops"));  //#3
    assertEquals(1, searcher.search(tq, 1).totalHits);

    PhraseQuery pq = new PhraseQuery();    //#4
    pq.add(new Term("content", "fox"));    //#4
    pq.add(new Term("content", "hops"));   //#4
    assertEquals(1, searcher.search(pq, 1).totalHits);
  }
}
```

#1 Analyze with SynonymAnalyzer
#2 Index single document
#3 Search for "hops"
#4 Search for "fox hops"

#1 We perform the analysis with a custom `SynonymAnalyzer`, using `MockSynonym-Engine`.

#3 A search for the word *hops* matches the document.

#4 A search for the phrase "fox hops" also matches.

The phrase "…fox jumps…" was indexed, and our `SynonymAnalyzer` injected *hops* in the same position as *jumps*. A `TermQuery` for *hops* succeeded, as did an exact `PhraseQuery` for "fox hops". Excellent!

Let's test it with `QueryParser`. We'll run two tests. The first one creates QueryParser using our `SynonymAnalyzer`, and the second one using `StandardAnalyzer`:

```
  public void testWithQueryParser() throws Exception {
    Query query = new QueryParser("content",                              // 1
                                  synonymAnalyzer).parse("\"fox jumps\"");  // 1
    assertEquals(1, searcher.search(query, 1).totalHits);                 // 1
    System.out.println("With SynonymAnalyzer, \"fox jumps\" parses to " +
                                       query.toString("content"));

    query = new QueryParser("content",                                    // 2
                            new StandardAnalyzer()).parse("\"fox jumps\""); // 2
    assertEquals(1, searcher.search(query, 1).totalHits);                 // 2
    System.out.println("With StandardAnalyzer, \"fox jumps\" parses to " +
                                       query.toString("content"));
```

```
}
```

#1 SynonymAnalyzer finds the document
#2 StandardAnalyzer also finds document

Both analyzers find the matching document just fine, which is great. The test produces the following output:

```
With SynonymAnalyzer, "fox jumps" parses to "fox (jumps hops leaps)"
With StandardAnalyzer, "fox jumps" parses to "fox jumps"
```

As expected, with `SynonymAnalyzer`, words in our query were expanded to their synonyms. `QueryParser` is smart enough to notice that the tokens produced by the analyzer have zero position increment, and when that happens inside a phrase query, it creates a `MultiPhraseQuery`, described in section 5.3.

However, this is in fact wasteful and unnecessary: we only need synonym expansion during indexing or during searching, not both. If you choose to expand during indexing, the disk space consumed by your index will be somewhat larger, but searching may be faster since there are fewer search terms to visit. However, since your synonyms have been baked into the index, you don't have the freedom to quickly change them and see the impact of such changes during searching. If instead you expand at search time, you can see fast turnaround when testing. These are simply tradeoffs, and which option is best is your decision based on your application's constraints!

Next we improve our AnalyzerUtils class to more easily see synonyms expansion during indexing.

4.6.1 Visualizing token positions

Our `AnalyzerUtils.tokensFromAnalysis` doesn't show us all the information when dealing with analyzers that set position increments other than 1. In order to get a better view of these types of analyzers, we add an additional utility method, `displayTokensWithPositions`, to `AnalyzerUtils`:

```
public static void displayTokensWithPositions
  (Analyzer analyzer, String text) throws IOException {
  AttributeSource[] tokens = tokensFromAnalysis(analyzer, text);

  int position = 0;

  for (int i = 0; i < tokens.length; i++) {
    AttributeSource token = tokens[i];

    TermAttribute term = (TermAttribute) token.addAttribute(TermAttribute.class);

    PositionIncrementAttribute posIncr =
      (PositionIncrementAttribute)
      token.getAttribute(PositionIncrementAttribute.class);

    int increment = posIncr.getPositionIncrement();

    if (increment > 0) {
```

```
        position = position + increment;
        System.out.println();
        System.out.print(position + ": ");
      }

      System.out.print("[" + term.term() + "] ");
    }
    System.out.println();
  }
```

We wrote a quick piece of code to see what our `SynonymAnalyzer` is really doing:

```
public class SynonymAnalyzerViewer {

  public static void main(String[] args) throws IOException {
    SynonymEngine engine = new TestSynonymEngine();

    AnalyzerUtils.displayTokensWithPositions(
      new SynonymAnalyzer(engine),
      "The quick brown fox jumps over the lazy dogs");
  }
}
```

And we can now visualize the synonyms placed in the same positions as the original words:

```
1: [quick] [speedy] [fast]
2: [brown]
3: [fox]
4: [jumps] [hops] [leaps]
5: [over] [above]
6: [lazy] [sluggish] [apathetic]
7: [dogs] [pooches] [canines]
```

Each number on the left represents the token position. The numbers here are continuous, but they wouldn't be if the analyzer left holes (as you'll see with the next custom analyzer). Multiple terms shown for a single position illustrates where synonyms were added.

4.7 Stemming analysis

Our final analyzer pulls out all the stops. It has a ridiculous, yet descriptive name: `PositionalPorterStopAnalyzer`. This analyzer removes stop words, leaving positional holes where words are removed, and also leverages a stemming filter.

The `PorterStemFilter` is shown in the class hierarchy in figure 4.3, but it isn't used by any built-in analyzer. It *stems* words using the Porter stemming algorithm created by Dr. Martin Porter, and it's best defined in his own words:

> The Porter stemming algorithm (or 'Porter stemmer') is a process for removing the commoner morphological and inflexional endings from words in English. Its main use is as part of a term normalisation process that is usually done when setting up Information Retrieval systems.[4]

In other words, the various forms of a word are reduced to a common root form. For example, the words *breathe*, *breathes*, *breathing*, and *breathed*, via the Porter stemmer, reduce to *breath*.

The Porter stemmer is one of many stemming algorithms. See section 8.3.1, page XXX, for coverage of an extension to Lucene that implements the Snowball algorithm (also created by Dr. Porter). KStem is another stemming algorithm that has been adapted to Lucene (search Google for KStem and Lucene).

We first show how to use `StopFilter` to leave holes whenever it removes a word. Then we'll describe the full analyzer and finally we'll talk about what to do about the missing positions.

4.7.1 Leaving holes with StopFilter

Stopword removal brings up an interesting issue: what happens to the holes left by the words removed? Suppose you index "one is not enough". The tokens emitted from `StopAnalyzer` will be `one` and `enough`, with `is` and `not` thrown away. By default, `StopAnalyzer` does not account for words removed, so the result is exactly as if you indexed "one enough". If you were to use `QueryParser` along with `StopAnalyzer`, this document would match phrase queries for "one enough", "one is enough", "one but not enough", and the original "one is not enough". Remember, `QueryParser` also analyzes phrases, and each of these reduces to "one enough" and matches the terms indexed.

Fortunately, if you call `setEnablePositionIncrements(true)`on your `StopFilter` instance then it will record positions that were skipped and properly set positionIncrement to reflect the gaps. This is illustrated from the output of `AnalyzerUtils.displayTokensWithPositions`:

```
2: [quick]
3: [brown]
4: [fox]
5: [jump]
6: [over]
8: [lazi]
9: [dog]
```

Positions 1 and 7 are missing due to the removal of *the*.

Since `StopAnalyzer` does not expose this option, this is a good reason to create your own analyzer chain using `StopFilter`.

4.7.2 Putting it together

This custom analyzer uses a stop-word removal filter, enabled to keep positional gaps, and fed from a `LowerCaseTokenizer`. The results of the stop filter are fed to the Porter stemmer. Listing 4.8 shows the full implementation of this sophisticated analyzer. `LowerCaseTokenizer` kicks off the analysis process,

[4] From Dr. Porter's website: http://www.tartarus.org/~martin/PorterStemmer/index.html.

feeding tokens through the stop-word removal filter and finally stemming the words using the built-in Porter stemmer.

Listing 4.8 PositionalPorterStopAnalyzer: removes stop words (leaving gaps) and stem words

```
public class PositionalPorterStopAnalyzer extends Analyzer {
  private Set stopWords;

  public PositionalPorterStopAnalyzer() {
    this(StopAnalyzer.ENGLISH_STOP_WORDS);
  }

  public PositionalPorterStopAnalyzer(String[] stopList) {
    stopWords = StopFilter.makeStopSet(stopList);
  }

  public TokenStream tokenStream(String fieldName, Reader reader) {
    StopFilter stopFilter = new StopFilter(new LowerCaseTokenizer(reader),
                                           stopWords);
    stopFilter.setEnablePositionIncrements(true);
    return new PorterStemFilter(stopFilter);
  }
}
```

Leaving gaps when stop words are removed makes logical sense but introduces new issues that we explore next.

4.7.3 Hole lot of trouble

As you saw with the `SynonymAnalyzer`, messing with token position information can cause trouble during searching. `PhraseQuery` and `QueryParser` are the two troublemakers. Exact phrase matches now fail, as illustrated in our test case:

```
public class PositionalPorterStopAnalyzerTest extends TestCase {
  private static PositionalPorterStopAnalyzer porterAnalyzer =
    new PositionalPorterStopAnalyzer();

  private IndexSearcher searcher;
  private QueryParser parser;

  public void setUp() throws Exception {

    RAMDirectory directory = new RAMDirectory();

    IndexWriter writer =
        new IndexWriter(directory,
                        porterAnalyzer,
                        IndexWriter.MaxFieldLength.UNLIMITED);

    Document doc = new Document();
    doc.add(new Field("contents",
                      "The quick brown fox jumps over the lazy dogs",
                      Field.Store.YES,
                      Field.Index.ANALYZED));
```

```
    writer.addDocument(doc);
    writer.close();
    searcher = new IndexSearcher(directory);
    parser = new QueryParser("contents",
                             porterAnalyzer);
  }

  public void testExactPhrase() throws Exception {
    Query query = parser.parse("\"over the lazy\"");

    assertEquals("exact match not found!", 0, TestUtil.hitCount(searcher, query));
  }
}
```

As shown, an exact phrase query didn't match. This is disturbing, of course. The difficulty lies deeper inside `PhraseQuery` and its current inability to deal with positional gaps. All terms in a `PhraseQuery` must be side by side, and in our test case, the phrase it's searching for is "over lazi" (stop word removed with remaining words stemmed).

`PhraseQuery` does allow a little looseness, called *slop*. This is covered in greater detail in section 3.4.5. Setting the slop to 1 allows the query to effectively ignore the single gap:

```
public void testWithSlop() throws Exception {
  parser.setPhraseSlop(1);

  Query query = parser.parse("\"over the lazy\"");

  assertEquals("hole accounted for", 1, TestUtil.hitCount(searcher, query));
}
```

The value of the phrase slop factor, in a simplified definition for this case, represents how many stop words could be present in the original text between indexed words. Introducing a slop factor greater than zero, however, allows even more inexact phrases to match. In this example, searching for "over lazy" also matches. With stop-word removal in analysis, doing *exact* phrase matches is, by definition, not possible: The words removed aren't there, so you can't know what they were.

The slop factor addresses the main problem with searching using stop-word removal that leaves holes; you can now see the benefit our analyzer provides, thanks to the stemming:

```
public void testStems() throws Exception {
  Query query = new QueryParser("contents", porterAnalyzer).parse(
                       "laziness");
  assertEquals("lazi", 1, TestUtil.hitCount(searcher, query));

  query = parser.parse("\"fox jumped\"");
  assertEquals("jump jumps jumped jumping", 1, TestUtil.hitCount(searcher, query));
}
```

Both *laziness* and the phrase "fox jumped" matched our indexed document, allowing users a bit of flexibility in the words used during searching.

Slop is not a great solution for this problem, since it fuzzes up all phrase matching. A better solution is to use *shingles*, which are compound tokens created from multiple adjacent tokens. Lucene has a sandbox module that simplifies adding shingles to your index, described in section 8.3.2. With shingles, stop words are combined with adjacent words to make new tokens, such as *the-quick*. At search time, the same expansion is used. This enables precise phrase matching, because the stop words are not in fact discarded, yet has good search performance because the number of documents containing *the-quick* is far fewer than the number containing the stopword *the* in any context. Nutch's document analysis, described in section 4.9, also uses shingles.

4.8 Language analysis issues

Dealing with languages in Lucene is an interesting and multifaceted issue. How can text in various languages be indexed and subsequently retrieved? As a developer building I18N-friendly applications based on Lucene, what issues do you need to consider?

You must contend with several issues when analyzing text in various languages. The first hurdle is ensuring that character-set encoding is done properly such that external data, such as files, are read into Java properly. During the analysis process, different languages have different sets of stop words and unique stemming algorithms. Perhaps accents should be removed from characters as well, which would be language dependent. Finally, you may require language detection if you aren't sure what language is being used. Each of these issues is ultimately up to the developer to address, with only basic building-block support provided by Lucene. However, a number of analyzers and additional building blocks such as `Tokenizer`s and `TokenStream`s are available in the Sandbox (discussed in section 8.3) and elsewhere online.

We first describe the Unicode character encoding, then discuss options for analyzing non-English languages, and in particular Asian languages, which present unique challenges. Finally we discuss options for mixing multiple languages in one index. We begin first with a brief introduction to Unicode and character encodings.

4.8.1 Unicode and encodings

Internally, Lucene stores all characters in the standard UTF-8 encoding. Java frees us from many struggles by automatically handling Unicode within `String`s and providing facilities for reading in external data in the many encodings. You, however, are responsible for getting external text into Java and Lucene. If you're indexing files on a file system, you need to know what encoding the files were saved as in order to read them properly. If you're reading HTML or XML from an HTTP server, encoding issues get a bit more complex. Encodings can be specified in an HTTP content-type header or specified within the document itself in the XML header or an HTML `<meta>` tag.

We won't elaborate on these encoding details, not because they aren't important, but because they're separate issues from Lucene. Please refer to appendix C for several sources of more detailed information on encoding topics. In particular, if you're new to I18N issues, read Joel Spolsky's excellent article "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" (http://www.joelonsoftware.com/articles/Unicode.html) and the Java language Internationalization tutorial (http://java.sun.com/docs/books/tutorial/i18n/intro/).

We'll proceed with the assumption that you have your text available as Unicode, and move on to the Lucene-specific language concerns.

4.8.2 Analyzing non-English languages

All the details of the analysis process apply when you're dealing with text in non-English languages. Extracting terms from text is the goal. With Western languages, where whitespace and punctuation are used to separate words, you must adjust stop-word lists and stemming algorithms to be specific to the language of the text being analyzed. You may also want to use the `ASCIIFoldingFilter`, which replaces accented characters with their unaccented counterparts.

Beyond the built-in analyzers we've discussed, the Sandbox provides many language-specific analyzers, under contrib/analyzers. These analyzers generally employ language-specific stemming and stop-word removal. Also freely available is the `SnowballAnalyzer` family of stemmers, which supports many European languages. We discuss `SnowballAnalyzer` in section 8.3.1.

4.8.3 Analyzing Asian languages

Asian languages, such as Chinese, Japanese, and Korean (also denoted as CJK), generally use ideograms rather than an alphabet to represent words. These pictorial words may or may not be separated by whitespace and thus require a different type of analysis that recognizes when tokens should be split. The only built-in analyzer capable of doing anything useful with Asian text is the `StandardAnalyzer`, which recognizes some ranges of the Unicode space as CJK characters and tokenizes them individually.

However, two analyzers in the Lucene Sandbox are suitable for Asian language analysis (see section 8.1 for more details on the Sandbox): `CJKAnalyzer` and `ChineseAnalyzer`. In our sample book data, the Chinese characters for the book *Tao Te Ching* were added to the title. Because our data originates in Java properties files, Unicode escape sequences are used:[5]

```
title=Tao Te Ching \u9053\u5FB7\u7D93
```

We used `StandardAnalyzer` for all tokenized fields in our index, which tokenizes each English word as expected (*tao*, *te*, and *ching*) as well as each of the Chinese characters as separate terms (*tao te ching*) even though there is no space between them. Our `ChineseTest` demonstrates that searching by the word *tao* using its Chinese representation works as desired:

```
public class ChineseTest extends TestCase {
  public void testChinese() throws Exception {
    IndexSearcher searcher = new IndexSearcher("build/index");
    assertEquals("tao", 1, TestUtil.hitCount(searcher, new TermQuery(new
Term("contents", "道"))));
  }
}
```

[5] `java.util.Properties` loads properties files using the ISO-8859-1 encoding but allows characters to be encoded using standard Java Unicode `\u` syntax. Java includes a `native2ascii` program that can convert natively encoded files into the appropriate format.

Note that our ChineseTest.java file was saved in UTF-8 format and compiled using the UTF-8 encoding switch (`-encoding utf8`) for the javac compiler. We had to ensure that the representations of the Chinese characters are encoded and read properly, and use a CJK-aware analyzer.

Similar to the `AnalyzerDemo` in listing 4.2, we created a `ChineseDemo` (listing 4.9) program to illustrate how various analyzers work with Chinese text. This demo uses AWT `Labels` to properly display the characters regardless of your locale and console environment.

Listing 4.9 ChineseDemo: illustrates what analyzers do with Chinese text

```
public class ChineseDemo {
  private static String[] strings = {"道德經"};  //1

  private static Analyzer[] analyzers = {
    new SimpleAnalyzer(),
    new StandardAnalyzer(),
    new ChineseAnalyzer (),    //2
    new CJKAnalyzer ()
  };

  public static void main(String args[]) throws Exception {

    for (int i = 0; i < strings.length; i++) {
      String string = strings[i];
      for (int j = 0; j < analyzers.length; j++) {
        Analyzer analyzer = analyzers[j];
        analyze(string, analyzer);
      }
    }

  }

  private static void analyze(String string, Analyzer analyzer)
        throws IOException {
    StringBuffer buffer = new StringBuffer();
    AttributeSource[] tokens =
      AnalyzerUtils.tokensFromAnalysis(analyzer, string);   //3
    for (int i = 0; i < tokens.length; i++) {
      TermAttribute term = (TermAttribute) tokens[i].getAttribute(TermAttribute.class);
      buffer.append("[");
      buffer.append(term.term());
      buffer.append("] ");
    }

    String output = buffer.toString();

    Frame f = new Frame();
    String name = analyzer.getClass().getName();
    f.setTitle(name.substring(name.lastIndexOf('.') + 1)
                + " : " + string);
    f.setResizable(false);

    Font font = new Font(null, Font.PLAIN, 36);
```

```
    int width = getWidth(f.getFontMetrics(font), output);

    f.setSize((width < 250) ? 250 : width + 50, 75);
    Label label = new Label(buffer.toString());   //4
    label.setSize(width, 75);
    label.setAlignment(Label.CENTER);
    label.setFont(font);
    f.add(label);

    f.setVisible(true);
  }

  private static int getWidth(FontMetrics metrics, String s) {
    int size = 0;
    for (int i = 0; i < s.length(); i++) {
      size += metrics.charWidth(s.charAt(i));
    }

    return size;
  }
}
```

#1 Chinese text to be analyzed
#2 Analyzers from Sandbox
#3 Retrieve tokens from analysis using AnalyzerUtils
#4 AWT Label displays analysis

`CJKAnalyzer` and `ChineseAnalyzer` are analyzers found in the Lucene Sandbox; they aren't included in the core Lucene distribution. `ChineseDemo` shows the output using an AWT Label component to avoid any confusion that might arise from console output encoding or limited fonts mangling things; you can see the output in figure 4.5.

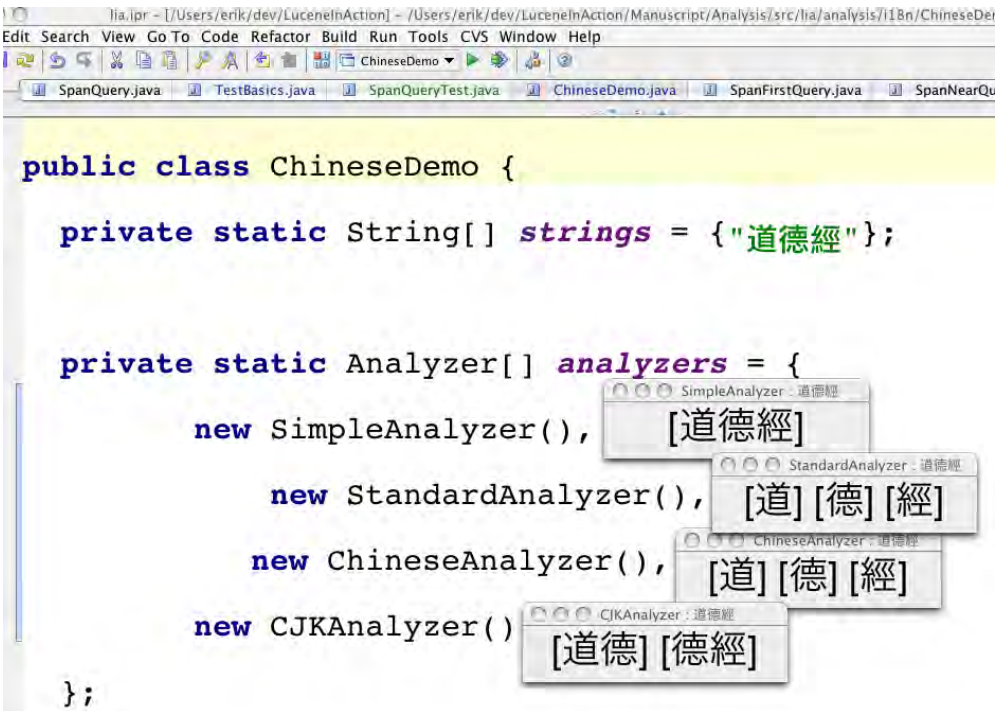


Figure 4.5 `ChineseDemo` illustrating analysis of the title *Tao Te Ching*

The `CJKAnalyzer` pairs characters in overlapping windows of two characters each. Many CJK words are two characters. By pairing characters in this manner, words are likely to be kept together (as well as disconnected characters, increasing the index size). The `ChineseAnalyzer` takes a simpler approach and, in our example, mirrors the results from the built-in `StandardAnalyzer` by tokenizing each Chinese character. Words that consist of multiple Chinese characters are split into terms for each component character.

4.8.4 Zaijian[6]

A major hurdle remains when you're dealing with various languages in a single index: handling text encoding. The `StandardAnalyzer` is still the best built-in general-purpose analyzer, even accounting for CJK characters; however, the Sandbox `CJKAnalyzer` seems better suited for Asian language analysis.

When you're indexing documents in multiple languages into a single index, using a per-`Document` analyzer is appropriate. You may also want to add a field to documents indicating their language; this field

[6] *Ziajian* means *good-bye* in Chinese.

can be used to filter search results or for display purposes during retrieval. In “Controlling date parsing locale” in section 3.5.5, we show how to retrieve the locale from a user’s web browser; this could be automatically used in queries.

One final topic is language detection. This, like character encodings, is outside the scope of Lucene, but it may be important to your application. We don’t cover language-detection techniques in this book, but it’s an active area of research with several implementations to choose from (see appendix C).

4.9 Nutch analysis

We don’t have the source code to Google, but we do have the open-source project Nutch, created by Lucene’s creator Doug Cutting. Nutch takes an interesting approach to analyzing text, specifically how it handles stop words, which it calls *common terms*. If all words are indexed, an enormous number of documents become associated with each common term, such as *the*. Querying for *the* is practically a nonsensical query, given that the majority of documents contain that term. When common terms are used in a query, but not within a phrase, such as *the quick brown* with no other adornments or quotes, common terms are discarded. However, if a series of terms is surrounded by double-quotes, such as “the quick brown”, a fancier trick is played, which we detail in this section.

Nutch combines an index-time analysis *bigram* (grouping two consecutive words as a single token) technique with a query-time optimization of phrases. This results in a far smaller document space considered during searching; for example, far fewer documents have *the quick* side by side than contain *the*. Using the internals of Nutch, we created a simple example to demonstrate the Nutch analysis trickery. Listing 4.10 first analyzes the phrase “The quick brown…” using the `NutchDocumentAnalyzer` and then parses a query of “the quick brown” to demonstrate the Lucene query created.

Listing 4.10 NutchExample: demonstrating the Nutch analysis and query-parsing techniques

```
public class NutchExample {

  public static void main(String[] args) throws IOException {
    Configuration conf = new Configuration();
    conf.addResource("nutch-default.xml");
    NutchDocumentAnalyzer analyzer = new NutchDocumentAnalyzer(conf);   //1

    TokenStream ts = analyzer.tokenStream("content", new StringReader("The quick brown
fox..."));
    int position = 0;
    while(true) {                                 // 2
      Token token = ts.next();
      if (token == null) {
        break;
      }
      int increment = token.getPositionIncrement();

      if (increment > 0) {
        position = position + increment;
        System.out.println();
        System.out.print(position + ": ");
      }
```

```
      System.out.print("[" +
                       token.termText() + ":" +
                       token.startOffset() + "->" +
                       token.endOffset() + ":" +
                       token.type() + "] ");
    }

    System.out.println();

    Query nutchQuery = Query.parse("\"the quick brown\"", conf);  // 3
    org.apache.lucene.search.Query luceneQuery;
    luceneQuery = new QueryFilters(conf).filter(nutchQuery); // 4
    System.out.println("Translated: " + luceneQuery);
  }
```

#1 Custom analyzer
#2 Display token details
#3 Parse to Nutch's Query
#4 Create corresponding translated Lucene Query

#1 Nutch uses a custom analyzer, `NutchDocumentAnalyzer`.

#2 Iterate through the tokens and print the details

#3 Create the Nutch query, and use Nutch's `QueryFilters` to translate the Query into the rewritten Lucene `Query`

The analyzer output shows how "the quick" becomes a bigram, but the word *the* isn't discarded. The bigram resides in the same token position as *the*:

```
1: [the:0->3:<WORD>] [the-quick:0->9:gram]
2: [quick:4->9:<WORD>]
3: [brown:10->15:<WORD>]
4: [fox:16->19:<WORD>]
```

Because additional tokens are created during analysis, the index is larger, but the benefit of this trade-off is that searches for exact-phrase queries are much faster. And there's a bonus: No terms were discarded during indexing.

During querying, phrases are also analyzed and optimized. The query output (recall from section 3.5.1 that `Query`'s `toString()` is handy) of the Lucene `Query` instance for the query expression `"the quick brown"` is

```
Translated: +(url:"the quick brown"^4.0 anchor:"the quick brown"^2.0
  content:"the-quick quick brown" title:"the quick brown"^1.5 host:"the quick
  brown"^2.0)
```

A Nutch query expands to search in the `url`, `anchor`, `title` and `host` fields as well, with higher boosts for those fields, using the exact phrase. The `content` field clause is optimized to only include the bigram of a position that contains an additional `<WORD>` type token.

This was a quick view of what Nutch does with indexing analysis and query construction. Nutch continues to evolve, optimize, and tweak the various techniques for indexing and querying. The bigrams

aren't taken into consideration except in the `content` field; but as the document base grows, whether optimizations are needed on other fields will be reevaluated. You can use the shingles sandbox package, covered in section 8.XXX, to take the same approach as Nutch.

4.10 Summary

Analysis, while only a single facet of using Lucene, is the aspect that deserves the most attention and effort. The words that can be searched are those emitted during indexing analysis: nothing more, nothing less. Sure, using `StandardAnalyzer` may do the trick for your needs, and it suffices for many applications. However, it's important to understand the analysis process. Users who take analysis for granted often run into confusion later when they try to understand why searching for "to be or not to be" returns no results (perhaps due to stop-word removal).

It takes less than one line of code to incorporate an analyzer during indexing. Many sophisticated processes may occur under the covers, such as stop-word removal and stemming of words. Removing words decreases your index size but can have a negative impact on precision querying.

Because one size doesn't fit all when it comes to analysis, you may need to tune the analysis process for your application domain. Lucene's elegant analyzer architecture decouples each of the processes internal to textual analysis, letting you reuse fundamental building blocks to construct custom analyzers. When you're working with analyzers, be sure to use our `AnalyzerUtils`, or something similar, to see first-hand how your text is tokenized. If you're changing analyzers, you should rebuild your index using the new analyzer so that all documents are analyzed in the same manner.

5

Advanced search techniques

This chapter covers

- Loading field values for all documents
- Sorting search results
- Span queries
- Function queries
- Filtering
- Multiple and remote index searching
- Leveraging term vectors

Many applications that implement search with Lucene can do so using the API introduced in chapter 3. Some projects, though, need more than the basic searching mechanisms. In this chapter, we explore the more sophisticated searching capabilities built into Lucene.

Spanning queries, Payload queries, function queries and a couple of odds and ends, `MultiPhraseQuery` and `MultiFieldQueryParser`, round out our coverage of Lucene's additional built-in capabilities. We begin by describing Lucene's field cache.

5.1 Field cache

Sometimes you need fast access to a certain field's value for every document. Lucene's normal inverted index can't do this, since it optimizes instead for fast access to all documents containing a given term. Stored fields and term vectors do let you access field values by document number; however they can be slow and are generally not recommended for more than a page worth of results.

Lucene's field cache, an advanced API, was created to address this need. Often your application will not use the field cache directly, but functionality you do use, such as sorting results by field values (covered in the next section), uses the field cache under the hood. We'll visit a number of classes in this chapter that use field cache internally, so it's important to understand the tradeoffs of the field cache. Let's first see how to use field cache directly, should we need access to a field's value for all documents.

5.1.1 Loading document values

You can easily use the field cache to load an array of native values for a given field, indexed by document number. For example, if every document has a field called "weight", you can get the weight for all documents like this:

```
float[] weights = ExtendedFieldCache.EXT_DEFAULT.getFloats(reader, "weight");
```

Then, simply reference weights[docID] whenever you need to know a document's weight value. One very important restriction is that all documents must have a single value for the specified field.

NOTE

Field cache can only be used on fields that have a single term. This typically means the field was indexed with Index.NOT_ANALYZED.

The field cache supports many native types: *byte*, *short*, *int*, *long*, *float*, *double*, *strings*, *StringIndex* (includes sort order of the string values). An "auto" method (`getAuto`) will peek at the first document in the index and attempt to guess the appropriate native type.

The first time the field cache is accessed for a given reader and field, the values for all documents are visited and loaded into memory as a single large array, and recorded into an internal cache keyed on the reader instance and the field name. This process can be quite time consuming, for a large index. Subsequent calls quickly return the same array from the cache. The cache entry is not cleared until the reader is closed and completely dereferenced by your application (a `WeakHashMap` is used under the hood).

It's important to factor in the memory usage of field cache. Numeric fields require the number of bytes for the native type, multiplied by the number of documents. For `String` types, each unique term is also cached for each document. For highly unique fields, such as "title", this can be a large amount of memory. The `StringIndex` field cache, which is used when sorting by a string field, also stores an additional int array holding the sort order for all documents.

NOTE

The field cache may consume quite a bit of memory, since each entry allocates an array of the native type, whose length is equal to the number of documents in the reader. `FieldCache` does not clear its entries until you close your reader and remove all references to that reader from your application.

5.2 Sorting search results

By default, Lucene sorts the documents in descending relevance score order, where the most relevant documents appearing first. For example, for a book search you may want to display search results grouped into categories, and within each category the books should be ordered by relevance to the query. Collecting all results and sorting them programmatically outside of Lucene is one way to accomplish this; however, doing so introduces a possible performance bottleneck if the number of results is enormous. In this section, we explore the various ways to sort search results, including sorting by one or more field values in either ascending or descending order.

We'll begin by showing how to specify a custom sort when searching. Then we visit two special sort orders: relevance (the default sort) and index order. Then we'll sort by a field's values, including optionally reversing the sort order. Next we'll see how to sort by multiple sort criteria. We then show how to specify the field's type or locale, which is important to ensure the sort order is correct. Finally we briefly describe the performance cost of sorting.

5.2.1 Using a sort

`IndexSearcher` contains several overloaded `search` methods. Thus far we've covered only the basic `search(Query, int)` method, which returns the top N results ordered by decreasing relevance. The sorting version of this method has the signature `search(Query, Filter, int, Sort)`. Filter, which we'll cover Section 5.5, should be null if you don't need to filter the results. Listing 5.1 demonstrates the use of the sorting `search` method, you can run this by typing *ant SortingExample* in the book's source code directory. The `displayResults` method uses the sorting `search` method and displays the search results. The examples following will use the `displayResults` method to illustrate how various sorts work.

Listing 5.1 Sorting example

```
public class SortingExample {
  private Directory directory;

  public SortingExample(Directory directory) {
    this.directory = directory;
  }

  public void displayResults(Query query, Sort sort)            // #1
      throws IOException {
    IndexSearcher searcher = new IndexSearcher(directory);

    TopDocs results = searcher.search(query, null, 20, sort);   // #2

    System.out.println("\nResults for: " +                      // #3
        query.toString() + " sorted by " + sort);

    System.out.println(StringUtils.rightPad("Title", 30) +
        StringUtils.rightPad("pubmonth", 10) +
        StringUtils.center("id", 4) +
        StringUtils.center("score", 15));
```

```
    PrintStream out = new PrintStream(System.out, true, "UTF-8");    // #4

    DecimalFormat scoreFormatter = new DecimalFormat("0.######");
    for (int i = 0; i < results.scoreDocs.length; i++) {
      int docID = results.scoreDocs[i].doc;
      float score = results.scoreDocs[i].score;
      Document doc = searcher.doc(docID);
      out.println(
          StringUtils.rightPad(                                              // #5
              StringUtils.abbreviate(doc.get("title"), 29), 30) +            // #5
          StringUtils.rightPad(doc.get("pubmonth"), 10) +                    // #5
          StringUtils.center("" + docID, 4) +                                // #5
          StringUtils.leftPad(                                               // #5
              scoreFormatter.format(score), 12));                            // #5
      out.println("   " + doc.get("category"));
      //System.out.println(searcher.explain(query, results.scoreDocs[i].doc));   // #6
    }

    searcher.close();
  }

  public static void main(String[] args) throws Exception {
    String earliest = "190001";
    String latest = "201012";
    Query allBooks = new RangeQuery("pubmonth", earliest, latest, true, true);
    //allBooks.setConstantScoreRewrite(true);                                  //#7

    FSDirectory directory = new FSDirectory(new
File(TestUtil.getBookIndexDirectory()), null);
    SortingExample example = new SortingExample(directory);

    example.displayResults(allBooks, Sort.RELEVANCE);

    example.displayResults(allBooks, Sort.INDEXORDER);

    example.displayResults(allBooks, new Sort("category"));

    example.displayResults(allBooks, new Sort("pubmonth", true));

    example.displayResults(allBooks,
        new Sort(new SortField[]{
          new SortField("category"),
          SortField.FIELD_SCORE,
          new SortField("pubmonth", SortField.INT, true)
        }));


    example.displayResults(allBooks, new Sort(new SortField[] {SortField.FIELD_SCORE,
new SortField("category")}));
  }
}
```

#1 Sort object encapsulates sorting info

#2 Sarch method that accepts Sort
#3 toString output
#4 Ensure unicode output is handled properly
#5 StringUtils provides columnar output
#6 Explanation commented out for now
#7 Use constant-scoring rewrite method

#1 The Sort object encapsulates an ordered collection of field sorting information.
#2 We call the overloaded `search` method with the `Sort` object.
#3 The `Sort` class has informative `toString()` output.
#4 We use `StringUtils` from Jakarta Commons Lang for nice columnar output formatting.
#5 Later you'll see a reason to look at the explanation of score. For now, it's commented out.
#6 We could have used the ConstantScoreRangeQuery instead of RangeQuery

Here's the `TestUtil.getBookIndexDirectory` method:

```
public static String getBookIndexDirectory() {
  // The build.xml ant script sets this property for us:
  return System.getProperty("index.dir");
}
```

If you've been running the examples using ant, that directory maps to `build/index` in the filesystem, which contains an index of all the sample book descriptions from the `data/` directory tree. Since our sample data set consists of only a handful of documents, the sorting examples use a query that matches all documents:

```
String earliest = "190001";
String latest = "201012";
Query allBooks = new RangeQuery("pubmonth", earliest, latest, true, true);
```

All books in our collection are in this publication month range. Next, the example runner is constructed from the sample book index included with this book's source code:

```
FSDirectory directory = new FSDirectory(new File(TestUtil.getBookIndexDirectory()),
null);
SortingExample example = new SortingExample(directory);
```

Now that you've seen how to use sorting, let's explore various ways search results can be sorted.

5.2.2 Sorting by relevance

Lucene sorts by decreasing relevance, also called *score* by default. Sorting by score relevance works by either passing `null` as the `Sort` object or using the default `Sort` behavior. Each of these variants returns results in the default score order. `Sort.RELEVANCE` is a shortcut to using `new Sort()`:

```
example.displayHits(allBooks, Sort.RELEVANCE);
```

```
example.displayHits(allBooks, new Sort());
```

There is overhead involved in using a `Sort` object, though, so stick to using `search(Query, int)` if you want to sort by relevance. The output of using `Sort.RELEVANCE` is as follows (notice the decreasing score column):

```
example.displayHits(allBooks, Sort.RELEVANCE);

Results for: pubmonth:[190001 TO 201012] sorted by <score>,<doc>
Title                          pubmonth   id      score
A Modern Art of Education      198106     0       0.86743
   /education/pedagogy
Imperial Secrets of Health... 199401      1       0.86743
   /health/alternative/chinese
Tao Te Ching 道德經              198810     2       0.86743
   /philosophy/eastern
Gödel, Escher, Bach: an Et... 197903      3       0.86743
   /technology/computers/ai
Mindstorms                     198001     4       0.86743
   /technology/computers/programming/education
Java Development with Ant      200208     5       0.86743
   /technology/computers/programming
JUnit in Action                200310     6       0.86743
   /technology/computers/programming
Lucene in Action               200406     7       0.86743
   /technology/computers/programming
Tapestry in Action             200403     9       0.86743
   /technology/computers/programming
Extreme Programming Explained 199910      8      0.626853
   /technology/computers/programming/methodology
The Pragmatic Programmer       199910     10     0.626853
   /technology/computers/programming
```

The output of `Sort`'s `toString()` shows `<score>,<doc>`. Score and index order are special types of sorting: The results are returned first in decreasing score order and, when the scores are identical, subsorted with a secondary sort by increasing document ID order. Document ID order is the order in which the documents were indexed. In our case, index order isn't relevant, and order is unspecified (see section 8.4 on the Ant `<index>` task, which is how we indexed our sample data).

As an aside, you may wonder why the score of the last two books is different from the rest. Our query was on a publication date range. Both of these books have the same publication month. A `RangeQuery` by default expands into a `BooleanQuery` matching any of the terms in the range (see section 3.xxx). The document frequency of the term *199910* in the `pubmonth` field is 2, which lowers the inverse document frequency (IDF) factor for those documents, thereby decreasing the score. We had the same curiosity when developing this example, and uncommenting the `Explanation` output in `displayHits` gave us the details to understand this effect. See section 3.3. for more information on the scoring factors. Alternatively, if you use the commented out `ConstantScoreRangeQuery`, which assigns the same score to all documents matching it, then the score becomes 1.0 for all documents. Section 3.4.2 describes the differences between `ConstantScoreRangeQuery` and `RangeQuery` in more detail.

5.2.3 Sorting by index order

If the order documents were indexed is relevant, you can use `Sort.INDEXORDER`. Note the increasing document ID column:

```
example.displayHits(allBooks, Sort.INDEXORDER);

Results for: pubmonth:[190001 TO 201012] sorted by <doc>
Title                          pubmonth   id      score
A Modern Art of Education      198106     0       0.86743
   /education/pedagogy
Imperial Secrets of Health... 199401      1       0.86743
   /health/alternative/chinese
Tao Te Ching 道德經              198810     2       0.86743
   /philosophy/eastern
Gödel, Escher, Bach: an Et... 197903      3       0.86743
   /technology/computers/ai
Mindstorms                     198001     4       0.86743
   /technology/computers/programming/education
Java Development with Ant      200208     5       0.86743
   /technology/computers/programming
JUnit in Action                200310     6       0.86743
   /technology/computers/programming
Lucene in Action               200406     7       0.86743
   /technology/computers/programming
Extreme Programming Explained 199910      8      0.626853
   /technology/computers/programming/methodology
Tapestry in Action             200403     9       0.86743
   /technology/computers/programming
The Pragmatic Programmer       199910     10     0.626853
   /technology/computers/programming
```

Document order may be interesting for an index that you build up once and never change. But if you need to re-index documents, document order typically will not work because newly indexed documents receive new document IDs and will be sorted last. So far we've only sorted by score, which was already happening without using the sorting facility, and document order, which is probably only marginally useful at best. Sorting by one of our own fields is really what we're after.

5.2.4 Sorting by a field

Sorting by a field first requires that you follow the rules for indexing a sortable field, as detailed in section 2.9. Our `category` field was indexed with `Field.Index.NOT_ANALYZED` and `Field.Store.YES`, allowing it to be used for sorting. To sort by a field, you must create a new `Sort` object, providing the field name:

```
example.displayHits(allBooks, new Sort("category"));

Results for: pubmonth:[190001 TO 201012] sorted by <string: "category">,<doc>
Title                          pubmonth   id      score
A Modern Art of Education      198106     0       0.86743
   /education/pedagogy
Imperial Secrets of Health... 199401      1       0.86743
   /health/alternative/chinese
```

```
Tao Te Ching 道德經              198810     2       0.86743
   /philosophy/eastern
Gödel, Escher, Bach: an Et... 197903     3       0.86743
   /technology/computers/ai
Java Development with Ant     200208     5       0.86743
   /technology/computers/programming
JUnit in Action               200310     6       0.86743
   /technology/computers/programming
Lucene in Action              200406     7       0.86743
   /technology/computers/programming
Tapestry in Action            200403     9       0.86743
   /technology/computers/programming
The Pragmatic Programmer      199910     10     0.626853
   /technology/computers/programming
Mindstorms                    198001     4       0.86743
   /technology/computers/programming/education
Extreme Programming Explained 199910     8      0.626853
   /technology/computers/programming/methodology
```

The results now appear sorted by our `category` field in increasing alphabetical order. Notice the sorted-by output: The `Sort` class itself automatically adds document ID as the final sort field when a single field name is specified, so the secondary sort within category is by document ID.

5.2.5 Reversing sort order

The default sort direction for sort fields (including relevance and document ID) is natural ordering. Natural order is descending for relevance but increasing for all other fields. The natural order can be reversed per field. For example, here we list books with the newest publications first:

```
example.displayHits(allBooks, new Sort("pubmonth", true));

Results for: pubmonth:[190001 TO 201012] sorted by <int: "pubmonth">!,<doc>
Title                         pubmonth   id      score
Lucene in Action              200406     7       0.86743
   /technology/computers/programming
Tapestry in Action            200403     9       0.86743
   /technology/computers/programming
JUnit in Action               200310     6       0.86743
   /technology/computers/programming
Java Development with Ant     200208     5       0.86743
   /technology/computers/programming
Extreme Programming Explained 199910     8      0.626853
   /technology/computers/programming/methodology
The Pragmatic Programmer      199910     10     0.626853
   /technology/computers/programming
Imperial Secrets of Health... 199401     1       0.86743
   /health/alternative/chinese
Tao Te Ching 道德經              198810     2       0.86743
   /philosophy/eastern
A Modern Art of Education     198106     0       0.86743
   /education/pedagogy
Mindstorms                    198001     4       0.86743
   /technology/computers/programming/education
```

```
Gödel, Escher, Bach: an Et... 197903     3       0.86743
   /technology/computers/ai
```

The exclamation point in `sorted by "pubmonth"!,<doc>` indicates that the `pubmonth` field is being sorted in reverse natural order (descending publication months, newest first). Note that the two books with the same publication month are sorted in document id order.

5.2.6 Sorting by multiple fields

Sorting by multiple fields is important whenever your primary sort leaves ambiguity when there are equal values. Implicitly we've been sorting by multiple fields, since the `Sort` object appends a sort by document ID in appropriate cases. You can control the sort fields explicitly using an array of `SortFields`. This example uses category as a primary alphabetic sort, with results within category sorted by score; finally, books with equal score within a category are sorted by decreasing publication month:

```
example.displayHits(allBooks,
     new Sort(new SortField[]{
       new SortField("category"),
       SortField.FIELD_SCORE,
       new SortField("pubmonth", SortField.INT, true)
     }));
```

```
Results for: pubmonth:[190001 TO 201012] sorted by <string: "category">,<score>,<int:
"pubmonth">!
Title                          pubmonth   id      score
A Modern Art of Education      198106     0       0.86743
   /education/pedagogy
Imperial Secrets of Health... 199401     1       0.86743
   /health/alternative/chinese
Tao Te Ching 道德經               198810     2       0.86743
   /philosophy/eastern
Gödel, Escher, Bach: an Et... 197903     3       0.86743
   /technology/computers/ai
Lucene in Action               200406     7       0.86743
   /technology/computers/programming
Tapestry in Action             200403     9       0.86743
   /technology/computers/programming
JUnit in Action                200310     6       0.86743
   /technology/computers/programming
Java Development with Ant      200208     5       0.86743
   /technology/computers/programming
The Pragmatic Programmer       199910     10     0.626853
   /technology/computers/programming
Mindstorms                     198001     4       0.86743
   /technology/computers/programming/education
Extreme Programming Explained 199910     8      0.626853
   /technology/computers/programming/methodology
```

The `Sort` instance internally keeps an array of `SortFields`, but only in this example have you seen it explicitly; the other examples used shortcuts to creating the `SortField` array. A `SortField` holds the field name, a field type, and the reverse order flag. `SortField` contains constants for several field types, including `SCORE`, `DOC`, `AUTO`, `STRING`, `BYTE`, `SHORT`, `INT`, `LONG`, `FLOAT`, and `DOUBLE`. `SCORE` and `DOC` are special types for sorting on relevance and document ID. `AUTO` is the type used by each of our other examples, which sort by a field name.

The type of field is automatically detected as `String`, `int`, or `float` based on the value of the first term in the field. If you're using strings that may appear as numeric in some fields, be sure to specify the type explicitly as `SortField.STRING`.

5.2.7 Selecting a sorting field type

By search time, the fields that can be sorted on and their corresponding types are already set. Indexing time is when the decision about sorting capabilities should be made; however, custom sorting implementations can do so at search time, as you'll see in section 6.1. Section 2.6 discusses index-time sorting design. By indexing an `Integer.toString` or `Float.toString`, sorting can be based on numeric values. In our example data, `pubmonth` was indexed as a `String` but is a valid, parsable `Integer`; thus it's treated as such for sorting purposes unless specified as `SortField.STRING` explicitly. Sorting by a numeric type consumes fewer memory resources than by `STRING`; section 5.1.9 discusses performance issues further.

It's important to understand that you index numeric values this way to facilitate sorting on those fields, not to constrain a search on a range of values. The numeric range query capability is covered in section 6.3.3; the padding technique will be necessary during indexing and searching in order to use numeric fields for searching. All terms in an index are `String`s; the sorting feature uses the standard `Integer` and `Float` constructors to parse the string representations.

When sorting by `String` values you may need to specify your own locale, which we cover next.

5.2.8 Using a nondefault locale for sorting

When you're sorting on a `SortField.STRING` type, order is determined under the covers using `String.compareTo` by default. However, if you need a different collation order, `SortField` lets you specify a `Locale`. A `Collator` is obtained for the provided locale using `Collator.getInstance(Locale)`, and the `Collator.compare` method determines the sort order. There are two overloaded `SortField` constructors for use when you need to specify locale:

```
public SortField (String field, Locale locale)
public SortField (String field, Locale locale, boolean reverse)
```

Both of these constructors imply the `SortField.STRING` type because locale applies only to string-type sorting, not to numerics.

5.2.9 Performance effect of sorting

Sorting by field value (ie, anything except relevance and document order) uses the field cache to retrieve values. This means the first query sorting by a given field and reader will be slower, and memory will be consumed holding onto the cache entries. If the first query is too slow, it's best to first "warm" your `IndexSearcher` before putting it into production, as described in 10.X. Sorting by `String`, especially for fields with many unique values, will consume the most memory.

5.3 Using MultiPhraseQuery

The built-in `MultiPhraseQuery` is definitely a niche query, but it's potentially useful. `MultiPhraseQuery` allows multiple terms per position, effectively the same as a `BooleanQuery` on

multiple non-required PhraseQuery clauses. For example, suppose we want to find all documents about speedy foxes, with *quick* or *fast* followed by *fox*. One approach is to do a "quick fox" OR "fast fox" query. Another option is to use MultiPhraseQuery. In our example, two documents are indexed with similar phrases. One document with uses “the quick brown fox jumped over the lazy dog”, and the other uses “the fast fox hopped over the hound” as shown in our test setUp() method:

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

  protected void setUp() throws Exception {
    Directory directory = new RAMDirectory();
    IndexWriter writer = new IndexWriter(directory,
                                         new WhitespaceAnalyzer(),
                                         IndexWriter.MaxFieldLength.UNLIMITED);
    Document doc1 = new Document();
    doc1.add(new Field("field",
              "the quick brown fox jumped over the lazy dog",
              Field.Store.YES, Field.Index.ANALYZED));
    writer.addDocument(doc1);
    Document doc2 = new Document();
    doc2.add(new Field("field",
              "the fast fox hopped over the hound",
              Field.Store.YES, Field.Index.ANALYZED));
    writer.addDocument(doc2);
    writer.close();

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

Knowing that we want to find documents about speedy foxes, MultiPhraseQuery lets us match phrases very much like PhraseQuery, but with a twist: each term position of the query can have multiple terms. This has the same set of hits as a BooleanQuery consisting of multiple PhraseQuerys combined with an OR operator. The following test method demonstrates the mechanics of using the MultiPhraseQuery API by adding one or more terms to a MultiPhraseQuery instance in order:

```
    public void testBasic() throws Exception {
    MultiPhraseQuery query = new MultiPhraseQuery();
    query.add(new Term[] {                        // #1
        new Term("field", "quick"),               // #1
        new Term("field", "fast")                 // #1
    });
    query.add(new Term("field", "fox"));          // #2
    System.out.println(query);

    TopDocs hits = searcher.search(query, 10);
    assertEquals("fast fox match", 1, hits.totalHits);

    query.setSlop(1);
    hits = searcher.search(query, 10);
    assertEquals("both match", 2, hits.totalHits);
```

```
  }
```

#1 Any of these terms may be in first position to match
#2 Only one in second position

Just as with `PhraseQuery`, the slop factor is supported. In `testBasic()`, the slop is used to match "quick brown fox" in the second search; with the default slop of zero, it doesn't match. For completeness, here is a test illustrating the described `BooleanQuery`, with a slop set for the phrase "quick fox":

```
  public void testAgainstOR() throws Exception {
    PhraseQuery quickFox = new PhraseQuery();
    quickFox.setSlop(1);
    quickFox.add(new Term("field", "quick"));
    quickFox.add(new Term("field", "fox"));

    PhraseQuery fastFox = new PhraseQuery();
    fastFox.add(new Term("field", "fast"));
    fastFox.add(new Term("field", "fox"));

    BooleanQuery query = new BooleanQuery();
    query.add(quickFox, BooleanClause.Occur.SHOULD);
    query.add(fastFox, BooleanClause.Occur.SHOULD);
    TopDocs hits = searcher.search(query, 10);
    assertEquals(2, hits.totalHits);
  }
```

One difference between `MultiPhraseQuery` and the `BooleanQuery` of `PhraseQuery`'s approach is that the slop factor is applied globally with `MultiPhraseQuery`—it's applied on a per-phrase basis with `PhraseQuery`.

Of course, hard-coding the terms wouldn't be realistic, generally speaking. One possible use of a `MultiPhraseQuery` would be to inject synonyms dynamically into phrase positions, allowing for less precise matching. For example, you could tie in the WordNet-based code (see section 8.6 for more on WordNet and Lucene). `QueryParser` produces a `MultiPhraseQuery` for search terms surrounded in double quotes when the analyzer it's using returns `positionIncrement` 0 for any of the tokens within the phrase:

```
  public void testQueryParser() throws Exception {
    TokenStream.setUseNewAPIDefault(true);
    SynonymEngine engine = new SynonymEngine() {
        public String[] getSynonyms(String s) {
          if (s.equals("quick"))
            return new String[] {"fast"};
          else
            return null;
        }
      };

    Query q = new QueryParser("field", new SynonymAnalyzer(engine)).parse("\"quick
fox\"");
```

```
    assertEquals("analyzed",
        "field:\"(quick fast) fox\"", q.toString());
    assertTrue("parsed as MultiPhraseQuery", q instanceof MultiPhraseQuery);
  }
```

Next we'll visit `MultiFieldQueryParser`, for querying on multiple fields.

5.4 Querying on multiple fields at once

In our book data, several fields were indexed. Users may want to query for terms regardless of which field they are in. One way to handle this is with `MultiFieldQueryParser`, which builds on `QueryParser`. Under the covers, it instantiates a `QueryParser` and parses the query expression for each field and then combines the resulting queries using a `BooleanQuery`. The default operator `OR` is used in the simplest `parse` method when adding the clauses to the `BooleanQuery`. For finer control, the operator can be specified for each field as required (`BooleanClause.Occur.MUST`), prohibited (`BooleanClause.Occur.MUST_NOT`), or normal (BooleanClause.Occur.SHOULD), using the constants from `MultiFieldQueryParser`.

Listing 5.2 shows this heavier `QueryParser` variant in use. The `testDefaultOperator()` method first parses the query `"development"` using both the `title` and `subjects` fields. The test shows that documents match based on either of those fields. The second test, `testSpecifiedOperator()`, sets the parsing to mandate that documents must match the expression in all specified fields.

Listing 5.2 MultiFieldQueryParser in action

```
  public void testDefaultOperator() throws Exception {
    Query query = new MultiFieldQueryParser(new String[]{"title", "subject"},
        new SimpleAnalyzer()).parse("development");

    IndexSearcher searcher = new IndexSearcher(TestUtil.getBookIndexDirectory());
    TopDocs hits = searcher.search(query, 10);

    assertTrue(TestUtil.hitsIncludeTitle(searcher, hits, "Java Development with Ant"));

    // has "development" in the subject field
    assertTrue(TestUtil.hitsIncludeTitle(searcher, hits, "Extreme Programming
Explained"));
  }

  public void testSpecifiedOperator() throws Exception {
    Query query = MultiFieldQueryParser.parse("development",
        new String[]{"title", "subject"},
        new BooleanClause.Occur[]{BooleanClause.Occur.MUST,
                  BooleanClause.Occur.MUST},
        new SimpleAnalyzer());

    IndexSearcher searcher = new IndexSearcher(TestUtil.getBookIndexDirectory());
    TopDocs hits = searcher.search(query, 10);

    assertTrue(TestUtil.hitsIncludeTitle(searcher, hits, "Java Development with Ant"));
    assertEquals("one and only one", 1, hits.scoreDocs.length);
  }
```

MultiFieldQueryParser has some limitations due to the way it uses QueryParser. You can't control any of the settings that QueryParser supports, and you're stuck with the defaults such as default locale date parsing and zero-slop default phrase queries.

NOTE

Generally speaking, querying on multiple fields isn't the best practice for user-entered queries. More commonly, all words you want searched are indexed into a contents or keywords field by combining various fields. A synthetic contents field in our test environment uses this scheme to put author and subjects together:

```
doc.add(new Field("contents", author + " " + subjects, Field.Store.NO,
Field.Index.ANALYZED));
```

We used a space (" ") between author and subjects to separate words for the analyzer. Allowing users to enter text in the simplest manner possible without the need to qualify field names, generally makes for a less confusing user experience.

If you choose to use MultiFieldQueryParser, be sure your queries are fabricated appropriately using the QueryParser and Analyzer diagnostic techniques shown in chapters 3 and 4. Plenty of odd interactions with analysis occur using QueryParser, and these are compounded when using MultiFieldQueryParser.

We'll now move onto span queries, advanced queries that allow you to match based on positional constraints.

5.5 Span queries

Lucene includes a whole family of queries based on SpanQuery. A *span* in this context is a starting and ending token position in a field. Recall from section 4.2.1 that tokens emitted during the analysis process include a position increment from the previous token. This position information, in conjunction with the new SpanQuery subclasses, allow for even more query discrimination and sophistication, such as all documents where "quick fox" is near "lazy dog".

Using the query types we've discussed thus far, it isn't possible to formulate such a query. Phrase queries could get close with something like "quick fox" AND "lazy dog", but these phrases may be too distant from one another to be relevant for our searching purposes.

Span queries track more than the documents that match: The individual spans, perhaps more than one per field, are tracked. Contrasting with TermQuery, which simply matches documents, for example, SpanTermQuery keeps track of the positions of each of the terms that match, for every document.

There are five subclasses of the base SpanQuery, shown in table 5.1.

Table 5.1 SpanQuery family

SpanQuery type	Description
SpanTermQuery	Used in conjunction with the other span query types. On its own, it's functionally equivalent to TermQuery.
SpanFirstQuery	Matches spans that occur within the first part of a field.
SpanNearQuery	Matches spans that occur near one another.
SpanNotQuery	Matches spans that don't overlap one another.
SpanOrQuery	Aggregates matches of span queries.

We'll discuss each of these SpanQuery types within the context of a JUnit test case, SpanQueryTest. In order to demonstrate each of these types, a bit of setup is needed as well as some helper assert methods to make our later code clearer, as shown in listing 5.3. We index two similar phrases in a field f as separate documents and create SpanTermQuerys for several of the terms for later use in our test methods. In addition, we add three convenience assert methods to streamline our examples.

Listing 5.3 SpanQuery demonstration infrastructure

```
public class SpanQueryTest extends TestCase {
  private RAMDirectory directory;
  private IndexSearcher searcher;
  private IndexReader reader;

  private SpanTermQuery quick;
  private SpanTermQuery brown;
  private SpanTermQuery red;
  private SpanTermQuery fox;
  private SpanTermQuery lazy;
  private SpanTermQuery sleepy;
  private SpanTermQuery dog;
  private SpanTermQuery cat;
  private Analyzer analyzer;

  protected void setUp() throws Exception {
    directory = new RAMDirectory();

    analyzer = new WhitespaceAnalyzer();
    IndexWriter writer = new IndexWriter(directory,
                                         analyzer,
                                         IndexWriter.MaxFieldLength.UNLIMITED);

    Document doc = new Document();
    doc.add(new Field("f",
```

```
            "the quick brown fox jumps over the lazy dog",
            Field.Store.YES, Field.Index.ANALYZED));
        writer.addDocument(doc);

        doc = new Document();
        doc.add(new Field("f",
            "the quick red fox jumps over the sleepy cat",
            Field.Store.YES, Field.Index.ANALYZED));
        writer.addDocument(doc);

        writer.close();

        searcher = new IndexSearcher(directory);
        reader = IndexReader.open(directory);

        quick = new SpanTermQuery(new Term("f", "quick"));
        brown = new SpanTermQuery(new Term("f", "brown"));
        red = new SpanTermQuery(new Term("f", "red"));
        fox = new SpanTermQuery(new Term("f", "fox"));
        lazy = new SpanTermQuery(new Term("f", "lazy"));
        sleepy = new SpanTermQuery(new Term("f", "sleepy"));
        dog = new SpanTermQuery(new Term("f", "dog"));
        cat = new SpanTermQuery(new Term("f", "cat"));
      }

      private void assertOnlyBrownFox(Query query)throws Exception {
        TopDocs hits = searcher.search(query, 10);
        assertEquals(1, hits.totalHits);
        assertEquals("wrong doc", 0, hits.scoreDocs[0].doc);
      }

      private void assertBothFoxes(Query query) throws Exception {
        TopDocs hits = searcher.search(query, 10);
        assertEquals(2, hits.totalHits);
      }

      private void assertNoMatches(Query query) throws Exception {
        TopDocs hits = searcher.search(query, 10);
        assertEquals(0, hits.totalHits);
      }
    }
```

With this necessary bit of setup out of the way, we can begin exploring span queries. First we'll ground ourselves with `SpanTermQuery`.

5.5.1 Building block of spanning, SpanTermQuery

Span queries need an initial leverage point, and `SpanTermQuery` is just that. Internally, a `SpanQuery` keeps track of its matches: a series of start/end positions for each matching document. By itself, a `SpanTermQuery` matches documents just like `TermQuery` does, but it also keeps track of position of the same terms that appear within each document.

Figure 5.1 illustrates the `SpanTermQuery` matches for this code:

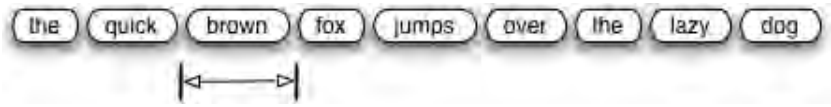


Figure 5.1 `SpanTermQuery` for *brown*

```
public void testSpanTermQuery() throws Exception {
  assertOnlyBrownFox(brown);
  dumpSpans(brown);
}
```

The *brown* `SpanTermQuery` was created in `setUp()` because it will be used in other tests that follow. We developed a method, `dumpSpans`, to visualize spans. The `dumpSpans` method uses some lower-level `SpanQuery` APIs to navigate the spans; this lower-level API probably isn't of much interest to you other than for diagnostic purposes, so we don't elaborate further on it. Each `SpanQuery` subclass sports a useful `toString()` for diagnostic purposes, which `dumpSpans` uses:

```
private void dumpSpans(SpanQuery query) throws IOException {
 Spans spans = query.getSpans(reader);
 System.out.println(query + ":");
 int numSpans = 0;

 TopDocs hits = searcher.search(query, 10);
 float[] scores = new float[2];
 for (int i = 0; i < hits.scoreDocs.length; i++) {
   scores[hits.scoreDocs[i].doc] = hits.scoreDocs[i].score;
 }

 while (spans.next()) {
   numSpans++;

   int id = spans.doc();
   Document doc = reader.document(id);

   // for simplicity - assume tokens are in sequential,
   // positions, starting from 0
   AttributeSource[] tokens = AnalyzerUtils.tokensFromAnalysis(
                                       analyzer, doc.get("f"));
   StringBuffer buffer = new StringBuffer();
   buffer.append("   ");
   for (int i = 0; i < tokens.length; i++) {
     if (i == spans.start()) {
       buffer.append("<");
     }
     buffer.append(AnalyzerUtils.getTerm(tokens[i]));
     if (i + 1 == spans.end()) {
       buffer.append(">");
     }
     buffer.append(" ");
   }
   buffer.append("(" + scores[id] + ") ");
   System.out.println(buffer);
```

```
//      System.out.println(searcher.explain(query, id));
    }

    if (numSpans == 0) {
      System.out.println("   No spans");
    }
    System.out.println();
  }
```

The output of `dumpSpans(brown)` is

```
f:brown:
   the quick <brown> fox jumps over the lazy dog (0.22097087)
```

More interesting is the `dumpSpans` output from a `SpanTermQuery` for *the*:

```
dumpSpans(new SpanTermQuery(new Term("f", "the")));

f:the:
   <the> quick brown fox jumps over the lazy dog (0.18579213)
   the quick brown fox jumps over <the> lazy dog (0.18579213)
   <the> quick red fox jumps over the sleepy cat (0.18579213)
   the quick red fox jumps over <the> sleepy cat (0.18579213)
```

Not only were both documents matched, but also each document had two span matches highlighted by the brackets. The basic `SpanTermQuery` is used as a building block of the other `SpanQuery` types. Let's see how to match only documents where the terms of interest occur in the beginning of the field.

5.5.2 Finding spans at the beginning of a field

To query for spans that occur within the first *n* positions of a field, use `SpanFirstQuery` . Figure 5.2 illustrates a `SpanFirstQuery`.

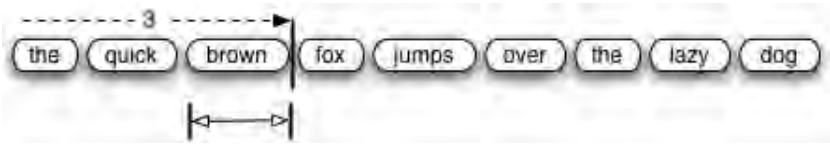


Figure 5.2 `SpanFirstQuery`

This test shows nonmatching and matching queries:

```
public void testSpanFirstQuery() throws Exception {
  SpanFirstQuery sfq = new SpanFirstQuery(brown, 2);
  assertNoMatches(sfq);

  sfq = new SpanFirstQuery(brown, 3);
  assertOnlyBrownFox(sfq);
}
```

No matches are found in the first query because the range of 2 is too short to find *brown*, but 3 is just long enough to cause a match in the second query (see figure 5.2). Any `SpanQuery` can be used within a `SpanFirstQuery`, with matches for spans that have an ending position in the first *n* (2 and 3 in this case) positions. The resulting span matches are the same as the original `SpanQuery` spans, in this case the same `dumpSpans()` output for *brown* as seen in section 5.4.1.

5.5.3 Spans near one another

A `PhraseQuery` (see section 3.4.5) matches documents that have terms near one another, with a slop factor to allow for intermediate or reversed terms. `SpanNearQuery` operates similarly to `PhraseQuery`, with some important differences. `SpanNearQuery` matches spans that are within a certain number of positions from one another, with a separate flag indicating whether the spans must be in the order specified or can be reversed. The resulting matching spans span from the start position of the first span sequentially to the ending position of the last span. An example of a `SpanNearQuery` given three `SpanTermQuery` objects is shown in figure 5.3.

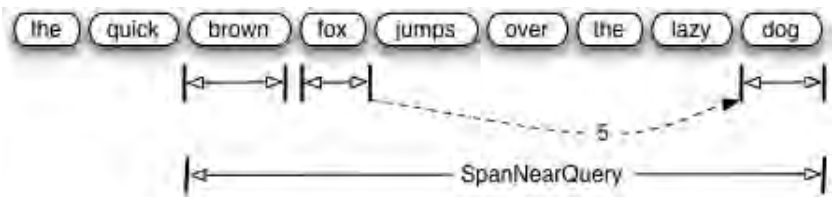


Figure 5.3 `SpanNearQuery`

Using `SpanTermQuery` objects as the `SpanQuery`s in a `SpanNearQuery` is much like a `PhraseQuery`. However, the `SpanNearQuery` slop factor is a bit less confusing than the `PhraseQuery` slop factor because it doesn't require at least two additional positions to account for a reversed span. To reverse a `SpanNearQuery`, set the `inOrder` flag (third argument to the constructor) to false. Listing 5.4 demonstrates a few variations of `SpanNearQuery` and shows it in relation to `PhraseQuery`.

Listing 5.4 SpanNearQuery

```
public void testSpanNearQuery() throws Exception {
SpanQuery[] quick_brown_dog =
    new SpanQuery[]{quick, brown, dog};
SpanNearQuery snq =
  new SpanNearQuery(quick_brown_dog, 0, true);                // #1
assertNoMatches(snq);
dumpSpans(snq);

snq = new SpanNearQuery(quick_brown_dog, 4, true);            // #2
assertNoMatches(snq);
dumpSpans(snq);

snq = new SpanNearQuery(quick_brown_dog, 5, true);            // #3
```

```
    assertOnlyBrownFox(snq);
    dumpSpans(snq);

    // interesting - even a sloppy phrase query would require
    // more slop to match
    snq = new SpanNearQuery(new SpanQuery[]{lazy, fox}, 3, false);// #4
    assertOnlyBrownFox(snq);
    dumpSpans(snq);

    PhraseQuery pq = new PhraseQuery();                          // #5
    pq.add(new Term("f", "lazy"));                               // #5
    pq.add(new Term("f", "fox"));                                // #5
    pq.setSlop(4);                                               // #5
    assertNoMatches(pq);

    pq.setSlop(5);                                               // #6
    assertOnlyBrownFox(pq);                                      // #6
  }
```

#1 Query for three successive terms
#2 Same terms, slop of 4
#3 SpanNearQuery matches
#4 Nested SpanTermQuery objects in reverse order
#5 Comparable PhraseQuery
#6 PhraseQuery, slop of 5

#1 Querying for these three terms in successive positions doesn't match either document.
#2 Using the same terms with a slop of 4 positions still doesn't result in a match.
#3 With a slop of 5, the `SpanNearQuery` has a match.
#4 The nested `SpanTermQuery` objects are in reverse order, so the `inOrder` flag is set to `false`. A slop of only 3 is needed for a match.
#5 Here we use a comparable `PhraseQuery`, although a slop of 4 still doesn't match.
#6 A slop of 5 is needed for a `PhraseQuery` to match.

We've only shown `SpanNearQuery` with nested `SpanTermQuery`s, but `SpanNearQuery` allows for any `SpanQuery` type. A more sophisticated `SpanNearQuery` is demonstrated later in listing 5.5 in conjunction with `SpanOrQuery`. Next we visit `SpanNotQuery`.

5.5.4 Excluding span overlap from matches

The `SpanNotQuery` excludes matches where one `SpanQuery` overlaps another. The following code demonstrates:

```
  public void testSpanNotQuery() throws Exception {
    SpanNearQuery quick_fox =
        new SpanNearQuery(new SpanQuery[]{quick, fox}, 1, true);
    assertBothFoxes(quick_fox);
    dumpSpans(quick_fox);

    SpanNotQuery quick_fox_dog = new SpanNotQuery(quick_fox, dog);
    assertBothFoxes(quick_fox_dog);
    dumpSpans(quick_fox_dog);
```

```
    SpanNotQuery no_quick_red_fox =
        new SpanNotQuery(quick_fox, red);
    assertOnlyBrownFox(no_quick_red_fox);
    dumpSpans(no_quick_red_fox);
  }
```

The first argument to the `SpanNotQuery` constructor is a span to *include*, and the second argument is the span to *exclude*. We've strategically added `dumpSpans` to clarify what is going on. Here is the output with the Java query annotated above each:

```
SpanNearQuery quick_fox =
        new SpanNearQuery(new SpanQuery[]{quick, fox}, 1, true);
spanNear([f:quick, f:fox], 1, true):
   the <quick brown fox> jumps over the lazy dog (0.18579213)
   the <quick red fox> jumps over the sleepy cat (0.18579213)

SpanNotQuery quick_fox_dog = new SpanNotQuery(quick_fox, dog);
spanNot(spanNear([f:quick, f:fox], 1, true), f:dog):
   the <quick brown fox> jumps over the lazy dog (0.18579213)
   the <quick red fox> jumps over the sleepy cat (0.18579213)

SpanNotQuery no_quick_red_fox =
        new SpanNotQuery(quick_fox, red);
spanNot(spanNear([f:quick, f:fox], 1, true), f:red):
   the <quick brown fox> jumps over the lazy dog (0.18579213)
```

The `SpanNear` query matched both documents because both have *quick* and *fox* within one position of one another. The first `SpanNotQuery`, `quick_fox_dog`, continues to match both documents because there is no overlap with the `quick_fox` span and *dog*. The second `SpanNotQuery`, `no_quick_red_fox`, excludes the second document because *red* overlaps with the `quick_fox` span. Notice that the resulting span matches are the original included span. The excluded span is only used to determine if there is an overlap and doesn't factor into the resulting span matches. Our final query is `SpanOrQuery`.

5.5.5 Spanning the globe

Finally there is `SpanOrQuery` , which aggregates an array of `SpanQuery`s. Our example query, in English, is `all documents that have "quick fox" near "lazy dog" or that have "quick fox" near "sleepy cat"`. The first clause of this query is shown in figure 5.4. This single clause is `SpanNearQuery` nesting two `SpanNearQuery`s, which each consist of two `SpanTermQuery`s.

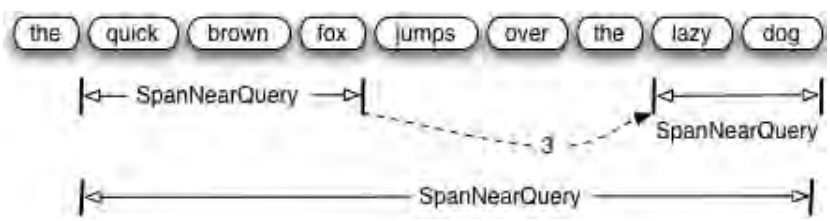


Figure 5.4 One clause of the `SpanOrQuery`

Our test case becomes a bit lengthier due to all the sub-`SpanQuery`s being built upon (see listing 5.5). Using `dumpSpans`, we analyze the code in more detail.

Listing 5.5 SpanOrQuery

```
public void testSpanOrQuery() throws Exception {
  SpanNearQuery quick_fox =
      new SpanNearQuery(new SpanQuery[]{quick, fox}, 1, true);

  SpanNearQuery lazy_dog =
      new SpanNearQuery(new SpanQuery[]{lazy, dog}, 0, true);

  SpanNearQuery sleepy_cat =
      new SpanNearQuery(new SpanQuery[]{sleepy, cat}, 0, true);

  SpanNearQuery qf_near_ld =
      new SpanNearQuery(
          new SpanQuery[]{quick_fox, lazy_dog}, 3, true);
  assertOnlyBrownFox(qf_near_ld);
  dumpSpans(qf_near_ld);

  SpanNearQuery qf_near_sc =
      new SpanNearQuery(
          new SpanQuery[]{quick_fox, sleepy_cat}, 3, true);
  dumpSpans(qf_near_sc);

  SpanOrQuery or = new SpanOrQuery(
      new SpanQuery[]{qf_near_ld, qf_near_sc});
  assertBothFoxes(or);
  dumpSpans(or);
}
```

We've used our handy `dumpSpans` a few times to allow us to follow the progression as the final OR query is built. Here is the output, followed by our analysis of it:

```
SpanNearQuery qf_near_ld =
        new SpanNearQuery(
            new SpanQuery[]{quick_fox, lazy_dog}, 3, true);
spanNear([spanNear([f:quick, f:fox], 1, true),
          spanNear([f:lazy, f:dog], 0, true)], 3, true):
   the <quick brown fox jumps over the lazy dog> (0.3321948)

SpanNearQuery qf_near_sc =
        new SpanNearQuery(
            new SpanQuery[]{quick_fox, sleepy_cat}, 3, true);
spanNear([spanNear([f:quick, f:fox], 1, true),
          spanNear([f:sleepy, f:cat], 0, true)], 3, true):
   the <quick red fox jumps over the sleepy cat> (0.3321948)

SpanOrQuery or = new SpanOrQuery(
        new SpanQuery[]{qf_near_ld, qf_near_sc});
spanOr([spanNear([spanNear([f:quick, f:fox], 1, true),
```

```
                spanNear([f:lazy, f:dog], 0, true)], 3, true),
    spanNear([spanNear([f:quick, f:fox], 1, true),
              spanNear([f:sleepy, f:cat], 0, true)], 3, true)]):
the <quick brown fox jumps over the lazy dog> (0.6643896)
the <quick red fox jumps over the sleepy cat> (0.6643896)
```

Two SpanNearQuerys are created to match *quick fox* near *lazy dog* (qf_near_ld) and *quick fox* near *sleepy cat* (qf_near_sc) using nested SpanNearQuerys made up of SpanTermQuerys at the lowest level. Finally, these two SpanNearQuery instances are combined within a SpanOrQuery, which aggregates all matching spans. Whew!

5.5.6 SpanQuery and QueryParser

QueryParser doesn't currently support any of the SpanQuery types, but the surround QueryParser in the sandbox does. We cover the surround parser in Section 8.XXX.

Recall from section 3.4.5 that PhraseQuery is impartial to term order when enough slop is specified. Interestingly, you can easily extend QueryParser to use a SpanNearQuery with SpanTermQuery clauses instead, and force phrase queries to only match fields with the terms in the same order as specified. We demonstrate this technique in section 6.3.4.

5.6 Filtering a search

Filtering is a mechanism of narrowing the search space, allowing only a subset of the documents to be considered as possible hits. They can be used to implement search-within-search features to successively search within a previous set of results or to constrain the document search space for security or external data reasons. A security filter is a powerful example, allowing users to only see search results of documents they own even if their query technically matches other documents that are off limits; we provide an example of a security filter in section 5.5.3.

You can filter any Lucene search, using the overloaded search methods that accept a Filter parameter. There are several built-in Filter implementations:

- PrefixFilter matches only documents containing terms in a specific field with a specific prefix.
- RangeFilter matches only documents containing terms within a specified range of terms.

  FieldCacheTermsFilter matches documents containing specific terms.
- QueryWrapperFilter uses the results of query as the searchable document space for a new query.
- CachingWrapperFilter is a decorator over another filter, caching its results to increase performance when used again.

Before you get concerned about mentions of caching results, rest assured that it's done with a tiny data structure (a DocIdBitSet) where each bit position represents a document.

Consider, also, the alternative to using a filter: aggregating required clauses in a BooleanQuery. In this section, we'll discuss each of the built-in filters as well as the BooleanQuery alternative, starting with RangeFilter.

5.6.1 Using RangeFilter

`RangeFilter` filters on a range of terms in a specific field. This is actually very useful, depending on the original type of the field. If the field is a date field, then you get a date range filter. If it's an integer field, you can filter by numeric range. If the field is simply textual, for example last names, then you can filter for all names within a certain alphabetic range such as M to Q.

Let's look at date filtering as an example. Indexing dates is covered in section 2.4. Having a `date` field, you filter as shown in `testDateFilter()` in Listing 5.6. Our book data indexes the last modified date of each book data file as a `modified` field, indexed as with `Field.Index.NOT_ANALYZED` and Field.Store.YES. We test the date RangeFilter by using an all-inclusive query, which by itself returns all documents.

Listing 5.6 Using RangeFilter to filter by date range

```
public class FilterTest extends TestCase {
  private RangeQuery allBooks;
  private IndexSearcher searcher;
  private int numAllBooks;

  protected void setUp() throws Exception {    // #1
    allBooks = new RangeQuery(
                    "pubmonth",
                    "190001",
                    "200512",
                    true, true);
    allBooks.setConstantScoreRewrite(true);
    searcher = new IndexSearcher(TestUtil.getBookIndexDirectory());
    numAllBooks = TestUtil.hitCount(searcher, allBooks);
  }

  public void testDateFilter() throws Exception {
    String jan1 = TestUtil.parseDate("2004-01-01");
    String jan31 = TestUtil.parseDate("2004-01-31");
    String dec31 = TestUtil.parseDate("2004-12-31");

    RangeFilter filter = new RangeFilter("modified", jan1, dec31, true, true);

    assertEquals("all modified in 2004",
                 numAllBooks,
                 TestUtil.hitCount(searcher, allBooks, filter));

    filter = new RangeFilter("modified", jan1, jan31, true, true);
    assertEquals("none modified in January",
                 0,
                 TestUtil.hitCount(searcher, allBooks, filter));
  }
}
```

#1 setUp() establishes baseline book count

#1 `setUp()` establishes a baseline count of all the books in our index, allowing for comparisons when we use an all inclusive date filter.

The first parameter to both of the `RangeFilter` constructors is the name of a date field in the index. In our sample data this field name is `modified`; this field is the last modified date of the source data file. The two final boolean arguments to the constructor for `RangeFilter`, `includeLower` and `includeUpper`, determine whether the lower and upper terms should be included or excluded from the filter. The `TestUtil.parseDate` method is very simple:

```
    public static String parseDate(String s) throws ParseException {
      return DateTools.dateToString(new SimpleDateFormat("yyyy-MM-
  dd").parse(s),DateTools.Resolution.MILLISECOND);
    }
```

Ranges can also be optionally open-ended.

OPEN-ENDED RANGE FILTERING

`RangeFilter` also supports open-ended ranges. To filter on ranges with one end of the range specified and the other end open, just pass null for whichever end should be open:

```
filter = new RangeFilter("modified", null, jan31, false, true);
filter = new RangeFilter("modified", jan1, null, true, false);
```

RangeFilter provides two static convenience methods to achieve the same thing:

```
filter = RangeFilter.Less("modified", jan31);
filter = RangeFilter.More("modified", jan1);
```

FIELDCACHERANGEFILTER

`FieldCacheRangeFilter` is another option for range filtering. It achieves exactly the same filtering, but does so with a very different implementation. It loads the `StringIndex` field cache entry under the hood, so it inherits the known benefits and limitations of field cache (described in section 5.1). While it's used in exactly the same way as `RangeFilter`, since the underlying implementation is very different, there are important tradeoffs. To check the filter for each document, it maps the upper and lower bounds to their corresponding points in the sort order, and then checks each document against these bounds. Each field must have only a single value.

TRIERANGEFILTER

A third option for range filtering is `TrieRangeFilter`, which lives in Lucene's sandbox. It's described in detail in section 8.12.5. It takes yet another interesting approach for range filtering, by pre-dividing the field's range of values into larger and larger ranges and then aggregating the ranges at search time, to achieve any desired lower and upper bound. Functionally, it's identical to `RangeFilter` and

`FieldCacheRangeFilter`. The tradeoff is a slight increase in index size, for likely very much faster range filtering. Generally for medium to large indexes, `TrieRangeFilter` is the best option.

Let's see how to filter by an arbitrary set of terms.

5.6.2 Filtering by specific terms

Sometimes you'd simply like to select specific terms to include in your filter. For example, perhaps your documents have Country as a field, and your search interface presents a checkbox allowing the user to pick and choose which countries to include in the search. There are actually two ways to achieve this.

The first approach is `FieldCacheTermsFilter`, which uses field cache under the hood (see section 5.1 for the known benefits and limitations of field cache). Simply instantiate it with the field (String) and an array of `String`:

```
public void testFieldCacheTermsFilter() throws Exception {
  Filter filter = new FieldCacheTermsFilter("pubmonth",
                                            new String[] {"199910",
                                                          "200406"});
  assertEquals("only 3 hits",
               3,
               TestUtil.hitCount(searcher, allBooks, filter));
}
```

All documents that have any of the specific terms in the specified field will be accepted. Note that the documents must have a single term value for each field. Under the hood, this filter loads all terms for all documents into the field cache the first time it's used during searching for a given field. This means the first search will be slower, but subsequent searches, which re-use the cache, will be very fast. The field cache is re-used even if you change which specific terms are included in the filter.

The second approach for filtering by terms is `TermsFilter`, which is included in Lucene's sandbox and described in more detail in section XXX. `TermsFilter` does not do any internal caching, and also allows filtering on fields that have more than one term. It's best to test both approaches for your application to see if there are any significant performance differences.

5.6.3 Using QueryWrapperFilter

`QueryFilter` uses the hits of one query to constrain available documents from a subsequent search. The result is a `DocIdSet` representing which documents were matched from the filtering query. Using a `QueryWrapperFilter`, we restrict the documents searched to a specific category:

```
public void testQueryWrapperFilter() throws Exception {
  TermQuery categoryQuery =
     new TermQuery(new Term("category", "/philosophy/eastern"));

  Filter categoryFilter = new QueryWrapperFilter(categoryQuery);

  assertEquals("only tao te ching",
               1,
               TestUtil.hitCount(searcher, allBooks, categoryFilter));
}
```

Here we're searching for all the books (see `setUp()` in listing 5.6) but constraining the search using a filter for a category which contains a single book. We explain the last assertion of `testQueryFilter()` shortly, in section 5.5.4.

`QueryWrapperFilter` can even replace `RangeFilter` usage, although it requires a few more lines of code, isn't nearly as elegant looking and likely has worse performance. The following code demonstrates date filtering using a `QueryWrapperFilter` on a `RangeQuery`:

```
  public void testQueryWrapperFilterWithRangeQuery() throws Exception {
    String jan1 = TestUtil.parseDate("2004-01-01");
    String dec31 = TestUtil.parseDate("2004-12-31");

    RangeQuery rangeQuery = new RangeQuery("modified", jan1, dec31,
                                           true, true);
    Filter filter = new QueryWrapperFilter(rangeQuery);
    assertEquals("all of 'em", numAllBooks, TestUtil.hitCount(searcher, allBooks,
filter));
  }
```

Let's see how to use filters for applying security constraints, also known as entitlements.

5.6.4 Security filters

Another example of document filtering constrains documents with security in mind. Our example assumes documents are associated with an owner, which is known at indexing time. We index two documents; both have the term *info* in their `keywords` field, but each document has a different owner:

```
public class SecurityFilterTest extends TestCase {

  private IndexSearcher searcher;

  protected void setUp() throws Exception {
    Directory directory = new RAMDirectory();
    IndexWriter writer = new IndexWriter(directory,
                                         new WhitespaceAnalyzer(),
                                         IndexWriter.MaxFieldLength.UNLIMITED);

    // Elwood
    Document document = new Document();
    document.add(new Field("owner", "elwood", Field.Store.YES,
Field.Index.NOT_ANALYZED));
    document.add(new Field("keywords", "elwood's sensitive info", Field.Store.YES,
Field.Index.ANALYZED));
    writer.addDocument(document);

    // Jake
    document = new Document();
    document.add(new Field("owner", "jake", Field.Store.YES,
Field.Index.NOT_ANALYZED));
    document.add(new Field("keywords", "jake's sensitive info", Field.Store.YES,
Field.Index.ANALYZED));
```

```
    writer.addDocument(document);

    writer.close();
    searcher = new IndexSearcher(directory);
  }
}
```

Using a `TermQuery` for *info* in the `keywords` field results in both documents found, naturally. Suppose, though, that Jake is using the search feature in our application, and only documents he owns should be searchable by him. Quite elegantly, we can easily use a `QueryWrapperFilter` to constrain the search space to only documents he is the owner of, as shown in listing 5.7.

Listing 5.7 Securing the search space with a filter

```
public void testSecurityFilter() throws Exception {
  TermQuery query = new TermQuery(new Term("keywords", "info"));     //#1

  assertEquals("Both documents match",                        //#2
               2,                                             //#2
               TestUtil.hitCount(searcher, query));           //#2

  Filter jakeFilter = new QueryWrapperFilter(                         //#3
    new TermQuery(new Term("owner", "jake")));                        //#3

  TopDocs hits = searcher.search(query, jakeFilter, 10);
  assertEquals(1, hits.totalHits);                                    //#4
  assertEquals("elwood is safe",                                      //#4
               "jake's sensitive info",                               //#4
      searcher.doc(hits.scoreDocs[0].doc).get("keywords"));           //#4
}
```

#1 TermQuery for "info"
#2 Returns documents containing "info"
#3 Filter
#4 Same TermQuery, constrained results

#1 This is a general `TermQuery` for *info*.

#2 All documents containing *info* are returned.

#3 Here, the filter constrains document searches to only documents owned by “jake”.

#4 Only Jake’s document is returned, using the same *info* `TermQuery`.

If your security requirements are this straightforward, where documents can be associated with users or roles during indexing, using a `QueryWrapperFilter` will work nicely. However, this scenario is oversimplified for most needs; the ways that documents are associated with roles may be quite a bit more dynamic. `QueryWrapperFilter` is useful only when the filtering constraints are present as field information within the index itself. In section 6.4, we develop a more sophisticated filter implementation

that leverages external information; this approach could be adapted to a more dynamic custom security filter.

5.6.5 A QueryWrapperFilter alternative

You can constrain a query to a subset of documents another way, by combining the constraining query to the original query as a *required* clause of a `BooleanQuery`. There are a couple of important differences, despite the fact that the same documents are returned from both. If you use `CachingWrapperFilter` around your `QueryWrapperFilter`, you can cache the set of documents allowed, probably speeding up successive searches using the same filter. In addition, normalized document scores are unlikely to be the same. The score difference makes sense when you're looking at the scoring formula (see section 3.3, page 78). The IDF factor may be dramatically different. When you're using `BooleanQuery` aggregation, all documents containing the terms are factored into the equation, whereas a filter reduces the documents under consideration and impacts the inverse document frequency factor.

This test case demonstrates how to "filter" using `BooleanQuery` aggregation and illustrates the scoring difference compared to `testQueryFilter`:

```
public void testFilterAlternative() throws Exception {
  TermQuery categoryQuery =
     new TermQuery(new Term("category", "/philosophy/eastern"));

  BooleanQuery constrainedQuery = new BooleanQuery();
  constrainedQuery.add(allBooks, BooleanClause.Occur.MUST);
  constrainedQuery.add(categoryQuery, BooleanClause.Occur.MUST);

  assertEquals("only tao te ching",
               1,
               TestUtil.hitCount(searcher, constrainedQuery));
}
```

The technique of aggregating a query in this manner works well with `QueryParser` parsed queries, allowing users to enter free-form queries yet restricting the set of documents searched by an API-controlled query. Better integration of `Query` and `Filter` classes is an active topic of discussion in Lucene, so you may very well see changes in this area. We'll describe `PrefixFilter` next.

5.6.6 PrefixFilter

`PrefixFilter` matches documents containing Terms starting with a specified prefix. We can use this to search for all books published any year in the 1900s:

```
public void testPrefixFilter() throws Exception {
  Filter prefixFilter = new PrefixFilter(new Term("pubmonth", "19"));
  assertEquals("only 19XX books",
               7,
               TestUtil.hitCount(searcher, allBooks, prefixFilter));
}
```

Next we show how to cache a filter for better performance.

5.6.7 Caching filter results

The biggest benefit from filters comes when they are cached and reused, using `CachingWrapperFilter`, which takes care of caching automatically (internally using a `WeakHashMap`, so that dereferenced entries get garbage collected). You can cache any Filter using `CachingWrappingFilter`. Filters cache by using the `IndexReader` as the key, which means searching should also be done with the same instance of `IndexReader` to benefit from the cache. If you aren't constructing `IndexReader` yourself, but rather are creating an `IndexSearcher` from a directory, you must use the same instance of `IndexSearcher` to benefit from the caching. When index changes need to be reflected in searches, discard `IndexSearcher` and `IndexReader` and reinstantiate.

To demonstrate its usage, we return to the date-range filtering example. We want to use `RangeFilter`, but we'd like to benefit from caching to improve performance:

```
public void testCachingWrapper() throws Exception {
  String jan1 = TestUtil.parseDate("2004-01-01");
  String dec31 = TestUtil.parseDate("2004-12-31");

  RangeFilter dateFilter =
      new RangeFilter("modified", jan1, dec31, true, true);

  CachingWrapperFilter cachingFilter = new CachingWrapperFilter(dateFilter);
  assertEquals("all of 'em",
               numAllBooks,
               TestUtil.hitCount(searcher, allBooks, cachingFilter));
}
```

Successive uses of the same `CachingWrapperFilter` instance with the same `IndexSearcher` instance will bypass using the wrapped filter, instead using the cached results.

5.6.8 Wrapping a Filter as a Query

We saw how to wrap a `Filter` as a `Query`. You can also do the reverse, using `ConstantScoreQuery` to turn any `Filter` into a `Query`, which you can then search on. The query matches only documents that are included in the Filter, and assigns all of them the score equal to the query boost. The queries that have a constant score mode (`PrefixQuery`, `RangeQuery`, `WildcardQuery` and `FuzzyQuery`) all simply use a `ConstantScoreQuery` wrapped around the corresponding filter.

5.6.9 Beyond the built-in filters

Lucene isn't restricted to using the built-in filters. An additional filter found in the Lucene Sandbox, `ChainedFilter`, allows for complex chaining of filters. We cover it in section 8.8.

Writing custom filters allows external data to factor into search constraints; however, a bit of detailed Lucene API know-how may be required to be highly efficient. We cover writing custom filters in section 6.4.

And if these filtering options aren't enough, Lucene adds another interesting use of a filter. The `FilteredQuery` filters a query, like `IndexSearcher`'s `search(Query, Filter, int)` can, except it is itself a query: Thus it can be used as a single clause within a `BooleanQuery`. Using `FilteredQuery`

seems to make sense only when using custom filters, so we cover it along with custom filters in section 6.4.

We are done with filters. Our next topic is function queries, which give you custom control over how documents are scored.

5.7 Custom scoring using function queries

Lucene's relevance scoring formula, which we discussed in Chapter 3, does a good job assigning relevance to each document based on how well the document matches the query. But what if you'd like to modify or override how this scoring is done? In section 5.1 we saw how you could change the default relevance sorting to sort by an arbitrary field, but what if you need even more flexibility? This is where function queries come in.

Function queries, in the package `org.apache.lucene.search.function`, allow you to programmatically customize how each document is scored. For example, `FieldScoreQuery` derives each document's score statically from a specific indexed field. The field should be numbers, indexed without norms and with a single token per document. Typically you would use `Field.Index.NOT_ANALYZED_NO_NORMS`. Let's look at a simple example. First, include the field "score" in your documents like this:

```
f = new Field("score", "42", Field.Store.NO, Field.Index.NOT_ANALZYED_NO_NORMS);
doc.add(f);
```

Then, create this function query:

```
Query q = new FieldScoreQuery("score", FieldScoreQuery.Type.BYTE);
```

That query matches all documents, assigning each a score according to the contents of its "score" field. You can also use the `SHORT`, `INT` or `FLOAT` types. Under the hood, this function query uses the same field cache used when you sort by a specified field, which means the first search that uses a given field will be slower as it must populate the cache. Subsequent searches based on the same IndexReader and field are then fast.

The above example is somewhat contrived, since you could simply sort by the score field, descending, to achieve the same thing. But function queries get more interesting when you combine them using the second type of function query, `CustomScoreQuery`. This query class lets you combine a normal Lucene query with one or more `ValueSourceQuery`s. A `ValueSourceQuery` is the super class of `FieldScoreQuery`, and is simply a query that matches all documents and returns an arbitrary pre-assigned score per document. `FieldScoreQuery` is one such example, where the score is derived from an indexed field. You could also provide values from some external source, for example a database.

We can now use the `FieldScoreQuery` we created above, and a `CustomScoreQuery`, to compute our own score:

```
Query q = queryParser.parse("my query text");
FieldScoreQuery qf = new FieldScoreQuery("score", FieldScoreQuery.Type.BYTE);
```

```
CustomScoreQuery customQ = new CustomScoreQuery(q,qf) {
  public float customScore(int doc, float subQueryScore, float valSrcScore) {
    return (float) (Math.sqrt(subQueryScore) * valSrcScore);
  }
};
```

In this case we create a normal query `q` by parsing the user's search text. We next create the same `FieldScoreQuery` we used above, which assigns static scores to documents according to the field `score`. Finally, we create a `CustomScoreQuery`, overriding the `customScore` method to compute our score for each matching document. In this case, we take the square root of the incoming query score and then multiply it by the static score provided by the `FieldScoreQuery`.

Listing 5.8 Using recency to boost search results

```
static class RecencyBoostingQuery extends CustomScoreQuery {

  int[] daysAgo;
  double multiplier;
  int maxDaysAgo;

  public RecencyBoostingQuery(Query q, int[] daysAgo, double multiplier,
                              int maxDaysAgo) {
    super(q);
    this.daysAgo = daysAgo;
    this.multiplier = multiplier;
    this.maxDaysAgo = maxDaysAgo;
  }

  public float customScore(int doc, float subQueryScore,
                           float valSrcScore) {
    if (daysAgo[doc] < maxDaysAgo) {                                  // #1
      float boost = (float) (multiplier * (maxDaysAgo-daysAgo[doc])   // #2
                             / maxDaysAgo);                           // #2
      return (float) (subQueryScore * (1.0+boost));
    } else
      return subQueryScore;                                           // #3
  }
};
```

#1 Check if book is new enough to get a boost
#2 Do a simple linear boost; other functions are possible
#3 Book is too old

A real-world use of `CustomScoreQuery` is to do custom document boosting. Listing 5.8 shows a new query class, `RecencyBoostingQuery`, that boosts documents that were modified more recently. In applications where documents have a clear timestamp, such as searching a news feed or press releases, boosting by relevance is very useful. The class requires you to externally pre-compute the `daysAgo` array, which for each doc id holds the number of days ago that the document was updated. Here's how we can use this to search our books index:

```
  final private static int MSEC_PER_DAY = 24*3600*1000;

  public void testRecency() throws Throwable {
    IndexReader r = IndexReader.open(bookDirectory);
    IndexSearcher s = new IndexSearcher(r);
    int maxDoc = s.maxDoc();
    int[] daysAgo = new int[maxDoc];
    //long now = new Date().getTime();
    long now = DateTools.stringToTime("200410");
    for(int i=0;i<maxDoc;i++) {                    // #1
      if (!r.isDeleted(i)) {
        long then = DateTools.stringToTime(r.document(i).get("pubmonth"));
        daysAgo[i] = (int) ((now - then)/MSEC_PER_DAY);
      }
    }

    QueryParser parser = new QueryParser("contents", new StandardAnalyzer());
    Query q = parser.parse("technology in action");  // #2
    Query q2 = new RecencyBoostingQuery(q, daysAgo,  // #3
                                        2.0, 2*365);
    TopDocs hits = s.search(q2, 5);
    //TopDocs hits = s.search(q, 5);
    for(int i=0;i<hits.scoreDocs.length;i++) {
      Document doc = r.document(hits.scoreDocs[i].doc);
      System.out.println((1+i) + ": " + doc.get("title") + ": score=" +
hits.scoreDocs[i].score);
    }
    s.close();
  }
```

#1 Build up daysAgo array, containing how many days ago each book was published
#2 Normal query
#3 Query that boosts by recency

In this example, we populate the `daysAgo` array by loading each stored document and processing the `pubmonth` field, but you could also derive it from any other source, for example an external database. In computing `daysAgo`, we pretend today is 10/1/2004 to better show the recency effect. In a production use you should use the current day. We instantiate the `RecencyBoostingQuery`, giving a boost factor of up to 2.0 for any book published within the past 2 years. When you run the test you'll see this output:

```
1: Lucene in Action: score=1.5612519
2: Tapestry in Action: score=1.4136308
3: JUnit in Action: score=1.1697354
4: Mindstorms: score=0.076921
5: Extreme Programming Explained: score=0.076921
```

If instead you run the commented out line, which runs the original unboosted query, you'll see this:

```
1: JUnit in Action: score=0.58567
2: Lucene in Action: score=0.58567
3: Tapestry in Action: score=0.58567
```

```
4: Mindstorms: score=0.076921
5: Extreme Programming Explained: score=0.076921
```

You can see that in the un-boosted query, the top 3 results were tied based on relevance. But after factoring in recency boosting, the scores were very different and the sort order changed (for the better, we might add!).

5.8 Searching across multiple Lucene indexes

If your architecture consists of multiple Lucene indexes, but you need to search across them using a single query with search results interleaving documents from different indexes, `MultiSearcher` is for you. In high-volume usage of Lucene, your architecture may partition sets of documents into different indexes.

5.8.1 Using MultiSearcher

With `MultiSearcher`, all indexes can be searched with the results merged in a specified (or descending-score) order. Using `MultiSearcher` is comparable to using `IndexSearcher`, except that you hand it an array of `IndexSearchers` to search rather than a single directory (so it's effectively a decorator pattern and delegates most of the work to the subsearchers).

Listing 5.8 illustrates how to search two indexes that are split alphabetically by keyword. The index is made up of animal names beginning with each letter of the alphabet. Half the names are in one index, and half are in the other. A search is performed with a range that spans both indexes, demonstrating that results are merged together.

Listing 5.9 Securing the search space with a filter

```
public class MultiSearcherTest extends TestCase {
  private IndexSearcher[] searchers;

  public void setUp() throws Exception {
    String[] animals = { "aardvark", "beaver", "coati",
                         "dog", "elephant", "frog", "gila monster",
                         "horse", "iguana", "javelina", "kangaroo",
                         "lemur", "moose", "nematode", "orca",
                         "python", "quokka", "rat", "scorpion",
                         "tarantula", "uromastyx", "vicuna",
                         "walrus", "xiphias", "yak", "zebra"};

    Analyzer analyzer = new WhitespaceAnalyzer();

    Directory aTOmDirectory = new RAMDirectory();     // #1
    Directory nTOzDirectory = new RAMDirectory();     // #1

    IndexWriter aTOmWriter = new IndexWriter(aTOmDirectory,
                                             analyzer,
                                             IndexWriter.MaxFieldLength.UNLIMITED);
    IndexWriter nTOzWriter = new IndexWriter(nTOzDirectory,
                                             analyzer,
                                             IndexWriter.MaxFieldLength.UNLIMITED);
```

```
    for (int i=animals.length - 1; i >= 0; i--) {
      Document doc = new Document();
      String animal = animals[i];
      doc.add(new Field("animal", animal, Field.Store.YES, Field.Index.NOT_ANALYZED));
      if (animal.compareToIgnoreCase("n") < 0) {
        aTOmWriter.addDocument(doc);                // #2
      } else {
        nTOzWriter.addDocument(doc);                // #2
      }
    }

    aTOmWriter.close();
    nTOzWriter.close();

    searchers = new IndexSearcher[2];
    searchers[0] = new IndexSearcher(aTOmDirectory);
    searchers[1] = new IndexSearcher(nTOzDirectory);
  }

  public void testMulti() throws Exception {

    MultiSearcher searcher = new MultiSearcher(searchers);

    // range spans documents across both indexes
    RangeQuery query = new RangeQuery("animal",   // #3
                                      "h",        // #3
                                      "t",        // #3
                                      true, true);// #3
    query.setConstantScoreRewrite(true);

    TopDocs hits = searcher.search(query, 10);
    assertEquals("tarantula not included", 12, hits.totalHits);
  }
}
```

#1 Two indexes
#2 Indexing halves of the alphabet
#3 Query spans both indexes

#1 This code uses two indexes.

#2 The first half of the alphabet is indexed to one index, and the other half is indexed to the other index.

#3 This query spans documents in both indexes.

The inclusive `ConstantScoreRangeQuery` matched animals that began with *h* through animals that began with *t*, with the matching documents coming from both indexes. A related class, `ParallelMultiSearcher`, achieves the same functionality as `MultiSearcher`, but uses multiple threads to gain concurrency.

5.8.2 Multithreaded searching using ParallelMultiSearcher

A multithreaded version of `MultiSearcher`, called `ParallelMultiSearcher`, spins a new thread for each Searchable and waits for them all to finish, when the search method is invoked. The basic search and search with filter options are parallelized, but searching with a `HitCollector` has not yet been parallelized.

Whether you'll see performance gains using `ParallelMultiSearcher` greatly depends on your architecture. Supposedly, if the indexes reside on different physical disks and you're able to take advantage of multiple CPUs, there may be improved performance; but in our tests with a single CPU, single physical disk, and multiple indexes, performance with `MultiSearcher` was slightly better than `ParallelMultiSearcher`.

Using a `ParallelMultiSearcher` is identical to using `MultiSearcher`. An example, using `ParallelMultiSearcher` remotely, is shown in listing 5.9.

SEARCHING MULTIPLE INDEXES REMOTELY

Lucene includes remote index searching capability through Remote Method Invocation (RMI). There are numerous other alternatives to exposing search remotely, such as through web services. This section focuses solely on Lucene's built-in capabilities; other implementations are left to your innovation (you can also borrow ideas from projects like Nutch; see section 10.1).

An RMI server binds to an instance of `RemoteSearchable`, which is an implementation of the `Searchable` interface just like `IndexSearcher` and `MultiSearcher`. The server-side `RemoteSearchable` delegates to a concrete `Searchable`, such as a regular `IndexSearcher` instance.

Clients to the `RemoteSearchable` invoke search methods identically to searching through an `IndexSearcher` or `MultiSearcher`, as shown throughout this chapter. Figure 5.5 illustrates one possible remote-searching configuration.

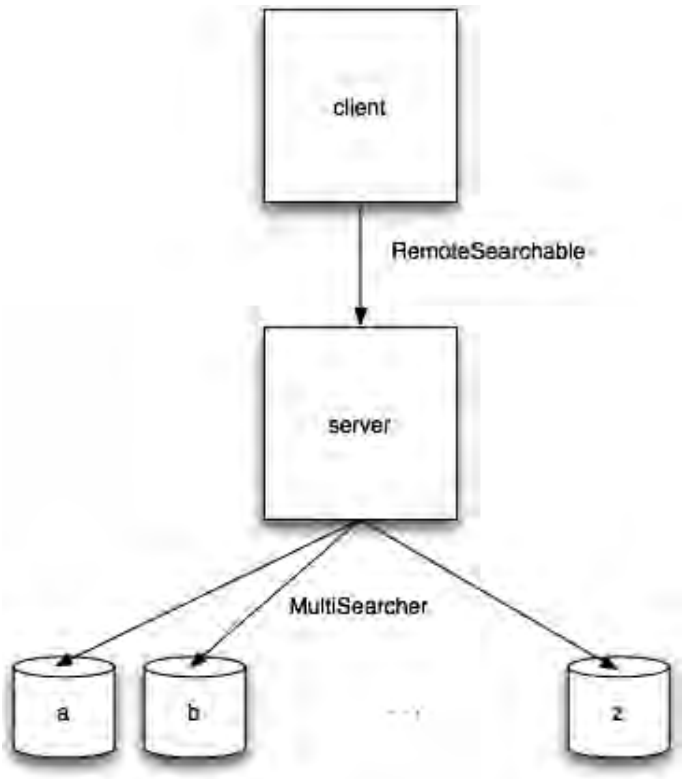


Figure 5.5 Remote searching through RMI, with the server searching multiple indexes

Other configurations are possible, depending on your needs. The client could instantiate a `ParallelMultiSearcher` over multiple remote (and/or local) indexes, and each server could search only a single index.

In order to demonstrate `RemoteSearchable` , we put together a multi-index server configuration, similar to figure 5.5, using both `MultiSearcher` and `ParallelMultiSearcher` in order to compare performance. We split the WordNet index (a database of nearly 44,000 words and their synonyms) into 26 indexes representing *A* through *Z*, with each word in the index corresponding to its first letter. The server exposes two RMI client-accessible `RemoteSearchables`, allowing clients to access either the serial `MultiSearcher` or the `ParallelMultiSearcher`.

`SearchServer` is shown in listing 5.9.

Listing 5.10 SearchServer: a remove search server using RMI

```
public class SearchServer {
  private static final String ALPHABET =
      "abcdefghijklmnopqrstuvwxyz";
```

```
  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("Usage: SearchServer <basedir>");
      System.exit(-1);
    }

    String basedir = args[0];   |#1
    Searchable[] searchables = new Searchable[ALPHABET.length()];
    for (int i = 0; i < ALPHABET.length(); i++) {
      searchables[i] = new IndexSearcher(
          new File(basedir,
              "" + ALPHABET.charAt(i)).getAbsolutePath());  |#2
    }

    LocateRegistry.createRegistry(1099);   |#3

    Searcher multiSearcher = new MultiSearcher(searchables);   |#4
    RemoteSearchable multiImpl =                               |#4
        new RemoteSearchable(multiSearcher);                   |#4
    Naming.rebind("//localhost/LIA_Multi", multiImpl);         |#4

    Searcher parallelSearcher =                                  |#5
        new ParallelMultiSearcher(searchables);                  |#5
    RemoteSearchable parallelImpl =                              |#5
        new RemoteSearchable(parallelSearcher);                  |#5
    Naming.rebind("//localhost/LIA_Parallel", parallelImpl);   |#5

    System.out.println("Server started");
  }
}
```

#1 Indexes under basedir
#2 Open IndexSearcher for each index
#3 Create RMI registry
#4 MultiSearcher over all indexes
#5 ParallelMultiSearcher over all indexes

#1 Twenty-six indexes reside under the `basedir`, each named for a letter of the alphabet.

#2 A plain `IndexSearcher` is opened for each index.

#3 An RMI registry is created.

#4 A `MultiSearcher` over all indexes, named `LIA_Multi`, is created and published through RMI.

#5 A `ParallelMultiSearcher` over the same indexes, named `LIA_Parallel`, is created and published.

Querying through `SearchServer` remotely involves mostly RMI glue, as shown in `SearchClient` in listing 5.10. Because our access to the server is through a `Remote-Searchable`, which is a lower-level

API than we want to work with, we wrap it inside a `MultiSearcher`. Why `MultiSearcher`? Because it's a wrapper over `Searchables`, making it as friendly to use as `IndexSearcher`.

Listing 5.11 SearchClient: accesses the RMI-exposed objects from SearchServer

```
public class SearchClient {
  private static HashMap searcherCache = new HashMap();

  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("Usage: SearchClient <query>");
      System.exit(-1);
    }

    String word = args[0];
    for (int i=0; i < 5; i++) {        |#1
      search("LIA_Multi", word);       |#1
      search("LIA_Parallel", word);    |#1
    }                                  |#1
  }

  private static void search(String name, String word)
      throws Exception {
    TermQuery query = new TermQuery(new Term("word", word));

    MultiSearcher searcher =
        (MultiSearcher) searcherCache.get(name);   |#2

    if (searcher == null) {
      searcher =
        new MultiSearcher(new Searchable[]{lookupRemote(name)});   |#3
      searcherCache.put(name, searcher);

    }

    long begin = new Date().getTime();    |#4
    Hits hits = searcher.search(query);   |#4
    long end = new Date().getTime();      |#4

    System.out.print("Searched " + name +
        " for '" + word + "' (" + (end - begin) + " ms): ");

    if (hits.length() == 0) {
      System.out.print("<NONE FOUND>");
    }

    for (int i = 0; i < hits.length(); i++) {
      Document doc = hits.doc(i);
      String[] values = doc.getValues("syn");
      for (int j = 0; j < values.length; j++) {
        System.out.print(values[j] + " ");
      }
    }
    System.out.println();
    System.out.println();
```

```
    // DO NOT CLOSE searcher!   |#5
  }

  private static Searchable lookupRemote(String name)
      throws Exception {
    return (Searchable) Naming.lookup("//localhost/" + name);   |#6
  }
}
```

#1 Multiple identical searches
#2 Cache searchers
#3 Wrap Searchable in MultiSearcher
#4 Time searching
#5 Don't close searcher
#6 RMI lookup

#1 We perform multiple identical searches to warm up the JVM and get a good sample of response time. The `MultiSearcher` and `ParallelMultiSearcher` are each searched.

#2 The searchers are cached, to be as efficient as possible.

#3 The remote `Searchable` is located and wrapped in a `MultiSearcher`.

#4 The searching process is timed.

#5 We don't close the searcher because it closes the remote searcher, thereby prohibiting future searches.

#6 Look up the remote interface.

WARNING

Don't `close()` the `RemoteSearchable` or its wrapping `MultiSearcher`. Doing so will prevent future searches from working because the server side will have closed its access to the index.

Let's see our remote searcher in action. For demonstration purposes, we ran it on a single machine in separate console windows. The server is started:

```
% java lia.advsearching.remote.SearchServer path/to/indexes/
Server started
```

The client connects, searches, outputs the results several times, and exits:

```
% java lia.advsearching.remote.SearchClient hello

Searched LIA_Multi for 'hello' (55 ms): hi howdy hullo

Searched LIA_Parallel for 'hello' (26 ms): hi howdy hullo

Searched LIA_Multi for 'hello' (8 ms): hi howdy hullo
```

```
Searched LIA_Parallel for 'hello' (28 ms): hi howdy hullo
Searched LIA_Multi for 'hello' (6 ms): hi howdy hullo
Searched LIA_Parallel for 'hello' (11 ms): hi howdy hullo
Searched LIA_Multi for 'hello' (6 ms): hi howdy hullo
Searched LIA_Parallel for 'hello' (12 ms): hi howdy hullo
Searched LIA_Multi for 'hello' (6 ms): hi howdy hullo
Searched LIA_Parallel for 'hello' (12 ms): hi howdy hullo
```

It's interesting to note the search times reported by each type of server-side searcher. The `ParallelMultiSearcher` is slower than the `MultiSearcher` in our environment (4 CPUs, single disk). Also, you can see the reason why we chose to run the search multiple times: The first search took much longer relative to the successive searches, which is probably due to JVM warmup. These results point out that performance testing is tricky business, but it's necessary in many environments. Because of the strong effect your environment has on performance, we urge you to perform your own tests with your own environment. Performance testing is covered in more detail in section 6.5.

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

We switch things up now and talk about term vectors, a topic you've already seen on the indexing side, in chapter 2.

5.9 Leveraging term vectors

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

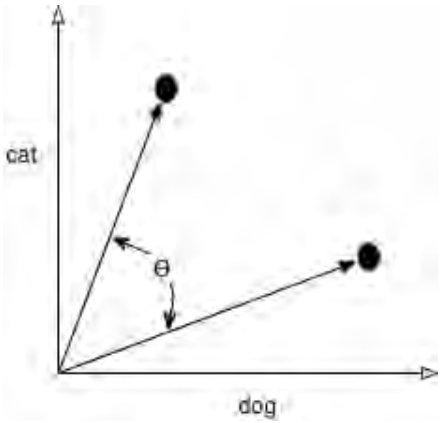


Figure 5.6 Term vectors for two documents containing the terms *cat* and *dog*

We covered how to enable indexing of term vectors in Section 2.2.1. We indexed the `title`, `author`, `subject` and `contents` fields with term vectors when indexing our book data. Retrieving term vectors for a field in a given document by ID requires a call to an `IndexReader` method:

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

A `TermFreqVector` instance has several methods for retrieving the vector information, primarily as matching arrays of `String`s and `int`s (the term value and frequency in the field, respectively). If you had also stored offsets and/or positions information with your term vectors, using `Field.TermVector.WITH_POSITIONS_OFFSETS` for example, then you'll get a `TermPositionVector` back when you load the term vectors. That class contains offset and position information for each occurrence of the terms in the document.

You can use term vectors for some interesting effects, such as finding documents "like" a particular document, which is an example of latent semantic analysis. We'll show how to find books similar to an existing one, as well as a proof-of-concept categorizer that can tell us the most appropriate category for a new book, as you'll see in the following sections. We wrap up with the `TermVectorMapper` classes for precisely controlling how term vectors are read from the index.

5.9.1 Books like this

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

Listing 5.12 Books like this

```
public class BooksLikeThis {

  public static void main(String[] args) throws IOException {
    String indexDir = System.getProperty("index.dir");

    FSDirectory directory = new FSDirectory(new File(indexDir), null);

    IndexReader reader = IndexReader.open(directory);
    int numDocs = reader.maxDoc();

    BooksLikeThis blt = new BooksLikeThis(reader);
    for (int i = 0; i < numDocs; i++) {                    // #1
      System.out.println();
      Document doc = reader.document(i);
      System.out.println(doc.get("title"));

      Document[] docs = blt.docsLike(i, 10);               // #2
      if (docs.length == 0) {
        System.out.println("  None like this");
      }
      for (int j = 0; j < docs.length; j++) {
        Document likeThisDoc = docs[j];
        System.out.println("  -> " + likeThisDoc.get("title"));
      }
    }
  }

  private IndexReader reader;
  private IndexSearcher searcher;

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

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

    String[] authors = doc.getValues("author");
    BooleanQuery authorQuery = new BooleanQuery();                 // #3
    for (int i = 0; i < authors.length; i++) {                     // #3
      String author = authors[i];                                  // #3
      authorQuery.add(new TermQuery(new Term("author", author)),   // #3
          BooleanClause.Occur.SHOULD);                             // #3
    }
    authorQuery.setBoost(2.0f);

    TermFreqVector vector =                                        // #4
        reader.getTermFreqVector(id, "subject");                   // #4

    BooleanQuery subjectQuery = new BooleanQuery();                // #4
    for (int j = 0; j < vector.size(); j++) {                      // #4
      TermQuery tq = new TermQuery(                                // #4
          new Term("subject", vector.getTerms()[j]));              // #4
      subjectQuery.add(tq, BooleanClause.Occur.SHOULD);            // #4
    }
```

```
      BooleanQuery likeThisQuery = new BooleanQuery();               // #5
      likeThisQuery.add(authorQuery, BooleanClause.Occur.SHOULD);    // #5
      likeThisQuery.add(subjectQuery, BooleanClause.Occur.SHOULD);   // #5

      // exclude myself
      likeThisQuery.add(new TermQuery(                                        // #6
          new Term("isbn", doc.get("isbn"))), BooleanClause.Occur.MUST_NOT); // #6

      // System.out.println("  Query: " +
      //    likeThisQuery.toString("contents"));
      TopDocs hits = searcher.search(likeThisQuery, 10);
      int size = max;
      if (max > hits.scoreDocs.length) size = hits.scoreDocs.length;

      Document[] docs = new Document[size];
      for (int i = 0; i < size; i++) {
        docs[i] = reader.document(hits.scoreDocs[i].doc);
      }

      return docs;
    }
  }
```

#1 Iterate over every book
#2 Look up books like this
#3 Boosts books by same author
#4 Use terms from "subject" term vectors
#5 Create final query
#6 Exclude current book

#1 As an example, we iterate over every book document in the index and find books like each one.

#2 Here we look up books that are like this one.

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

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

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

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

In #3, we used a different way to get the value of the `author` field. It was indexed as multiple fields, in the manner (shown in more detail in section 8.4) where the original author string is a comma-separated list of author(s) of a book:

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

```
    }
```

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

```
A Modern Art of Education
  -> Mindstorms

Imperial Secrets of Health and Longevity
  None like this
Tao Te Ching 道德經
  None like this

Gšdel, Escher, Bach: an Eternal Golden Braid
  None like this

Mindstorms
  -> A Modern Art of Education

Java Development with Ant
  -> Lucene in Action
  -> JUnit in Action
  -> Extreme Programming Explained

JUnit in Action
  -> Java Development with Ant

Lucene in Action
  -> Java Development with Ant

Extreme Programming Explained
  -> The Pragmatic Programmer
  -> Java Development with Ant

Tapestry in Action
  None like this

The Pragmatic Programmer
  -> Extreme Programming Explained
```

If you'd like to see the actual query used for each, uncomment the output lines toward the end of the `docsLike`.

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

The sandbox contains a useful `Query` implementation, `MoreLikeThisQuery`, doing the same thing as our `BooksLikeThis` class, but more generically. `BooksLikeThis` is clearly hardwired to fields like "subject" and "author", from our books index. But `MoreLikeThis` makes this generic so it works well on

any index. Section 8.12.1 describes this in more detail. Another Sandbox package, the `Highlighter`, described in Section 8.7, also uses of term vectors to find term occurrences for highlighting.

Let's see another example usage of term vectors: automatic category assignment.

5.9.2 What category?

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

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

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

}
```

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

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

```
  protected void setUp() throws Exception {
    categoryMap = new TreeMap();

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

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

```
  private void buildCategoryVectors() throws IOException {
    IndexReader reader = IndexReader.open(TestUtil.getBookIndexDirectory());

    int maxDoc = reader.maxDoc();
```

```
    for (int i = 0; i < maxDoc; i++) {
      if (!reader.isDeleted(i)) {
        Document doc = reader.document(i);
        String category = doc.get("category");

        Map vectorMap = (Map) categoryMap.get(category);
        if (vectorMap == null) {
          vectorMap = new TreeMap();
          categoryMap.put(category, vectorMap);
        }

        TermFreqVector termFreqVector =
            reader.getTermFreqVector(i, "subject");

        addTermFreqToMap(vectorMap, termFreqVector);
      }
    }
  }
```

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

```
  private void addTermFreqToMap(Map vectorMap,
                                TermFreqVector termFreqVector) {
    String[] terms = termFreqVector.getTerms();
    int[] freqs = termFreqVector.getTermFrequencies();

    for (int i = 0; i < terms.length; i++) {
      String term = terms[i];

      if (vectorMap.containsKey(term)) {
        Integer value = (Integer) vectorMap.get(term);
        vectorMap.put(term,
            new Integer(value.intValue() + freqs[i]));
      } else {
        vectorMap.put(term, new Integer(freqs[i]));
      }
    }
  }
```

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

$$\cos\theta = \frac{A \cdot B}{\| A \| \| B \|}$$

Figure 5.8 Formula for computing the angle between two vectors

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

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

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

      while (categoryIterator.hasNext()) {
        String category = (String) categoryIterator.next();
//        System.out.println(category);

        double angle = computeAngle(words, category);
//        System.out.println(" -> angle = " + angle + " (" + Math.toDegrees(angle) +
")");
        if (angle < bestAngle) {
          bestAngle = angle;
          bestCategory = category;
        }
      }

      return bestCategory;
    }
```

We assume that the subject string is in a whitespace-separated form and that each word occurs only once. The angle computation takes these assumptions into account to simplify a part of the computation. Finally, computing the angle between an array of words and a specific category is done in `computeAngle`, shown in listing 5.1.2.

Listing 5.13 Computing term vector angles for a new book against a given category

```
    private double computeAngle(String[] words, String category) {
      // assume words are unique and only occur once

      Map vectorMap = (Map) categoryMap.get(category);

      int dotProduct = 0;
      int sumOfSquares = 0;
      for (int i = 0; i < words.length; i++) {
        String word = words[i];
        int categoryWordFreq = 0;

        if (vectorMap.containsKey(word)) {
          categoryWordFreq =
              ((Integer) vectorMap.get(word)).intValue();
        }

        dotProduct += categoryWordFreq;  // optimized because we assume frequency in
```

```
words is 1
      sumOfSquares += categoryWordFreq * categoryWordFreq;
    }


    double denominator;
    if (sumOfSquares == words.length) {
      // avoid precision issues for special case
      denominator = sumOfSquares; // sqrt x * sqrt x = x
    } else {
      denominator = Math.sqrt(sumOfSquares) *
                    Math.sqrt(words.length);
    }

    double ratio = dotProduct / denominator;

    return Math.acos(ratio);
  }
```

#1 Assume each word has frequency 1
#2 Shortcut to prevent precision issue

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

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

You should be aware that computing term vector angles between two documents or, in this case, between a document and an archetypical category, is computation-intensive. It requires square-root and inverse cosine calculations and may be prohibitive in high-volume indexes. We finish our coverage of term vectors with the `TermVectorMapper` class.

5.9.3 TermVectorMapper

Sometimes, the parallel array structure returned by `IndexReader.getTermFreqVector` may not be convenient for your application. Perhaps instead of sorting by `Term`, you'd like to sort the term vectors according to your own criteria. Or maybe you'd like to only load certain terms of interest. All of these can be done with a recent addition to Lucene, `TermVectorMapper`. This is an abstract base class which, when passed to `IndexReader.getTermFreqVector` methods, separately receives each term, with optional positions and offsets and can choose to store the data in its own manner. Table 5.ZZZ describes the methods that a concrete `TermVectorMapper` implementation (subclass) should implement.

setDocumentNumber	Called once per document to tell you which document is currently being loaded
setExpectations	Called once per field to tell you how many terms

	occur in the field, and whether postions and offsets are stored
map	Called once per term to provide the actual term vectors data
isIgnoringPositions	You should return false only if you need to see the positions data for the term vectors
isIgnoringOffsets	You should return false only if you need to see the offsets data for the term vectors

Lucene includes a few public core implementations of `TermVectorMapper`, described in Table 5.AAA. You can also create your own implementation.

PositionBasedTermVectorMapper	For each field, stores a map from the Integer position to terms and optionally offsets that occurred at that position.
SortedTermVectorMapper	Merges term vectors for all fields into a single SortedSet, sorted according to a Comparator that you specify. One comparator is provided in the Lucene core, TermVectorEntryFreqSortedComparator, which sorts first by frequency of the term and second by the Term itself.
FieldSortedTermVectorMapper	Just like SortedTermVectorMapper, except the fields are not merged together and instead each field stores its sorted terms separately.

When you load stored fields, you likewise have specific control using `FieldSelector`.

5.10 Loading fields with FieldSelector

We've talked about reading a `Document` from the index using an `IndexReader`. You know that the document returned differs from the original document indexed in that it only has those fields you chose to store at indexing time, using `Field.Store.YES`. Under the hood, Lucene writes these fields into the index and then `IndexReader` reads them.

Unfortunately, reading a Document can be fairly time consuming, especially if you need to read many of them per search and if your documents have many stored fields. Often, a document may have one or two large stored fields, holding the actual textual content for the document, and a number of smaller "metadata" fields such as title, category, author, published date, etc. When presenting the search results, you might only need the metadata fields and so loading the very large fields is costly and unnecessary. This is where `FieldSelector` comes in. `FieldSelector`, which is in the

`org.apache.lucene.document` package, allows you to load a specific restricted set of fields for each document. It's an interface with a single simple method:

```
FieldSelectorResult accept(String fieldName);
```

Concrete classes implementing this interface return a `FieldSelectorResult` describing whether the specified fieldname should be loaded, and, how. `FieldSelectorResult`, in turn, contains 7 static values, described in Table 5.XXX.

Table 5.XXX: `FieldSelectorResult` options when loading a stored field

LOAD	Load the field
LAZY_LOAD	Load the field, lazily. The actual contents of the field won't be read until Field.stringValue() or Field.binaryValue() is called.
NO_LOAD	Skip loading the field
LOAD_AND_BREAK	Load this field and don't load any of the remaining fields.
LOAD_FOR_MERGE	Used internally to load a field during segment merging; this skips decompressing compressed fields.
SIZE	Read only the size of the field, then add a binary field with a 4-byte array encoding that size
SIZE_AND_BREAK	Like SIZE, but don't load any of the remaining fields.

When loading stored fields with a `FieldSelector`, `IndexReader` steps through the fields one by one for the document, invoking `FieldSelector` on each field and choosing to load the field or not based on the returned result.

There are several builtin concrete classes implementing `FieldSelector`, described in Table 5.YYY. It's also straightforward to create your own implementation.

Table 5.XXX: Core `FieldSelector` implementations

LoadFirstFieldSelector	Loads only the first field encountered.
MapFieldSelector	You specify the String names of the fields you want to LOAD; all other fields are skipped.
SetBasedFieldSelector	You specify two sets: the first set are fields to LOAD and the second set are fields to LAZY_LOAD.

While `FieldSelector` will save some time during loading fields, just how much time is very application dependent. Much of the cost when loading stored fields is in seeking the file pointers to the places in the index where all fields are stored, so you may find you don't save that much time skipping fields. Test on your application to find the right tradeoff.

5.11 Summary

This chapter has covered some diverse ground, highlighting Lucene's additional built-in search features. We touched on Lucene's field cache, which allows you to load into memory an array of a given field's value for all documents. Sorting is a flexible way to control the ordering of search results.

We described a number of advanced queries. `MultiPhraseQuery` generalizes `PhraseQuery` by allowing more than one term at the same position within a phrase. The `SpanQuery` family leverages term-position information for greater searching precision. `MultiFieldQueryParser` is another `QueryParser` that matches against more than one field. Function queries let you programmatically customize how documents are scored.

Filters constrain document search space, regardless of the query, and you can either create your own `Filter` (described in section 6.4), or use one of Lucene's many built-in ones. We saw how to wrap a `Query` as a `Filter`, and vice versa, as well as how to cache filters for fast reuse.

Lucene includes support for multiple (including parallel) and remote index searching, giving developers a head start on distributed and scalable architectures. The term vectors enable interesting effects, such as "like this" term vector angle calculations. Finally we showed how to fine-tune the loading of term vectors and stored fields by using `TermVectorMapper` and `FieldSelector`.

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

6

Extending search

This chapter covers

- Creating a custom sort
- Using a HitCollector
- Customizing QueryParser
- Use positional payloads
- Testing performance

Just when you thought we were done with searching, here we are again with even more on the topic! Chapter 3 discussed the basics of Lucene's built-in capabilities, and chapter 5 went well beyond the basics into Lucene's more advanced searching features. In those two chapters, we explored only the built-in features. Lucene also has several nifty extension points.

Our first custom extension demonstrates Lucene's custom sorting hooks, allowing us to implement a search that returns results in ascending geographic proximity order from a user's current location. Next, implementing your own `HitCollector` bypasses simple collection of the top N scoring documents; this is effectively an event listener when matches are detected during searches.

`QueryParser` is extensible in several useful ways, such as for controlling date parsing and numeric formatting, as well as for disabling potential performance degrading queries such as wildcard and fuzzy queries or using your own Query subclasses when creating Query instances. Custom filters allow information from outside the index to factor into search constraints, such as factoring some information present only in a relational database into Lucene searches.

And finally, we explore Lucene performance testing using JUnitPerf. The performance-testing example we provide is a meaningful example of testing actually becoming a design tool rather than an after-the-fact assurance test.

6.1 Using a custom sort method

If sorting by score, ID, or field values is insufficient for your needs, Lucene lets you implement a custom sorting mechanism by providing your own subclass of the `FieldcomparatorSource` abstract base class. Custom sorting implementations are most useful in situations when the sort criteria can't be determined during indexing.

An interesting idea for a custom sorting mechanism is to order search results based on geographic distance from a given location.[1] The given location is only known at search time. We've created a simplified demonstration of this concept using the important question, "What Mexican food restaurant is nearest to me?" Figure 6.1 shows a sample of restaurants and their fictitious grid coordinates on a sample 10x10 grid.[2]

[1]Thanks to Tim Jones (the contributor of Lucene's sort capabilities) for the inspiration.

[2]These are real (tasty!) restaurants in Tucson, Arizona, a city Erik used to call home.

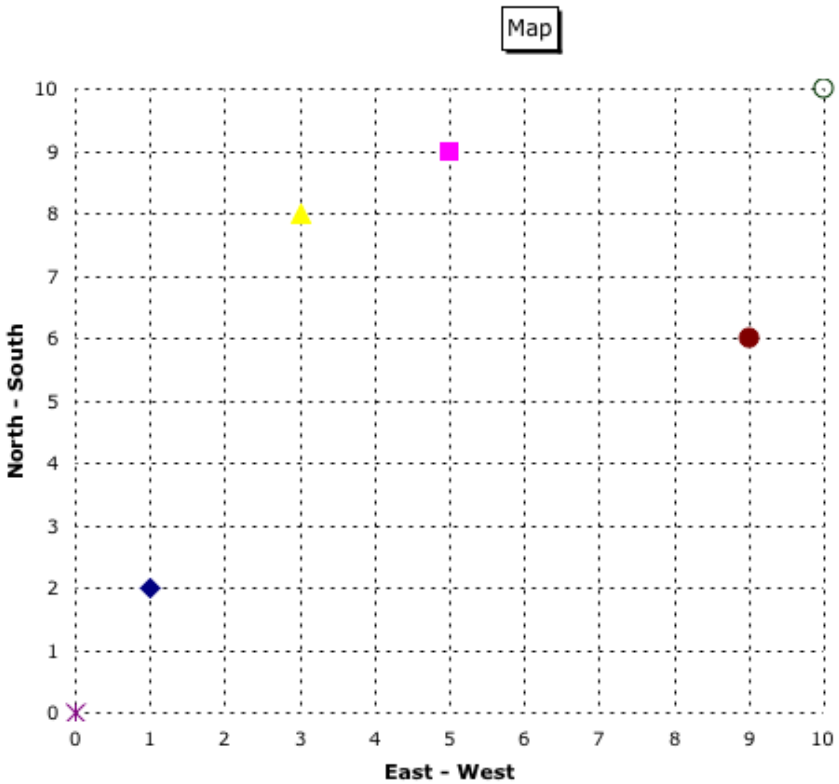


Figure 6.1 Which Mexican restaurant is closest to home (at 0,0) or work (at 10,10)?

The test data is indexed as shown in listing 6.1, with each place given a name, location in X and Y coordinates, and a type. The `type` field allows our data to accommodate other types of businesses and could allow us to filter search results to specific types of places.

Listing 6.1 Indexing geographic data

```
public class DistanceSortingTest extends TestCase {
  private RAMDirectory directory;
  private IndexSearcher searcher;
  private Query query;

  protected void setUp() throws Exception {
    directory = new RAMDirectory();
    IndexWriter writer =
        new IndexWriter(directory, new WhitespaceAnalyzer(),
                        IndexWriter.MaxFieldLength.UNLIMITED);
    addPoint(writer, "El Charro", "restaurant", 1, 2);
    addPoint(writer, "Cafe Poca Cosa", "restaurant", 5, 9);
```

```
    addPoint(writer, "Los Betos", "restaurant", 9, 6);
    addPoint(writer, "Nico's Taco Shop", "restaurant", 3, 8);

    writer.close();

    searcher = new IndexSearcher(directory);

    query = new TermQuery(new Term("type", "restaurant"));
  }

  private void addPoint(IndexWriter writer,
                        String name, String type, int x, int y)
      throws IOException {
    Document doc = new Document();
    doc.add(new Field("name", name, Field.Store.YES, Field.Index.NOT_ANALYZED));
    doc.add(new Field("type", type, Field.Store.YES, Field.Index.NOT_ANALYZED));
    doc.add(new Field("location", x + "," + y, Field.Store.YES,
Field.Index.NOT_ANALYZED));
    writer.addDocument(doc);
  }
}
```

The coordinates are indexed into a single location field as a string *x*, *y*. The location could be encoded in numerous ways, but we opted for the simplest approach for this example. Next we write a test that we use to assert that our sorting implementation works appropriately:

```
  public void testNearestRestaurantToHome() throws Exception {
    Sort sort = new Sort(new SortField("location",
        new DistanceComparatorSource(0, 0)));

    TopDocs hits = searcher.search(query, null, 10, sort);

    assertEquals("closest",
                 "El Charro", searcher.doc(hits.scoreDocs[0].doc).get("name"));
    assertEquals("furthest",
                 "Los Betos", searcher.doc(hits.scoreDocs[3].doc).get("name"));
  }
```

Home is at coordinates (0,0). Our test has shown that the first and last documents in the returned are the ones closest and furthest from home. *Muy bien!* Had we not used a sort, the documents would have been returned in insertion order, since the score of each hit is equivalent for the restaurant-type query. The distance computation, using the basic distance formula, is done under our custom `DistanceComparatorSource`, shown in listing 6.2.

Listing 6.2 DistanceComparatorSource

```
public class DistanceComparatorSource
  extends FieldComparatorSource {                // #1
  private int x;
  private int y;

  public DistanceComparatorSource(int x, int y) { // #2
    this.x = x;
```

```
    this.y = y;
  }

  public org.apache.lucene.search.FieldComparator newComparator(java.lang.String
fieldName, org.apache.lucene.index.IndexReader[] subReaders,
                                                                int numHits, int
sortPos, boolean reversed) throws IOException {       // #3
    return new DistanceScoreDocLookupComparator(fieldName,
                                                numHits);
  }

  private class DistanceScoreDocLookupComparator  // #4
      extends FieldComparator {
    private float[] distances;                    // #5
    private float[] values;                       // #6
    private float bottom;                         // #7
    String fieldName;

    public DistanceScoreDocLookupComparator(
                String fieldName, int numHits) throws IOException {
      values = new float[numHits];
      this.fieldName = fieldName;
    }

    public void setNextReader(IndexReader reader, int docBase, int numSlotsFull) throws
IOException {
      final TermEnum enumerator =
          reader.terms(new Term(fieldName, ""));
      distances = new float[reader.maxDoc()];
      if (distances.length > 0) {
        TermDocs termDocs = reader.termDocs();
        try {
          if (enumerator.term() == null) {
            throw new RuntimeException(
                "no terms in field " + fieldName);
          }
          do {                                        // #8
            Term term = enumerator.term();            // #8
            if (term.field() != fieldName) break;     // #8
            termDocs.seek(enumerator);                // #8
            while (termDocs.next()) {                 // #9
              String[] xy = term.text().split(",");   // #9
              int deltax = Integer.parseInt(xy[0]) - x;  // #9
              int deltay = Integer.parseInt(xy[1]) - y;  // #9

              distances[termDocs.doc()] = (float) Math.sqrt( // #9 #10
                  deltax * deltax + deltay * deltay); // #9 #10
            }                                         // #9
          } while (enumerator.next());                // #9
        } finally {
          termDocs.close();
        }
      }
    }

    public int compare(int slot1, int slot2) {          // #11
```

```
      if (values[slot1] < values[slot2]) return -1;     // #11
      if (values[slot1] > values[slot2]) return 1;      // #11
      return 0;                                         // #11
    }

    public void setBottom(int slot) {                   // #12
      bottom = values[slot];
    }

    public int compareBottom(int doc, float score) {    // #13
      if (bottom < distances[doc]) return -1;           // #13
      if (bottom > distances[doc]) return 1;            // #13
      return 0;                                         // #13
    }

    public void copy(int slot, int doc, float score) {  // #14
      values[slot] = distances[doc];                    // #14
    }

    public Comparable value(int slot) {                 // #15
      return new Float(values[slot]);                   // #15
    }                                                   // #15

    public int sortType() {
      return SortField.CUSTOM;
    }
  }

  public String toString() {
    return "Distance from ("+x+","+y+")";
  }
}
```

#1 Implement FieldComparatorSource
#2 Give constructor base location
#3 newComparator
#4 FieldComparator
#5 Array of distances
#6 Distances for top N best documents so far
#7 Worst distance in the top N best documents so far
#8 Iterate over terms
#9 Iterate over documents containing current term
#10 Compute and store distance
#11 compare two docs in the top N
#12 Record worst scoring doc in the top N
#13 Compare new doc to worst scoring doc
#14 Insert new doc into top N
#15 Extract value from top N

#1 First we implement `FieldComparatorSource`.

#2 The constructor is handed the base location from which results are sorted by distance.

#3 This is `FieldComparatorSource`'s only method.

#4 This is our custom `FieldComparator` implementation.

#5 Here we create an array of distances.

#6 Here we store the top N distances seen, so far.

#7 Here we store the worst distance in the top N queue, so far.

#8 We iterate over all the terms in the specified field.

#9 Next, we iterate over every document containing the current term.

#10 We compute and store the distance.

#11 The `compare` method is used by the high-level searching API when the actual distance isn't needed.

#12 Record which slot is the bottom (worst) in the queue.

#13 Compares a document to the bottom of the queue.

#14 Installs a document into the queue.

#15 The `value` method is used by the lower-level searching API when the distance value is desired.

The sorting infrastructure within Lucene interacts with the `FieldComparator` API in order to sort matching documents. For performance reasons, this API is more complex than one would otherwise expect. In particular, the comparator is made aware of the size of the top N queue (passed as the `numHits` argument to `newComparator`) being maintained within Lucene. In addition, the comparator is notified every time a new segment is searched (with the `setNextReader` method).

While searching, when a document is competitive it is inserted into the queue at a given slot, determined by Lucene. Your comparator is asked to compare hits within the queue (compare), set the bottom (worst scoring entry) slot in the queue (`setBottom`), compare a hit to the bottom of the queue (`compareBottom`), and copy a new hit into the queue (`copy`).

With each reader (segment) that's visited, `DistanceScoreDocLookupComparator` implementation makes space to store a `float` for every document and computes the distance from the base location to each document containing the specified sort field (`location` in our example). In a homogeneous index where all documents have the same fields, this would involve computing the distance for every document. Given these steps, it's imperative that you're aware of the resources utilized to sort; this topic is discussed in more detail in section 5.1.9 as well as in Lucene's Javadocs.

Sorting by runtime information such as a user's location is an incredibly powerful feature. At this point, though, we still have a missing piece: What is the distance from each of the restaurants to our current location? When using the `TopDocs`-returning `search` methods, we can't get to the distance computed. However, a lower-level API lets us access the values used for sorting.

6.1.1 Accessing values used in custom sorting

Beyond the `IndexSearcher.search` methods you've seen thus far, some lower-level methods are used internally to Lucene and aren't that useful to the outside. The exception enters with accessing custom sorting values, like the distance to each of the restaurants computed by our custom comparator source. The signature of the method we use, on `IndexSearcher`, is:

```
public TopFieldDocs search(Query query, Filter filter,
                           final int nDocs, Sort sort)
```

`TopFieldDocs` contains the total number of `ScoreDocs`, the `SortField` array used for sorting, and an array of `FieldDoc` (subclass of `ScoreDoc`) objects. A `FieldDoc` encapsulates the computed raw score, document ID, and an array of `Comparables` with the value used for each `SortField`. `TopFieldDocs` and `FieldDoc` are specific to searching with a `Sort`. Rather than concerning ourselves with the details of the API, which you can get from Lucene's Javadocs or the source code, let's see how to really use it.

Listing 6.3's test case demonstrates the use of `TopFieldDocs` and `FieldDoc` to retrieve the distance computed during sorting, this time sorting from Work at location (10,10).

Listing 6.3 Accessing custom sorting values for search results

```
public void testNeareastRestaurantToWork() throws Exception {
  Sort sort = new Sort(new SortField("location",
      new DistanceComparatorSource(10, 10)));

  TopFieldDocs docs = searcher.search(query, null, 3, sort);  // #1

  assertEquals(4, docs.totalHits);              // #2
  assertEquals(3, docs.scoreDocs.length);       // #3

  FieldDoc fieldDoc = (FieldDoc) docs.scoreDocs[0];     // #4

  assertEquals("(10,10) -> (9,6) = sqrt(17)",
      new Float(Math.sqrt(17)),
      fieldDoc.fields[0]);                          // #5

  Document document = searcher.doc(fieldDoc.doc);  // #6
  assertEquals("Los Betos", document.get("name"));

  //dumpDocs(sort, docs);
}
```

#1 Specify maximum hits returned
#2 Total number of hits
#3 Return total number of documents
#4 Get sorting values

#5 Give value of first computation
#6 Get Document

#1 This lower-level API requires that we specify the maximum number of hits returned.

#2 The total number of hits is still provided because all hits need to be determined to find the three best ones.

#3 The total number of documents (up to the maximum specified) are returned.

#4 `docs.scoreDocs(0)` returns a `ScoreDoc` and must be cast to `FieldDoc` to get sorting values.

#5 The value of the first (and only, in this example) `SortField` computation is available in the first `fields` slot.

#6 Getting the actual `Document` requires another call.

6.2 Developing a custom HitCollector

In most applications with full-text search, users are looking for the most relevant documents from a query. The most common usage pattern is such that only the first few highest-scoring hits are visited. In some scenarios, though, users want to be shown all documents (by ID) that match a query without needing to access the contents of the document; search filters, discussed in section 5.5, may use `HitCollectors` efficiently in this manner. Another possible use, which we demonstrate in this section, is accessing every document's contents from a search in a direct fashion.

Using a `TopDocs`-returning `search` method will work to collect all documents if you traverse all the results and process them manually, although you're incurring the cost of sorting by relevence. Using a custom `HitCollector` class avoids this cost. We show you two simple custom `HitCollectors`, `BookLinkCollector` and `AllDocCollector`.

6.2.1 About BookLinkCollector

We've developed a custom `HitCollector`, called `BookLinkCollector`, which builds a map of all unique URLs and the corresponding book titles matching a query. The `collect(int, float)` method must be implemented from the `HitCollector` interface. `BookLinkCollector` is shown in listing 6.4.

Listing 6.4 Custom HitCollector: collects all book hyperlinks

```
public class BookLinkCollector extends HitCollector {
  private IndexSearcher searcher;
  private HashMap documents = new HashMap();

  public BookLinkCollector(IndexSearcher searcher) {
    this.searcher = searcher;
  }

  public void collect(int id, float score) {
    try {
      Document doc = searcher.doc(id);          // #1
```

```
        documents.put(doc.get("url"), doc.get("title"));
        System.out.println(doc.get("title") + ":" + score);
      } catch (IOException e) {
        // ignore
      }
    }

    public Map getLinks() {
      return Collections.unmodifiableMap(documents);
    }
  }
```

#1 Access documents by ID

Our collector collects all book titles (by URL) that match the query.

6.2.2 Using BookLinkCollector

Using a `HitCollector` requires the use of `IndexSearcher`'s `search` method variant as shown here:

```
  public void testCollecting() throws Exception {
    TermQuery query = new TermQuery(new Term("contents", "junit"));
    IndexSearcher searcher = new IndexSearcher(TestUtil.getBookIndexDirectory());

    BookLinkCollector collector = new BookLinkCollector(searcher);
    searcher.search(query, collector);

    Map linkMap = collector.getLinks();
    assertEquals("Java Development with Ant",
                 linkMap.get("http://www.manning.com/antbook"));;

    TopDocs hits = searcher.search(query, 10);
    TestUtil.dumpHits(searcher, hits);

    searcher.close();
  }
```

Calling `IndexSearcher.doc(n)` or `IndexReader.document(n)` in the `collect` method can slow searches by an order of magnitude, so be sure your situation requires access to all the documents. In our example, we're sure we want the title and URL of each document matched. Stopping a `HitCollector` midstream is a bit of a hack, though, because there is no built-in mechanism to allow for this. To stop a `HitCollector`, you must throw a runtime exception and be prepared to catch it where you invoke `search`.

Here's the `TestUtil.dumpHits` method:

```
  public static void dumpHits(IndexSearcher searcher, TopDocs hits) throws IOException
{
    if (hits.totalHits == 0) {
      System.out.println("No hits");
    }

    for (int i=0; i < hits.totalHits; i++) {
      Document doc = searcher.doc(hits.scoreDocs[i].doc);
```

```
        System.out.println(hits.scoreDocs[i].score + ":" + doc.get("title"));
      }
    }
```

It simply iterates through the results, printing the score and title of each.

Filters (see section 5.5), such as `QueryFilter`, can use a `HitCollector` to set bits on a `DocIdSet` when documents are matched, and don't access the underlying documents directly; this is a highly efficient use of `HitCollector`. Let's look at a very simple custom `HitCollector` next.

6.2.3 AllDocCollector

Sometimes you'd like to simple record every single matching document for a search, and you know the number of matches will not be very large. Here's a simple class, `AllDocCollector`, to do just that:

```
public class AllDocCollector extends HitCollector {
  List<ScoreDoc> docs = new ArrayList<ScoreDoc>();
  public void collect(int doc, float score) {
    if (score > 0.0f) {
      docs.add(new ScoreDoc(doc, score));
    }
  }

  public void reset() {
    docs.clear();
  }

  public List<ScoreDoc> getHits() {
    return docs;
  }
}
```

You simply instantiate it, pass it to the search, and then use the `getHits()` method to retrieve all hits. Next we discuss useful ways to extend `QueryParser`.

6.3 Extending QueryParser

In section 3.5, we introduced `QueryParser` and showed that it has a few settings to control its behavior, such as setting the locale for date parsing and controlling the default phrase slop. `QueryParser` is also extensible, allowing subclassing to override parts of the query-creation process. In this section, we demonstrate subclassing `QueryParser` to disallow inefficient wildcard and fuzzy queries, custom date-range handling, and morphing phrase queries into `SpanNearQuery`s instead of `PhraseQuery`s.

6.3.1 Customizing QueryParser's behavior

Although `QueryParser` has some quirks, such as the interactions with an analyzer, it does have extensibility points that allow for customization. Table 6.1 details the methods designed for overriding and why you may want to do so.

Table 6.1 `QueryParser`'s extensibility points

Method	Why override?
getFieldQuery(String field, Analyzer analyzer, String queryText) or getFieldQuery(String field, Analyzer analyzer, String queryText, int slop)	These methods are responsible for the construction of either a `TermQuery` or a `PhraseQuery`. If special analysis is needed, or a unique type of query is desired, override this method. For example, a `SpanNearQuery` can replace `PhraseQuery` to force ordered phrase matches.
getFuzzyQuery(String field, String termStr, float minSimilarity)	Fuzzy queries can adversely affect performance. Override and throw a `ParseException` to disallow fuzzy queries.
getPrefixQuery(String field, String termStr)	This method is used to construct a query when the term ends with an asterisk. The term string handed to this method doesn't include the trailing asterisk and isn't analyzed. Override this method to perform any desired analysis.
getRangeQuery(String field, String start, String end, boolean inclusive)	Default range-query behavior has several noted quirks (see section 3.5.5). Overriding could: Lowercase the start and end terms Use a different date format Handle number ranges by padding to match how numbers were indexed

getBooleanQuery(List clauses) or getBooleanQuery(List clauses, boolean disableCoord)	Constructs a BooleanQuery given the clauses.
getWildcardQuery(String field, String termStr)	Wildcard queries can adversely affect performance, so overridden methods could throw a `ParseException` to disallow them. Alternatively, since the term string isn't analyzed, special handling may be desired.

All of the methods listed return a `Query`, making it possible to construct something other than the current subclass type used by the original implementations of these methods. Also, each of these methods may throw a `ParseException` allowing for error handling.

`QueryParser` also has extensibility points for instantiating each query type. These differ from the points listed in table 6.1 in that they simply create the requested query type and return it. Overriding these is useful if you simply want to change which Query class is used for each type of query without altering the logic of what query is constructed. These methods are `newBooleanQuery`, `newTermQuery`, `newPhraseQuery`, `newMultiPhraseQuery`, `newPrefixQuery`, `newFuzzyQeury`, `newRangeQuery`, `newMatchAllDocsQuery` and `newWildcardQuery`. For example, if whenever a `TermQuery` is created by `QueryParser` you'd like to instantiate your own subclass of `TermQuery`, simply override `newTermQuery`.

6.3.2 Prohibiting fuzzy and wildcard queries

The subclass in listing 6.5 demonstrates a custom query parser subclass that disables fuzzy and wildcard queries by taking advantage of the `ParseException` option.

Listing 6.5 Disallowing wildcard and fuzzy queries

```
public class CustomQueryParser extends QueryParser {
  public CustomQueryParser(String field, Analyzer analyzer) {
    super(field, analyzer);
  }

  protected final Query getWildcardQuery(String field, String termStr) throws
ParseException {
    throw new ParseException("Wildcard not allowed");
  }
```

```
    protected Query getFuzzyQuery(String field, String term, float minSimilarity) throws
  ParseException {
      throw new ParseException("Fuzzy queries not allowed");
    }
```

To use this custom parser and prevent users from executing wildcard and fuzzy queries, construct an instance of `CustomQueryParser` and use it exactly as you would `QueryParser`, as shown in the following code:

```
  public void testCustomQueryParser() {
    CustomQueryParser parser =
        new CustomQueryParser("field", analyzer);
    try {
      parser.parse("a?t");
      fail("Wildcard queries should not be allowed");
    } catch (ParseException expected) {
      // expected
      assertTrue(true);
    }

    try {
      parser.parse("xunit~");
      fail("Fuzzy queries should not be allowed");
    } catch (ParseException expected) {
      // expected
      assertTrue(true);
    }
  }
```

With this implementation, both of these expensive query types are forbidden, giving you some peace of mind in terms of performance and errors that may arise from these queries expanding into too many terms. Our next example of `QueryParser` extension shows how to tweak how `RangeQuery` is created.

6.3.3 Handling numeric field-range queries

Lucene is all about dealing with text. You've seen in several places how dates can be handled, which amounts to their being converted into a text representation that can be ordered alphabetically. Handling numbers is basically the same, except implementing a conversion to a text format is left up to you.

In this section, our example scenario indexes an integer `id` field so that range queries can be performed. If we indexed `toString` representations of the integers 1 through 10, the order in the index would be 1, 10, 2, 3, 4, 5, 6, 7, 8, 9—not the intended order at all. However, if we pad the numbers with leading zeros so that all numbers have the same width, the order is correct: 01, 02, 03, and so on. You'll have to decide on the maximum width your numbers need; we chose 10 digits and implemented the following `pad(int)` utility method:[3]

[3]Lucene stores term information with prefix compression so that no penalty is paid for large shared prefixes like this zero padding.

```
public class NumberUtils {
  private static final DecimalFormat formatter =
      new DecimalFormat("0000000000");

  public static String pad(int n) {
    return formatter.format(n);
  }
}
```

The numbers need to be padded during indexing. This is done in our test `setUp()` method on the `id` keyword field:

```
public class AdvancedQueryParserTest extends TestCase {
  private Analyzer analyzer;
  private RAMDirectory directory;

  protected void setUp() throws Exception {
    analyzer = new WhitespaceAnalyzer();

    directory = new RAMDirectory();
    IndexWriter writer = new IndexWriter(directory, analyzer,
                                         IndexWriter.MaxFieldLength.UNLIMITED);
    for (int i = 1; i <= 500; i++) {
      Document doc = new Document();
      doc.add(new Field("id", NumberUtils.pad(i), Field.Store.YES,
Field.Index.NOT_ANALYZED));
      writer.addDocument(doc);
    }
    writer.close();
  }
}
```

With this index-time padding, we're only halfway there. A query expression for IDs 37 through 346 phrased as `id:[37 TO 346]` won't work as expected with the default `RangeQuery` created by `QueryParser`. The values are taken literally and aren't padded as they were when indexed. Fortunately we can fix this problem in our `CustomQueryParser` by overriding the `getRangeQuery()` method:

```
    protected Query getRangeQuery(String field,
                                  String part1,
                                  String part2,
                                  boolean inclusive) throws ParseException {
    if ("id".equals(field)) {
      try {
        int num1 = Integer.parseInt(part1);
        int num2 = Integer.parseInt(part2);
        return new RangeQuery(field,
                              NumberUtils.pad(num1),
                              NumberUtils.pad(num2),
                              inclusive, inclusive);
      } catch (NumberFormatException e) {
        throw new ParseException(e.getMessage());
```

```
      }
    }

    return super.getRangeQuery(field, part1, part2,
                               inclusive);
  }
```

This implementation is specific to our `id` field; you may want to generalize it for more fields. If the field isn't `id`, it delegates to the default behavior. The `id` field is treated specially, and the pad function is called just as with indexing. The following test case shows that the range query worked as expected, and you can see the results of the padding using `Query`'s `toString(String)` method:

```
  public void testIdRangeQuery() throws Exception {
    CustomQueryParser parser =
        new CustomQueryParser("field", analyzer);

    Query query = parser.parse("id:[37 TO 346]");

    assertEquals("padded", "id:[0000000037 TO 0000000346]",
                           query.toString("field"));

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

    assertEquals(310, hits.totalHits);
  }
```

Our test shows that we've succeeded in allowing sensible-looking user-entered range queries to work as expected. Our final QueryParser customization shows how to replace the default `PhraseQuery` with `SpanNearQuery`.

6.3.4 Allowing ordered phrase queries

When `QueryParser` parses a single term, or terms within double quotes, it delegates the construction of the `Query` to a `getFieldQuery` method. Parsing an unquoted term calls the `getFieldQuery` method without the slop signature (slop makes sense only on multiterm phrase query); parsing a quoted phrase calls the `getFieldQuery` signature with the slop factor, which internally delegates to the nonslop signature to build the query and then sets the slop appropriately. The `Query` returned is either a `TermQuery` or a `PhraseQuery`, by default, depending on whether one or more tokens are returned from the analyzer.[4] Given enough slop, `PhraseQuery` will match terms out of order in the original text. There is no way to force a `PhraseQuery` to match in order (except with slop of 0 or 1). However, `SpanNearQuery` does allow in-order matching. A straightforward override of `getFieldQuery` allows us to replace a `PhraseQuery` with an ordered `SpanNearQuery`:

```
  protected Query getFieldQuery(String field, String queryText, int slop) throws
ParseException {
```

[4] A `PhraseQuery` could be created from a single term if the analyzer created more than one token for it.

```
    // let QueryParser's implementation do the analysis
    Query orig = super.getFieldQuery(field, queryText, slop);  // #1

    if (!(orig instanceof PhraseQuery)) {         // #2
      return orig;                                // #2
    }                                             // #2

    PhraseQuery pq = (PhraseQuery) orig;
    Term[] terms = pq.getTerms();                 // #3
    SpanTermQuery[] clauses = new SpanTermQuery[terms.length];
    for (int i = 0; i < terms.length; i++) {
      clauses[i] = new SpanTermQuery(terms[i]);
    }

    SpanNearQuery query = new SpanNearQuery(      // #4
                    clauses, slop, true);         // #4

    return query;
  }
```

#1 Delegate to QueryParser's implementation
#2 Only override PhraseQuery
#3 Pull all terms
#4 Create SpanNearQuery

#1 We delegate to `QueryParser`'s implementation for analysis and determination of query type.

#2 Here we override `PhraseQuery` and return anything else right away.

#3 We pull all terms from the original `PhraseQuery`.

#4 Finally, we create a `SpanNearQuery` with all the terms from the original `PhraseQuery`.

Our test case shows that our custom `getFieldQuery` is effective in creating a `SpanNearQuery`:

```
  public void testPhraseQuery() throws Exception {
    CustomQueryParser parser =
        new CustomQueryParser("field", analyzer);

    Query query = parser.parse("singleTerm");
    assertTrue("TermQuery", query instanceof TermQuery);

    query = parser.parse("\"a phrase\"");
    assertTrue("SpanNearQuery", query instanceof SpanNearQuery);
  }
```

Another possible enhancement would add a toggle switch to the custom query parser, allowing the in-order flag to be controlled by the user of the API.

6.4 Using a custom filter

If all the information needed to perform filtering is in the index, there is no need to write your own filter because the `QueryFilter` can handle it. However, there are good reasons to factor external information into a custom filter. Using our book example data and pretending we're running an online bookstore, we want users to be able to search within our special hot deals of the day. One option is to store the specials flag in an index field. However, the specials change frequently. Rather than reindex documents when specials change, we opt to keep the specials flagged in our (hypothetical) relational database.

To do this right, we want it to be test-driven and demonstrate how our `SpecialsFilter` can pull information from an external source without even having an external source! Using an interface, a mock object, and good ol' JUnit, here we go. First, here's the interface for retrieving specials:

```
public interface SpecialsAccessor {
  String[] isbns();
}
```

Since we won't have an enormous amount of specials at one time, returning all the ISBNs of the books on special will suffice.

Now that we have a retrieval interface, we can write our custom filter, `SpecialsFilter`. Filters extend from the `org.apache.lucene.search.Filter` class and must implement the `getDocIdSet(IndexReader reader)` method, returning a `DocIdSet`. Bit positions match the document numbers. Enabled bits mean the document for that position is available to be searched against the query, and unset bits mean the document won't be considered in the search. Figure 6.2 illustrates an example `SpecialsFilter` that sets bits for books on special (see listing 6.6).

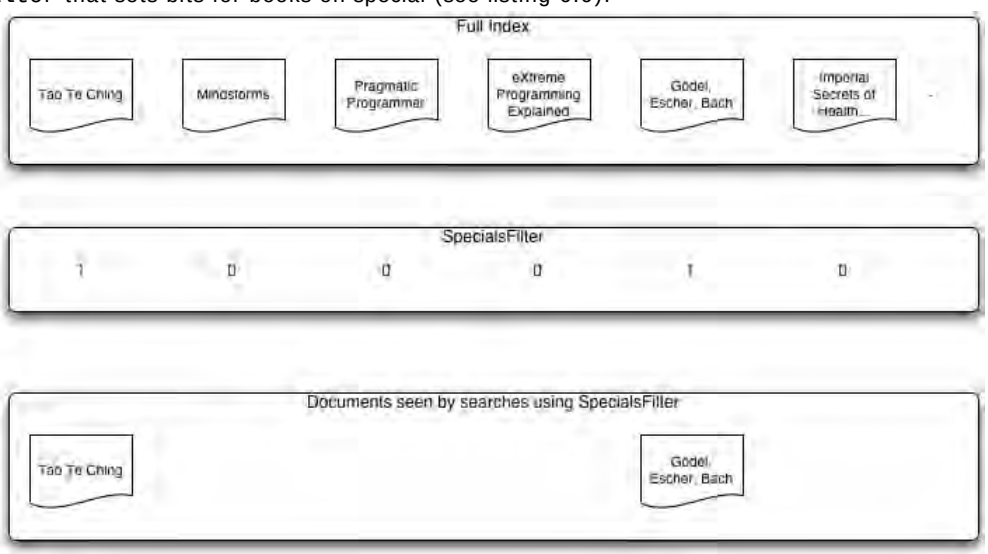


Figure 6.2 Filtering for books on special

Listing 6.6 SpecialFilter: a custom filter that retrieves information from an external source

```
public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
  OpenBitSet bits = new OpenBitSet(reader.maxDoc());

  String[] isbns = accessor.isbns();                  // #1

  int[] docs = new int[1];
  int[] freqs = new int[1];

  for (int i = 0; i < isbns.length; i++) {
    String isbn = isbns[i];
    if (isbn != null) {
      TermDocs termDocs =
        reader.termDocs(new Term("isbn", isbn));      // #2
      int count = termDocs.read(docs, freqs);
      if (count == 1) {                               // #3
        bits.set(docs[0]);                            // #3
      }                                               // #3
    }
  }

  return bits;
}
```

#1 Fetch ISBNs
#2 Jump to term
#3 Set corresponding bit

#1 Here, we fetch the ISBNs of the specials we want to enable for searching.

#2 `isbn` is indexed as a `NOT_ANALYZED` field and is unique, so we use `IndexReader` to jump directly to the term.

#3 With the matching document found, we set its corresponding bit.

To test that our filter is working, we created a simple `TestSpecialsAccessor` to return a specified set of ISBNs, giving our test case control over the set of specials:

```
public class TestSpecialsAccessor implements SpecialsAccessor {
  private String[] isbns;

  public TestSpecialsAccessor(String[] isbns) {
    this.isbns = isbns;
  }

  public String[] isbns() {
    return isbns;
  }
}
```

Here's how we test our `SpecialsFilter`, using the same `setUp()` that the other filter tests used:

```
public void testCustomFilter() throws Exception {
  String[] isbns = new String[] {"0060812451", "0465026567"};

  SpecialsAccessor accessor = new TestSpecialsAccessor(isbns);
  Filter filter = new SpecialsFilter(accessor);
  TopDocs hits = searcher.search(allBooks, filter, 10);
  assertEquals("the specials", isbns.length, hits.totalHits);
}
```

We use a generic query that is broad enough to retrieve all the books, making assertions easier to craft; but because our filter trimmed the search space, only the specials are returned. With this infrastructure in place, implementing a `SpecialsAccessor` to retrieve a list of ISBNs from a database should be easy; doing so is left as an exercise for the savvy reader.

Note that we made an important implementation decision *not* to cache the `DocIdSet` in `SpecialsFilter`. Decorating `SpecialsFilter` with a `CachingWrapperFilter` frees us from that aspect.

6.4.1 Using a filtered query

To add to the *filter* terminology overload, one final option is `FilteredQuery`.[5] `FilteredQuery` inverts the situation that searching with a `Filter` presents. Using a `Filter` an `IndexSearcher`'s `search` method applies a single filter during querying. Using the `FilteredQuery`, though, you can apply a `Filter` to a particular query clause of a `BooleanQuery`.

Let's take the `SpecialsFilter` as an example again. This time, we want a more sophisticated query: books in an education category on special, or books on Logo.[6] We couldn't accomplish this with a direct query using the techniques shown thus far, but `FilteredQuery` makes this possible. Had our search been only for books in the education category on special, we could have used the technique shown in the previous code snippet, instead.

Our test case, in listing 6.7, demonstrates the described query using a `BooleanQuery` with a nested `TermQuery` and `FilteredQuery`.

Listing 6.7 Using a `FilteredQuery`

```
public void testFilteredQuery() throws Exception {
  String[] isbns = new String[] {"0854402624"};  // Steiner     // #1

  SpecialsAccessor accessor = new TestSpecialsAccessor(isbns);
  Filter filter = new SpecialsFilter(accessor);
```

[5] We're sorry! We know that `Filter`, `QueryFilter`, `FilteredQuery`, and the completely unrelated `TokenFilter` names can be confusing.

[6] Erik began his programming adventures with Logo on an Apple][e. Times haven't changed much; now he tinkers with StarLogo on a PowerBook.

```
    WildcardQuery educationBooks =                                  // #2
      new WildcardQuery(new Term("category", "*education*"));       // #2
    FilteredQuery edBooksOnSpecial =                                // #2
        new FilteredQuery(educationBooks, filter);                  // #2

    TermQuery logoBooks =                                           // #3
        new TermQuery(new Term("subject", "logo"));                 // #3

    BooleanQuery logoOrEdBooks = new BooleanQuery();                    // #4
    logoOrEdBooks.add(logoBooks, BooleanClause.Occur.SHOULD);           // #4
    logoOrEdBooks.add(edBooksOnSpecial, BooleanClause.Occur.SHOULD);    // #4

    TopDocs hits = searcher.search(logoOrEdBooks, 10);
    System.out.println(logoOrEdBooks.toString());
    assertEquals("Papert and Steiner", 2, hits.totalHits);
  }
```

#1 Rudolf Steiner's book
#2 All education books on special
#3 All books with "logo" in subject
#4 Combine queries

#1 This is the ISBN number for Rudolf Steiner's *A Modern Art of Education*.

#2 We construct a query for education books on special, which only includes Steiner's book in this example.

#3 We construct a query for all books with *logo* in the subject, which only includes *Mindstorms* in our sample data.

#4 The two queries are combined in an OR fashion.

The `getDocIdSet()` method of the nested `Filter` is called each time a `FilteredQuery` is used in a search, so we recommend that you use a caching filter if the query is to be used repeatedly and the results of a filter don't change.

We'll switch to an advanced means of customization and a relatively new feature in lucene, *payloads*.

6.5 Payloads

Payloads are an advanced feature in Lucene that enables an application to store an arbitrary byte array for every occurrence of a term during indexing. This byte array is entirely opaque to Lucene: it's simply stored at each `Term` position, during indexing, and then can be retrieved during searching. Otherwise the core Lucene functionality doesn't do anything with the payload or make any assumptions about its contents. This means you can store arbitrary encoded data that is important to your application, and then use it during searching, either to decide which documents are included in the search results, or to alter how matched documents are scored, or, both.

Let's see how to use payloads for position-specific boosting, whereby matched documents can be boosted when the specific terms that matched were "important". Imagine we are indexing mixed documents, where some of them are bulletins (weather warnings) while others are more ordinary

documents. You'd like a search for "warning" to give extra boost when it occurs in a bulletin document. Another example is boosting terms that were bolded or italicized in the original text, or contained within a title or header tag for HTML documents. While you could use field boosting to achieve this, that'd require you to separate out all the important terms into entirely separate fields, which is often not feasible or desired. Payloads lets you solve this by boosting on a term by term basis within a single field.

The first step is to create an analyzer that attaches payloads to certain tokens. The `TokenStream` for such an analyzer should define the `PayloadAttribute`, and then create a `Payload` instance when appropriate and set the payload using `PayloadAttribute.setPayload` inside the `incrementToken` method. Payloads are created with the following constructors:

```
public Payload(byte[] data)
Payload(byte[] data, int offset, int length)
```

It's perfectly fine to set a null payload for some tokens. In fact, for application where there is a common "default value", it's best to represent that default value as a null payload, instead of a payload with the default value encoded into it, to save space in your index. Lucene simply records that there is no payload available at that position.

The sandbox, under contrib/analyzers, includes several useful `TokenFilters`, as shown in Table 6.2. These classes simply translate certain existing attributes of a `Token`, such as type and start/end offset, into a corresponding payload.

Table 6.2 `TokenFilter` in contrib/analyzers that encode certain `TokenAttributes` as payloads

`NumericPayloadTokenFilter`	Encodes a float payload for those tokens matching the specified token type.
`TypeAsPayloadTokenFilter`	Encodes the token's type as a payload on every token.
`TokenOffsetPayloadTokenFilter`	Encodes the start and end offset of each token into its payload
`PayloadHelper`	Static methods to encode and decode ints and floats into byte array payloads.

Quite often, as is the case in our example, the logic you need to create a payload requires more customization. In our case, we want to create a payload for those term occurrences that should be boosted, containing the boost score, and set no payload for all other terms. Fortunately, it's straightforward to create your own `TokenFilter` to implement such logic. Listing 6.8 shows our own `BulletinPayloadsAnalyzer`.

Our logic is quite simple: if the document is a bulletin, which is simplistically determined by checking whether the contents start with the prefix "Bulletin:", then we attach a payload that encodes a float boost to any occurrence of the term "warning". We use `PayloadHelper` to encode the float into a byte array.

Listing 6.8 Custom filter and analyzer to attach payloads to the token "warning" inside bulletin documents

```
public class BulletinPayloadsFilter extends TokenFilter {

  private TermAttribute termAtt;
  private PayloadAttribute payloadAttr;
  private boolean isBulletin;
  private Payload boostPayload;

  BulletinPayloadsFilter(TokenStream in, float warningBoost) {
    super(in);
    payloadAttr = (PayloadAttribute) addAttribute(PayloadAttribute.class);
    termAtt = (TermAttribute) addAttribute(TermAttribute.class);
    boostPayload = new Payload(PayloadHelper.encodeFloat(warningBoost));
  }

  void setIsBulletin(boolean v) {
    isBulletin = v;
  }

  public final boolean incrementToken() throws IOException {
    if (input.incrementToken()) {
      if (isBulletin && termAtt.term().equals("warning"))           // #1
        payloadAttr.setPayload(boostPayload);                       // #1
      else
        payloadAttr.clear();                                        // #2
      return true;
    } else
      return false;
  }
}

public class BulletinPayloadsAnalyzer extends Analyzer {
  private boolean isBulletin;
  private float boost;

  BulletinPayloadsAnalyzer(float boost) {
    this.boost = boost;
  }

  void setIsBulletin(boolean v) {
    isBulletin = v;
  }

  public TokenStream tokenStream(String fieldName, Reader reader) {
    BulletinPayloadsFilter stream = new BulletinPayloadsFilter(new
StandardAnalyzer().tokenStream(fieldName, reader), boost);
    stream.setIsBulletin(isBulletin);
    return stream;
  }
}
```

#1 If document is a bulletin, and term is warning, record payload boost
#2 Clear payload to get no boost

Using this analyzer, we can get our payloads into the index. But how do we use the payloads during searching to boost scores for certain matches? Fortunately, Lucene provides a simple builtin query `BoostingTermQuery`, in the package `org.apache.lucene.search.payloads`, for precisely this purpose. This query is just like `TermQuery`, in that it matches all documents containing the specified term, except when scoring each occurrence of the term in a document it also invokes the `Similarity.scorePayload`, giving you a chance to define how a given payload should boost the score of the match. Let's create our own `Similarity` class, subclassing `DefaultSimilarity`, that overrides `scorePayload`:

```
public class BoostingSimilarity extends DefaultSimilarity {
  public float scorePayload(String fieldName, byte[] payload, int offset, int length) {
    if (payload != null)
      return PayloadHelper.decodeFloat(payload, offset);
    else
      return 1.0F;
  }
}
```

We again use `PayloadHelper`, this time to decode the byte array back into a float. `BoostingTermQuery` takes the average of all floats returned by `Similarity.scorePayload` for each document and then multiplies the normal score that a TermQuery returns by that average.

Listing 6.9 Test case showing how to use payloads to boost certain term occurrences

```
public class PayloadsTest extends TestCase {

  Directory dir;
  IndexWriter writer;
  BulletinPayloadsAnalyzer analyzer;

  protected void setUp() throws Exception {
    super.setUp();
    TokenStream.setUseNewAPIDefault(true);
    dir = new RAMDirectory();
    analyzer = new BulletinPayloadsAnalyzer(5.0F);
    writer = new IndexWriter(dir, analyzer, IndexWriter.MaxFieldLength.UNLIMITED);
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    writer.close();
  }

  void addDoc(String title, String contents) throws IOException {
    Document doc = new Document();
    doc.add(new Field("title",
                      title,
                      Field.Store.YES,
                      Field.Index.NO));
    doc.add(new Field("contents",
```

```
                    contents,
                    Field.Store.NO,
                    Field.Index.ANALYZED));
    analyzer.setIsBulletin(contents.startsWith("Bulletin:"));
    writer.addDocument(doc);
  }

  public void testBoostingTermQuery() throws Throwable {
    addDoc("Hurricane warning", "Bulletin: A hurricane warning was issued at 6 AM for
the outer great banks");
    addDoc("Warning label maker", "The warning label maker is a delightful toy for your
precocious six year old's warning needs");
    addDoc("Tornado warning", "Bulletin: There is a tornado warning for Worcester
county until 6 PM today");

    writer.commit();

    IndexSearcher searcher = new IndexSearcher(dir);
    searcher.setSimilarity(new BoostingSimilarity());

    Term warning = new Term("contents", "warning");

    Query query1 = new TermQuery(warning);
    System.out.println("\nTermQuery results:");
    TestUtil.dumpHits(searcher,
                    searcher.search(query1, 10));

    Query query2 = new BoostingTermQuery(warning);
    System.out.println("\nBoostingTermQuery results:");
    TestUtil.dumpHits(searcher,
                    searcher.search(query2, 10));
  }
}
```

Now that we have all the pieces, let's pull it together into a test case as shown in Listing 6.9. We index three documents, two of which are bulletins. Next, we do two searches, printing the results. The first search is a normal `TermQuery`, which should return the 2nd document as the top result, because it contains two occurrences of the term "warning". The second query is a `BoostingTermQuery` that boosts the occurrence of warning in each bulletin by 5.0 boost (passed as the single argument to `BulletinPayloadsAnalyzer`). Running this test produces this output:

```
TermQuery results:
0.2518424:Warning label maker
0.22259936:Hurricane warning
0.22259936:Tornado warning

BoostingTermQuery results:
0.7870075:Hurricane warning
0.7870075:Tornado warning
0.17807949:Warning label maker
```

Indeed, `BoostingTermQuery` caused the two bulletins to get much higher scores, bringing them to the top of the results!

While `BoostingTermQuery` is the simplest way to use payloads to alter scoring of documents, the `SpanQuery` classes have also been extended on an initial, experimental basis, to include payloads. Each `SpanQuery` class implements the method `getPayloadSpans`, to retrieve all matching spans for the query, along with the payloads contained within each span that matched. At this point, none of the `SpanQuery` classes make use of the payloads. It's up to you to subclass a `SpanQuery` class and override the `getSpans` method if you'd like to filter documents that match based on payload, or override the `SpanScorer` class to provide custom scoring based on the payloads contained within each matched span. These are very advanced use cases, and only a few users have ventured into this territory, so your best bet for inspiration is to spend some quality time on Lucene's users list.

The final Lucene API that has been extended with payloads is the `TermPositions` iterator. This is an advanced internal API that allows you to step through the posting list for a specific term, retrieving each document that matched plus all positions, along with their payload, of that term's occurrences in the document. `TermPositions` has these added methods:

```
boolean isPayloadAvailable()
int getPayloadLength()
byte[] getPayload(byte[] data, int offset)
```

Note that once you've called `getPayload()` you cannot call it again until you've advanced to the next position by calling `nextPosition()`. Each payload can only be retrieved once.

Payloads are still under active development and exploration, in order to provide more core support to make use of payloads for either result filtering or custom scoring. Until the core support is fully fleshed out, you'll need to use the extension points described here to take advantage of this powerful feature. And stay tuned on the user's list!

Next we show some tools to evaluate search performance using the JUnitPerf framework.

6.6 Performance testing

Lucene is fast and scalable. But how fast is it? Is it fast enough? Can you guarantee that searches are returned within a reasonable amount of time? How does Lucene respond under load?

If your project has high performance demands, you've done the right thing by choosing Lucene, but don't let performance numbers be a mystery. There are several ways Lucene's performance can be negatively impacted by how you use it—like using fuzzy or wildcard queries or a range query, as you'll see in this section.

We've been highlighting unit testing throughout the book using the basics of JUnit. In this section, we utilize another unit-testing gem, JUnitPerf. JUnitPerf, a JUnit decorator, allows JUnit tests to be measured for load and speed.

6.5.1 Testing the speed of a search

We've discussed how `FuzzyQuery` and `WildcardQuery` have the potential to get out of control. In a similar fashion, `RangeQuery` can, too: As it enumerates all the terms in the range, it forms a `BooleanQuery` that can potentially be large.

The infamous Mike "addicted to the green bar" Clark has graciously donated some Lucene performance tests to us.[7] Let's examine a concrete example in which we determine that a searching performance issue is caused by how we index, and find out how we can easily fix this issue. We rely on JUnitPerf to identify the issue and ensure that it's fixed and stays fixed.

We're indexing documents that have a last-modified timestamp. For example purposes, we index a sample of 1,000 fabricated documents with timestamps increasing in 1-second increments, starting yesterday:

```
Calendar timestamp = GregorianCalendar.getInstance();
timestamp.set(Calendar.DATE,                              // #1
              timestamp.get(Calendar.DATE) - 1);          // #1
for (int i = 0; i < size; i++) {
  timestamp.set(Calendar.SECOND,                          // #2
                timestamp.get(Calendar.SECOND) + 1);      // #2
  String now = DateTools.dateToString(timestamp.getTime(),
                                      DateTools.Resolution.MILLISECOND);
  Document document = new Document();
  document.add(new Field("last-modified", now,
                         Field.Store.YES, Field.Index.NOT_ANALYZED));
  writer.addDocument(document);
}
```

#1 Yesterday
#2 Increase 1 second

Let's create a test up front to ensure that our search is returning the expected results by searching over a timestamp range that encompasses all documents indexed:

```
public void testSearchByTimestamp() throws Exception {
  Search s = new Search();
  TopDocs hits = s.searchByTimestamp(janOneTimestamp,
                                     todayTimestamp);
  assertEquals(1000, hits.totalHits);
}
```

`searchByTimestamp` performs a `RangeQuery`:[8]

[7] Mike is the coauthor of *Bitter EJB* (Manning) and the author of *Pragmatic Automation* (Pragmatic Bookshelf); http://www.clarkware.com.

[8] We're intentionally skipping bits of Mike's test infrastructure to keep our discussion focused on the performance-testing aspect rather than get bogged down following his nicely decoupled code. See the "about this book" section at the beginning of the book for details on obtaining the full source code.

```
public TopDocs searchByTimestamp(Date begin, Date end)
    throws Exception {
  String beginTerm =
      DateTools.dateToString(begin, DateTools.Resolution.MILLISECOND);
  String endTerm =
      DateTools.dateToString(end, DateTools.Resolution.MILLISECOND);

  Query query = new RangeQuery("last-modified",
                               beginTerm, endTerm,
                               true, true);

  return newSearcher(
           index.byTimestampIndexDirName()).search(query, 10);
}
```

At this point, all is well. We've indexed 1,000 documents and found them all using an encompassing date `RangeQuery`. Ship it! Whoa…not so fast…what if we had indexed 2,000 documents? Try changing the arg value from 1000 to 2000 under the build-perf-index task in build.xml, then run "ant clean test –Dtest=Search". Here's what happens when we run the `testSearchByTimestamp()` test method:

```
org.apache.lucene.search.BooleanQuery$TooManyClauses: maxClauseCount is set to 1024
     at org.apache.lucene.search.BooleanQuery.add(BooleanQuery.java:163)
     at org.apache.lucene.search.BooleanQuery.add(BooleanQuery.java:154)
     at org.apache.lucene.search.MultiTermQuery.rewrite(MultiTermQuery.java:84)
     at org.apache.lucene.search.IndexSearcher.rewrite(IndexSearcher.java:162)
     at org.apache.lucene.search.Query.weight(Query.java:94)
     at org.apache.lucene.search.Searcher.createWeight(Searcher.java:185)
     at org.apache.lucene.search.Searcher.search(Searcher.java:136)
     at org.apache.lucene.search.Searcher.search(Searcher.java:146)
     at lia.extsearch.perf.Search.searchByTimestamp(Search.java:43)
     at lia.extsearch.perf.SearchTest.testSearchByTimestamp(SearchTest.java:27)
```

Our dataset is only 2,000 documents, which is in general no problem for Lucene to handle. But, by default a `RangeQuery` internally rewrites itself to a `BooleanQuery` with a SHOULD clause for every term in the range. That is, with 2,000 documents being indexed, the `searchByTimestamp()` method will cause 2,000 OR'd `TermQuery`s nested in a `BooleanQuery`. This exceeds the default limit of 1,024 clauses to a `BooleanQuery`, which prevents queries from getting carried away.

Fortunately, one simple fix for this would be to call RangeQuery.setConstantScoreRewrite, which has no limit on the number of terms that fall within the range. The search would then run fine. Even so, it's still in your interest to reduce the number of terms in the range so that the rewrite process runs as quickly as possible. Let's see how to do that next.

MODIFYING THE INDEX

For searching purposes, though, the goal is to be able to search by date range. It's unlikely we'll need to search for documents in a range of seconds, so using this fine-grained timestamp isn't necessary. In fact,

it's problematic. Indexing 1,000 or 2,000 documents in successive second timestamp increments gives each document a completely unique term, all within the span of less than an hour's worth of timestamps.

Since searching by day, not second, is the real goal, let's index the documents by day instead:

```
String today = Search.today();

for (int i = 0; i < size; i++) {
  Document document = new Document();
  document.add(new Field("last-modified",
                         today, Field.Store.YES,
                         Field.Index.NOT_ANALYZED));
  writer.addDocument(document);
}
```

Here, `today` is set to YYYYMMDD format. Remember, terms are sorted alphabetically, so numbers need to take this into account (see section 6.3.3 for a number-padding example):

```
public static String today() {
  SimpleDateFormat dateFormat =
      (SimpleDateFormat) SimpleDateFormat.getDateInstance();
  dateFormat.applyPattern("yyyyMMdd");
  return dateFormat.format(todayTimestamp());
}
```

Notice that we're using a `String` value for `today` (such as 20040715) rather than using the `DateField.dateToString()` method. Regardless of whether you index by timestamp or by YYYYMMDD format, the documents all have the same year, month, and day; so in our second try at indexing a last-modified field, there is only a single term in the index, not thousands. This is a dramatic improvement that's easily spotted in JUnitPerf tests. You can certainly keep a timestamp field in the document, too—it just shouldn't be a field used in range queries unless you actually require per-second resolution.

TESTING THE TIMESTAMP-BASED INDEX

Listing 6.8 is a JUnitPerf `TimedTest`, testing that our original 1,000 documents are found in 100 milliseconds or less.

Listing 6.10 JUnitPerf-decorated timed test

```
public class SearchTimedTest {

  public static Test suite() {

    int maxTimeInMillis = 100;

    Test test = new SearchTest("testSearchByTimestamp");
    //Test test = new SearchTest("testSearchByDay");

    TestSuite suite = new TestSuite();
    suite.addTest(test);                                  // #1
    suite.addTest(new TimedTest(test, maxTimeInMillis));  // #2

    return suite;
```

```
  }
}
```

#1 Warmup test
#2 Wrap test in TimedTest

#1 We first run one test to warm up the JVM prior to timing.

#2 Then, we wrap the simple test inside a `TimedTest`, asserting that it runs in 100 milliseconds or less.

This test fails because it exceeds the 100-millisecond constraint:

```
Testcase: testSearchByTimestamp(lia.extsearch.perf.SearchTest):  FAILED
Maximum elapsed time exceeded! Expected 100ms, but was 149ms.
junit.framework.AssertionFailedError: Maximum elapsed time exceeded! Expected 100ms,
but was 149ms.
     at com.clarkware.junitperf.TimedTest.runUntilTestCompletion(Unknown Source)
     at com.clarkware.junitperf.TimedTest.run(Unknown Source)
```

The test failed, but not by much. Of course, when 2,000 documents are attempted it fails horribly with a `TooManyClauses` exception.

TESTING THE DATE-BASED INDEX

Now let's write a unit test that uses the YYYYMMDD range:

```
public void testSearchByDay() throws Exception {
  Search s = new Search();
  TopDocs hits = s.searchByDay("20040101", today);
  assertEquals(1000, hits.totalHits);
}
```

The value of `today` in `testSearchByDay()` is the current date in YYYYMMDD format. Now we replace one line in `SearchTimedTest` with a `testSearchByDay()`:

```
Test test = new SearchTest("testSearchByDay");
```

Our `SearchTimedTest` now passes with flying colors (see figure 6.3 for timings of `SearchTest` under load).

6.5.2 Load testing

Not only can JUnitPerf decorate a test and assert that it executes in a tolerated amount of time, it can also perform load tests by simulating a number of concurrent users. The same decorator pattern is used as with a `TimedTest`. Decorating a `TimedTest` with a `LoadTest` is the general usage, as shown in listing 6.9.

Listing 6.11 Load test

```
public class SearchLoadTest {

  public static Test suite() {

    int maxTimeInMillis = 100;
    int concurrentUsers = 10;
    Test test = new SearchTest("testSearchByDay");

    TestSuite suite = new TestSuite();
    suite.addTest(test);
    Test timedTest = new TimedTest(test, maxTimeInMillis); |#1
    LoadTest loadTest = new LoadTest(timedTest, concurrentUsers);
  |#2
    suite.addTest(loadTest);


    return suite;
  }
      }
```

#1 Wrap basic test with TimedTest
#2 Wrap TimedTest in LoadTest

#1 We wrap the basic test (ensuring that 1,000 hits are found) with a `TimedTest`.

#2 Then we wrap the `TimedTest` in a `LoadTest`, which executes the `TimedTest` 10 times concurrently.

`SearchLoadTest` executes `testSearchByDay()` 10 times concurrently, with each thread required to execute in under 100 milliseconds. It should be no surprise that switching the `SearchLoadTest` to run `SearchTest.testSearchByTimestamp()` causes a failure, since it fails even the `SearchTimedTest`. The timings of each `SearchTest`, run as 10 concurrent tests, are shown in figure 6.3.

Test	Time elapsed
Total:	**0.154 s**
testSearchByDay	0.016 s
testSearchByDay	0.009 s
testSearchByDay	0.006 s
testSearchByDay	0.005 s
testSearchByDay	0.006 s
testSearchByDay	0.007 s
testSearchByDay	0.069 s
testSearchByDay	0.005 s
testSearchByDay	0.009 s
testSearchByDay	0.022 s

Figure 6.3 Performance test results for 10 concurrent `SearchTest`s, each required to complete in 100 milliseconds or less

The results indicate that each test performed well under the 100-millisecond requirement, even running under concurrent load.

6.5.3 QueryParser again!

QueryParser rears its ugly head again with our changed date format. The built-in date-range handling parses DateFormat.SHORT formats into the DateField text conversions. It would be nice to let users enter a typical date format like 1/1/04 and have it converted to our revised date format of YYYYMMDD. This can be done in a similar fashion to what we did in section 6.3.3 to pad integers for range queries. The desired effect is shown in the following test:

```
    public void testQueryParsing() throws Exception {
      SmartDayQueryParser parser =
          new SmartDayQueryParser("contents",
              new StandardAnalyzer());
      parser.setLocale(Locale.US);

      Query query =
          parser.parse("last-modified:[1/1/04 TO 2/29/04]");

      assertEquals("last-modified:[20040101 TO 20040229]",
          query.toString("contents"));
    }
```

Now that we have our desired effect coded as a test case, let's make it pass by coding SmartDayQueryParser.

UNDERSTANDING SMARTDAYQUERYPARSER

The SmartDayQueryParser is a simple adaptation of the built-in QueryParser's getRangeQuery method:

```
  public class SmartDayQueryParser extends QueryParser {
    public static final DateFormat formatter =
        new SimpleDateFormat("yyyyMMdd");

    public SmartDayQueryParser(String field, Analyzer analyzer) {
      super(field, analyzer);
    }

    protected Query getRangeQuery(String field, String part1, String part2, boolean
  inclusive)
                                           throws ParseException {
      try {
        DateFormat df =
            DateFormat.getDateInstance(DateFormat.SHORT,
                getLocale());
        df.setLenient(true);
        Date d1 = df.parse(part1);
        Date d2 = df.parse(part2);
        part1 = formatter.format(d1);
        part2 = formatter.format(d2);
```

```
      } catch (Exception ignored) {
      }

      return new RangeQuery(field, part1, part2,
                            inclusive, inclusive);
    }
  }
```

The only difference between our overridden `getRangeQuery` and the original implementation is the use of YYYYMMDD formatting.

6.5.4 Morals of performance testing

In addition to testing whether Lucene can perform acceptably with your environment and data, unit performance testing assists (as does basic JUnit testing) in the design of your code. In this case, you've seen how our original method of indexing dates was less than desirable even though our first unit test succeeded with the right number of results. Only when we tested with more data or with time and load constraints did an issue present itself. We could have swept the data failure under the rug temporarily by setting `BooleanQuery`'s `setMaxClauseCount(int)` to `Integer.MAX_VALUE`. However, we wouldn't be able to hide a performance test failure.

We strongly encourage you to adopt unit testing in your projects and to continue to evolve the testing codebase into performance unit testing. As you can tell from the code examples in this book, we are highly test-centric, and we also use tests for learning purposes by exploring APIs. Lucene itself is built around a strong set of unit tests, and it improves on a regular basis.

6.6 Summary

Lucene offers developers extreme flexibility in searching capabilities, so much so that this is our 3rd (and final!) chapter covering search. Custom sorting is straightforward and useful when the built-in sorting by relevance or field values is not sufficient. Custom `HitCollector` implementations let you efficiently do what you want with each search hit as it's found, while custom Filters allow you to pull in any external information to construct a filter.

By extending `QueryParser` you can refine how it constructs queries, in order to prevent certain kinds of queries or alter how each `Query` is constructed. Finally we showed how the advanced payloads functionality can be used for refined control over which terms in a document are more important than others, based on their positions.

Equipped with the searching features from this chapter and chapters 3 and 5, you have more than enough power and flexibility to integrate Lucene searching into your applications.

7

Extracting document text with Tika

This chapter covers

- Tika's logical design
- Tika's built-in tool for text extraction
- Tika's APIs for text extraction
- Tika's limitations
- Alternative text extraction tools

One of the more mundane yet vital steps when building a search application is extracting text from the documents you need to index. You might be lucky to have an application whose content is already in textual format or whose documents are always the same format, such as XML files or as rows in a database. If you are unlucky, you must instead accept the surprisingly wide plethora of document formats that are popular today such as Outlook, Word, Excel, PowerPoint, Visio, Flash, PDF, Open Office, RTF, HTML and even archive file formats like Tar and Zip. Even seemingly textual formats, like XML or HTML, present challenges as you must take care not to accidentally include any tags or JavaScript sources, etc. The plain text format might seem easiest of all, yet frequently it is difficult to determine its character encoding.

In the past it was necessary to "go it alone": track down your own document filters, one by one, and interact with their unique and interesting APIs in order to extract the text you need. You'd also need to detect the document type yourself. Fortunately, we now have a nice framework called Tika which handles most of the work for you.

7.1 What is Tika?

Tika was added to the Lucene umbrella in October 2008, after graduating from the Apache incubator. The most recent release as of March 2009 is 0.3. Development continues at a rapid pace, and it's expected there will be

non-back-compatible changes in the march to the 1.0 release, so be sure to check Tika's website at http://lucene.apache.org/tika for the latest documentation.

Tika is actually a framework that hosts plugin parsers for each supported document type. The framework presents the same standard API to the application for extracting text and metadata from a document, and under the hood the plugin parser interacts with the external library using the custom API exposed by that library. This lets your application use the same uniform API regardless of document type. When you need to extract text from a document, Tika finds the right parser for the document (details on this shortly).

Being a framework, Tika doesn't do any of the actual document filtering itself. Rather, it relies on external open-source projects and libraries to do the heavy lifting. Table 7.1 lists the formats supported as of the 0.3 release, along with which project or library the document parser is based upon. There is support for many common document formats, and new formats are added frequently, so check online for the latest list.

In addition to extracting the body text for a document, Tika also extracts metadata values for most document types. Tika represents metadata as a single `String <-> String` map, with constants exposed for the common metadata keys, listed in Table 7.2. These constants are defined in the `Metadata` class in the `org.apache.tika.metadata` package. However, not all parsers can extract metadata, and when they do, they may extract to different metadata keys than you expect. In general the area of metadata extraction is still in flux in Tika, so it's best to test parsing some samples of your documents to understand what metadata is exposed.

Table 7.1: Supported document formats and the library used to parse them

Format	Library
Microsoft Office OLE2 Compound Document Format (Excel, Word, PowerPoint, Visio, Outlook)	Apache POI
Microsoft Office 2007 OOXML	Apache POI
Adobe Portable Document Format (PDF)	PDFBox
Rich Text Format (RTF) – currently body text only (no metadata)	Java Swing API (RTFEditorKit)
Plain Text	ICU4J library
HTML	CyberNeko library
XML	Java's javax.xml classes
ZIP Archives	Java's builtin ZIP classes
TAR Archives	Apache Ant
GZIP compression	Java's built-in support (GZIPInputStream)
BZIP2 compression	Apache Ant
Image formats (metadata only)	Java's javax.imageio classes
Java class files	ASM library (JCR-1522)
Java JAR files	ZIP + Java Class files
MP3 audio (ID3v1 tags)	Implemented directly
Open Document	Parses XML directly
Microsoft Office 2007 XML (in progress)	Apache POI
Adobe Flash (in progress)	
MIDI files (embedded text, eg song lyrics)	Java's built-in support (javax.sound.midi.*)

Table 7.2: Metadata keys that Tika extracts

Metadata Constant	Description
RESOURCE_KEY_NAME	The name of the file or resource that contains the document. A client application can set this property to allow the parser to use file name heuristics to determine the format of the document. The parser implementation may set this property if the file format contains the canonical name of the file (for example the Gzip format has a slot for the file name).
CONTENT_TYPE	The declared content type of the document. A client application can set this property based on an HTTP Content-Type header, for example. The declared content type may help the parser to correctly interpret the document. The parser implementation sets this property to the content type according to which the document was parsed.
CONTENT_ENCODING	The declared content encoding of the document. A client application can set

	this property based on an HTTP Content-Type header, for example. The declared content type may help the parser to correctly interpret the document. The parser implementation sets this property to the content type according to which the document was parsed.
TITLE	The title of the document. The parser implementation sets this property if the document format contains an explicit title field.
AUTHOR	The name of the author of the document. The parser implementation sets this property if the document format contains an explicit author field.
MSOffice.*	Defines additional metadata from Microsoft Office: APPLICATION_NAME, CHARACTER_COUNT, COMMENTS, KEYWORDS, LAST_AUTHOR, LAST_PRINTED, LAST_SAVED, PAGE_COUNT, REVISION_NUMBER, TEMPLATE, WORD_COUNT

Let's drill down into how Tika models a document's logical structure, and what concrete API is used to expose this.

7.1.1 Tika's logical design and API

Tika uses the XHTML (Extensible Hypertext Markup Language) standard to model all documents, regardless of their original format. XHTML is a markup language that combines the best of XML and HTML: because an XHTML document is valid XML, it can be programmatically processed using standard XML tools, and because it is also valid HTML it can be rendered with a web browser. With XHTML, a document is cast to this logical structure:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>...</title>
  </head>
  <body>
    ...
  </body>
</html>
```

Within the `<body>...</body>` there are other tags (`<p>`, `<h1>`, `<div>` etc.) representing internal document structure.

This is the logical structure of an XHTML document, but how does Tika actually deliver that to your application? The answer is SAX (Simple API for XML), another well established standard used by XML parsers. With SAX, as an XML document is parsed, the parser invokes methods on an instance implementing the `org.xml.sax.ContentHandler`. This is a very scalable approach for parsing XML documents since it enables the application to pick and choose what should be done with each element, as it is encountered. Arbitrarily large documents can be processed with minimal consumption of RAM.

The primary interface to Tika is the surprisingly simple `parse` method (in the `org.apache.tika.parser.Parser` class):

```
void parse(InputStream stream
           ContentHandler handler,
           Metadata metadata)
```

`Tika` reads the bytes for the document from the `InputStream`, but will not close it. It's recommended you close the stream using a `try/finally` clause.

The document parser then decodes the bytes, translates the document into the logical XHTML structure, and invokes the SAX API via the provided `ContentHandler`. The final parameter, metadata, is used bidirectionally: input details, such as specified `Content-Type` (from an HTTP Server) or filename (if known) are set before invoking parse, and then any metadata encountered while Tika is processing the document will be recorded and returned.

You can see Tika itself is simply a conduit: it doesn't do anything with the document text it encounters except invoke the `ContentHandler`. It's then up to your application to provide a `ContentHandler` that actually does something of interest with the resulting elements and text. However, `Tika` includes some helpful utility classes that implement `ContentHandler` for common cases. For example, `BodyContentHandler` gathers all text within the `<body>...</body>` part of the document and forwards it to another handler, `OutputStream`, `Writer`, or an internal string buffer for later retrieval.

If you know for certain which document type you are dealing with, you can create the right parser (for example, `PDFParser`, `OfficeParser`, `HtmlParser`, etc) directly and then invoke its `parse` method. If you are unsure of the document's type, `Tika` provides an `AutoDetectParser`, which is a `Parser` implementation that uses various heuristics to determine the document's type and apply the correct parser.

`Tika` tries to autodetect things like document format and the encoding of the character set (for text/plain documents). Still if you have pre-existing information about your documents, such as the original filename (containing a possibly helpful extension) or the character encoding, it's best if you provide this information via the Metadata input so Tika may make use this. The filename should be added under `Metadata.RESOURCE_NAME_KEY`; content-type should be added under `Metadata.CONTENT_TYPE`, and the content encoding should be added under `Metadata.CONTENT_ENCODING`.

It's time to get our feet wet! Let's walk through the installation process for Tika.

7.2 Installing Tika

You'll need a build of `Tika`. The source code with this book includes the 0.3 release of Tika, in the lib directory, but likely you're staring at a newer release. The binary builds for `Tika` are included in Maven 2 repository, which you may either download directly or reference in your application if you are already using Maven 2.

Building Tika from sources is also straightforward, although you should check Getting Started on the Tika website for any changes since this was written. Download the source release (for example, apache-tika-0.3-src.tar.gz for 0.3) and extract it. Tika uses Apache's Maven 2 build system, and requires Java 5 or higher, so you'll need to first install those dependencies. Then run "`mvn install`" from within the Tika source directory you unpacked above. That command will download a bunch of dependencies into your Maven area, compile `Tika`'s sources, run tests, and finally produce these two build artifacts in the subdirectory "target":

- `tika-0.3.jar` -- contains the compiled Java classes & interfaces for the `org.apache.tika` package, as well as the default Tika configuration settings
- `tika-0.3-standalone.jar` – contains classes for Tika as well as all the dependencies, collected into a single JAR file; this is the most convenient way to use Tika
- `tika-0.3-jdk14.jar` -- this is a retro-translated version of the JAR file, that enables Tika to run in a 1.4 JRE. This uses the http://retrotranslator.sourceforge.net software, run on the first build artifact.

It's recommended that you simply use the `tika-0.3-standalone.jar`, since it has all dependencies contained within it. If for some reason that's not possible, you can use maven to gather all dependency jars into the `target/dependencies` directory.

> **NOTE**
>
> You can gather all required dependency JARs by running mvn dependency:copy-dependencies. This will copy the required JARS out of your maven area and into the target/dependencies directory. This is very useful if you intend to run Tika outside of Maven 2.

If all goes well, you'll see "`BUILD SUCCESSFUL`" printed at the end.

Now that we've built Tika, it's time to finally extract some text! We'll start with Tika's built-in text extraction tool.

7.3 Tika's built-in text extraction tool

`Tika` comes with a simple built-in tool allowing you to extract text from documents in the local filesystem or via URL. This tool creates an `AutoDetectParser` to filter the document, and then provides a few options for interacting with the results. The tool can run either with a dedicated graphical user interface (GUI), or in a command-line only mode that can be chained together, using pipes, with other command-line tools. To run the tool with a GUI:

```
java -jar lib/tika-0.3-standalone.jar --gui
```

This brings up a simple GUI window, to which you can drag and drop files in order to test how the filters work with them. Figure 7.1 shows the window after dragging a draft of this chapter (as a Microsoft Word document) onto the window. The window has multiple tabs showing different text extracted during filtering:

- Formatted text renders the XHTML, rendered with Java's builtin `javax.swing.JEditorPane` as `text/html` content
- Plain text shows only the text and whitespace parts, extracted from the XHTML document
- Structured text shows the raw XHTML source
- Metadata contains all metadata fields extracted from the document
- Errors describes any errors encountered while parsing the document

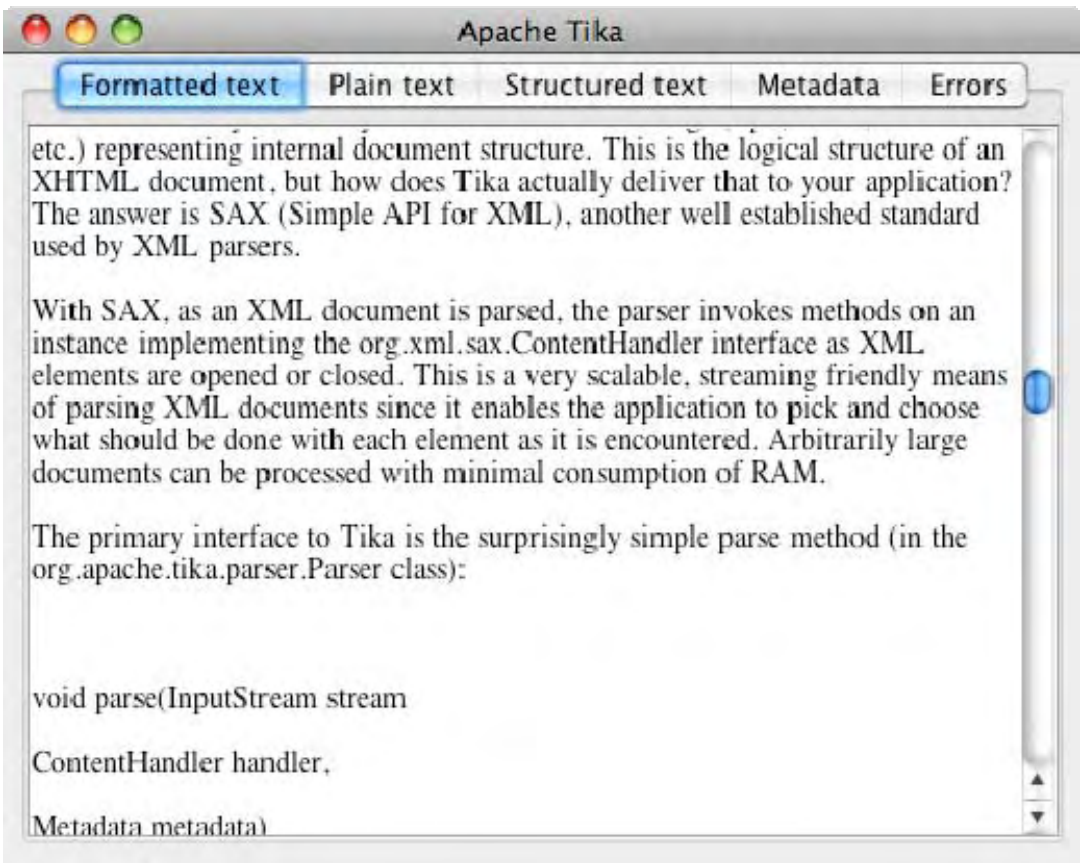


Figure 7.1 Tika's built-in text extraction tool

While the GUI tool is a great way to quickly test Tika on a document, it's often more useful to use the command-line only invocation, for example like this:

```
cat Document.pdf | java -jar target/tika-0.3-standalone.jar -
```

This prints the full XHTML output from the parser (the extra '-' on the end of the command tells the tool to read the document from the standard input; you could also provide the filename directly instead of piping its contents into the command). This tool accepts various command-line options to change its behavior:

- --help or -? prints the full usage

- --verbose or –v prints debug messages
- --gui or –g runs the GUI
- --xml or –x outputs the XHTML content (this is the default behavior). This corresponds to the Structured text tab from the GUI.
- --html or –h outputs the HTML content, which is a simplified version of the XHTML content. This corresponds to the Formatted text (rendered as HTML) from the GUI.
- --text or –t outputs the plain text content. This corresponds to the Plain text tab from the GUI.
- --metadata or –m outputs only the metadata keys and values. This corresponds to the Metadata tab from the GUI.

You could use Tika's command-line tool as the basis of your text extraction solution. It's very simple to use and fast to deploy. But if you need more control over which parts of the text are used, or which metadata fields to keep, you'll need to use Tika's programmatic API, which we cover next.

7.4 Extracting text programmatically

We've seen Tika's simple `parse` API, which is the core of any text extraction based on Tika. But what about the rest of the text extraction process? How can you build a Lucene document from a SAX `ContentHandler`? That's what we'll now do in this section.

The basic approach is straightforward. You have a source for the document, which you must open as an `InputStream`. Then you create an appropriate `ContentHandler` for your application, or use one of the utility classes provided with Tika. Finally, you build the Lucene Document instance from the metadata and text encountered by the `ContentHandler`. Let's make this all concrete: recall that the `Indexer` tool from Chapter 1 has the limitation that it can only index plain text files (with the extension .txt). `TikaIndexer`, shown in Listing 7.1, now fixes that! Let's walk through the approach:

1. Subclass the original `Indexer` and override two methods: the `acceptFile` method is changed to always return true, so that we attempt to index all files encountered; the `getDocument` method is changed to use `Tika` to extract the text.
2. Override the static `main()` method, to first print all all mime types that Tika can parse, and then to create `TikaIndexer` and invoke its index method.
3. In `getDocument`, we create a `Metadata` instance and record the filename in it so `AutoDetectParser` can use the file's extension to aid in choosing the right parser.
4. Create the `InputStream` to read the file's contents.
5. Instantiate `AutoDetectParser`, which detects the document's type and then delegates to the appropriate parser.
6. Use the utility class `BodyContentHandler` to get the text from the body of the document.
7. Invoke the `parse` method, and then retrieve the text from the body of the document by calling `handler.toString()` and adding that as the contents field in a newly created Lucene Document.
8. Iterate over all metadata items and add stored but un-indexed fields to the document. Those metadata fields that are designated as textual, via `textualMetadataFields`, are also indexed (appended) into the "contents" field.
9.

This example will work very well, but there are a few things you should fix before using it for real in production:

1. Catch and handle the various exceptions that may be thrown by `parser.parse`. If the document is corrupt you'll hit a `TikaException`. If there was a problem reading the bytes from the `InputStream`, you'll hit an `IOException`. You may hit class loader exceptions if the required parser could not be located or instantiated.
2. Be more selective about which metadata fields you want in our index, and how you'd like to index them. This is vey much application dependent.
3. Be more selective about which text is indexed. Right now `TikaIndexer` simply appends together all text from the document into the "contents" field, by adding more than one instance of that field name to the

document. You may instead want to handle different sub-structure of the document differently, and perhaps use an analyzer that sets a `positionIncrementGap` so that phrases and span queries cannot match across two different "contents" fields.

4. Add any custom logic to filter out known "uninteresting" portions of text documents, for example standard headers and footer text that appear in all documents.
5. If your document's text could be very large in size, consider using the `ParsingReader` utility class (described next in section 7.4.1) instead.

NOTE

Since Tika is advancing so quickly, it's likely by the time you read this there is a good out-of-the-box integration of Lucene and `Tika`, so be sure to check at http://lucene.apache.org/tika.

Listing 7.1: Class to extract text from arbitrary documents and index it with Lucene

```
public class TikaIndexer extends Indexer {

  private boolean DEBUG = false;                     //1

  static Set textualMetadataFields = new HashSet();  //2
  static {                                           //2
    textualMetadataFields.add(Metadata.TITLE);       //2
    textualMetadataFields.add(Metadata.AUTHOR);      //2
    textualMetadataFields.add(Metadata.COMMENTS);    //2
    textualMetadataFields.add(Metadata.KEYWORDS);    //2
    textualMetadataFields.add(Metadata.DESCRIPTION); //2
    textualMetadataFields.add(Metadata.SUBJECT);     //2
  }

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      throw new Exception("Usage: java " + TikaIndexer.class.getName()
        + " <index dir> <data dir>");
    }

    TikaConfig config = TikaConfig.getDefaultConfig();  //3
    List<String> parsers = new ArrayList(config.getParsers().keySet());  //3
    Collections.sort(parsers);                          //3
    Iterator<String> it = parsers.iterator();           //3
    System.out.println("Mime type parsers:");           //3
    while(it.hasNext()) {                               //3
      System.out.println("  " + it.next());             //3
    }                                                   //3
    System.out.println();                               //3

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

    long start = new Date().getTime();
    TikaIndexer indexer = new TikaIndexer(indexDir);
    int numIndexed = indexer.index(dataDir);
    long end = new Date().getTime();

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

  public TikaIndexer(String indexDir) throws IOException {
    super(indexDir);
  }

  protected boolean acceptFile(File f) {
    return true;                                    //4
  }

  protected Document getDocument(File f) throws Exception {

    Metadata metadata = new Metadata();
```

```
      metadata.set(Metadata.RESOURCE_NAME_KEY,        //5
                   f.getCanonicalPath());

      // If you know content type (eg because this document
      // was loaded from an HTTP server), then you should also
      // set Metadata.CONTENT_TYPE

      // If you know content encoding (eg because this
      // document was loaded from an HTTP server), then you
      // should also set Metadata.CONTENT_ENCODING

      InputStream is = new FileInputStream(f);
      Parser parser = new AutoDetectParser();
      ContentHandler handler = new BodyContentHandler();

      try {
        parser.parse(is, handler, metadata);
      } finally {
        is.close();
      }

      Document doc = new Document();

      doc.add(new Field("contents", handler.toString(), Field.Store.NO, Field.Index.ANALYZED));

      if (DEBUG) {
        System.out.println("  all text: " + handler.toString());
      }

      for(String name : metadata.names()) {         //6
        String value = metadata.get(name);

        if (textualMetadataFields.contains(name)) {
          doc.add(new Field("contents", value,      //7
                            Field.Store.NO, Field.Index.ANALYZED));
        }

        doc.add(new Field(name, value, Field.Store.YES, Field.Index.NO));

        if (DEBUG) {
          System.out.println("  " + name + ": " + value);
        }
      }

      if (DEBUG) {
        System.out.println();
      }

      return doc;
    }
  }
```

#1 Change to true to see text for each document
#2 Which metadata fields are textual
#3 List all mime types handled by Tika
#3 Always attempt to index the file
#4 Tell Tika the filename
#5 Index textual metadata fields
#6 Append to contents field

As you can see, it's quite simple using Tika's programmatic APIs to extract text and build a Lucene document. In our example, we used the `parse` API from `AutoDetectParser`, but Tika also provides some utility APIs that might be a useful alternate path for your application.

7.4.1 The ParsingReader utility class

The `ParsingReader` class, in the `org.apache.tika.parser` package, is a very useful drop-in class. It parses the document but exposes a `Reader` to read the text out. Since Lucene's Field can index text directly from a Reader, this is a very simple way to index the text with Lucene.

`ParsingReader` has a clever implementation. When created, it spawns a background thread to parse the document, using the `BodyContentHandler`. The resulting text is written to a `PipedWriter` (from `java.io`), and then a corresponding `PipedReader` is returned back to you. Because of this streaming implementation, the full text of the document is never materialized at once. Instead, the text is created as the `Reader` consumes it, with a small shared buffer. This means even documents that parse to an exceptionally large amount of text will use very little memory during filtering.

During creation, `ParsingReader` also attempts to process all metadata for the document, so after it's created but before indexing the document you should call the `getMetadata()` method and add any important metadata to your document. Note that this is a new feature available starting with Tika's 0.3 release; prior releases don't set the metadata until after text is read from the `Reader`.

This class may be a great fit for your application. However, because a thread is spawned for every document, and because `PipedWriter` and `PipedReader` are used, it's likely net indexing throughput is slower than if you simply materialize the full text yourself up front (eg with `StringBuilder`). Still, if materializing the full text up front is out of the question, because your documents may be unbounded in size, then `ParsingReader` is a real life saver.

7.4.2 Customizing parser selection

Tika's `AutoDetectParser` first determines the mime type of the document, through various heuristics, and then uses that mime type to look up the appropriate parser. To do that lookup, `Tika` uses an instance of `TikaConfig`, which is a simple class that loads the mapping of mime type to parser class via an XML file. The default `TikaConfig` class can be obtained with the static `getDefaultConfig` method, which in turn simply loads the file `tika-config.xml` that comes with Tika. Since this is an XML file, you can easily open it with your favorite text editor to see which mime types Tika can presently handle. We also used `TikaConfig`'s `getParsers` method in Listing 7.1 to list the mime types.

If you'd like to change which parser is used for a given mime-type, or, add your own parser to handle a certain mime-type, simply create your own corresponding XML file, and instantiate your own `TikaConfig` from that file. Then, when creating `AutoDetectParser`, pass in your `TikaConfig` instance.

This wraps up our coverage of using `Tika` for extracting text. Let's change gears now and consider some of `Tika`'s limitations.

7.5 Tika's limitations

Being very new, Tika has a few known challenges that it's working through. Some of these issues are simply a by-product of its design and won't change with time without major changes, while others are solvable problems and likely resolved by the time you read this.

The first challenge is loss of document structure in certain situations. In general, some documents may have far richer structure than the simple standard XHTML model used by `Tika`. In our example, `addressbook.xml` has rich structure, containing 2 entries each with rich specific fields. But `Tika` regularizes this down to a fixed XHTML structure, thus losing some information. If you need to make use of this structure, you're better off interacting directly with an XML parser.

Another limitation is the astounding number of dependencies when using Tika. If you use the standalone jar, this results in a very large number of classes in that jar. If you're not using the standalone JAR, then you'll need man JAR files on your CLASSPATH. In part this is simply because `Tika` relies on numerous external packages to do the actual parsing. But it's also because these external libraries often do far more than `Tika` requires. For example PDFBox and Apache POI understand document fonts, layouts, embedded graphics, etc., and are able to create new documents in the binary format or modify existing documents. `Tika` only requires a small portion of this (the "extract text" part), yet these libraries don't typically factor that out as a standalone component. As a result, numerous excess classes and JARs end up on the CLASSPATH which could cause problems if they conflict with other JARs in your application.

Another challenge is certain document parsers, such as Microsoft's OLE2 Compound Document Format, require full random access to the document's bytes, which `InputStream` doesn't expose. In such cases `Tika` currently copies all bytes from the stream into a temporary file, which is then opened directly for random access. A future improvement, possibly already done by the time you read this, will allow you to pass a random access stream directly to `Tika` (if your document is already stored and accessible via a random access file), to avoid this unnecessary copy.

Let's look at some alternatives to Tika.

7.6 Alternatives

While Tika is our favorite way to extract text from documents, there are some interesting alternatives. The Aperture open-source project, hosted by SourceForge at http://aperture.sourceforge.net, has support for a wide variety of document formats and is able to extract text content and metadata. Furthermore, while Tika focuses only on text extraction, Aperture also provides crawling support, meaning it can connect to file systems, Web servers, IMAP mail servers, Outlook and iCal file and crawl for all documents within these systems.

There are also commercial document filtering libraries, such as Stellent's filters (now part of Oracle) and KeyView filters (now part of Autonomy). While these are closed solutions, and could be fairly expensive to license, they may be a fit for your application.

Finally, there are numerous individual open-source parsers out there for handling document types. It's entirely possible your document type already has a good open-source parser that simply hasn't yet been integrated with Tika. If you find one, you should consider building the Tika plugin for it and donating it back, or even simply calling attention to the parser on Tika's developers mailing list.

7.7 Summary

There are a great many popular document formats in the world. In the past, extracting text from these documents was a real sore point in building a search application. But today, we have Tika, which makes text extraction surprisingly simple. We've seen Tika's command-line tool, which could be the basis of a quick integration with your application, as well as an example using Tika's APIs that with some small modifications could easily be the core of text extraction for your search application. Using Tika to handle text extraction allows you to spend more time on the truly important parts of your search application.

In the next chapter we'll look at ports of Lucene to other programming languages and environments.

8

Tools and extensions

This chapter covers

- Using Lucene's Sandbox components
- Highlighting hits in your search results
- Correcting the spelling of search text
- Viewing index details using Luke
- Variety of Query and Analyzer implementations
- Working with third-party Lucene tools

You've built an index, but can you browse or query it without writing code? Absolutely! In this chapter, we'll discuss three tools to do this. Do you need analysis beyond what the built-in analyzers provide? Several specialized analyzers for many languages are available in Lucene's Sandbox. How about providing term highlighting in search results? We've got that, too!

This chapter examines third-party (non-core-Lucene) software as well as several Sandbox projects. Apache hosts a separate subversion directory, contrib, where add-ons to Lucene are kept. Deliberate care was taken with the design of Lucene to keep the core source code cohesive yet extensible. We're taking the same care in this book by keeping an intentional separation between what is in the core of Lucene and the tools and extensions that have been developed to augment it.

8.1 Playing in Lucene's Sandbox

In an effort to accommodate the increasing contributions to the Lucene project that are above and beyond the core codebase, a contrib directory was created to house them. The Sandbox is continually evolving, making it tough to write about concretely. We'll cover the stable pieces and allude to the other interesting

bits. We encourage you, when you need additional Lucene pieces, to consult the Sandbox repository and familiarize yourself with what is there—you may find that one missing piece you need. And in the same vein, if you've developed Lucene pieces and want to share the maintenance efforts, contributions are more than welcome.

Table 8.1 lists the current major contents of the Sandbox with pointers to where each is covered in this book.

Table 8.1 Major Sandbox component cross reference *(continued)*

Sandbox area	Description	Coverage
analyzers	Analyzers for various languages	Section 8.3
ant	An Ant `<index>` task	Section 8.4
benchmark	Support for running repeatable performance tests	Appendix D
db	Berkeley DB Directory implementation	Section 8.9
highlighter	Search result snippet highlighting	Section 8.7
javascript	Query builder and validator for web browsers	Section 8.5
lucli	Command-line interface to interact with an index	Section 8.2.1
Luke	Graphical interface to interact with an index	Section 8.2.2
limo	Web-application (WAR) for interacting with an index	Section 8.2.3
miscellaneous	A few odds and ends, including the `ChainedFilter`	Section 8.8
snowball	Sophisticated family of stemmers and wrapping analyzer	Section 8.3.1
shingles	Token filter to create shingles (single token from multiple adjacent tokens) f rom another Token Stream	Section 8.3.2
spatial	Adds geographic search to Lucene	Section 8.16

surround	QueryParser that can create span queries	Section 8.14
Ngrams	Builds tokens from adjacent letters	Section 8.3.3
Memory indices	Create custom memory-based indexes for fast searching	Section 8.10
Query extensions	MoreLikeThis, FuzzyLikeThisQuery, BoostingQuery	Section 8.12
spellchecker	Correct spelling of terms in the user's query	Section 8.11
xml-query-parser	Creating a Query from XML strings	Section 8.13
WordNet	Utility to build a synonym index from WordNet database	Section 8.6

There are a few more Sandbox components than those we cover in this chapter. Refer to the Sandbox directly to dig around and to see any new goodies since this was printed. The benchmark package is so useful we dedicate a separate appendix (D) to it. We begin with some useful tools for peeking into your Lucene index.

8.2 Interacting with an index

You've created a great index. Now what? Wouldn't it be nice to browse the index and perform ad hoc queries? You will, of course, write Java code to integrate Lucene into your applications, and you could fairly easily write utility code as a JUnit test case, a command-line utility, or a web application to interact with the index. Thankfully, though, some nice utilities have already been created to let you interact with Lucene file system indexes. We'll explore three such utilities, each unique and having a different type of interface into an index:

- *lucli (Lucene Command-Line Interface)*—A CLI that allows ad-hoc querying and index inspection
- *Luke (Lucene Index Toolbox)*—A desktop application with nice usability
- *LIMO (Lucene Index Monitor)*—A web interface that allows remote index browsing

8.2.1 lucli: a command-line interface

Rather than write code to interact with an index, it can be easier to do a little command-line tap dancing for ad-hoc searches or to get a quick explanation of a score. The Sandbox contains the Lucene Command-Line Interface (lucli) contribution from Dror Matalon. Lucli lets you scroll through a history of commands and reexecute a previously entered command to enhance its usability.

Using the WordNet index we'll build in section 8.6 as an example, listing 8.1 demonstrates an interactive session.

Listing 8.1 lucli in action

```
% java lucli.Lucli

Lucene CLI. Using directory 'index'. Type 'help' for instructions.

lucli> index /lucene/wordnetindex           #1
Lucene CLI. Using directory '/lucene/wordnetindex'. Type 'help' for instructions.
Index has 44931 documents
All Fields:[syn, word]
Indexed Fields:[word]

lucli> search jump                          #2
Searching for: syn:jump word:jump           #3
1 total matching documents
-------------------------------------
---------------- 1 score:1.0---------------------
syn:alternate
syn:bound
syn:chute
syn:derail
syn:jumping
syn:jumpstart
syn:leap
syn:parachute
syn:parachuting
syn:rise
syn:saltation
syn:skip
syn:spring
syn:start
syn:startle
word:jump
################################################

lucli> help                                   #4
help
     count: Return the number of hits for a search. Example: count foo
     explain: Explanation that describes how the document scored against query.
Example: explain foo
     help: Display help about commands
     index: Choose a different lucene index. Example index my_index
     info: Display info about the current Lucene index. Example: info
     optimize: Optimize the current index
     quit: Quit/exit the program
     search: Search the current index. Example: search foo
     terms: Show the first 100 terms in this index. Supply a field name to only show
terms in a specific field. Example: terms
     tokens: Does a search and shows the top 10 tokens for each document. Verbose!
Example: tokens foo

lucli> explain dog                            #5
explain dog
Searching for: syn:dog word:dog
1 total matching documents
Searching for: word:dog
```

```
-----------------------------------
--------------- 1 score:1.0---------------------
syn:andiron
syn:blackguard
syn:bounder
syn:cad
syn:chase
syn:click
syn:detent
syn:dogtooth
syn:firedog
syn:frank
syn:frankfurter
syn:frump
syn:heel
syn:hotdog
syn:hound
syn:pawl
syn:tag
syn:tail
syn:track
syn:trail
syn:weenie
syn:wiener
syn:wienerwurst
word:dog
Explanation:11.019736 = (MATCH) fieldWeight(word:dog in 12176), product of:
  1.0 = tf(termFreq(word:dog)=1)
  11.019736 = idf(docFreq=1, numDocs=44931)
  1.0 = fieldNorm(field=word, doc=12176)

################################################
```

#1 Open existing index by path
#2 Perform search
#3 Query on all terms
#4 lucli explanations of commands
#5 Search, and explain results

Lucli is a fairly simple tool, but it has enough functionality to be very useful, especially if you're running through a limited terminal connection and unable to run the full user-interface that tools like Luke require. Lucli uses the `MultiFieldQueryParser` for search expressions and is hard-coded to use `StandardAnalyzer` with the parser. Our next tool is the wildly popular Luke.

8.2.2 Luke: the Lucene Index Toolbox

Andrzej Bialecki created Luke (found at `http://www.getopt.org/luke/`), an elegant Lucene index browser. This gem provides an intimate view inside a file system–based index from an attractive desktop Java application (see figure 8.1). We highly recommend having Luke handy when you're developing with Lucene because it allows for ad-hoc querying and provides insight into the terms and structure in an index.

Figure 8.1 Luke's About page

Luke has become a regular part of our Lucene development toolkit. Its tabbed and well integrated user interface allows for rapid browsing and experimentation. Luke can force an index to be unlocked when opening, optimize an index, and also delete and undelete documents, so it's really only for developers or, perhaps, system administrators. But what a wonderful tool it is!

You can launch Luke via Java WebStart from the Luke web site or install it locally. In any event, it requires JRE 1.5 or later to run. It's a single JAR file that can be launched directly (by double-clicking from a file-system browser, if your system supports that) or running `java -jar luke.jar` from the command line. The latest version at the time of this writing is 0.9.2; it embeds Lucene 2.4.1. A separate JAR is available without Lucene embedded; you can use it if you wish to use a different version of Lucene.[1] Of course, the first thing Luke needs is a path to the index file, as shown in the file-selection dialog in figure 8.2.

[1]The usual issues of Lucene version and index compatibility apply.

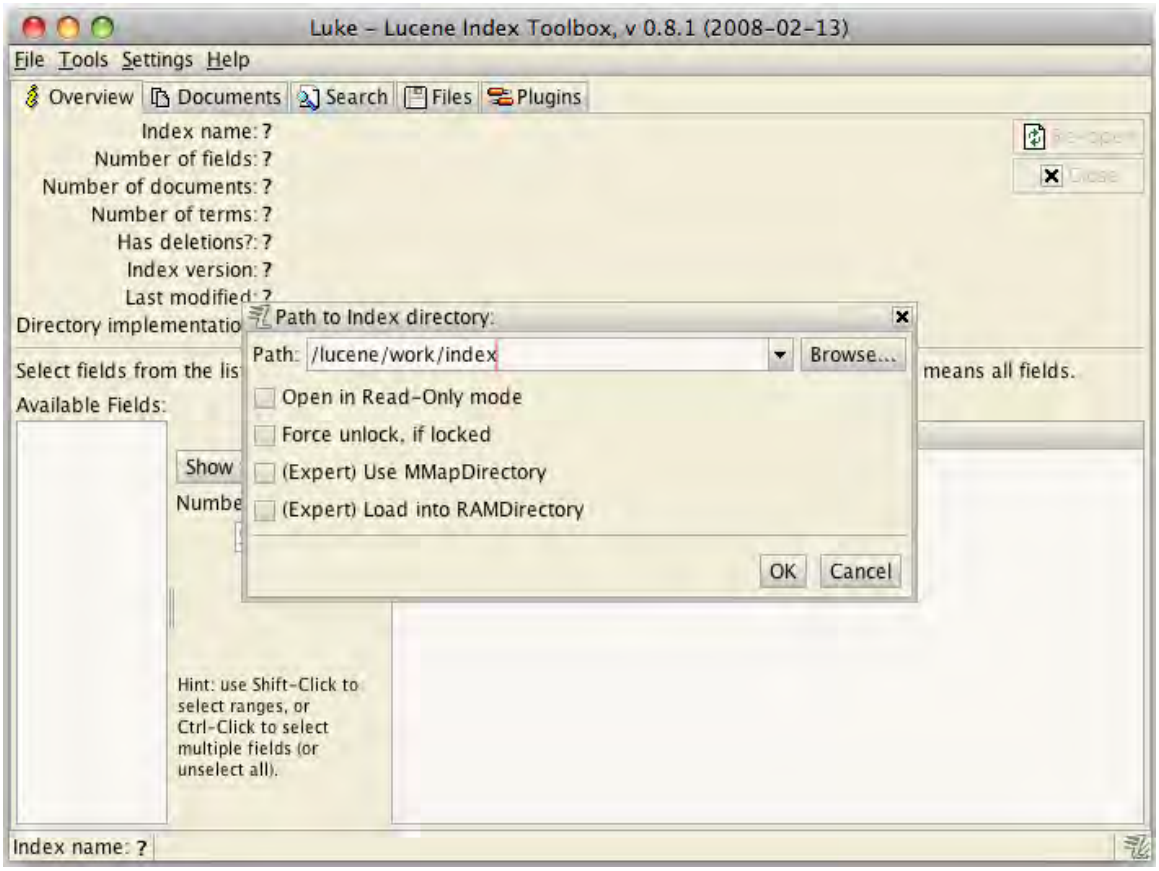


Figure 8.2 Luke: opening an index

Luke's interface is nicely interconnected so that you can jump from one view to another in the same context. The interface is divided into five tabs: Overview, Documents, Search, Files, and Plugins. The Tools menu provides options to optimize the current index, undelete any documents flagged for deletion, and switch the index between compound and standard format.

OVERVIEW: SEEING THE BIG PICTURE

Luke's Overview tab shows the major pieces of a Lucene index, including the number of fields, documents, and terms (figure 8.3). The top terms in one or more selected fields are shown in the "Top ranking terms" pane. Double-clicking a term opens the Documents tab for the selected term, where you can browse all documents containing that term. Right-clicking a term brings up a menu with three options: "Show all term docs" opens the Search tab for that term so all documents appear in a list, "Browse term docs"

opens the Documents tab for the selected term, and “Copy to clipboard” copies the term to the clipboard so you can then paste it elsewhere.

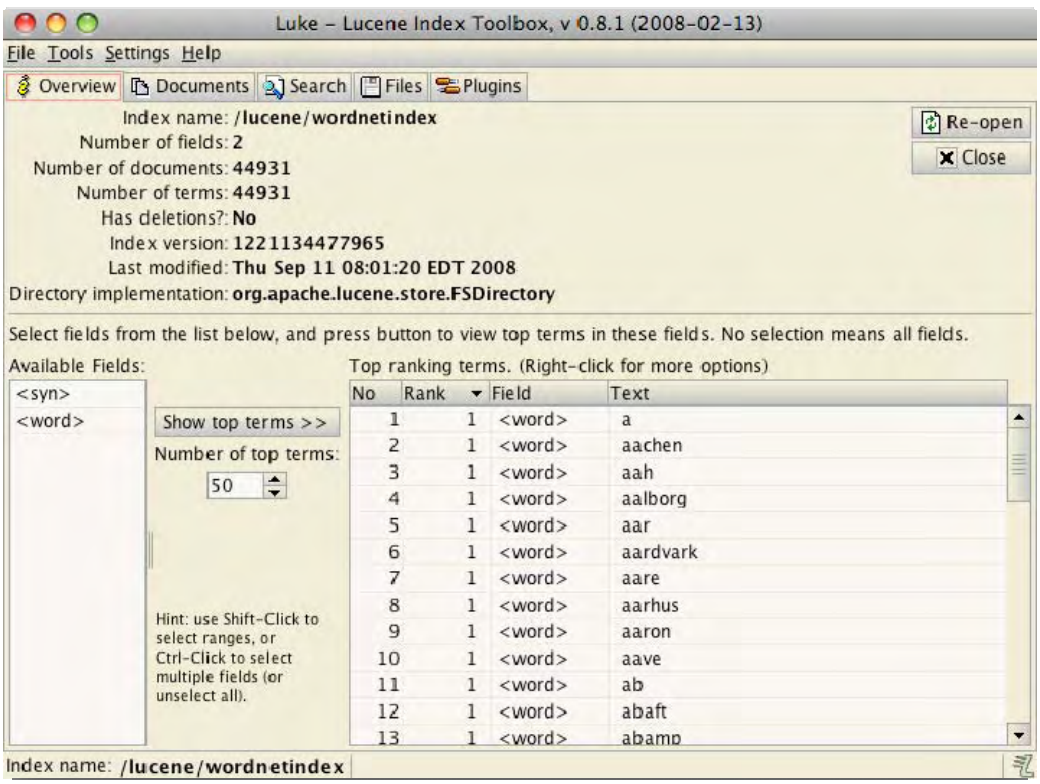


Figure 8.3 Luke: index overview, allowing you to browse fields and terms

DOCUMENT BROWSING

The Documents tab is Luke’s most sophisticated screen, where you can browse documents by document number and by term (see figure 8.4). Browsing by document number is straightforward; you can use the arrows to navigate through the documents sequentially. The table at the bottom of the screen shows all stored fields for the currently selected document.

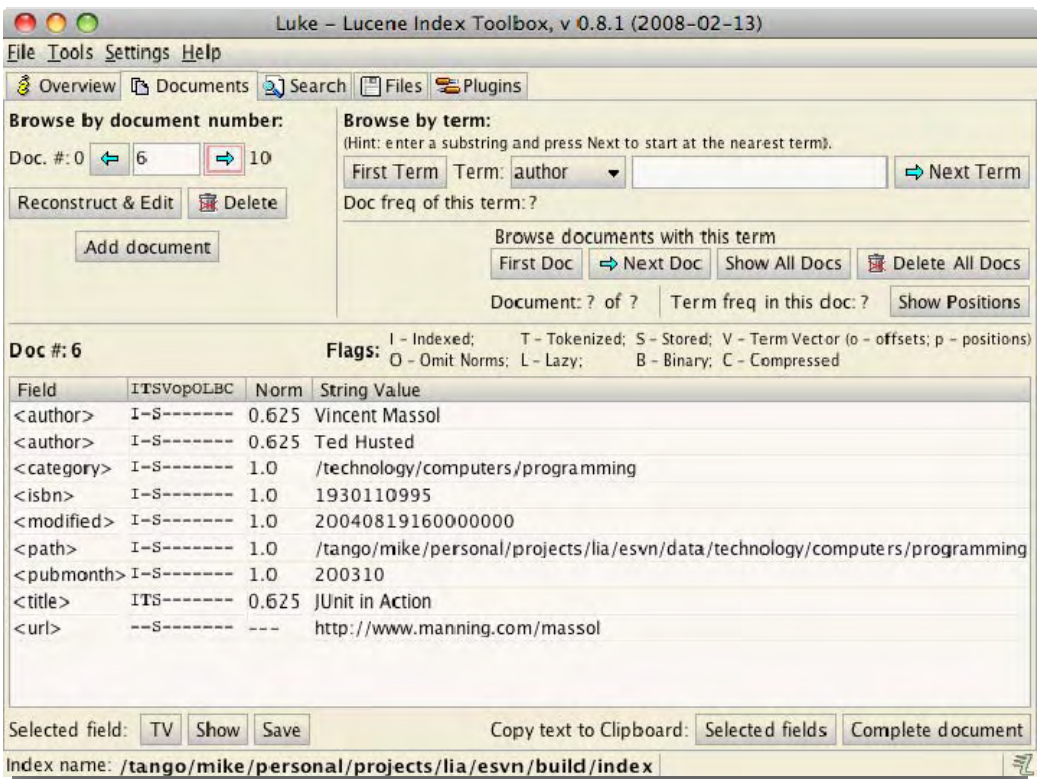


Figure 8.4 Luke's Documents tab: feel the power!

Browsing by term is trickier; you can go about it several ways. Clicking First Term navigates the term selection to the first term in the index. You can scroll through terms by clicking the Next Term button. The number of documents containing a given term is shown as the "Doc freq of this term" value. To select a specific term, type all but the last character in the text box, click Next Term, and navigate forward until you find the desired term.

Just below the term browser is the term document browser, which lets you navigate through the documents containing the term you selected. The First Doc button selects the first document that contains the selected term; and, as when you're browsing terms, Next Doc navigates forward.

The selected document, or all documents containing the selected term, can also be deleted from this screen (use caution if this is a production index, of course!).

Another feature of the Documents tab is the “Copy text to Clipboard” feature. All fields shown, or the selected fields, may be copied to the clipboard. For example, copying the entire document to the clipboard places the following text there:

```
stored/uncompressed,indexed<author:Erik Hatcher>
stored/uncompressed,indexed<author:Steve Loughran>
stored/uncompressed,indexed<category:/technology/computers/programming>
stored/uncompressed,indexed<isbn:1930110588>
stored/uncompressed,indexed<modified:20040819160000000>
stored/uncompressed,indexed<path:/tango/mike/personal/projects/lia/esvn/data/technology
/computers/programming/jdwa.properties>
stored/uncompressed,indexed<pubmonth:200208>
stored/uncompressed,indexed,tokenized<title:Java Development with Ant>
stored/uncompressedurl:http://www.manning.com/antbook
```

NOTE

Luke can only work within the constraints of a Lucene index, and unstored fields don’t have the text available in its original form. The terms of those fields, of course, are navigable with Luke, but those fields aren’t available in the document viewer or for copying to the clipboard (for example, our `contents` field in this case).

Clicking the *Show All Docs* button shifts the view to the Search tab with a search on the selected term, such that all documents containing this term are displayed. If a field’s term vectors have been stored, the Field’s Term Vector button displays a window showing terms and frequencies.

One final feature of the Documents tab is the “Reconstruct & Edit” button. Clicking this button opens a document editor allowing you to edit (delete and re-add) the document in the index or add a new document. Figure 8.5 shows a document being edited.

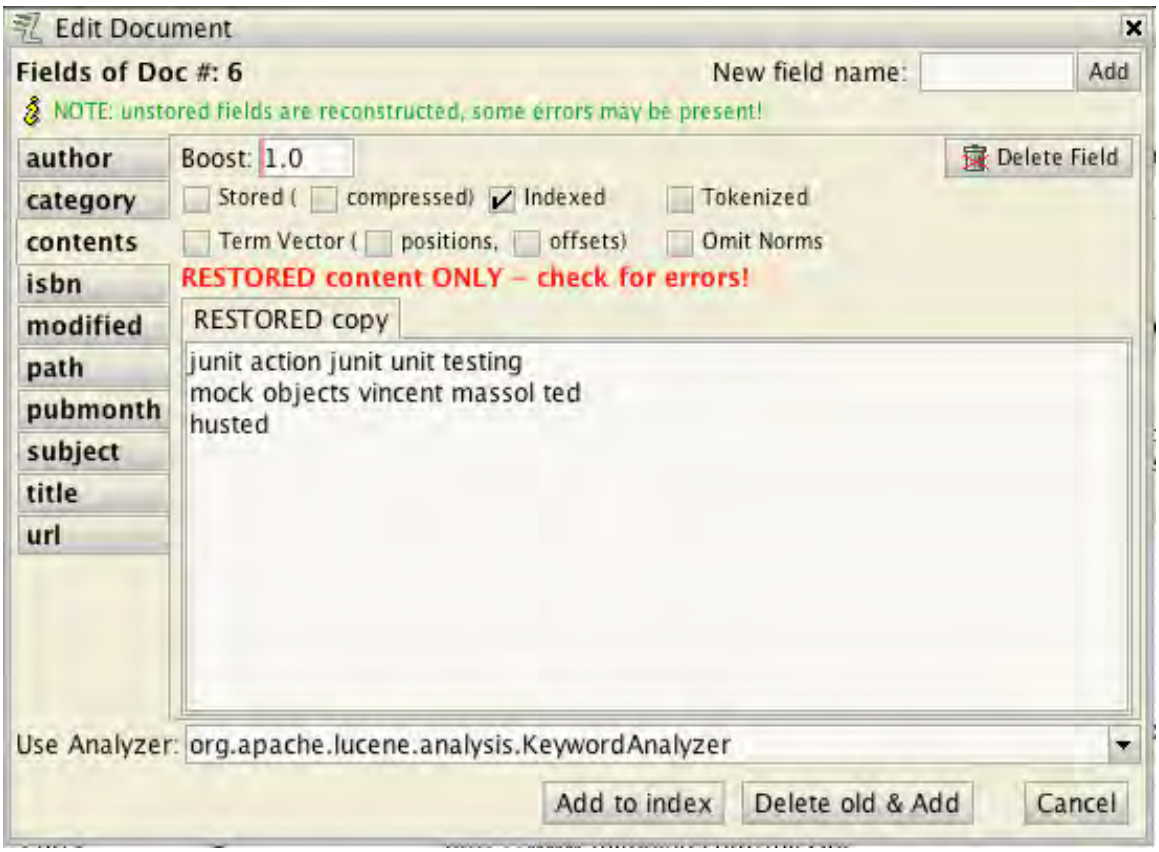


Figure 8.5 Document editor

Luke reconstructs fields that were tokenized but not stored, by aggregating in position order all the terms that were indexed. Reconstructing a field is a potentially lossy operation, and Luke warns of this when you view a reconstructed field (for example, if stop words were removed or tokens were stemmed during the analysis process then the original value cannot be reconstructed).

STILL SEARCHING OVER HERE, BOSS

We've already shown two ways to automatically arrive at the Search tab: choosing "Show all term docs" from the right-click menu of the "Top ranking terms" section of the Overview tab, and clicking Show All Docs from the term browser on the Documents tab.

You can also use the Search tab manually, entering `QueryParser` expression syntax along with your choice of `Analyzer` and default field. Click Search when the expression and other fields are as desired. The bottom table shows all the documents from the search hits, as shown in figure 8.6.

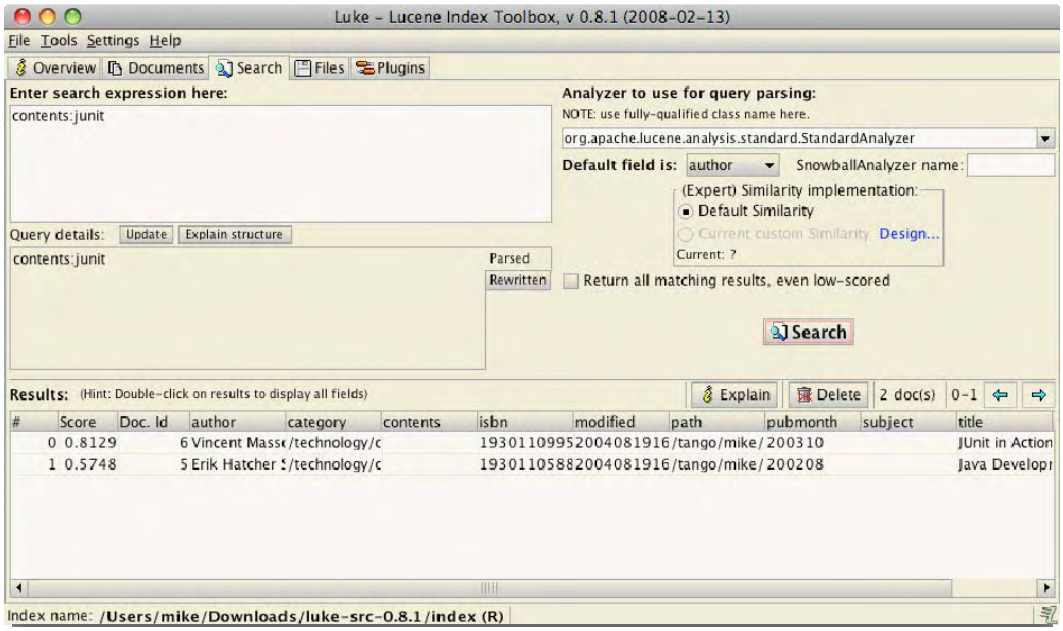


Figure 8.6 Searching: an easy way to experiment with `QueryParser`

Double-clicking a document shifts back to the Documents tab with the appropriate document preselected. It's useful to interactively experiment with search expressions and see how `QueryParser` reacts to them (but be sure to commit your assumptions to test cases, too!). Luke shows all analyzers it finds in the classpath, but only analyzers with no-arg constructors may be used with Luke. Luke also provides insight into document scoring with the explanation feature.

To view score explanation, select a result and click the Explanation button; an example is shown in figure 8.7.

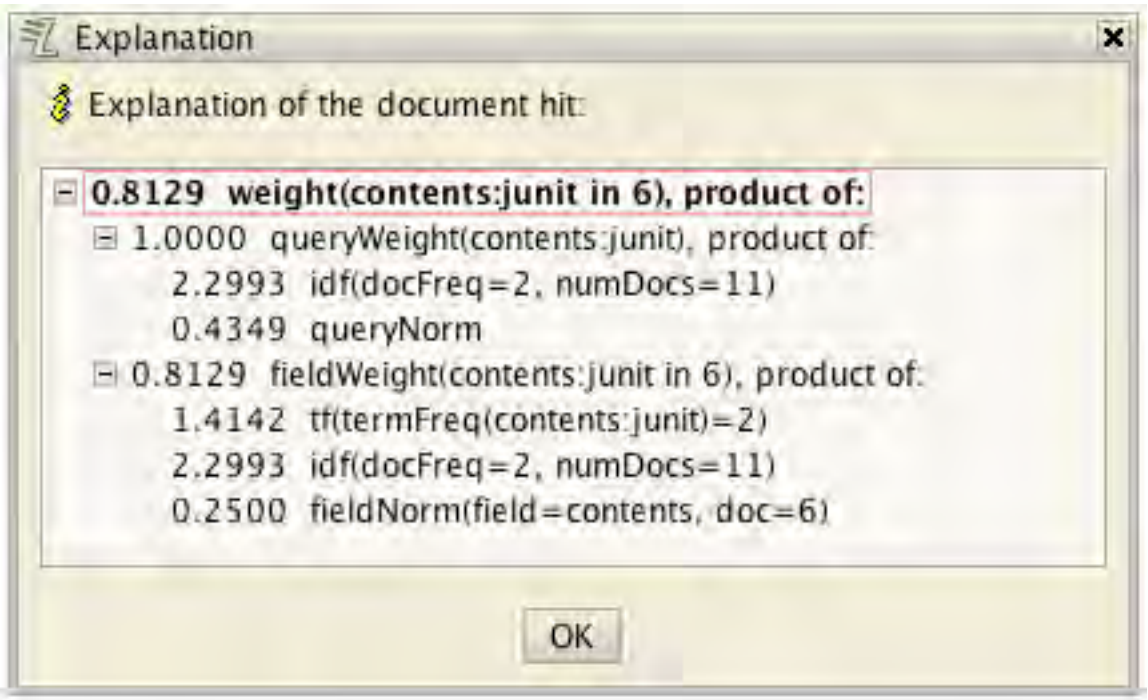


Figure 8.7 Lucene's scoring explanation

FILES VIEW

The final view in Luke displays the files (and their sizes) that make up the internals of a Lucene index directory. The total index size is also shown, as you can see in figure 8.8.

Figure 8.8 Luke's Files view shows how big an index is.

PLUGINS VIEW

As if the features already described about Luke weren't enough, Andrzej has gone the extra kilometer and added a plug-in framework so that others can add tools to Luke. Five plug-ins comes built in. Analyzer Tool has the same purpose as the `AnalyzerDemo` developed in section 4.2.3, showing the results of the analysis process on a block of text. As an added bonus, highlighting a selected token is a mere button-click away, as shown in figure 8.9. Scripting Luke lets interactively run JavaScript code accessing Luke's internals. Custom Similarity allows you to code up your own Similarity implementation in JavaScript, which is then compiled and accessible in the Search panel. Vocabulary Analysis Tool and Zipf distribution are two tools that show graphs of term statistics from the index.

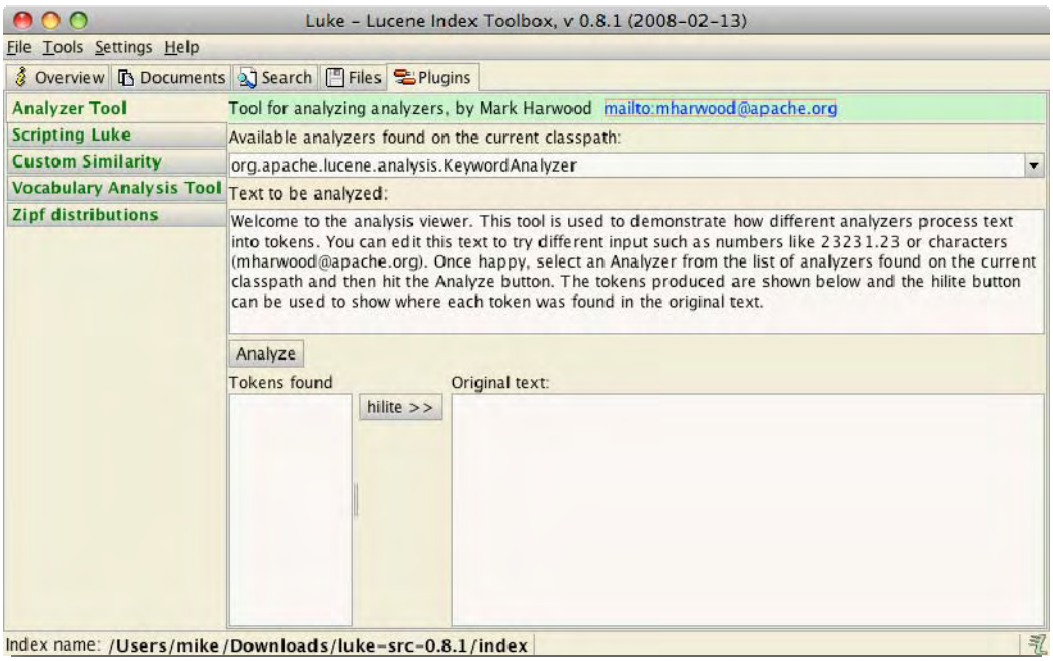


Figure 8.9 Builtin Luke plug-ins

Consult the Luke documentation and source code for information on how to develop your own plug-in. We conclude this section with a third tool, LIMO.

8.2.3 LIMO: Lucene Index Monitor

Julien Nioche is the creator of Lucene Index Monitor (LIMO).[2] It's available online at http://limo.sourceforge.net/. LIMO provides a web browser interface to Lucene indexes, giving you a quick look at index status information such as whether an index is locked, the last modification date, the number of documents, and a field summary. In addition, a rudimentary document browser lets you scroll through documents sequentially. When you cannot use Luke, because the index is on a remote server and not accessible to your local computer, LIMO is a good fallback since it runs as a webapp on the server.

Figure 8.10 shows the initial page, where you can select one or more preconfigured indexes.

[2] LIMO v0.61 is the most recent version at the time of this writing.

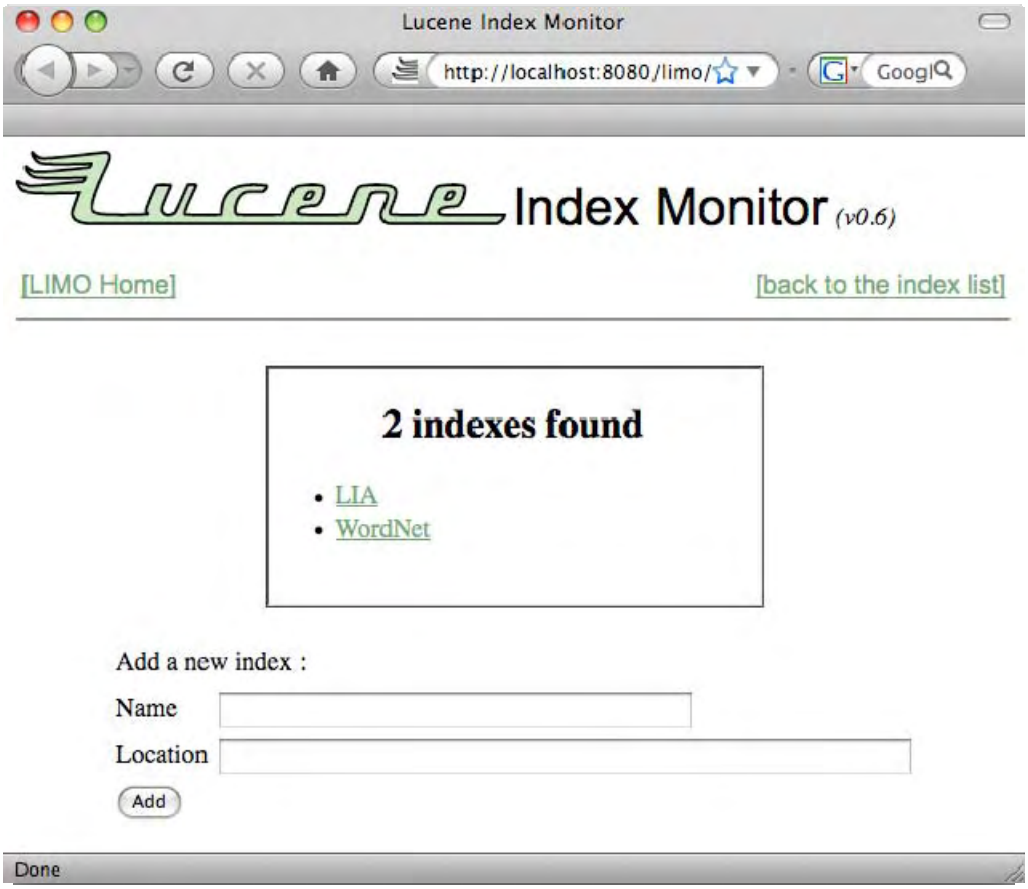


Figure 8.10 LIMO: selecting an index

To install LIMO, follow these steps:

1. Download the LIMO distribution, which is a WAR file.
2. Expand the WAR file in the Tomcat webapps/limo webapps directory.
3. Start the web container and point your browser to the appropriate URL (http://localhost:8080/limo, if you are using Tomcat and you're on the same computer)

The version of LIMO that we used embeds Lucene 2.0; if you need to use a newer version of Lucene than LIMO embeds, replace the Lucene JAR in WEB-INF/lib by removing the existing file and adding a newer one.

When you start, Limo presents the index selection page. The first step you must do is enter the path and name for the index you wish to browse. These paths are simply saved into the limo.properties file under webapps/limo. Once you've entered one or more indices, you can select the one you'd like to browse.

BROWSING AN INDEX

Clicking on an index brings you to LIMO's index summary and document browser view. Figure 8.11 shows a sample.

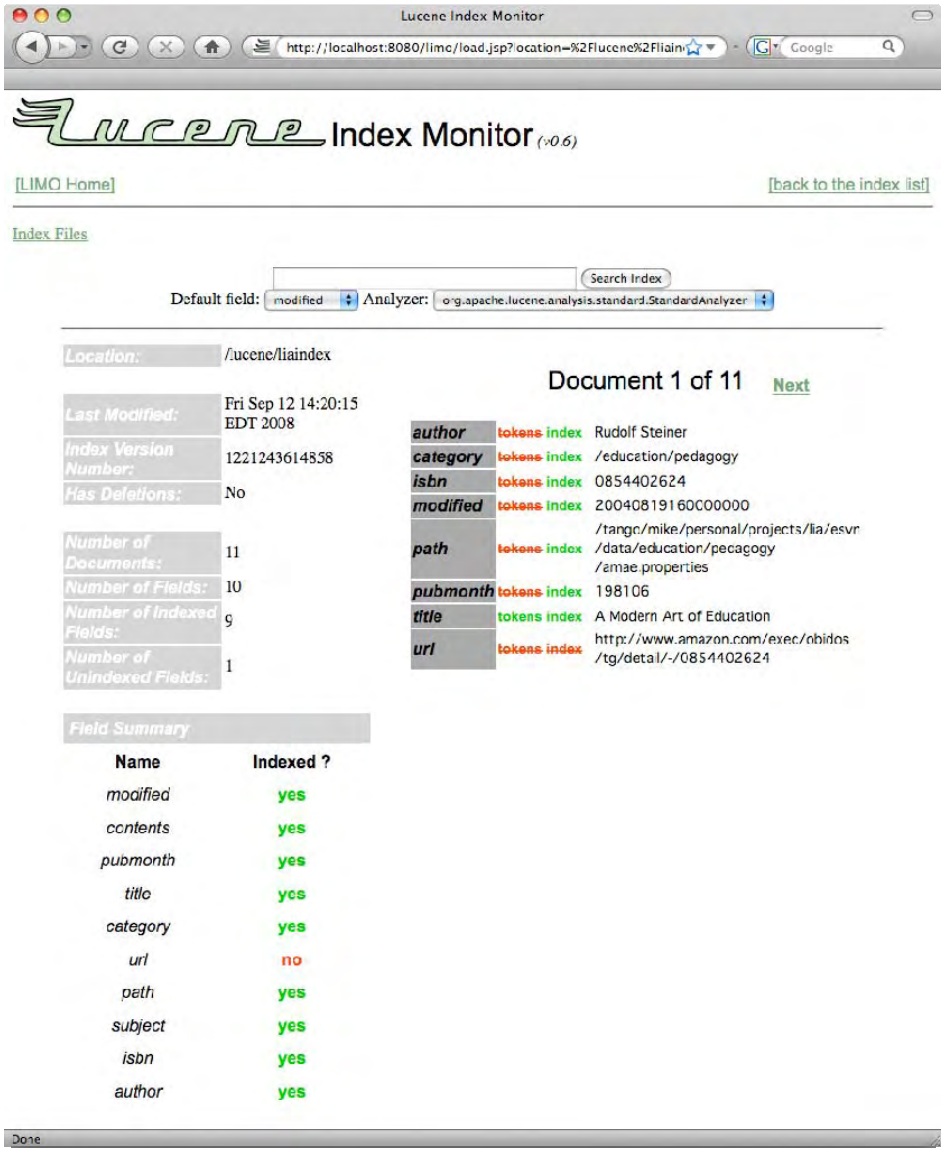


Figure 8.11 Cruising in the LIMO

Click the Prev and Next links to navigate through the documents. All the fields are shown on the right, indicating whether they are stored and/or indexed.

Next, you can run a search by entering your search text into the text box, selecting the default field to search as well as the analyzer you'd like to use, and then clicking the Search Index button. This brings you to the search results page, with highlighting, as shown in Figure 8.12. On the far right of each search hit are three links: view lets you view that document in the document browsing page; explain provides an explanation of the score for that hit, in the bottom left of the page; and reconstruct rebuilds the document from the index, in the bottom right of the page.

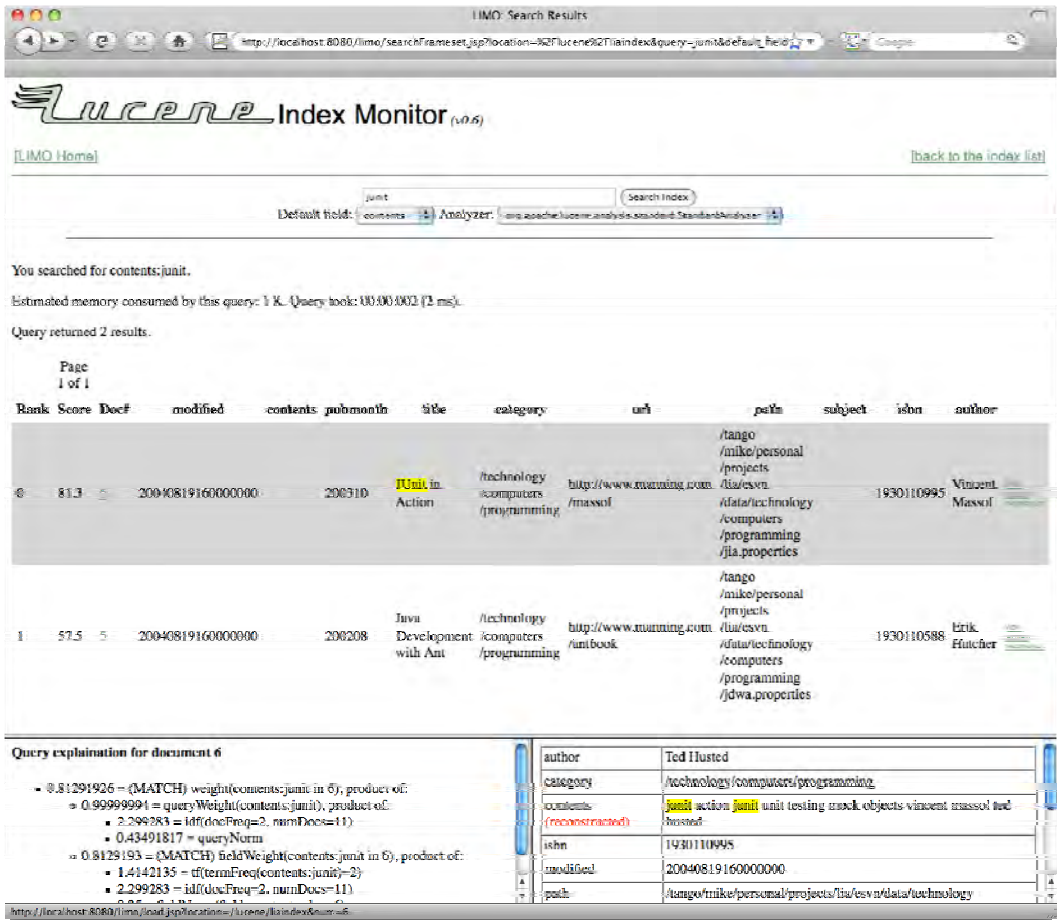


Figure 8.12 LIMO's search page

USING LIMO

LIMO's user interface isn't fancy, but it does the job. You may want to have LIMO installed on a secured Tomcat instance on a production server. Being able to get a quick view of how many documents are in an index, whether it's locked, and when it was last updated can be helpful for monitoring purposes. Also, using the LIMO JSP pages as a basis for building your own custom monitoring view could be a time saver. Because LIMO functions as a web application and doesn't allow any destructive operations on an index, it provides a handy and safe way to peek into a remote index.

We now switch to the numerous options in the sandbox for analysis.

8.3 Analyzers, tokenizers, and TokenFilters, oh my

The more analyzers, the merrier, we always say. And the Sandbox doesn't disappoint in this area: It houses numerous language-specific analyzers, a few related filters and tokenizers, and the slick Snowball algorithm analyzers. The analyzers are listed in table 8.2.

Table 8.2 Sandbox analyzers

Analyzer[3]	`TokenStream` flow
`org.apache.lucene.analysis.br.BrazilianAnalyzer`	`StandardTokenizer` ‹ `StandardFilter` ‹ `StopFilter` (custom stop table) ‹ `BrazilianStemFilter` ‹ `LowerCaseFilter`
`org.apache.lucene.analysis.cjk.CJKAnalyzer`	`CJKTokenizer` ‹ `StopFilter` (custom English stop words ironically)
`org.apache.lucene.analysis.cn.ChineseAnalyzer`	`ChineseTokenizer` ‹ `ChineseFilter`
`org.apache.lucene.analysis.cz.CzechAnalyzer`	`StandardTokenizer` ‹ `StandardFilter` ‹ `LowerCaseFilter` ‹ `StopFilter` (custom stop list)
`Org.apache.lucene.analysis.de.GermanAnalyzer`	`StandardTokenizer` ‹ `StandardFilter` ‹ `LowerCaseFilter` ‹ `StopFilter` (custom stop list) > GermanStemFilter

[3] *Note the different package name for the `SnowballAnalyzer`—it is housed in a different sandbox directory than the others.*

Org.apache.lucene.analysis.el.GreekAnalyzer	StandardTokenizer > GreekLowerCaseFilter > StopFilter (custom stop list)
Org.apache.lucene.analysis.ngram.*	Breaks characters of a single word into a series of character ngrams. This can be useful for spell correction and live auto completion.
org.apache.lucene.analysis.nl.DutchAnalyzer	StandardTokenizer ‹ StandardFilter ‹ StopFilter (custom stop table)‹ DutchStemFilter
org.apache.lucene.analysis.fr.FrenchAnalyzer	StandardTokenizer ‹ StandardFilter ‹ StopFilter (custom stop table)‹ FrenchStemFilter ‹ LowerCaseFilter
Org.apache.lucene.analysis.ru.RussianAnalyzer	RussianLetterTokenizer > RussianLowerCaseFilter > StopFilter (custom stop list) > RussianStemFilter
Org.apache.lucene.analysis.th.ThaiAnalyzer	StandardFilter > ThaiWordFilter > StopFilter (English stop words)
Org.apache.lucene.analysis.compound.*	Two different TokenFilters that decompose compound words you find in many Germanic languages to the word parts. There are two approaches (one using hyphenation based grammar to detect word parts; the other using a word-based dictionary).
Org.apache.lucene.wikipedia.analysis.WikipediaTokenizer	Similar to StandardTokenizer, except it adds further specialization to process the Wikipedia-specific tokens that appear in the XML export of the Wikipedia corpus
Org.apache.lucene.analysis.sinks.DateRecognizerSinkTokenizer.java	A SinkTokenizer (see Section 4.XXX) that only accepts tokens that are valid dates (using java.text.DateFormat)
Org.apache.lucene.analysis.sinks.TokenRangeSinkTokenizer	A SinkTokenizer (see Section 4.XXX) that only accepts tokens within a certain range

Org.apache.lucene.analysis.sinks.TokenTypeSinkTokenizer	A SinkTokenizer (see Section 4.XXX) that only accepts tokens of a specific type as returned by Token.type()
Org.apache.lucene.analysis.shingle.*	Tokenizers that create shingles (n-grams from multiple tokens) from another TokenStream
Org.apache.lucene.analysis.payloads.*	TokenFilters that carry over token attributes as payloads; these are described in section 6.5.
org.apache.lucene.analysis.snowball.SnowballAnalyzer	StandardTokenizer ‹ StandardFilter ‹ LowerCaseFilter [‹ StopFilter] ‹ SnowballFilter

The language-specific analyzers vary in how they tokenize. The Brazilian and French analyzers use language-specific stemming and custom stop-word lists. The Czech analyzer uses standard tokenization, but also incorporates a custom stop word list. The Chinese and CJK (Chinese-Japanese-Korean) analyzers tokenize double-byte characters as a single token to keep a logical character intact. We demonstrate analysis of Chinese characters in section 4.8.3, illustrating how these two analyzers work.

Each of these analyzers, including the SnowballAnalyzer discussed in the next section, lets you customize the stop-word list just as the StopAnalyzer does (see section 4.3.1). Most of these analyzers do quite a bit in the filtering process. If the stemming or tokenization is all you need, borrow the relevant pieces, and construct your own custom analyzer from the parts here. Section 4.6 covers creating custom analyzers.

We'll give special attention here to the snowball analyzers and shingle and ngram filters.

8.3.1 SnowballAnalyzer

The SnowballAnalyzer deserves special mention because it serves as a driver of an entire family of stemmers for different languages. Stemming was first introduced in section 4.7. Dr. Martin Porter, who also developed the Porter stemming algorithm, created the Snowball algorithm.[4] The Porter algorithm was designed for English only; in addition, many "purported" implementations don't adhere to the definition faithfully.[5] To address these issues, Dr. Porter rigorously defined the Snowball system of stemming

[4]The name *Snowball* is a tribute to the string-manipulation language SNOBOL.

[5]From http://snowball.tartarus.org/texts/introduction.html

algorithms. Through these algorithmic definitions, accurate implementations can be generated. In fact, the snowball project in Lucene's Sandbox has a build process that can pull the definitions from Dr. Porter's site and generate the Java implementation.

One of the test cases demonstrates the result of the English stemmer stripping off the trailing *ming* from *stemming* and the *s* from *algorithms*:

```
public void testEnglish() throws Exception {
  Analyzer analyzer = new SnowballAnalyzer("English");

  assertAnalyzesTo(analyzer,
      "stemming algorithms", new String[] {"stem", "algorithm"});
}
```

`SnowballAnalyzer` has two constructors; both accept the stemmer name, and one specifies a `String[]` stop-word list to use. Many unique stemmers exist for various languages. The non-English stemmers include Danish, Dutch, Finnish, French, German, German2, Hungarian, Italian, Kp (Kraaij-Pohlmann algorithm for Dutch), Norwegian, Portuguese, Romanian, Russian, Spanish, Swedish and Turkish. There are a few English-specific stemmers named English, Lovins, and Porter. These exact names are the valid argument values for the name argument to the `SnowballAnalyzer` constructors. Here is an example using the Spanish stemming algorithm:

```
public void testSpanish() throws Exception {
  Analyzer analyzer = new SnowballAnalyzer("Spanish");

  assertAnalyzesTo(analyzer,
      "algoritmos", new String[] {"algoritm"});
}
```

If your project demands stemming, we recommend that you give the Snowball analyzer your attention first since an expert in the stemming field developed it. And, as already mentioned but worth repeating, you may want to use the clever piece of this analyzer (the `SnowballFilter`) wrapped in your own custom analyzer implementation. Several sections in chapter 4 discuss writing custom analyzers in great detail.

8.3.2 Shingle filters

Shingles are single tokens constructed from multiple adjacent tokens. They are similar to letter ngrams, used by the spellchecker package (Section 8.11) and the ngram tokenizers (Section 8.3.3) in that they make new tokens by combining multiple adjacent things. However, while the ngram tokenizers operate on letters, shingles operate on whole words. For example the sentence "please divide this sentence into shingles" might be tokenized into the shingles "please divide", "divide this", "this sentence", "sentence into" and "into shingles".

Why would you ever want to do such a thing? One common reason is to speed up phrase searches, especially for phrases involving common terms. For example, consider a search for the exact phrase "Wizard of Oz". Since the word "of" is incredibly common, including it in the phrase search will require

Lucene to visit and filter out a great many occurrences that do not match the phrase, which is costly. If, instead, you had indexed the tokens “wizard of” and “of oz”, those tokens occur far less frequently and your phrase search can run very quickly. The Nutch search engine, covered in Section 4.9, creates shingles for exactly this reason. You could also simply discard such common terms as stop words, using a StopFilter during analysis. But then you must take care to add slop into your phrase searches, as covered in section 4.7.3. Unlike shingles which correctly provide an exact match to your search, the slop solution is able to match other phrases.

Another interesting use of shingles is document clustering in order to group similar or near-duplicate documents together. This is important for large collections of documents where duplicates may accidentally sneak in, which happens frequently when crawling for content through Web servers that construct documents dynamically. Often slightly different URLs can yield the same underlying content, perhaps with a different header added in. Much like using term vectors to find similar documents (Section XXX) the approach is to represent each document by its salient shingles and then search for other documents that have similar shingles with similar frequency.

Listing 8.2 Using NGram filters

```
public class NGramTest extends LiaTestCase {

  private static class NGramAnalyzer extends Analyzer {
    public TokenStream tokenStream(String fieldName, Reader reader) {
      return new NGramTokenFilter(new KeywordTokenizer(reader), 2, 4);
    }
  }

  private static class FrontEdgeNGramAnalyzer extends Analyzer {
    public TokenStream tokenStream(String fieldName, Reader reader) {
      return new EdgeNGramTokenFilter(new KeywordTokenizer(reader),
EdgeNGramTokenFilter.Side.FRONT, 1, 4);
    }
  }

  private static class BackEdgeNGramAnalyzer extends Analyzer {
    public TokenStream tokenStream(String fieldName, Reader reader) {
      return new EdgeNGramTokenFilter(new KeywordTokenizer(reader),
EdgeNGramTokenFilter.Side.BACK, 1, 4);
    }
  }

  public void testNGramTokenFilter24() throws IOException {
    AnalyzerUtils.displayTokensWithPositions(new NGramAnalyzer(), "lettuce");
  }

  public void testEdgeNGramTokenFilterFront() throws IOException {
    AnalyzerUtils.displayTokensWithPositions(new FrontEdgeNGramAnalyzer(), "lettuce");
  }

  public void testEdgeNGramTokenFilterBack() throws IOException {
    AnalyzerUtils.displayTokensWithPositions(new BackEdgeNGramAnalyzer(), "lettuce");
  }
```

```
}
```

8.3.3 Ngram filters

The ngram filters take a single token and emit a series of letter ngram tokens, which are combinations of N adjacent letters into a single token. Listing 8.2 shows how to use these unusual filters. The `testNGramTokenFilter24` method creates an `NGramTokenFilter` to generate all letter ngrams of length 2, 3 or 4, on the word "lettuce". The resulting output is this:

```
1: [le]
2: [et]
3: [tt]
4: [tu]
5: [uc]
6: [ce]
7: [let]
8: [ett]
9: [ttu]
10: [tuc]
11: [uce]
12: [lett]
13: [ettu]
14: [ttuc]
15: [tuce]
```

Note that each larger ngram series is positioned after the previous series. A more natural approach would be to have the ngram's position be set to the character position where it had started in the word, but unfortunately at this time there's no option to do this (it is however a known limitation, so by the time you read this it may be fixed).

The `EdgeNGramFilter` is similar, except it only generates ngrams anchored to the start or end of the word. Here's the output of the `testEdgeNGramTokenFilterFront`:

```
1: [l]
2: [le]
3: [let]
4: [lett]
```

And `testEdgeNGramTokenFilterBack`:

```
1: [e]
2: [ce]
3: [uce]
4: [tuce]
```

8.3.4 Obtaining the Sandbox analyzers

Depending on your needs, you may want JAR binary distributions of these analyzers or raw source code from which to borrow ideas. Section 8.10 provides details on how to access the Sandbox SVN repository

and how to build binary distributions. Within the repository, the Snowball analyzer resides in contrib/snowball; the other analyzers discussed here are in contrib/analyzers. There are no external dependencies for these analyzers other than Lucene itself, so they are easy to incorporate. A test program called TestApp is included for the Snowball project. It's run in this manner:

```
> java –cp dist/snowball.jar net.sf.snowball.TestApp
Usage: TestApp <stemmer name> <input file> [-o <output file>]

> java -cp dist/snowball.jar
ž  net.sf.snowball.TestApp Lovins spoonful.txt
... output of stemmer applied to specified file
```

The Snowball TestApp bypasses SnowballAnalyzer. Only the Snowball stemmer itself is used with rudimentary text splitting at whitespace.

Our next package extends ant with tasks to control indexing.

8.4 Java Development with Ant and Lucene

A natural integration point with Lucene incorporates document indexing into a build process. As part of *Java Development with Ant* (Hatcher and Loughran, Manning Publications, 2002), Erik created an Ant task to index a directory of file-based documents. This code has since been enhanced and is maintained in the Sandbox.

Why index documents during a build process? Imagine a project that is providing an embedded help system with search capability. The documents are probably static for a particular version of the system, and having a read-only index created at build-time fits perfectly. For example, what if the Ant, Lucene, and other projects had a domain-specific search on their respective web sites? It makes sense for the searchable documentation to be the latest release version; it doesn't need to be dynamically updated.

We first describe the <index> task, then show how to create a custom document handler, and finally show how to install this package.

8.4.1 Using the <index> task

Listing 8.2 shows a simplistic Ant 1.6.x–compatible build file that indexes a directory of text and HTML files.

Listing 8.3 using the Ant <index> task

```
<?xml version="1.0"?>
<project name="ant-example" default="index">

  <description>
    Lucene Ant index example
  </description>

  <property name="index.base.dir" location="build"/>    |#1
  <property name="files.dir" location="."/>             |#2

  <target name="index">
    <mkdir dir="${index.base.dir}"/>
```

```
    <index index="${index.base.dir}/index"
           xmlns="antlib:org.apache.lucene.ant">
      <fileset dir="${files.dir}"/>
    </index>
  </target>

</project>
```

#1 Parent of index directory
#2 Root directory of documents to index

The Ant integration is Ant 1.6 Antlib compatible, as seen with the `xmlns` specification. The legacy `<taskdef>` method can still be used, too. Listing 8.2 shows the most basic usage of the `<index>` task, minimally requiring specification of the index directory and a fileset of files to consider for indexing. The default file-handling mechanism indexes only files that end with .txt or .html.[6] Table 8.3 lists the fields created by the index task and the default document handler. Only `path` and `modified` are fixed fields; the others come from the document handler.

Table 8.3 `<index>` task default fields

Field name	Field type	Comments
path	Keyword	Absolute path to a file
modified	`Keyword` (as `Date`)	Last-modified date of a file
title	Text	`<title>` in HTML files; and filename for .txt files.
Contents	Text	Complete contents of .txt files; parsed `<body>` of HTML files
rawcontents	UnIndexed	Raw contents of the file

It's very likely that the default document handler is insufficient for your needs. Fortunately, a custom document handler extension point exists.

[6] JTidy is currently used to extract HTML content for indexing. See section 7.4 for more on indexing HTML.

8.4.2 Creating a custom document handler

A swappable document-handler facility is built into the <index> task, allowing custom implementations to handle different document types and control the Lucene fields created.[7] Not only can the document handler be specified, configuration parameters can be passed to the custom document handler. We used the Ant <index> task, as shown in listing 8.3, to build the index used in the majority of the code for this book.

Listing 8.4 Use of the <index> task to build the sample index for this book

```
<target name="build-index" depends="compile">
  <typedef resource="org/apache/lucene/ant/antlib.xml"> |#1
    <classpath>
    |#1
      <path refid="compile.classpath"/>
    |#1
      <pathelement location="${build.dir}/classes"/>
    |#1
    </classpath>
    |#1
  </typedef>
    |#1

  <index index="${build.dir}/index"
         documenthandler="lia.common.TestDataDocumentHandler">   |#2
    <fileset dir="${data.dir}"/>

    <config basedir="${data.dir}"/>                      |#3
  </index>
</target>
```

#1 <typedef>
#2 Use custom document handler
#3 basedir configuration property

#1 We use <typedef> because we need an additional dependency added to the classpath for our document handler. If we didn't need a custom document handler, the <typedef> would be unnecessary.

#2 We use a custom document handler to process files differently.

#3 Here we hand our document handler a configuration property, basedir. This allows relative paths to be extracted cleanly.

[7] The <index> task document handler facility was developed long before the framework Otis built in chapter 7. At this point, the two document-handling frameworks are independent of one another, although they're similar and can be easily merged.

The directory, referred to as ${data.dir}, contains a hierarchy of folders and .properties files. Each .properties file contains information about a single book, as in this example:

```
title=Tao Te Ching \u9053\u5FB7\u7D93
isbn=0060812451
author=Stephen Mitchell
subject=taoism
pubmonth=198810
url=http://www.amazon.com/exec/obidos/tg/detail/-/0060812451
```

The folder hierarchy serves as meta-data also, specifying the book categories. Figure 8.12 shows the sample data directory. For example, the .properties example just shown is the ttc.properties file that resides in the data/philosophy/eastern directory. The base directory points to data and is stripped off in the document handler as shown in listing 8.4.

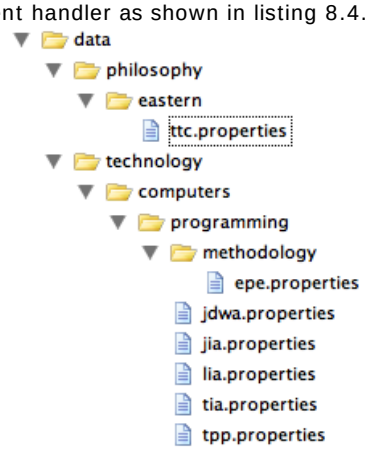


Figure 8.13 Sample data directory structure, with the file path specifying a category

To write a custom document handler, pick one of the two interfaces to implement. If you don't need any additional meta-data from the Ant build file, implement `DocumentHandler`, which has the following single method returning a Lucene `Document` instance:

```
public interface DocumentHandler {
  Document getDocument(File file)
           throws DocumentHandlerException;
}
```

Implementing `ConfigurableDocumentHandler` allows the `<index>` task to pass additional information as a `java.util.Properties` object:

```
public interface ConfigurableDocumentHandler
                 extends DocumentHandler {
  void configure(Properties props);
}
```

Configuration options are passed using a single `<config>` subelement with arbitrarily named attributes. The `<config>` attribute names become the keys to the properties. Our complete `TestDataDocumentHandler` class is shown in listing 8.4.

Listing 8.5 TestDataDocumentHandler: how we built our sample index

```
public class TestDataDocumentHandler
             implements ConfigurableDocumentHandler {
  private String basedir;

  public Document getDocument(File file)
           throws DocumentHandlerException {
    Properties props = new Properties();
    try {
      props.load(new FileInputStream(file));
    } catch (IOException e) {
      throw new DocumentHandlerException(e);
    }

    Document doc = new Document();

    // category comes from relative path below the base directory
    String category = file.getParent().substring(basedir.length());   |#1
    category = category.replace(File.separatorChar,'/');               |#1


    String isbn = props.getProperty("isbn");             |#2
    String title = props.getProperty("title");           |#2
    String author = props.getProperty("author");         |#2
    String url = props.getProperty("url");               |#2
    String subject = props.getProperty("subject");       |#2
    String pubmonth = props.getProperty("pubmonth");     |#2

    doc.add(Field.Keyword("isbn", isbn));                          |#3
    doc.add(Field.Keyword("category", category));                  |#3
    doc.add(Field.Text("title", title));                           |#3

    // split multiple authors into unique field instances
         |#3
    String[] authors = author.split(",");                          |#3
    for (int i = 0; i < authors.length; i++) {                     |#3
      doc.add(Field.Keyword("author", authors[i]));|#3
    }
|#3

    doc.add(Field.UnIndexed("url", url));                          |#3
    doc.add(Field.UnStored("subject", subject, true));             |#3  |#4


    doc.add(Field.Keyword("pubmonth", pubmonth));                  |#3

    doc.add(Field.UnStored("contents",                             |#3  |#5
        aggregate(new String[] { title, subject, author})));       |#3  |#5
```

```
    return doc;
  }
  private String aggregate(String[] strings) {
    StringBuffer buffer = new StringBuffer();

    for (int i = 0; i < strings.length; i++) {
      buffer.append(strings[i]);
      buffer.append(" ");
    }

    return buffer.toString();
  }

  public void configure(Properties props) {
    this.basedir = props.getProperty("basedir");
  }
}
```

#1 Get category
#2 Pull fields
#3 Add fields to Document instance
#4 Flag subject field
#5 Add contents field

#1 We base the category on the relative path from the base data directory, ensuring that forward slashes are used as separators.

#2 Here we pull each field from the values in the .properties file.

#3 We add each field to the `Document` instance; note the different types of fields used.

#4 The `subject` field is flagged for term vector storage.

#5 The `contents` field is an aggregate field: We can search a single field containing both the author and subject.

When you use a custom document handler, in addition to the fields the handler creates, the `<index>` task automatically adds `path` and `modified` fields. These two fields are used for incremental indexing, allowing only newly modified files to be processed.

The build file can also control the analyzer and merge factor. The merge factor defaults to 20, but you can set it to another value by specifying `mergeFactor= "..."` as an attribute to the `<index>` task. The analyzer is specified in one of two ways. The built-in analyzers are available using `analyzer="..."`, where the value is `simple`, `standard`, `stop`, `whitespace`, `german`, or `russian`. If you need to use any other analyzer, specify `analyzerClass="..."` instead, with the fully qualified class name. Currently, only analyzers that have a no-argument constructor can be used with `<index>`; this rules out using the `SnowballAnalyzer` directly, for example.

There are several interesting possibilities, thanks to the flexibility of the `<index>` task, such as indexing documentation in multiple languages. You may have documents separated by directory structure (`docs/en`, `docs/fr`, `docs/nl`, and so on), by filename (`index.html.en`, `index.html.fr`, and so

on), or by some other scheme. You could use the <index> task multiple times in a build process to build a separate index for each language, or you could write them all to the same index and use a different analyzer for each language.

8.4.3 Installation

The <index> task requires three libraries and at least Ant 1.5.4 (although Ant 1.6 or higher is recommended to take advantage of the Antlib feature). The Lucene JAR, JTidy's JAR, and the JAR of the <index> task itself are required. Obtain these JARs, place them in a single directory together, and use the -lib Ant 1.6 command-line switch to point to this directory (or use <taskdef> with the proper classpath). See section 8.10 for elaboration on how to obtain JARs from the Sandbox component, and refer to Ant's documentation and Manning's *Java Development with Ant* for specifics on working with Ant.

Let's see next how to build a search UI with the JavaScript utilities sandbox package.

8.5 JavaScript browser utilities

Integrating Lucene into an application often requires placing a search interface in a web application. QueryParser is handy, and it's easy to expose a simple text box allowing the user to enter a query; but it can be friendlier for users to see query options separated into fields, such as a date-range selection in conjunction with a text box for free-text searching. The JavaScript utilities in the Sandbox assist with browser-side usability in constructing and validating sophisticated expressions suitable for QueryParser. We'll describe how a query is created and validated, how to escape special characters, and finally how to use the JavaScript sources.

8.5.1 JavaScript query construction and validation

As we've explored in several previous chapters, exposing QueryParser directly to end users can lead to confusion. If you're providing a web interface to search a Lucene index, you may want to consider using the nicely done JavaScript query constructor and validator in the Sandbox, originally written by fellow Lucene developer Kelvin Tan. The javascript Sandbox project includes a sample HTML file that mimics Google's advanced searching options, as shown in figure 8.14.

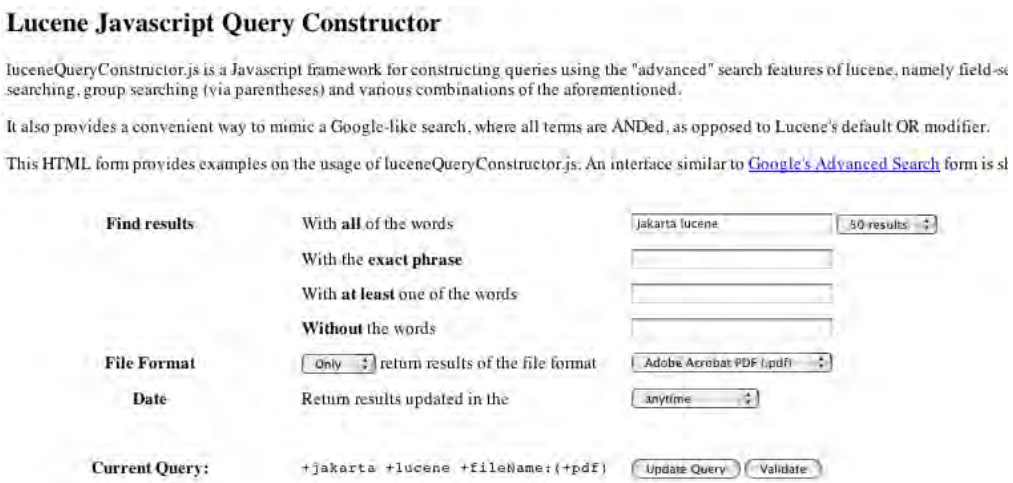


Figure 8.14 JavaScript example

The query constructor supports all HTML fields including text and hidden fields, radio buttons, and single and multiple selects. Each HTML field must have a corresponding HTML field named with the suffix *Modifier*, controlling how the terms are added to the query. The modifier field can be a hidden field to prevent a user from controlling it, as in the case of the text fields in figure 8.12. The constructed query is placed in an HTML field (typically a hidden one), which is handed to `QueryParser` on the server side.

The query validator uses regular expressions to do its best approximation of what is acceptable to `QueryParser`. Both JavaScript files allow customization with features like debug mode to alert you to what is happening, modifier field suffixes, specifying whether to submit the form upon construction, and more. The Java-Script files are well documented and easy to drop into your own environment.

At the time of this writing, the javascript Sandbox was being enhanced. Rather than show potentially out-of-date HTML, we refer you to the examples in the Sandbox when you need this capability.

8.5.2 Escaping special characters

`QueryParser` uses many special characters for operators and grouping. The characters must be escaped if they're used in a field name or as part of a term (see section 3.5 for more details on `QueryParser` escape characters). Using the luceneQueryEscaper.js support from the Sandbox, you can escape a query string.

You should use the query escaper only on fields or strings that should not contain any Lucene special characters already. For example, it would be incorrect to escape a query built with the query constructor, since any parentheses and operators it added would be subsequently escaped.

8.5.3 Using JavaScript support

Adding JavaScript support to your HTML file only requires grabbing the JavaScript files (see section 8.10) and referring to them in the <head> section in this manner:

```
<script type="text/javascript"
        src="luceneQueryConstructor.js"></script>
<script type="text/javascript"
        src="luceneQueryValidator.js"></script>
<script type="text/javascript" src="luceneQueryEscaper.js"></script>
```

Call `doMakeQuery` to construct a query and `doCheckLuceneQuery` to validate a query. Both methods require a form field argument that specifies which field to populate or validate. To escape a query, call `doEscapeQuery` with the form field or a text string (it detects the type); the escaped query string will be returned.

8.6 Synonyms from WordNet

What a tangled web of words we weave. A system developed at Princeton University's Cognitive Science Laboratory, driven by Psychology Professor George Miller, illustrates the net of synonyms.[8] WordNet represents word forms that are interchangeable, both lexically and semantically. Google's define feature (type **define: *word*** as a Google search, and see for yourself) often refers users to the online WordNet system, allowing you to navigate word interconnections. Figure 8.14 shows the results of searching for *search* at the WordNet site.

[8] Interestingly, this is the same George Miller who reported on the phenomenon of seven plus or minus two chunks in immediate memory.

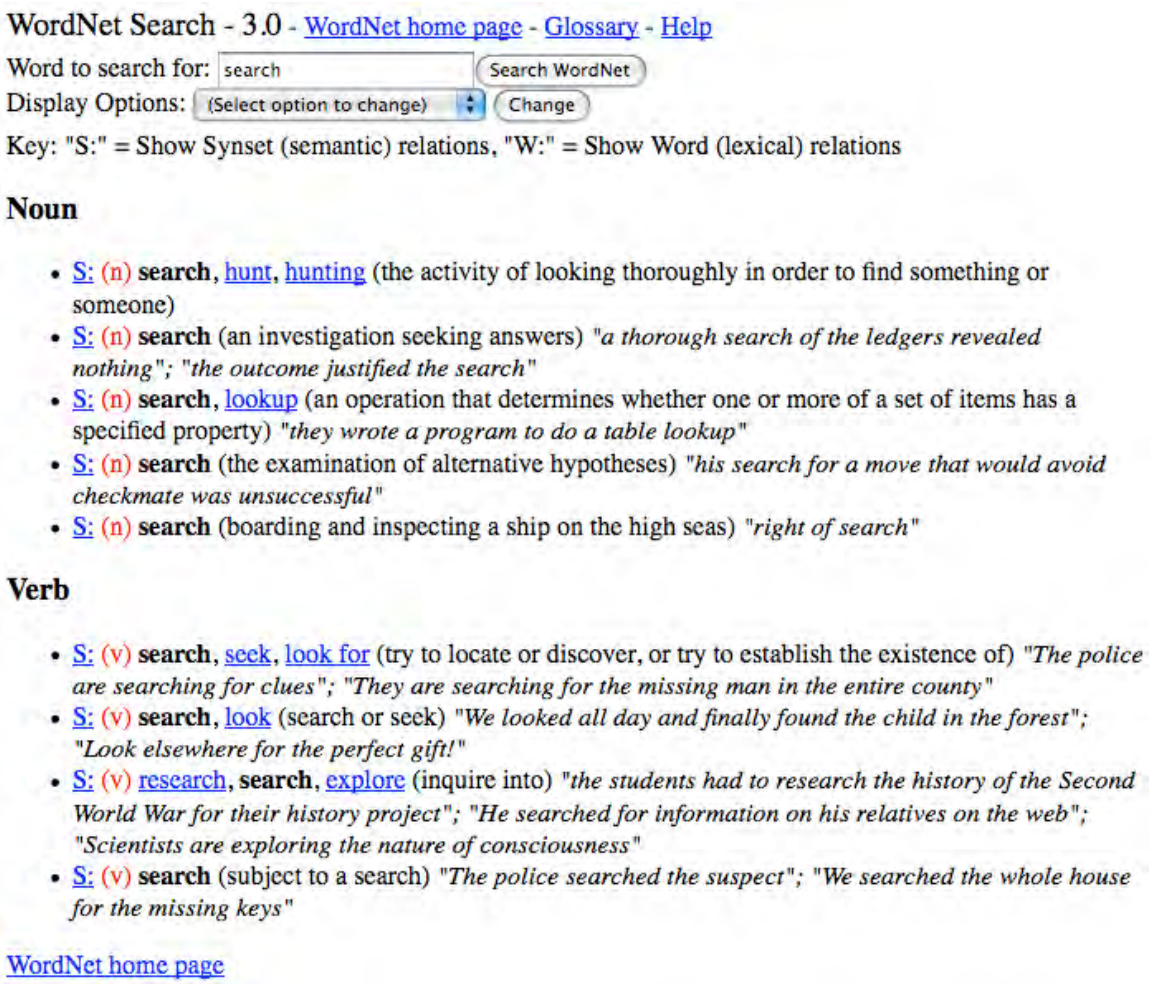


Figure 8.15 Caught in the WordNet: word interconnections for *search*

What does all this mean to developers using Lucene? With Dave Spencer's contribution to Lucene's Sandbox, the WordNet synonym database can be churned into a Lucene index. This allows for rapid synonym lookup—for example, for synonym injection during indexing or querying (see section 8.6.2 for such an implementation). We first see how to build an index containing WordNet's synonyms, then how to use these synonyms during analysis, and finally an unusual example of what you can do with a WordNet index.

8.6.1 Building the synonym index

To build the synonym index, follow these steps:

1. Download and expand the the Prolog version of WordNet, currently distributed as the file WNprolog-3.0.tar.gz from the WordNet site at http://www.cogsci.princeton.edu/~wn.
2. Obtain the binary (or build from source; see section 8.10) of the Sandbox WordNet package.
3. Un-tar the file you downloaded. It should produce a subdirectory, prolog, that has many files. We are only interested in the wn_s.pl file. Build the synonym index using the Syns2Index program from the command line. The first parameter points to the wn_s.pl file and the second argument specifies the path where the Lucene index will be created:

```
java org.apache.lucene.wordnet.Syns2Index prolog/wn_s.pl wordnetindex
```

The Syns2Index program converts the WordNet Prolog synonym database into a standard Lucene index with an indexed field `word` and unindexed fields `syn` for each document. Version 3.0 of WordNet produces 44,930 documents, each representing a single word; the index size is approximately 2.9 MB, making it compact enough to load as a `RAMDirectory` for speedy access.

A second utility program in the WordNet Sandbox area lets you look up synonyms of a word. Here is a sample lookup of a word near and dear to our hearts:

```
java org.apache.lucene.wordnet.SynLookup wordnetindex search

Synonyms found for "search":
explore
hunt
hunting
look
lookup
research
seek
```

Figure 8.16 shows these same synonyms graphically using Luke.

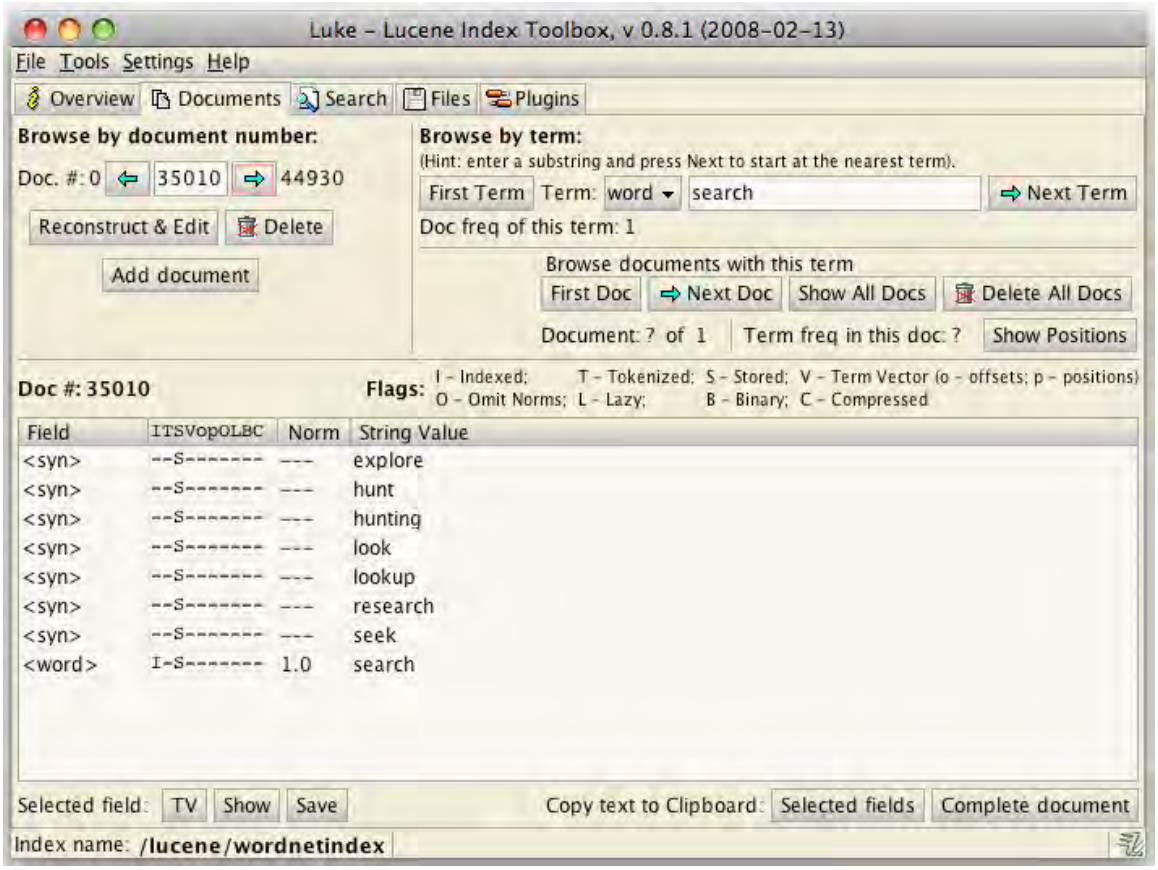


Figure 8.16 Cool app Luke: inspecting WordNet synonyms

To use the synonym index in your applications, borrow the relevant pieces from `SynLookup`, as shown in listing 8.5.

Listing 8.6 Looking up synonyms from a WordNet-based index

```
public class SynLookup {

  public static void main(String[] args) throws IOException {
    if (args.length != 2) {
      System.out.println(
    "java org.apache.lucene.wordnet.SynLookup <index path> <word>");
    }

    FSDirectory directory = FSDirectory.getDirectory(args[0], false);
```

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

    String word = args[1];
    Hits hits = searcher.search(
      new TermQuery(new Term("word", word)));

    if (hits.length() == 0) {
      System.out.println("No synonyms found for " + word);
    } else {
      System.out.println("Synonyms found for \"" + word + "\":");
    }
    for (int i = 0; i < hits.length(); i++) {                |#1
      Document doc = hits.doc(i);                            |#1
                                                             |#1
      String[] values = doc.getValues("syn");                |#1
                                                             |#1
      for (int j = 0; j < values.length; j++) {              |#1
        System.out.println(values[j]);                       |#1
      }                                                      |#1
    }                                                        |#1

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

#1 Enumerate synonyms for word

The SynLookup program was written for this book, but it has been added into the WordNet Sandbox codebase.

8.6.2 Tying WordNet synonyms into an analyzer

The custom SynonymAnalyzer from section 4.6 can easily hook into WordNet synonyms using the SynonymEngine interface. Listing 8.6 contains the WordNetSynonymEngine, which is suitable for use with the SynonymAnalyzer.

Listing 8.7 WordNetSynonymEngine

```
public class WordNetSynonymEngine implements SynonymEngine {
  RAMDirectory directory;
  IndexSearcher searcher;

  public WordNetSynonymEngine(File index) throws IOException {
    FSDirectory fsDir = new FSDirectory(index, null);
    directory = new RAMDirectory(fsDir);  // #1
    fsDir.close();
    searcher = new IndexSearcher(directory);
  }

  public String[] getSynonyms(String word) throws IOException {

    ArrayList synList = new ArrayList();

    AllDocCollector collector = new AllDocCollector();  // #2
```

```
      searcher.search(new TermQuery(new Term("word", word)), collector);

      List<ScoreDoc> hits = collector.getHits();
      Iterator<ScoreDoc> it = hits.iterator();
      while(it.hasNext()) {                    // #3
        ScoreDoc hit = it.next();
        Document doc = searcher.doc(hit.doc);

        String[] values = doc.getValues("syn");

        for (int j = 0; j < values.length; j++) {  // #4
          synList.add(values[j]);
        }
      }

      return (String[]) synList.toArray(new String[0]);
    }
  }
```

#1 Load synonym index into RAM for rapid access
#2 Collect every matching document
#3 Iterate over matchin documents
#4 Record synonyms

We use the `AllDocCollector` from section 6.XXX to keep all synonyms.

Adjusting the `SynonymAnalyzerViewer` from section 4.6 to use the `WordNetSynonymEngine`, our sample output is as follows:

```
1: [quick] [warm] [straightaway] [spry] [speedy] [ready] [quickly] [promptly] [prompt]
[nimble] [immediate] [flying] [fast] [agile]
2: [brown] [embrown] [brownness] [brownish] [browned]
3: [fox] [9][trick] [throw] [slyboots] [fuddle] [fob] [dodger] [discombobulate] [confuse]
[confound] [befuddle] [bedevil]
4: [jumps]
5: [over] [terminated] [o] [ended] [concluded] [complete]
6: [lazy] [slothful] [otiose] [indolent] [faineant]
7: [dogs]
```

Interestingly, WordNet synonyms do exist for *jump* and *dog* (see the lucli output in listing 8.1), but only in singular form. Perhaps stemming should be added to our `SynonymAnalyzer` prior to the `SynonymFilter`, or maybe the `WordNetSynonymEngine` should be responsible for stemming words before looking them up in the WordNet index. These are issues that need to be addressed based on your environment. This emphasizes again the importance of the analysis process and the fact that it deserves your attention.

[9]We've apparently befuddled or outfoxed the WordNet synonym database because the synonyms injected for *fox* don't relate to the animal noun we intended.

Let's use the WordNet index for an unusual application.

8.6.3 Calling on Lucene

With the ubiquity of mobile devices and their shrinking size, we need clever text-input methods. The T9 interface present on most phones is far more efficient than requiring exact character input.[10] As a prototype of something potentially useful, we put Lucene and WordNet under a cell-phone-like Swing interface, as shown in figure 8.16.[11]

Figure 8.17 Cell-phone-like Swing interface

The buttons 2–9 are mapped to three or four letters of the alphabet each, identical to an actual phone. Each click of these numbers appends the selected digit to an internal buffer; a Lucene search is performed to match words for those digits. The buttons that aren't mapped to letters are used for additional capabilities: 1 scrolls the view through the list of matching words (the status bar shows how many words match the digits entered); the asterisk (*) backspaces one digit, undoing the last number entered; 0 enables debugging diagnostic output to the console; and pound (#) clears all digits entered, allowing you to start a new entry.

CONSTRUCTING THE T9 INDEX

We wrote a utility class to preprocess the original WordNet index into a specialized T9 index. Each word is converted into a `t9` keyword field. Each word, its T9 equivalent, and the text length of the word are indexed, as shown here:

```
Document newDoc = new Document();
```

[10] T9 is an input method that maps each numeric button to multiple letters of the alphabet. A series of numbers logically corresponds to a subset of sensible words. For example, 732724 spells *search*.

[11] Many thanks to Dave Engler for building the base Swing application framework.

```
Field wordField = new Field("word", word,
                            Field.Store.YES,
                            Field.Index.NOT_ANALYZED_NO_NORMS);
wordField.setOmitTf(true);
newDoc.add(wordField);

Field t9 = new Field("t9", t9(word),
                     Field.Store.YES,
                     Field.Index.NOT_ANALYZED_NO_NORMS);
t9.setOmitTf(true);
newDoc.add(t9);

Field length = new Field("length", Integer.toString(word.length()),
                         Field.Store.NO,
                         Field.Index.NOT_ANALYZED_NO_NORMS);
length.setOmitTf(true);
newDoc.add(length);
```

We omit norms and set `omitTf` on the word and length fields, since they are only used for sorting. For the t9 field we omit norms because every field has the same length. The `t9` method is not shown, but it can be obtained from the book's source code distribution (see the "About this book" section). The word length is indexed as its `Integer.toString()` value to allow for sorting by length using the sort feature discussed in section 5.1.

SEARCHING FOR WORDS WITH T9

To have a little fun with Lucene, we query for a sequence of digits using a `Boolean-Query` with a slight look-ahead so a user doesn't have to enter all the digits. For example, if the digits 73272 are entered, *search* is the first word shown, but two others also match (*secpar*[12] and *peasant*). The query uses a boosted `TermQuery` on the exact digits (to ensure that exact matches come first) and wildcard queries matching words with one or two more characters more. Here's the `BooleanQuery` code:

```
BooleanQuery query = new BooleanQuery();
Term term = new Term("t9", number);
TermQuery termQuery = new TermQuery(term);
termQuery.setBoost(2.0f);
WildcardQuery plus1 = new WildcardQuery(new Term("t9", number + "?"));
WildcardQuery plus2 = new WildcardQuery(new Term("t9", number + "??"));
query.add(termQuery, BooleanClause.Occur.SHOULD);
query.add(plus1, BooleanClause.Occur.SHOULD);
query.add(plus2, BooleanClause.Occur.SHOULD);
```

The search results are sorted first by score, then by length, and finally alphabetically within words of the same length:

---

[12] "A unit of astronomical length based on the distance from Earth at which stellar parallax is 1 second of arc; equivalent to 3.262 light years" (according to a Google **define: secpar** result from WordNet).

```
Hits hits = searcher.search(query,
    new Sort(new SortField[] {SortField.FIELD_SCORE,
                              new SortField("length",
                                  SortField.INT),
                              new SortField("word")}));
```

Search results are timed and cached. The status bar displays the time the search took (often under 5ms). The cache allows the user to scroll through words.

8.7 Highlighting query terms

Giving end users some context around hits from their searches is friendly and, more important, useful. A prime example is Google search results. Each hit, as shown in Figure 1.1, includes up to three lines of the matching document highlighting the terms of the query. Often a brief glimpse of the surrounding context of the search terms is enough to know if that result is worth investigating further. Like spell correction, covered in Section 8.11, the Web search engines have established this feature as a baseline requirement that all other search engines are expected to have.

What's commonly referred to as highlighting in fact consists of two separate functions. First is dynamic fragmenting, which means picking a few choice sentences out of a large text that best match the search query. Some search applications skip this step, and instead fallback on a static abstract or summary for each document, but generally that gives a worse user experience because it's static. The second function is highlighting, whereby specific words in context of their surrounding text are called out, often with bolding and a colored background, so the user's eyes can quickly jump to specific words that matched.

These two functions are fully independent. For example, you may apply highlighting to a title field without deriving fragments from it, because you always want to present the full title. Or, for a field that has a large amount of text, you would first fragment it and then apply the highlighting.

Thanks to Mark Harwood's contribution, the Sandbox includes infrastructure to fragment and highlight text based on a Lucene query. Figure 8.17 is an example of using Highlighter on part of the text from this section, based on a term query for *highlighting*. The source code for this is shown in Listing 8.7.

We begin by an overview of the components used during highlighting, then show a simple example of highlighter in action, including how to use cascading style sheets to control the mechanics of highlighting. We wrap up showing you how to highlight actual search results.

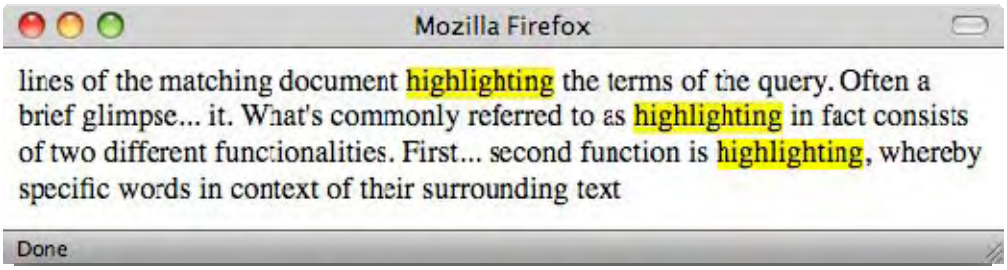


Figure 8.18 Highlighting query terms

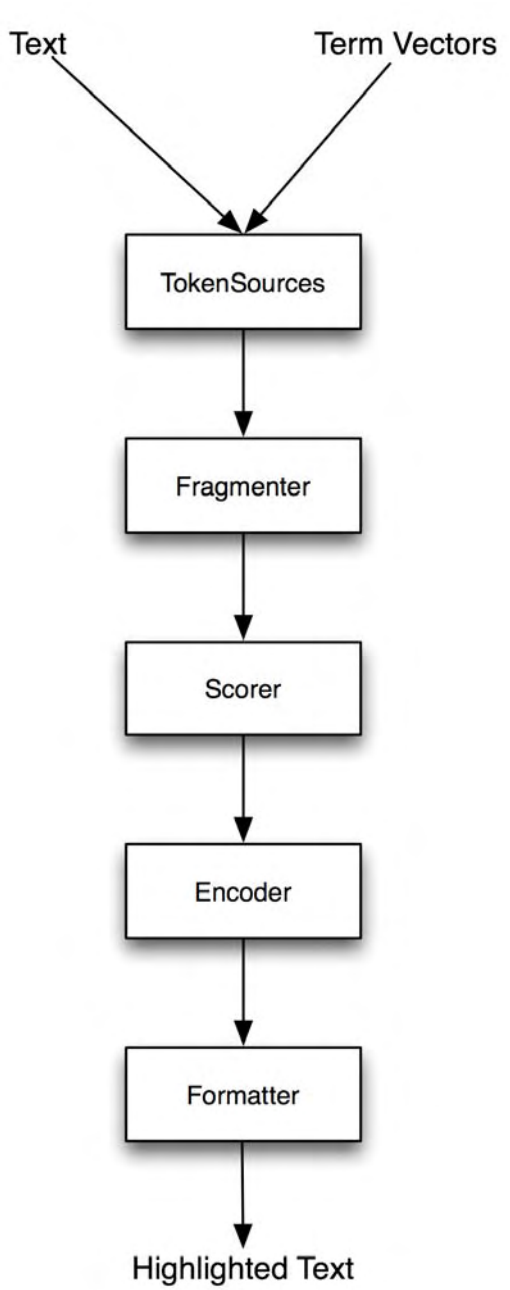


Figure 8.19 Java classes and interfaces used by Highlighter

8.7.1 Highlighter Components

The Highlighter code is a sophisticated and flexible utility, and is well factored to break out the separate steps necessary during fragmentation and highlighting. Figure 8.19 shows the steps used by the Highlighter class to compute the highlighted text. Let's walk through each.

TOKENSOURCES

Highlighting requires two separate inputs: the actual full original text (a `String`) to work on, and a `TokenStream` derived from that text. Typically you would store the full text as a stored field in the index, but if you have an alternate external store, for example a database, that works fine as well. Just be sure that source can deliver the text for N documents per search quickly enough.

To create the `TokenStream`, you could simply re-analyze the text. Alternatively, since you presumably had already analyzed the text during indexing, you can derive the `TokenStream` from previously stored term vectors (see section 2.8), as long as you used `Field.TermVector.WITH_POSITIONS_OFFSETS`. The convenient `TokenSources` class in the `Highlighter` package has various static convenience methods that will extract a `TokenStream` from an index using whichever of these sources is appropriate. You can also create your own `TokenStream` separately if you'd like. Generally, term vectors will give you the fastest performance, but they do consume additional space in the index.

Highlighter relies on the start and end offset of each `Token` from the token stream, to locate the exact character slices to highlight in the original input text. So it's crucial that your analyzer sets `startOffset` and `endOffset` on each token correctly, as character offsets! If these are not right, you'll see non-word pieces of text highlighted, or you may hit a `InvalidTokenOffsetsException` during highlighting. The core Lucene analyzers all set the offsets properly, so this normally is not a problem unless you've created your own analyzer. The next component, `Fragmenter`, breaks the original text into small pieces called fragments.

FRAGMENTER

`Fragmenter` is a Java interface in the highlighter package whose purpose is to split the original string into separate fragments for consideration. `NullFragmenter` is one concrete class implementing this interface that simply returns the entire string as a single fragment. This is appropriate for title fields and other short text fields, where you wish to show the full text. `SimpleFragmenter` is another concrete class that breaks the text up into fixed-size fragments by character length, with no effort to spot sentence boundaries. You can specify how many characters per fragment (the default is 100). However, this fragmenter is a little too simple: it does not take into account positional constraints of the query when creating the fragments, which means for phrase queries and span queries, a matching span will easily be broken across two fragments.

Fortunately, the final fragmenter, `SimpleSpanFragmenter`, resolves that problem by attempting to make fragments that always include the spans matching each document. You'll have to pass in a `SpanScorer` (see next section) so it knows where the span matches are.

If you don't set a `Fragmenter` on your `Highlighter` instance, it uses `SimpleFragmenter` by default. Although it does not exist currently in the `Highlighter` package, a good implementation of

Fragmenter would be one that attempts to produce fragments on sentence boundaries. Highlighter then takes each fragment produced by the fragmenter and passes each to the Scorer.

SCORER

The output of the Fragmenter is a series of text fragments from which highlighter must pick the best one(s) to present. To do this, Highlighter asks the Scorer, a Java interface, to score each fragment. The Highlighter package provides two concrete implementations: QueryScorer, which scores each fragment based on how many terms from the provided Query appear in the fragment, and SpanScorer, which attempts to only assign scores to actual term occurrences that contributed to the match for the document. When combined with SimpleSpanFragmenter, SpanScorer is usually the best option since true matches are highlighted.

QueryScorer uses the terms from the query; it extracts them from primitive term, phrase, and Boolean queries and weighs them based on their corresponding boost factor. A query must be rewritten in its most primitive form for QueryScorer to be happy. For example, wildcard, fuzzy, prefix, and range queries rewrite themselves to a BooleanQuery of all the matching terms. Call Query.rewrite(IndexReader), which translates the query into primitive form, to rewrite a query prior to passing the Query to QueryScorer (unless you're sure the query is already a primitive one).

SpanScorer extracts matching spans for the query and then uses these spans to score each fragment. Fragments that did not in fact match the query, even if they contain a subset of the terms from the query, receive a score of 0.0. If you use the simpler QueryScorer, you'll find that a PhraseQuery can show fragments that do not actually show the entire phrase, which is terribly disconcerting and trust eroding to the end user. Note, however, that because SpanScorer is specific to each matching document, since it enumerates the specific matching spans, it must be instantiated for every document you need to highlight. Because of these benefits, it's strongly recommended that you use SpanScorer instead of QueryScorer. All of our examples below use SpanScorer.

At this point Highlighter chooses the best scoring fragments to present to the user. All that remains is to properly format them.

ENCODER

The Encoder Java interface has a simple purpose to encode the original text into the external format. There are two concrete implementations: DefaultEncoder, which is used by default in Highlighter, simply does nothing with the text. SimpleHTMLEncoder encodes the text as HTML, escaping any special characters such as < and > and &, and non-ascii characters. Once encoder is done, the final step is to format the fragments for presentation.

FORMATTER

Finally, the Formatter Java interface takes each fragment of text as a String, as well as the terms to be highlighted, and actually renders the highlighting. Highlighter provides 3 concrete classes to choose from. SimpleHTMLFormatter simply puts a pre and post tag to wrap around each hit. The default constructor will use the <b> (bold) HTML tag. GradientFormatter uses different shades of background color to indicate how strong each hit was, using the <font> HTML tag. Finally, SpanGradientFormatter does the same thing but uses the HTML tag because some browsers may not render the <font> tag

correctly. You can of course also create your own class implementing the Formatter API. Let's see a real example.

8.7.2 Putting it all together

Now that you understand the logical steps of the highlighting process, let's look at some concrete examples. The simplest example of Highlighter returns the best fragment, surrounding each matching term with HTML <B> tags:

```
String text = "The quick brown fox jumps over the lazy dog";

TermQuery query = new TermQuery(new Term("field", "fox"));

TokenStream tokenStream =
    new SimpleAnalyzer().tokenStream("field",
        new StringReader(text));

SpanScorer scorer = new SpanScorer(query, "field",
                                   new CachingTokenFilter(tokenStream));
Fragmenter fragmenter = new SimpleSpanFragmenter(scorer);
Highlighter highlighter = new Highlighter(scorer);
highlighter.setTextFragmenter(fragmenter);
assertEquals("The quick brown <B>fox</B> jumps over the lazy dog",
    highlighter.getBestFragment(tokenStream, text));
```

The previous code produces this output:

```
The quick brown <B>fox</B> jumps over the lazy dog
```

In this simple example, our text was a fixed string and we derived a `TokenStream` by using `SimpleAnalyzer`. To successfully highlight terms, the terms in the `Query` need to match `Token`s emitted from the `TokenStream`. The same text must be used to generate the `TokenStream` as is used for the original text to highlight.

We then create a `SpanScorer` to score fragments. `SpanScorer` requires you to wrap the `TokenStream` in a `CachingTokenFilter` as it needs to process the tokens more than once. Using the `SpanScorer`, we create a `SimpleSpanFragmenter` to break the text into fragments. In this simple example, since the text is so small, the fragmenter is pointless as the entire text will become the one and only fragment; for example, we could have simply used `NullFragmenter` instead. Finally, we create `Highlighter`, set our fragmenter, then ask it to for the best scoring fragment.

Next we show how to use Cascading Style Sheets to control how highlighting is done.

8.7.3 Highlighting with CSS

Using <B> tags to surround text that will be rendered by browsers is a reasonable default. Fancier styling should be done with cascading style sheets (CSS) instead. Our next example, which was used to generate the highlighted result shown in Figure 8.18, uses custom begin and end tags to wrap highlighted terms with a using the custom CSS class highlight. Using CSS attributes, the color and formatting of

highlighted terms is decoupled from highlighting, allowing much more control for the web designers who are tasked with beautifying our search results page.

Listing 8.7 demonstrates the use of a custom Fragmenter, setting the fragment size to 70, and a custom Formatter to style highlights with CSS. Note that this is a contrived example, since the content to be highlighted is a static string in the source code. In our first example, only the best fragment was returned, but Highlighter shines in returning multiple fragments. `HighlightIt`, in listing 8.7, uses the `Highlighter` method to concatenate the best fragments with an ellipsis (…) separator; however you could also have a String[] returned by not passing in a separator, so that your code could deal with each fragment individually.

Listing 8.8 Highlighting terms using cascading style sheets

```
public class HighlightIt {
  private static final String text =
    "Giving end users some context around hits from their searches is friendly and,
more important, useful. A prime example is Google search results. Each hit, as shown in
Figure 1.1, includes up to three lines of the matching document highlighting the terms
of the query. Often a brief glimpse of the surrounding context of the search terms is
enough to know if that result is worth investigating further.  Like spell correction,
covered in Section 8.11, the Web search engines have established this feature as a
requirement that all other search engines are expected to have: users now expect it. "
+
    "What's commonly referred to as highlighting in fact consists of two different
functionalities.  First is dynamic fragmenting, which means picking a few sentences out
of a large text that best match the search query.  Some search applications skip this
step, and instead fallback on a static abstract or summary for each document, but
generally that gives a worse user experience because it's static." +
    "The second function is highlighting, whereby specific words in context of their
surrounding text are called out, often with bolding and a colored background, so the
user's eyes can quickly jump to specific words that matched.  These two functions are
somewhat independent.  For example, you may apply highlighting to a title field without
excerpting it, because you always want to present the full title.  Or, for a field that
has a large amount of text, you would first excerpt it and then apply the
highlighting.";

  public static void main(String[] args) throws Exception {

    if (args.length != 1) {
      System.err.println("Usage: HighlightIt <filename>");
      System.exit(-1);
    }

    String filename = args[0];

    //String searchText = "\"search engines\"";
    String searchText = "highlighting";                                    // #1
    QueryParser parser = new QueryParser("f", new StandardAnalyzer());// #1
    Query query = parser.parse(searchText);                                // #1

    SimpleHTMLFormatter formatter =                                        // #2
      new SimpleHTMLFormatter("<span class=\"highlight\">",                // #2
                              "</span>");                                  // #2
```

```
    TokenStream tokenStream = new StandardAnalyzer()                 // #3
        .tokenStream("f", new StringReader(text));                   // #3

    CachingTokenFilter tokens = new CachingTokenFilter(tokenStream);  // #4

    SpanScorer scorer = new SpanScorer(query, null, tokens);         // #5
    Fragmenter fragmenter = new SimpleSpanFragmenter(scorer);        // #6

    Highlighter highlighter = new Highlighter(formatter, scorer);    // #7
    highlighter.setTextFragmenter(fragmenter);                       // #7

    String result =                                                  // #8
        highlighter.getBestFragments(tokens, text, 3, "...");        // #8

    FileWriter writer = new FileWriter(filename);
    writer.write("<html>");
    writer.write("<style>\n" +                                       // #9
        ".highlight {\n" +                                           // #9
        " background: yellow;\n" +                                   // #9
        "}\n" +                                                      // #9
        "</style>");                                                 // #9
    writer.write("<body>");
    writer.write(result);
    writer.write("</body></html>");
    writer.close();
  }
}
```

#1 Create the query
#2 Customize surrounding tags
#3 Tokenize text
#4 Create cached TokenStream
#5 Create SpanScorer
#6 Use SimpleSpanFragmenter to fragment
#7 Create highlighter with fragmenter
#8 Highlight best 3 fragments
#9 Write highlighted HTML

#2 We customize the surrounding tags for each highlighted term.

#3 Here we tokenize the original text, using `StandardAnalyzer`.

#4 Create `CachingTokenFilter` required by `SpanScorer`

#8 We highlight the best three fragments, separating them with an ellipsis (…).

#9 Finally we write the highlighted HTML to a file, as shown in figure 8.17.

In neither of our examples did we perform a search and highlight actual hits. The text to highlight was hard-coded. This brings up an important issue when dealing with the Highlighter: where to get the text to highlight in a real search application? This is addressed in the next section.

8.7.4 Highlighting Search Results

Whether to store the original field text in the index is up to you (see section 2.2.1 for field indexing options). If the original text isn't stored in the index (generally because of size considerations), you'll have to retrieve the text to be highlighted from its original source. Take care to ensure that the retrieved text is always identical to the text that had been indexed. If the original text is stored with the field, it can be retrieved directly from the `Document` obtained from the search, as shown in the following piece of code:

```
public void testHits() throws Exception {
  IndexSearcher searcher = new IndexSearcher(TestUtil.getBookIndexDirectory());
  TermQuery query = new TermQuery(new Term("title", "action"));
  TopDocs hits = searcher.search(query, 10);

  Highlighter highlighter = new Highlighter(null);
  Analyzer analyzer = new SimpleAnalyzer();

  for (int i = 0; i < hits.scoreDocs.length; i++) {
    Document doc = searcher.doc(hits.scoreDocs[i].doc);
    String title = doc.get("title");

    TokenStream stream = TokenSources.getAnyTokenStream(searcher.getIndexReader(),
                                                        hits.scoreDocs[i].doc,
                                                        "title",
                                                        doc,
                                                        analyzer);
    SpanScorer scorer = new SpanScorer(query, "title",
                                       new CachingTokenFilter(stream));
    Fragmenter fragmenter = new SimpleSpanFragmenter(scorer);
    highlighter.setFragmentScorer(scorer);
    highlighter.setTextFragmenter(fragmenter);

    String fragment =
        highlighter.getBestFragment(stream, title);

    System.out.println(fragment);
  }
}
```

With our sample book index, the output is

```
JUnit in <B>Action</B>
Lucene in <B>Action</B>
Tapestry in <B>Action</B>
```

Notice that we use the convenient `TokenSources.getAnyTokenStream` method to derive a `TokenStream` from our original text. Under the hood, this method first tries to retrieve the term vectors from the index. As long as you indexed the document's field with

`Field.TermVector.WITH_POSITIONS_OFFSETS` then term vectors are used to reconstruct the `TokenStream`. Otherwise, the analyzer you pass in is used to re-analyze the text. Whether to index term vectors or re-analyze the text is an application dependent decision: run your own tests to measure the difference in run time and index size for each approach. In our case, since we did index the title field with term vectors in the books index, term vectors are used to create the token stream.

Next we show a useful means of cascading more than one filter together.

8.8 Chaining filters

Using a search filter, as we've discussed in section 5.5, is a powerful mechanism for selectively narrowing the document space to be searched by a query. The Sandbox contains an interesting meta-filter in the misc project, contributed by Kelvin Tan, which chains other filters together and performs AND, OR, XOR, and ANDNOT bit operations between them. `ChainedFilter`, like the built-in `CachingWrapperFilter`, isn't a concrete filter; it combines a list of filters and performs a desired bit-wise operation for each successive filter, allowing for sophisticated combinations.

It's slightly involved to demonstrate `ChainedFilter` because it requires a diverse enough dataset to showcase how the various scenarios work. We've set up an index with 500 documents including a `key` field with values 1 through 500; a `date` field with successive days starting from January 1, 2009; and an `owner` field with the first half of the documents owned by bob and the second half owned by sue:

```
public class ChainedFilterTest extends TestCase {
  public static final int MAX = 500;
  private RAMDirectory directory;
  private IndexSearcher searcher;
  private Query query;
  private Filter dateFilter;
  private Filter bobFilter;
  private Filter sueFilter;

  public void setUp() throws Exception {
    directory = new RAMDirectory();
    IndexWriter writer =
       new IndexWriter(directory, new WhitespaceAnalyzer(),
                       IndexWriter.MaxFieldLength.UNLIMITED);

    Calendar cal = Calendar.getInstance();
    cal.set(2009, 1, 1, 0, 0); // 2009 January 01

    for (int i = 0; i < MAX; i++) {
      Document doc = new Document();
      doc.add(new Field("key", "" + (i + 1),
                        Field.Store.YES, Field.Index.NOT_ANALYZED));
      doc.add(new Field("owner", (i < MAX / 2) ? "bob" : "sue",
                        Field.Store.YES, Field.Index.NOT_ANALYZED));
      doc.add(new Field("date", DateTools.timeToString(cal.getTimeInMillis(),
DateTools.Resolution.DAY),
                        Field.Store.YES, Field.Index.NOT_ANALYZED));
      writer.addDocument(doc);

      cal.add(Calendar.DATE, 1);
```

```
      }

      writer.close();

      searcher = new IndexSearcher(directory);

      // query for everything to make life easier
      BooleanQuery bq = new BooleanQuery();
      bq.add(new TermQuery(new Term("owner", "bob")), BooleanClause.Occur.SHOULD);
      bq.add(new TermQuery(new Term("owner", "sue")), BooleanClause.Occur.SHOULD);
      query = bq;

      // date filter matches everything too
      cal.set(2099, 1, 1, 0, 0); // 2099 January 01
      dateFilter = RangeFilter.Less("date", DateTools.timeToString(cal.getTimeInMillis(),

  DateTools.Resolution.DAY));
      bobFilter = new CachingWrapperFilter(
          new QueryWrapperFilter(
                new TermQuery(new Term("owner", "bob"))));
      sueFilter = new CachingWrapperFilter(
          new QueryWrapperFilter(
                new TermQuery(new Term("owner", "sue"))));
    }
  }
```

In addition to the test index, `setUp` defines an all-encompassing query and some filters for our examples. The query searches for documents owned by either bob or sue; used without a filter, it will match all 500 documents. An all-encompassing `DateFilter` is constructed, as well as two `QueryFilter`s, one to filter on owner bob and the other for sue.

Using a single filter nested in a `ChainedFilter` has no effect beyond using the filter without `ChainedFilter`, as shown here with two of the filters:

```
    public void testSingleFilter() throws Exception {
      ChainedFilter chain = new ChainedFilter(
                                            new Filter[] {dateFilter});
      TopDocs hits = searcher.search(query, chain, 10);
      assertEquals(MAX, hits.totalHits);

      chain = new ChainedFilter(new Filter[] {bobFilter});
      assertEquals(MAX / 2, TestUtil.hitCount(searcher, query, chain), hits.totalHits);
    }
```

The real power of `ChainedFilter` comes when we chain multiple filters together. The default operation is OR, combining the filtered space as shown when filtering on bob or sue:

```
  public void testOR() throws Exception {
      ChainedFilter chain = new ChainedFilter(
                new Filter[] {sueFilter, bobFilter});

      assertEquals("OR matches all", MAX, TestUtil.hitCount(searcher, query, chain));
    }
```

Rather than increase the document space, AND can be used to narrow the space:

```
public void testAND() throws Exception {
  ChainedFilter chain = new ChainedFilter(
            new Filter[] {dateFilter, bobFilter}, ChainedFilter.AND);
  TopDocs hits = searcher.search(query, chain, 10);
  assertEquals("AND matches just bob", MAX / 2, hits.totalHits);
  Document firstDoc = searcher.doc(hits.scoreDocs[0].doc);
  assertEquals("bob", firstDoc.get("owner"));
}
```

The `testAND` test case shows that the `dateFilter` is AND'd with the `bobFilter`, effectively restricting the search space to documents owned by bob since the `dateFilter` is all encompassing. In other words, the intersection of the provided filters is the document search space for the query. Filter bit sets can be XOR'd (exclusively OR'd, meaning one or the other, but not both):

```
public void testXOR() throws Exception {
  ChainedFilter chain = new ChainedFilter(
    new Filter[]{dateFilter, bobFilter}, ChainedFilter.XOR);
  TopDocs hits = searcher.search(query, chain, 10);
  assertEquals("XOR matches sue", MAX / 2, hits.totalHits);
  Document firstDoc = searcher.doc(hits.scoreDocs[0].doc);
  assertEquals("sue", firstDoc.get("owner"));
}
```

The `dateFilter` XOR'd with `bobFilter` effectively filters for owner sue in our test data. And finally, the ANDNOT operation allows only documents that match the first filter but not the second filter to pass through:

```
public void testANDNOT() throws Exception {
  ChainedFilter chain = new ChainedFilter(
    new Filter[]{dateFilter, sueFilter},
      new int[] {ChainedFilter.AND, ChainedFilter.ANDNOT});

  TopDocs hits = searcher.search(query, chain, 10);
  assertEquals("ANDNOT matches just bob",
               MAX / 2, hits.totalHits);
  Document firstDoc = searcher.doc(hits.scoreDocs[0].doc);
  assertEquals("bob", firstDoc.get("owner"));
}
```

In `testANDNOT`, given our test data, all documents in the date range except those owned by sue are available for searching, which narrows it down to only documents owned by bob.

Depending on your needs, the same effect can be obtained by combining query clauses into a `BooleanQuery` or using `FilteredQuery` (see section 6.4.1). Keep in mind the performance caveats to using filters; and, if you're reusing filters without changing the index, be sure you're using a caching filter.

`ChainedFilter` doesn't cache, but wrapping it in a `CachingWrappingFilter` will take care of that aspect.

Let's look at some alternate `Directory` implementations next.

8.9 Storing an index in Berkeley DB

The Chandler project (http://chandlerproject.org) is an ongoing effort to build an open-source Personal Information Manager. Chandler aims to manage diverse types of information such as email, instant messages, appointments, contacts, tasks, notes, web pages, blogs, bookmarks, photos, and much more. It's an extensible platform, not just an application. As you suspected, search is a crucial component to the Chandler infrastructure.

The Chandler codebase uses Python primarily, with hooks to native code where necessary. We're going to jump right to how the Chandler developers use Lucene; refer to the Chandler site for more details on this fascinating project. Andi Vajda, one of Chandler's key developers, created PyLucene to enable full access to Lucene's APIs from Python. PyLucene is an interesting port of Lucene to Python; we cover it in full detail in section 9.6.

Chandler's underlying repository uses Oracle's Berkeley DB in a vastly different way than a traditional relational database, inspired by RDF and associative databases. Andi created a Lucene directory implementation that uses Berkeley DB as the underlying storage mechanism. An interesting side-effect of having a Lucene index in a database is the transactional support it provides. Andi donated his implementation to the Lucene project, and it's maintained in the contrib/db/bdb area of the Sandbox.

Berkeley DB, at release 4.7.25 as of this writing, is written in C, but provides full Java API access via JNI. The "contrib/db/bdb" sandbox package provides access via this API. Berkeley DB also has a "Java edition", which is written entirely in Java, so no JNI access is required and the code exists in a single JAR file. Aaron Donovan ported the contrib/db/bdb to the "Java edition" under the "contrib/db/bdb-je" directory. The example below shows how to use the Java edition version of Berkeley DB but the API for the original Berkeley DB is very similar. We provide the corresponding examples for both indexing and searching with the source code that comes with this book.

8.9.1 Coding to JEDirectory

`JEDirectory`, which is a Directory implementation that stores its files in the Berkeley DB Java Edition, is more involved to use than the built-in `RAMDirectory` and `FSDirectory`. It requires constructing and managing two Berkeley DB Java API objects, `Environment` and `Database`. Listing 8.8 shows `JEDirectory` being used for indexing.

Listing 8.9 Indexing with JEDirectory

```
public class BerkeleyDbIndexer {
  public static void main(String[] args) throws IOException, DatabaseException {
    if (args.length != 1) {
      System.err.println("Usage: BerkeleyDbIndexer <index dir>");
      System.exit(-1);
    }

    File indexFile = new File(args[0]);
```

```
    if (indexFile.exists()) {
      File[] files = indexFile.listFiles();

      for (int i = 0; i < files.length; i++)
        if (files[i].getName().startsWith("__"))
          files[i].delete();
      indexFile.delete();
    }

    indexFile.mkdir();

    EnvironmentConfig envConfig = new EnvironmentConfig();
    DatabaseConfig dbConfig = new DatabaseConfig();

    envConfig.setTransactional(true);
    envConfig.setInitializeCache(true);
    envConfig.setInitializeLocking(true);
    envConfig.setInitializeLogging(true);
    envConfig.setAllowCreate(true);
    envConfig.setThreaded(true);
    dbConfig.setAllowCreate(true);
    dbConfig.setType(DatabaseType.BTREE);

    Environment env = new Environment(indexFile, envConfig);

    Transaction txn = env.beginTransaction(null, null);
    Database index = env.openDatabase(txn, "__index__", null, dbConfig);
    Database blocks = env.openDatabase(txn, "__blocks__", null, dbConfig);
    txn.commit();

    txn = env.beginTransaction(null, null);
    DbDirectory directory = new DbDirectory(txn, index, blocks);

    IndexWriter writer = new IndexWriter(directory,
                                         new StandardAnalyzer(),
                                         true,
                                         IndexWriter.MaxFieldLength.UNLIMITED);

    Document doc = new Document();
    doc.add(new Field("contents", "The quick brown fox...", Field.Store.YES,
Field.Index.ANALYZED));
    writer.addDocument(doc);

    writer.optimize();
    writer.close();

    directory.close();
    txn.commit();

    index.close();
    blocks.close();
    env.close();

    System.out.println("Indexing Complete");
  }
```

```
}
```

Once you have an instance of `JEDirectory`, using it with Lucene is no different than using the built-in `Directory` implementations. Searching with `JEDirectory` uses the same mechanism (see `BerkeleyDBJESearcher` in the source code with this book).

8.10 Fast memory based indices

In Section 2.10 we showed how to use `RAMDirectory` to load an index entirely in RAM. This is especially convenient if you have a pre-built index living on disk and you'd like to slurp the whole thing into RAM for faster searching. However, because `RAMDirectory` still treats all data from the index as files, there is actually significant overhead during searching for Lucene to decode this file structure for every query. This is where two interesting Sandbox contributions come in: `MemoryIndex` and `InstantiatedIndex`.

`MemoryIndex`, contributed by Wolfgang Hoschek, is a very fast RAM-only index designed to test whether a single document matches a query. It's only able to index and search a single document. You instantiate the `MemoryIndex`, and then use its `addField` method to add the document's fields into it. Then, use its search methods to search with an arbitrary Lucene query. This method returns a float relevance score, where 0.0 means there was no match.

`InstantiatedIndex`, contributed by Karl Wettin, is similar, except it's able to index and search multiple documents. You first create an `InstantiatedIndex`, which is analogous to RAMDirectory in that it is the common store that a writer and reader share. Then, create an `InstantiatedIndexWriter` to index documents. Finally, create an `InstantiatedIndexReader`, and then an IndexSearcher from that, to run arbitrary Lucene searches.

Under the hood, both of these contributions represents all aspects of a Lucene index using linked in-memory Java data structures, instead of separate index files like `RAMDirectory`. This makes searching much faster than `RAMDirectory`, at the expense of more RAM consumption. In many cases, especially if the index is small, the documents you'd like to search have very high turnover, short index to search delay is required, and you have plenty of RAM, one of these classes may be a perfect fit.

8.11 Spell correction

Spell correction is something users now take for granted in today's search engines. Enter a mis-spelled word into Google and you'll get back a helpful "Did you mean…?" with your typo corrected as a hyperlink. You can then conveniently click it to express "yes, indeed, I did!". Google's spell correction is so effective that you can rely on it to correct your typos! Spelling correction is such a wonderfully simple and intuitive must-have feature to the end user. But, as a developer, just how do you implement it? Fortunately, Lucene has the spellchecker Sandbox contribution, created by David Spencer, for just this purpose.

Web search engines spend a lot of energy tuning their spell correction algorithms, and it shows. Generally you get a very good experience, and indeed this sets the very high bar for how all the world's search applications are expected to behave. Let's walk through the typical steps during spell correction, including generation of possible suggestions, selection of the best one for each mis-spelled word, presenting the choice to the user, and some other possible extensions to try. Along the way we'll see how the spellchecker sandbox package tackles each.

8.11.1 Generating suggestions list

You might assume the very first step is to decide whether or not spell correction is even necessary. But that's hard to determine up front, and it's usually more effective to simply always run through the steps and then use the score of each potential suggestion to decide whether they should be presented to the user. The first step is to generate a raw set of possible suggestions. The spellchecker module works with one term at a time, so if the query has multiple terms you'll have to consider each, separately (though see "Further things to try" below for some ideas on handling multi-term queries).

You will need a source dictionary of "valid" words. You could try to find a generic known accurate dictionary, but it's hard to find such dictionaries that will exactly match your search domain and it's even harder to keep such a dictionary current over time. A more powerful means of deriving a dictionary is to use your search index to gather all unique terms seen during indexing from a particular field. This is the approach used by spellchecker.

Given the dictionary, you must enumerate the suggestions. You could use a phonetic approach, such as the "sounds like" matching we explored in section 4.5. Another approach, which is the one used by spellchecker, is to use letter ngrams to identify similar words. A letter ngram is all sub-sequences of N letters in length, where N varies between a min and max size. Using this approach, the ngrams for all words in the dictionary are indexed into a separate spellchecker index. This is usually a fast operation, and so the application's indexing process would rebuild the entire spell check index whenever the main index is updated.

Let's walk through an example. Say our dictionary contains the word "lettuce". Table 8.20 shows the 3grams and 4grams that are added into the spellchecker index. In this case, our "document" is the word lettuce whose indexed tokens are the generated 3grams and 4grams. Next, imagine the user searches for "letuce", whose ngrams are shown in Table 8.21. To find the suggestions, the ngrams for "letuce" are used to run a search against the spellcheck index. Since many of the ngrams are shared ("let", "tuc", "uce" and "tuce") the correct word "lettuce" will be returned with a high relevance score

Word	lettuce
gram3	let, ett, ttu, tuc, uce
gram4	lett, ettu, ttuc, tuce

Table 8.20 ngrams for the word "lettuce"

Word	letuce
gram3	let, etu, tuc, uce
gram4	letu, etuc, tuce

Table 8.21 ngrams for the mis-spelled word "letuce"

Fortunately, the spellchecker module handles all this ngram processing for you, under the hood (though the NGramTokenizer and EdgeNGramTokenizer, described in Section XXX, let you create your own ngrams if you want to take a more custom approach). Creating the spellchecker index is surprisingly simple. Listing 8.9 shows how to do so, using the terms from an existing lucene index. Run it like this:

```
java lia.tools.CreateSpellCheckerIndex ./spellindex /lucene/wordnetindex word
```

This creates a spellchecker index, stored in the local directory spellindex, by enumerating all unique terms seen in the wordnet index (`/lucene/wordnetindex`) in the "word" field. It produces output like this:

```
Now build SpellChecker index...
took 3018 milliseconds
```

Note that if you have an alternate source of words, or perhaps you'd like to use terms from a Lucene index but filter certain ones out, you can create your own class implementing the `Dictionary` interface (in the `org.apache.lucene.search.spell` package) and pass that to `SpellChecker` instead.

Listing 8.9 Using the spellchecker Sandbox module to create the spell checker index.

```
public class CreateSpellCheckerIndex {

  public static void main(String[] args) throws IOException {

    if (args.length != 3) {
      System.out.println("Usage: java lia.tools.SpellCheckerTest SpellCheckerIndexDir
IndexDir IndexField");
      System.exit(1);
    }

    String spellCheckDir = args[0];
    String indexDir = args[1];
    String indexField = args[2];

    System.out.println("Now build SpellChecker index...");
    Directory dir = new FSDirectory(new File(spellCheckDir), null);
    SpellChecker spell = new SpellChecker(dir);     //#1
    long startTime = System.currentTimeMillis();
    IndexReader r = IndexReader.open(indexDir);     //#2
    try {
      spell.indexDictionary(new LuceneDictionary(r, indexField));  //#3
    } finally {
      r.close();
    }
    dir.close();
    long endTime = System.currentTimeMillis();
    System.out.println("  took " + (endTime-startTime) + " milliseconds");
  }
```

#1 Create SpellChecker on its directory
#2 Open IndexReader containing words to add to spell dictionary

#3 Add all words from the specified fields into the spell checker index

```
}
```

Listing 8.10 Finding the list of respelled words candidates using the spellchecker index.

```
public class SpellCheckerExample {

  public static void main(String[] args) throws IOException {

    if (args.length != 2) {
      System.out.println("Usage: java lia.tools.SpellCheckerTest SpellCheckerIndexDir
wordToRespell");
      System.exit(1);
    }

    String spellCheckDir = args[0];
    String wordToRespell = args[1];

    Directory dir = new FSDirectory(new File(spellCheckDir), null);
    SpellChecker spell = new SpellChecker(dir);  //#1

    spell.setStringDistance(new LevensteinDistance());  //#2
    //spell.setStringDistance(new JaroWinklerDistance());

    String[] suggestions = spell.suggestSimilar(wordToRespell, 5); //#3
    System.out.println(suggestions.length + " suggestions for '" + wordToRespell +
"':");
    for(int i=0;i<suggestions.length;i++)
      System.out.println("  " + suggestions[i]);
  }
}
```

#1 Create SpellCheck from existing spell check index
#2 Sets the string distance metric used to rank the suggestions
#3 Generate respelled candidates

The next step is to pick the best suggestion.

8.11.2 Select the best suggestion

From the first step, using the letter ngram approach, we can now generate a set of suggestions for each term in the user's query. Listing 8.10 shows how to generate respellings with spellchecker, using the spellchecker index created by Listing 8.9. For example, run it like this:

```
java lia.tools.SpellCheckerExample spellindex letuce
```

and it prints this result:

```
5 suggestions for 'letuce':
  lettuce
  seduce
```

```
reduce
deduce
letch
```

Unfortunately, you don't have the luxury of showing many spell suggestions to the user. Typically you can either present no choice (if you determine all terms in the query seem properly spelled, or there were no good spelling candidates found), or a single suggestion, back to the user.

While the ngram approach is good for enumerating potential respellings, its relevance ranking is generally not good for selecting the best one. Typically, a different distance metric is used to resort the suggestions, according to how similar each is to the original term. One common metric is the Levenshtein metric, which we used in Section 3.4.7 to search for similar terms. This is the default metric used by SpellChecker, and generally it works well. You can also use the `JaroWinkler` distance (see http://en.wikipedia.org/wiki/Jaro-Winkler), which is provided in the `spellchecker` package, or you could implement your own string similarity metric. The array of suggestions returned by `SpellChecker.suggestSimilar` is sorted by decreasing similarity according to the distance metric, so you simply pick the first result to present as the suggested respelling.

The final step is to present the respelled option to the user.

8.11.3 Presenting the result to the user

Finally, once you have your single best respelling candidate, you first need to decide if it's good enough for presenting to the user. `SpellChecker` does not return the distance between each suggestion and the original user's term, though you could simply recomputed that by calling the `getDistance` method on the `StringDistance` you are using. `SpellChecker` also has an alternative `suggestSimilar` method that takes additional arguments in order to restrict the suggestions to those terms that are more frequent than the original term; this way you will only present a suggested respelling if it occurred more frequently than the original term, which is one good way to decide whether a candidate is worth presenting. It also has a `setAccuracy` method to set the minimum relevance of each suggestion. You could also run the original search, and then if it returns 0, or very few results, try the respelled search to see if it returns more, and use that to bias the decision.

Next, assuming you have a suggestion worth presenting, what exactly should your application do? One option, if you are very confident of the respelling, is to automatically respell the term. But be sure to clearly tell the user at the top of the search results that this was done, and give the user a quick link to forcefully do the original search. Alternatively, you could search exactly as the user requested but present a "Did you mean…" with the respelling. Finally, you could search for both the original query plus the respelled query, or'd together perhaps with different boosts.

Typically a search application will choose up-front one of these options. But modern Web search engines seem to make this choice dynamically, per query, depending on some measure of confidence of the respelling. Go ahead and try some searches in http://google.com and http://search.live.com. For example, on Google a search for the mis-spelled "Levenstein distance" will silently also search for the proper spelling "Levenshtein distance" plus the original mis-spelled term. If instead you search for "hippopatomus" (misspelled), Google will faithfully search for your mis-spelled term, which has thousands

of hits, while presenting a "Did you mean: hippopotamus" link to search on the corrected term. Curiously, the same two searches on http://search.live.com have the opposite behavior. Of course by the time you read this likely you'll see different behavior.

We'll end this important topic with some further ideas to explore.

8.11.4 Further things to try

Implementing spell correction is in fact very challenging, and we've touched on a few of the issues above. The spellchecker sandbox package gives you a great start. But you may want to explore some of the possible improvements below in your application. If you find success with one of these, or something else, please donate it back if possible:

- If you have high search traffic, consider using the terms from your user's queries to help rank the best suggestion. In applications whose dictionary changes quickly with time, such as a news search engine for current events, this is especially compelling.

- Instead of respelling each term separately, consider factoring in the other terms to bias the suggestions of each term. One way is to compute term co-occurrence statistics up front for every pair of terms X and Y, to measure how many documents contain both terms X and Y. Then, when sorting the suggestions take these statistics into account with the other terms in the user's query. If a user enters the misspelled query "harry poter" you'd really like to suggest "harry potter" instead of other choices like "harry spotter".

- The dictionary you use for spell correction is very important. When you use terms from an existing index, you can easily import mis-spellings from the content you had indexed if the content is "dirty". You can also accidentally import terms that you may never want to suggest, for example SKU numbers or stock ticker symbols. Try to filter such terms out, or only accepting terms that occurred above a certain frequency.

- If you high search traffic, you can train your spellchecker according to how users click on the "Did you mean…" link, biasing future suggestions based on how users have accepted suggestions in the past.

- If your search application has entitlements (restricting which content a user can see based on her entitlement) then take care to keep the spellchecker dictionary separate for different user classes. A single global dictionary can accidentally "leak" information across entitlement classes which could cause real problems.

- Tweak how you compute the confidence of each suggestion. The spellcheck module currently relies entirely on the StringDistance score for this, but you could imagine improving this by combining StringDistance with the frequency of this tem in the index to get a better confidence.

8.12 Fun and interesting Query Extensions

The queries sandbox package provides some interesting additions to Lucene's core queries, contributed by Mark Harwood and Uwe Schindler, including `MoreLikeThis`, `FuzzyLikeThisQuery`, `BoostingQuery`, `TermsFilter`, and `TrieRangeFilter`.

8.12.1 MoreLikeThis

The `MoreLikeThis` class captures all the logic for finding similar documents to an existing document. In Section 5.7.1 we saw the BooksLikeThis example to accomplish the same functionality, but `MoreLikeThis` is more general and will work with any Lucene index. Listing 8.10 shows how to do the same thing as `BooksLikeThis` using `MoreLikeThis`.

The approach is exactly the same: enumerate terms from the provided document and build up a `Query` to find similar documents. `MoreLikeThis` is more flexible: if you give it a docID and an `IndexReader` instance, it will iterate through any field that is stored or has indexed term vectors, to locate the terms for that document. For stored fields it must re-analyze the text, so be sure to set the analyzer first if the default `StandardAnalyzer` is not appropriate. `MoreLikeThis` is able to find similar documents to an arbitrary String or the contents of a provided File or URL, as well.

Remember `MoreLikeThis` will usually return the exact same document back (if your search was based on a document in the index), so be sure to filter it out in your presentation.

Listing 8.10 Using the MoreLikeThis class to find similar documents

```
public class SpellCheckerExample {

  public static void main(String[] args) throws IOException {

    if (args.length != 2) {
      System.out.println("Usage: java lia.tools.SpellCheckerTest SpellCheckerIndexDir
wordToRespell");
      System.exit(1);
    }

    String spellCheckDir = args[0];
    String wordToRespell = args[1];

    Directory dir = new FSDirectory(new File(spellCheckDir), null);
    SpellChecker spell = new SpellChecker(dir);  //#1

    spell.setStringDistance(new LevensteinDistance());  //#2
    //spell.setStringDistance(new JaroWinklerDistance());

    String[] suggestions = spell.suggestSimilar(wordToRespell, 5); //#3
    System.out.println(suggestions.length + " suggestions for '" + wordToRespell +
"':");
    for(int i=0;i<suggestions.length;i++)
      System.out.println("  " + suggestions[i]);
  }
}
```

#1 Create SpellCheck from existing spell check index

#2 Sets the string distance metric used to rank the suggestions
#3 Generate respelled candidates

8.12.2 FuzzyLikeThisQuery

The `FuzzyLikeThisQuery` combines `MoreLikeThis` and `FuzzyQuery`. It allows you to build a query by adding in arbitrary text, which is analyzed by default with `StandardAnalyzer`. The tokens derived from that analysis are then "fuzzed" using the same process that `FuzzyQuery` uses. Finally, from these terms the most differentiating terms are selected and searched on. This query can be a useful alternative when end users are unfamiliar with the standard `QueryParser` boolean search syntax.

8.12.3 BoostingQuery

The `BoostingQuery` allows you to run a primary `Query`, but selectively demote search results matching a second `Query`. Use it like this:

```
Query balancedQuery = new BoostingQuery(positiveQuery, negativeQuery, 0.01f);
```

where `positiveQuery` is your primary query, `negativeQuery` matches those documents you'd like to demote, and 0.01f is the factor you'd like use when demoting. All documents matching `negativeQuery` alone will not be included in the results. All documents matching `positiveQuery` alone will be included with their original score. Finally, all documents matching both will have their score demoted by the specified factor.

`BoostingQuery` is similar to creating a boolean query with the `negativeQuery` added as a NOT clause, except instead of excluding outright those documents matching `negativeQuery`, `BoostingQuery` includes those documents with a weaker score.

8.12.4 TermsFilter

`TermsFilter` is a filter that matches any arbitrary set of terms you specify. It's like a `RangeFilter` that does not require the terms to be in a contiguous sequence. You simply construct the `TermsFilter`, one by one add the terms you'd like to filter on by calling the `addTerm` method, and then use that filter when searching. An example might be a collection of primary keys from a database query result or perhaps a choice of "category" labels picked by the end user.

8.12.5 TrieRangeQuery

`TrieRangeQuery` first appeared in Lucene 2.9, contributed by Uwe Schindler. It provides range filtering on 32 and 64 bit numeric fields (`int`, `long`, `float`, `double`) and any other types, such as `java.util.Date`, that can be accurately represented as `int` or `long`. While this is precisely the same functionality as `RangeQuery`, `TrieRangeQuery` takes a drastically different approach: it indexes additional tokens derived from the number that represent successively larger ranges the number falls within. This range aggregation at indexing time increases the size of the index somewhat, since additional tokens are present, but results in far faster range searches, especially when your numeric field is fine-

grained and you need to filter on large ranges. In such situations `RangeQuery` is typically unacceptably slow, while `TrieRangeQuery` is exceptionally fast.

At search time, the requested range is translated into an equivalent union of these pre-indexed ranges, which typically results in far fewer terms that need to be searched. Fortunately, all this magic happens under the hood and is exposed with a simple API. Listing 8.XXX shows how to actually use `TrieRangeQuery`.

Listing 8.XXX Using `TrieRangeFilter`

```
public class TrieRangeTest extends TestCase {

  private final static boolean VERBOSE = false;

  private IndexSearcher searcher1;     // #1
  private IndexSearcher searcher2;     // #2

  private static final DecimalFormat formatter =
      new DecimalFormat("000");

  public void setUp() throws Exception {
    RAMDirectory dir1 = new RAMDirectory();
    RAMDirectory dir2 = new RAMDirectory();

    IndexWriter writer1 = new IndexWriter(dir1, new WhitespaceAnalyzer(),
                                          IndexWriter.MaxFieldLength.UNLIMITED);

    IndexWriter writer2 = new IndexWriter(dir2, new WhitespaceAnalyzer(),
                                          IndexWriter.MaxFieldLength.UNLIMITED);

    for(int i=0; i<1000; i++) {
      Document doc1 = new Document();                              // #3
      Field field1 = new Field("number",                           // #3
                               formatter.format(i),                // #3 #4
                               Field.Store.NO,                     // #3
                               Field.Index.NOT_ANALYZED_NO_NORMS); // #3
      field1.setOmitTermFreqAndPositions(true);                    // #3
      doc1.add(field1);                                            // #3
      writer1.addDocument(doc1);                                   // #3

      Document doc2 = new Document();                              // #5
      TokenStream stream = new LongTrieTokenStream((long) i, 4);   // #5
      Field field2 = new Field("number", stream);                  // #5
      field2.setOmitNorms(true);                                   // #5
      field2.setOmitTermFreqAndPositions(true);                    // #5
      doc2.add(field2);                                            // #5
      writer2.addDocument(doc2);                                   // #5

      if (VERBOSE && i == 123) {
        System.out.println("Terms generated for long value: " + i);
        stream.reset();
        AnalyzerUtils.displayTokens(stream);
```

```
          System.out.println();
        }
      }
      writer1.close();
      writer2.close();

      searcher1 = new IndexSearcher(dir1);
      searcher2 = new IndexSearcher(dir2);

      System.out.println("Without trie index: " +
                         dir1.sizeInBytes() +
                         " bytes");

      System.out.println("   With trie index: " +
                         dir2.sizeInBytes() +
                         " bytes");
    }

    public void testQuery() throws Exception {

      LongTrieRangeQuery q =  new LongTrieRangeQuery("number",       // #6
                                      4,                             // #6
                                      128L, 256L,                    // #6
                                      true, true);                   // #6
      assertEquals(129, TestUtil.hitCount(searcher2, q));

      RangeQuery rq = new RangeQuery("number",                       // #7
                                     "128", "256",                   // #7
                                     true, true);                    // #7
      rq.setConstantScoreRewrite(true);                              // #7
      assertEquals(129, TestUtil.hitCount(searcher1, rq));           // #7
    }
  }

  #1 Searches normal index
  #2 Searches Trie index
  #3 Add normal int field
  #4 Zero-pad to 3 digits
  #5 Add trie-encoded field
  #6 Test Trie query
  #7 Test RangeQuery
```

When you run the tests, it produces this output (in addition to the tests passing):

```
Without trie index: 20480 bytes
   With trie index: 39936 bytes
```

As you can see, for this simple index, adding the trie fields almost doubled the size of the index. But, fear not! In a real index, that has additional text fields and a larger set of documents, the percentage increase in index size will be much smaller. Normally the amazing gain in search performance more than makes up for this.

In the `setUp` method we create two indexes. The first one is a normal Lucene index, which we use for running `RangeQuery`. You can see that we had to zero-pad the numbers, using `DecimalFormat`, to ensure their lexicographic sort matches the numeric sort, as described in section 2.8. In the second index, we use `LongTrieTokenStream` to index additional tokens. Conveniently, no zero padding is necessary!

As you can see, we handle our numeric field like a tokenized text by supplying a special `TokenStream` (see [chapter about analyzers]) to the `Field` constructor to index our long value:

- `new LongTrieTokenStream(long value, int precisionStep)` – this creates a `TokenStream` that encodes and enumerates the additional "range tokens" to be added to our document. The `precisionStep` controls how many ranges should be created; a larger value makes fewer ranges, which reduces the size of the index but makes searching slower. Values between 1 and 8 are reasonable; 4 is a good starting default. See the javadocs for more details on this parameter. `value` is the long you wish to index. Like `Field` and `Document`, `LongTrieTokenStream` can be reused for indexing later documents. Just set the new numeric value using `setValue(long value)` and consume the stream again.
- `new Field(String name, TokenStream stream)` – this constructor of `Field` is not often used, but gives us the possibility to pass a pre-tokenized field value to our `Document`; `field` is the name of your field, and `stream` is the reference to our `LongTrieTokenStream`. As this constructor does not supply the possibility to disable norms and tern frequency, it is recommeneded to do this using `setOmitNorms(true)` and `setOmitTermFreqAndPositions(true)`.

When you need to query a range at search time, simply create `LongTrieRangeQuery` with the same arguments that you'd pass to `RangeQuery`, except for a new integer argument (`precisionStep`) passed after the field name. That argument must be equal to, or a multiple of, the `precisionStep` you passed to `LongTrieTokenStream`. Just like `precisionStep` used during indexing, the larger this value is, the slower the searching will be as more ranges will need to be visited. Alternatively, you can use `LongTrieRangeFilter` as a counterpart for `RangeFilter`.

OTHER NUMERIC TYPES

In addition to `long`, which we cover above, the `trie` package also provides matching methods for `int`s (`IntTrieTokenStream, IntTrieRangeQuery, IntTrieRangeFilter`). To handle `float` values, a utility class `TrieUtils` provides `floatToSortableInt`, which encodes the `float` as an `int` that will sort properly. The inverse method, `sortableIntToFloat`, does the reverse. No information is lost in this conversion. There are also the same methods for converting `double` to `long` (`doubleToSortableLong`) and back (`sortableLongToDouble`). In general, any type that can be represented without loss of precision as an int or long can be handled in this way. For example, if you need to index `java.util.Date` values, you can use the `getTime()` method of `java.util.Date` to translate it into a long.

SORTING

The encoding, the `trie` package uses, preserves the sort order of the original value. However, because the tokens in the index are encoded, you must use a custom parser to decode them back to their original values. Fortunately, `TrieUtils` exposes these helpful methods:

```
SortField getLongSortField(final String field, boolean reverse)
SortField getIntSortField(final String field, boolean reverse)
```

Once you have the `SortField`, create a `Sort` instance from it (possibly involving other fields if you are performing a multi-field sort) and then run your search. Under the hood, `TrieUtils` will parse the encoded values back to their original values for sorting. Section 5.XXX provides more details on sorting.

If you are curious, set VERBOSE to `true` in the test to see just what terms are added to your document. It will produce output like this:

```
Terms generated for long value: 123
[ NUL NUL NUL NUL NUL NUL NUL NUL {] [NUL NUL NUL NUL NUL NUL NUL •] [(@NUL NUL NUL NUL
NUL NUL NUL ] [,┘NUL NUL NUL NUL NUL NUL NUL ] [0 NUL NUL NUL NUL NUL NUL ] [4NUL NUL
NUL NUL NUL NUL ] [8┼NUL NUL NUL NUL NUL ] [<NUL NUL NUL NUL NUL ] [NUL NUL NUL NUL ]
[D@NUL NUL NUL ] [H┘NUL NUL NUL ] [L NUL NUL ] [PNUL NUL ] [T┼NUL ] [XNUL ] []
```

As you can see, the field values are unreadable! Don't worry, that's expected. All that really matters is that the Trie queries and `SortField` parsers know how to read them. You'll see similar values if you look at your index with Luke, for example.

Our next package, XML Query Parser, is another option for building your search user interface.

8.13 XML Query Parser: Beyond "one box" search interfaces

(This section was contributed by Mark Harwood.)

The standard Lucene QueryParser is ideal for creating the classic "onebox" search interface provided by web search engines, such as Google. However, many search applications are more complex than this and require a custom search form to capture criteria with widgets such as:

- Drop-down list boxes e.g. choice of gender: male/female
- Radio buttons or check-boxes e.g. "include fuzzy matching?"
- Calendars to select dates or ranges of dates
- Maps to define locations
- Separate "free-text" input boxes for targeting different fields e.g. title or author.

All of the criteria from these HTML form elements must be brought together somehow to form a Lucene search request. There are fundamentally 3 approaches to constructing this request, as shown in Figure 8.22.

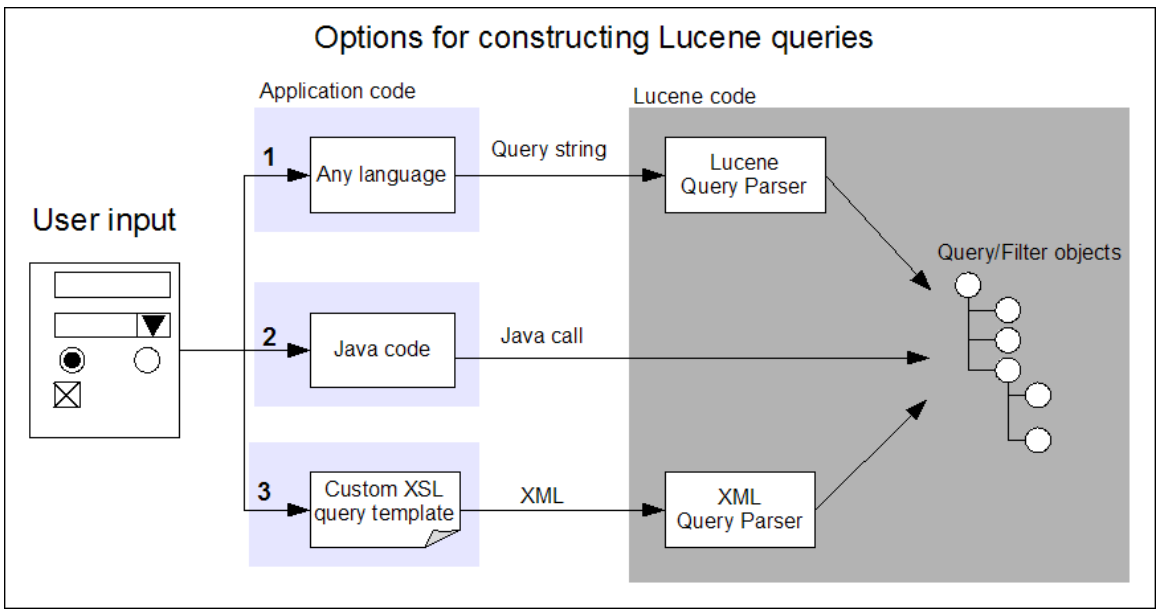


Figure 8.22: Three common options for building a Lucene query from a search user interface.

Options 1 and 2 above both have drawbacks. The standard QueryParser syntax can only be used to instantiate a limited range of Lucene's available queries and filters. Option 2 embeds all of the query logic in Java code, where it can be hard to read or maintain. There are many examples that show it's desirable to avoid using Java code to assemble complex collections of objects. Often a domain-specific text file provides a cleaner syntax and eases maintenance. Further examples include Spring configuration files, XUL frameworks, Ant build files or Hibernate database mappings. The sandbox `XmlQueryParser` does exactly this, enabling option 3 from Figure 8.22, for Lucene.

We'll start with a brief example, then show a full example of how `XmlQueryParser` is used, and end with options for extending `XmlQueryParser` with new `Query` types. Here's a simple example XML query that combines a Lucene query and filter, enabling you to express a Lucene Query without any Java code:

```
<FilteredQuery>
   <Query>
      <UserQuery fieldName="text">"Swimming pool"</UserQuery>
   </Query>
   <Filter>
      <TermsFilter fieldName="dayOfWeek">monday friday</TermsFilter>
   </Filter>
```

```
</FilteredQuery>
```

`XmlQueryParser` parses such XML and produces a Query object for you, and the sandbox includes a full DTD to formally specify the out-of-the-box tags, as well as full HTML documentation, including examples, for all tags.

But how can you produce this XML from a Web search UI in the first place? There are various approaches but one simple approach is to use the Extensible Stylesheet Language (XSL) to define query templates as text files that can be populated with user input at run-time. Let's walk through an example web application. This example is derived from the web demo available in the `XmlQueryParser` sandbox sources.

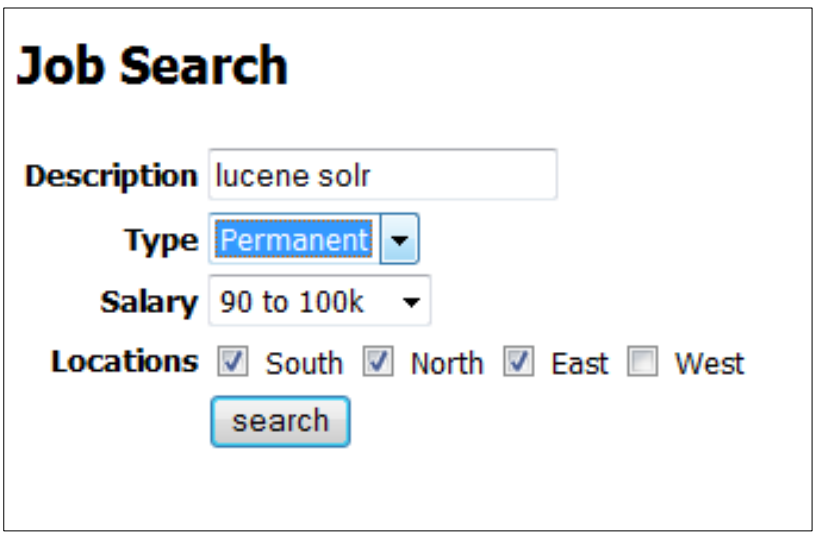


Figure 8.23 An example search user interface for a job search site.

8.13.1 Using XmlQueryParser

Consider the Web-based form user interface shown in Figure 8.23. Let's create a servlet that can handle this "job search" form. The good news is this code should also work, unchanged, with your own choice of form. Our Java servlet begins with some initialization code:

```
public void init(ServletConfig config) throws ServletException {
  super.init(config);
  try {
    openExampleIndex();
```

```
    //Load and cache choice of XSL query template using QueryTemplateManager
    queryTemplateManager=new QueryTemplateManager(

getServletContext().getResourceAsStream("/WEB-INF/query.xsl"));

    //initialize an XML Query Parser for use by all threads
    xmlParser=new CorePlusExtensionsParser(defaultFldName,analyzer);

  } catch (Exception e) {
    throw new ServletException("Error loading query template",e);
  }
}
```

The initialization code performs three basic operations:

- Opening the search index. Our method (not shown here) simply opens a standard IndexSearcher and caches this in our servlet's instance data.
- Loading a Query template using the QueryTemplateManager class. This class will be used later to help construct queries.
- Creating an XML query parser. The CorePlusExtensionsParser class used here provides an XML query parser that is pre-configured with support for all the core Lucene queries and filters and also those from Lucene's “contrib” section (we will examine how to add support for custom queries later).

Having initialized our servlet we now add code to handle search requests:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {

  //Take all completed form fields and add to a Properties object
  Properties completedFormFields=new Properties();
  Enumeration pNames = request.getParameterNames();
  while(pNames.hasMoreElements()){
    String propName=(String) pNames.nextElement();
    String value=request.getParameter(propName);
    if((value!=null)&&(value.trim().length()>0)){
      completedFormFields.setProperty(propName, value);
    }
  }

  try{
    //Create an XML query by populating template with given user criteria
    org.w3c.dom.Document xmlQuery=
      queryTemplateManager.getQueryAsDOM(completedFormFields);

    //Parse the XML to produce a Lucene query
    Query query=xmlParser.getQuery(xmlQuery.getDocumentElement());

    //Run the query
    TopDocs topDocs = searcher.search(query,10);
```

```
    //and package the results and forward to JSP
    if(topDocs!=null)      {
      ScoreDoc[] sd = topDocs.scoreDocs;
      Document[] results=new Document[sd.length];
      for (int i = 0; i < results.length; i++) {
        results[i]=searcher.doc(sd[i].doc);
        request.setAttribute("results", results);
      }
    }
    RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher("/index.jsp");
    dispatcher.forward(request,response);
  }
  catch(Exception e){
    throw new ServletException("Error processing query",e);
  }
}
```

First, a `java.util.Properties` object is populated with all of the form values where the user provided some choice of criteria. This code should work with any HTML form. The Properties object is then passed to the `QueryTemplateManager` to populate the search template and create an XML document which represents our query logic. The XML document is then passed to the query parser to create a `Query` object for use in searching. The remainder of the method is typical Servlet code used to package results and pass them on to a JSP for display.

Having set up our servlet we can now take a closer look at the custom query logic we need for our "job search" and how this is expressed in the "query.xsl" query template. The XSL language in the query template allows us to perform the following operations:

- Test for the presence of input values with "if" statements
- Substitute input values into the output XML document
- Manipulate input values e.g. splitting strings and zero-padding numbers
- Loop around sections of content using "for each" statements

We will not attempt to document all of the XSL language here but clearly the shortlist above lets us perform the majority of operations that we typically need to transform user input into queries. The XSL statements that control the construction of our query clauses can be differentiated from the query clauses because they are all prefixed with the `<xsl:` tag. Our `query.xsl` file is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/Document">
    <BooleanQuery>
      <!-- Clause if user selects a preference for type of job - apply choice of
           permanent/contract filter and cache -->
      <xsl:if test="type">
        <Clause occurs="must">
```

```
          <ConstantScoreQuery>
            <CachedFilter>
              <TermsFilter fieldName="type"><xsl:value-of select="type"/></TermsFilter>
            </CachedFilter>
          </ConstantScoreQuery>
        </Clause>
      </xsl:if>

      <!-- Use standard Lucene query parser for any job description input -->
      <xsl:if test="description">
        <Clause occurs="must">
          <UserQuery fieldName="description">
            <xsl:value-of select="description"/>
          </UserQuery>
        </Clause>
      </xsl:if>

      <!-- If any of the location fields are set OR them ALL in a
           Boolean filter and cache individual filters -->
      <xsl:if test="South|North|East|West">
        <Clause occurs="must">
          <ConstantScoreQuery>
            <BooleanFilter>
              <xsl:for-each select="South|North|East|West">
                <Clause occurs="should">
                  <CachedFilter>
                    <TermsFilter fieldName="location">
                      <xsl:value-of select="name()"/>
                    </TermsFilter>
                  </CachedFilter>
                </Clause>
              </xsl:for-each>
            </BooleanFilter>
          </ConstantScoreQuery>
        </Clause>
      </xsl:if>

      <xsl:if test="salaryRange">
        <Clause occurs="must">
          <ConstantScoreQuery>
            <RangeFilter fieldName="salary" >
              <xsl:attribute name="lowerTerm">
                <xsl:value-of
                   select='format-number( substring-before(salaryRange,"-"), "000" )'
/>
              </xsl:attribute>
              <xsl:attribute name="upperTerm">
                <xsl:value-of
                   select='format-number( substring-after(salaryRange,"-"), "000" )' />
              </xsl:attribute>
            </RangeFilter>
          </ConstantScoreQuery>
        </Clause>
      </xsl:if>
    </BooleanQuery>
  </xsl:template>
```

```
</xsl:stylesheet>
```

The above template conditionally outputs clauses depending on the presence of user input. The logic behind each of the clauses is as follows:

- **Job Type: a**s a field with only two possible values (“permanent” or “contract”) this can be an expensive query clause to run because a search will typically match half of all the documents in our search index. If our index is very large then this can involve reading millions of document ids from the disk. For this reason we use a cached filter for these search terms. Any filter can be cached in memory for reuse simply by wrapping it in a <CachedFilter> tag.
- **Job description: b**eing a free-text field the standard Lucene query syntax is useful for allowing the user to express their criteria. The contents of the <UserQuery> tag are passed to a standard Lucene QueryParser to interpret the user's search.
- **Job location:** like the job type field, the job location field is a field with a limited choice of values which benefit from caching as a filter. Unlike the job type field however, multiple choices of field value can be selected for a location. We use a BooleanFilter to OR multiple Filter clauses together.
- **Job Salary: j**ob salaries are handled as a RangeFilter clause. The input field from the search form requires some manipulation in the XSL template before it can be used. The salary range value arrives from our search form as a single string value such as “90-100”. Before we can construct a Lucene request we must split this into an Upper and Lower value and make sure both values are zero-padded to comply with Lucene's requirement for these to be lexicographically ordered. Fortunately these operations can be performed using built-in XSL functions.

Let’s see how to extend `XmlQueryParser`.

8.13.2 Extending the XML query syntax

Adding support for new tags in the query syntax or changing the classes that support the existing tags is a relatively simple task. As an example, we will add support for a new XML tag to simplify the creation of date-based filters. Our new tag allows us to express date ranges in relation to today's date such as “last week's news” or “people aged between 30 and 40”. For example, in our job search application we might want to add a filter using syntax like this:

```
<Ago fieldName="dateJobPosted" timeUnit="days" from="0" to="7"/>
```

Each tag in the XML syntax has an associated “Builder” class which is used to parse the content. The Builders are registered simply by adding the object with the name of the tag it supports to the parser. So in order to register a new builder for the above “Ago” tag we would need to include a line like the following in the initialization method of our servlet:

```
xmlParser.addFilterBuilder("Ago", new AgoFilterBuilder());
```

The AgoFilterBuilder class is a simple object which is used to parse any XML tags with the value “Ago”. For those familiar with the XML DOM interface the code presented below should be straight-forward.

```
public class AgoFilterBuilder implements FilterBuilder {

  static HashMap<String,Integer> timeUnits=new HashMap<String,Integer>();
  @Override
    public Filter getFilter(Element element) throws ParserException {
    String fieldName=DOMUtils.getAttributeWithInheritanceOrFail(element,
                                                                "fieldName");
    String timeUnit=DOMUtils.getAttribute(element, "timeUnit", "days");
    Integer calUnit=timeUnits.get(timeUnit);
    if(calUnit==null){
      throw new ParserException("Illegal time unit:"
                               +timeUnit+" - must be days, months or years");
    }
    int agoStart=DOMUtils.getAttribute(element, "from",0);
    int agoEnd=DOMUtils.getAttribute(element, "to", 0);
    if(agoStart<agoEnd) {
      //swap order
      int oldAgoStart=agoStart;
      agoStart=agoEnd;
      agoEnd=oldAgoStart;
    }
    SimpleDateFormat sdf=new SimpleDateFormat("yyyyMMdd");

    Calendar start=Calendar.getInstance();
    start.add(calUnit, agoStart*-1);

    Calendar end=Calendar.getInstance();
    end.add(calUnit, agoEnd*-1);

    return new RangeFilter(fieldName,
                           sdf.format(start.getTime()),
                           sdf.format(end.getTime()),
                           true,true);
  }

  static{
    timeUnits.put("days", Calendar.DAY_OF_YEAR);
    timeUnits.put("months",Calendar.MONTH);
    timeUnits.put("years", Calendar.YEAR);
  }
}
```

Our `AgoFilterBuilder` is called by the XML parser every time an “Ago” tag is encountered and it is expected to return a Lucene Filter object given an XML DOM element. The class `DOMUtils` simplifies the code involved in extracting parameters. Our `AgoFilterBuilder` reads the “to”, “from” and “timeUnit” attributes using `DOMUtils` to provide default values if no attributes are specified. Our code simplifies application logic for specifying “to” and “from” values by swapping the values if they are out of order.

An important consideration in coding Builder classes is that they should be thread-safe. For this reason our class creates a `SimpleDateFormat` object for each request rather than holding a single object in instance data as `SimpleDateFormat` is not thread-safe.

Our Builder is relatively simple because the XML tag does not permit any child queries or filters to be nested inside it. The `BooleanQueryBuilder` class in Lucene's contrib section provides an example of a more complex XML tag which supports nested Query objects. These sorts of Builder class must be initialized with a `QueryBuilderFactory` which is used to find the appropriate Builder to handle each of the nested query tags. Next we look at an alternate `QueryParser` that can produce span queries.

8.14 Surround query language

(This section was written by Paul Elschot)

As we saw in section 5.XXX, span queries offer some advanced possibilities for positional matching. Unfortunately, Lucene's `QueryParser` is unable to produce span queries. That's where the Surround `QueryParser`, contributed by Paul Elschot, comes in. The Surround `QueryParser` defines an advanced textual language to describe span queries.

Let's walk through an example to get a sense of the query language accepted by the Surround `QueryParser`. Suppose a meteorologist wants to find documents on "temperature inversion". In the documents this "inversion" can also be expressed as "negative gradient", and each word can occur in various inflected forms.

This query in the surround query language can be used for the "temperature inversion" concept: `5n(temperat*, (invers* or (negativ* 3n gradient*)))`. This query will match the following sample texts:

- Even when the temperature is high, its inversion would...
- A negative gradient for the temperature.

But this will not match the following text, because there's nothing to match "gradient":

- A negative temperature.

This shows the power of spans: they allow word combinations in proximity ("negative gradient") to be treated as synonyms of single words ("inversion") or of other words in proximity.

You'll notice the Surround syntax is very different from Lucene's built-in `QueryParser`. First off, operators, such as 5n, may be in prefix notation, meaning they come first, followed by their sub-queries in parenthesis, for example `5n(…,…)`. The parentheses for the prefix form gave the name Surround to the language, as they surround the underlying Lucene spans.

The 3n operator is used in infix notation, meaning it's written in-between the two subqueries. Either notation is allowed in the surround query language. The 5n and 3n operators create an unordered

SpanNearQuery containing the specified sub-queries, meaning they only match when their sub-queries have spans within 5 or 3 positions of one another. If you replace “n” with “w” then an ordered SpanNearQuery is created. The prefixed number may be from 1 to 99; if you leave off the number (and just type “n” or “w”), then the default is 1, meaning the sub-queries must be adjacent.

Continuing the example, suppose the meteorologist wants to find documents that match the above “negative gradient” and two more concepts, "low pressure", and "rain". In the documents these concepts can be also expressed in plural or verb form and by synonyms such as "depression" for "low pressure" and "precipitation" for "rain". Also all three concepts should occur at most 50 words away from each other:

```
50n( (low w pressure*) or depression*,
     5n(temperat*, (invers* or (negativ* 3n gradient*))),
     rain* or precipitat*)
```

This matches the following sample texts:

- Low pressure, temperature inversion and rain.
- When the temperature has a negative height gradient above a depression no precipitation is expected.

But it will not match this text because the word "gradient" is in wrong place (further than 3 positions away), leading to improved precision in query results:

- When the temperature has a negative height above a depression no precipitation gradient is expected.

Just like the built-in QueryParser, Surround also accepts parentheses to nest queries, field: text to restrict to a specific field, * and ? as wildcards, and boolean AND, OR and NOT operators, as well as the caret ^ for boosting sub-queries. When no proximity is used, the Surround QueryParser produces the same boolean and term queries as the built-in QueryParser. In proximity subqueries, wildcards and "or" map to SpanOrQuery, and single terms map to SpanTermQuery. Due to limitations of the Lucene spans package, the operators "and", "not" and "^" cannot be used in subqueries of the proximity operators.

It should be noted that the Lucene spans package is generally not as efficient as the phrase queries used by the standard query parser. And the more complex the query, the higher its execution time. Because of this, it is recommended to provide the user with the possibility to use filters.

Unlike the standard QueryParser, the Surround parser does not use an analyzer. This means that the user will have to know precisely how terms are indexed. For indexing texts to be queried by the Surround language it is recommended to use a lowercasing analyzer that removes only the most frequently occurring punctuations. Such an analyzer is assumed in the above examples. Using analyzers this way gives very good control over the query results, at the expense of having to use more wildcards during searching.

With the possibility of nested proximity queries, the need to know precisely what is indexed, the need to use parentheses, commas and wildcards, and the preference for additional use of filters, the Surround query language is not intended for the casual user. However, for those users that are willing to spend more effort on their queries so they can save time by having higher precision results, this query language can be a good fit.

For a more complete description of the Surround query language, have a look at the `README.txt` file that comes with the source code. To use Surround, make sure that the surround contrib package is on the CLASSPATH and follow the example java code to obtain a normal Lucene query:

```
import org.apache.lucene.queryParser.surround.parser.QueryParser;
import org.apache.lucene.queryParser.surround.query.SrndQuery;
import org.apache.lucene.queryParser.surround.query.BasicQueryFactory;
import org.apache.lucene.queryParser.surround.query.TooManyBasicQueries;
import org.apache.lucene.search.Query;

  {
     String queryText = "5d(temperat*, (invers* or (negativ* 3d gradient*)))";
     SrndQuery srndQuery = QueryParser.parse(queryText);

     int maxBasicQueries = 1000; // to limit expansions of truncations during
Query.rewrite()
     BasicQueryFactory bqFactory = new BasicQueryFactory(maxBasicQueries);

     String defaultFieldName = "txt"; // as prepared by the IndexWriter
     Query luceneQuery = srndQuery.makeLuceneQueryField(defaultFieldName, bqFactory);

     // and use luceneQuery as usual, possibly
     // catching TooManyBasicQueries during the Query.rewrite() that is normally
     // done during query search.
  }
```

And, of course, extensions to this query language are welcome. Our final sandbox package is spatial lucene.

8.16 Spatial Lucene

(This section was written by Patrick O'Leary)

Over the past decade, web search has transformed itself from finding a basic web page, to finding specific results in a certain topic. Video search, medical search, image search, news, sports etc.: each of these is referred to as a vertical search. One that stands out is local search: Local search is the use of specialized search techniques that allow users to submit geographically constrained searches against a structured database of local business listings[13].

Lucene now contains a sandbox package to enable local search, called spatial lucene. Spatial lucene started with the donation of local lucene from Patrick O'Leary (www.gissearch.com) and is expected to

[13] Wikipedia provides more details at http://en.wikipedia.org/wiki/Local_search_(Internet).

grow in capabilities over time. If you need to find 'shoe stores' that exist within 10 miles of location X, then spatial lucene will do that.

While by no means a full GIS (geographical information system) solution, spatial lucene supports these functions:

- Radial based searching, for example "show me only restaurants within 2 miles from a specified location". This defines a filter covering a circular area.
- Sorting by distance, so locations closer to a specified origin are sorted first.
- Boosting by distance, so locations closer to a specified origin receive a larger boost.

The real challenge with spatial search is that for every query that arrives, a different origin is required. Life would be very simple if the origin were fixed, as we could compute and store all distances in the index. But because distance is a very dynamic value, changing with every query as the origin changes, spatial Lucene must take a far more dynamic approach that requires special logic during indexing as well as searching. We'll visit this logic here, as well as touch on some of the performance consideration implied by spatial Lucene's approach. Let's first see how to index documents for spatial search.

8.16.2 Indexing spatial data

In order to use spatial lucene, you must first geo-code locations in your documents. This means a textual location, such as "77 Massachusetts Ave" or "the Louvre" must be translated into its corresponding latitude and longitude. Some methods for geo-coding are described at http://www.gissearch.com/geocode. This process must be done outside of spatial lucene, which only operates on locations represented as latitudes and longitudes.

Now what does spatial lucene do with each location? One simple way approach would be to load each document's location, compute its distance on the fly, and use that for filtering, sorting or boosting. This approach will work, but it results in rather poor performance. Instead, spatial Lucene implements interesting transformations during indexing, including both projection and hierarchical tries and grids, that allow for faster searching.

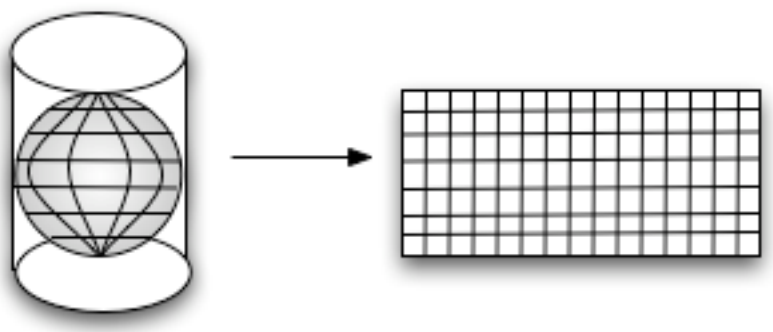


Figure 8.XXX Projecting the globe

PROJECTING THE GLOBE

In order to compute distances, we first must "flatten" the globe using a mathematical process called "projection", depicted in Figure XXX. This is similar to having a light shine through a transparent globe and 'projected' onto a flat canvas. By unfolding the globe into a flat surface the methods for selecting bounding boxes are much more uniform.

There are two common projections. The first is the sinusoidal projection (http://en.wikipedia.org/wiki/Sinusoidal_projection), which keeps an even spacing of the projection. However, it will cause a distortion of the image, giving it a "pinched" look. The second projection is the Mercator projection (http://en.wikipedia.org/wiki/Mercator_projection), used because it gives a regular rectangular view of the globe. However it does not correctly scale to certain areas of the planet. If for example you look at a global projection of the Earth on Google maps, and compare it to the spherical projection in Google Earth, you will see Greenland in Google maps rectangular projection about the size of North America, whereas in Google Earth, it's about 1/3 the size. Spatial lucene has a built-in implementation for the Sinusoidal projection, which we use in our example.

The next step is to map each location to a series of grid boxes.

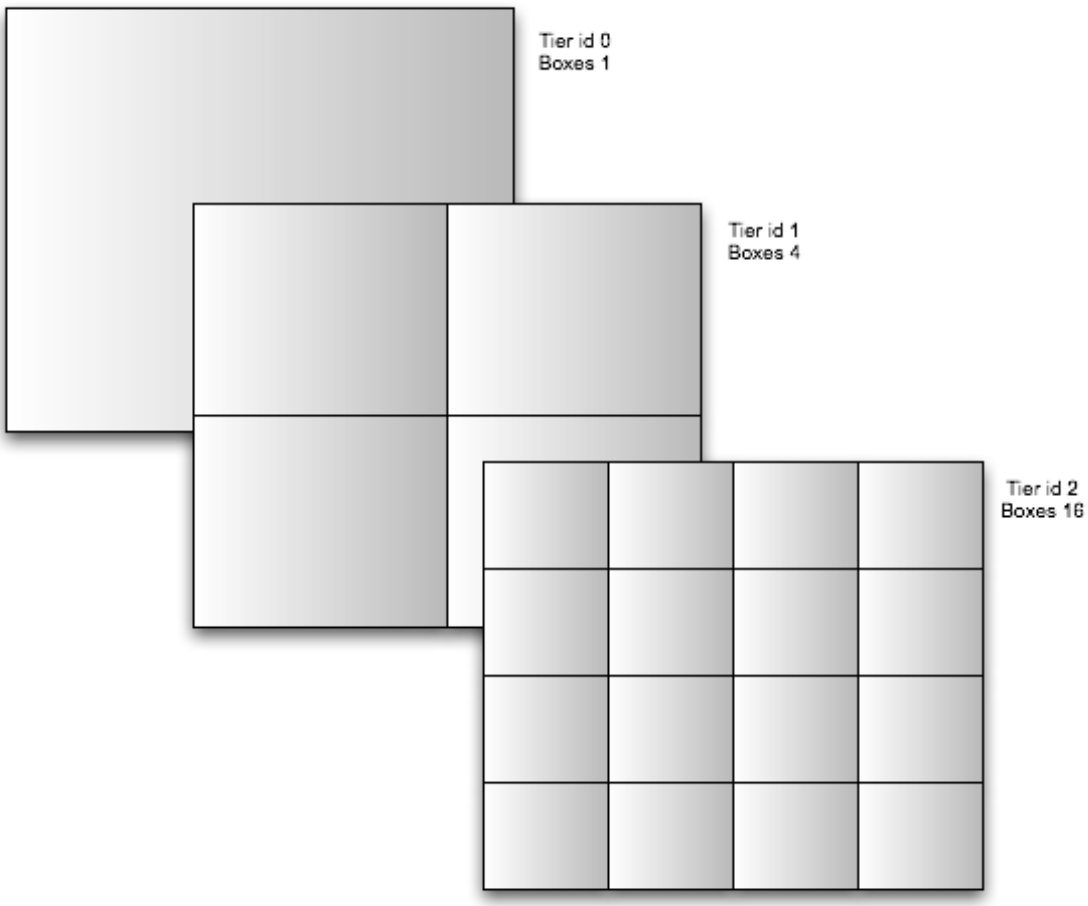


Figure 8.XXX Tiers and grid boxes

TIERS AND GRID BOXES

Once each location is flattened through projection, it is mapped a hierarchical series of tiers and grid boxes as shown in Figure 8.XXX. Tiers divide the 2d grid into smaller and smaller square grid boxes, where each grid box is assigned a unique id; as each tier gets higher the grid boxes become finer.

This allows quick retrieval of locations stored at different levels of granularity. For instance, imagine you have 1 million documents representing different parts of the US, and want every document that has a location in the West Coast of the US. If you were storing just the raw document locations, you would have to iterate through every one of those million documents to if its location is inside your search radius.

But using grids you can say: my search radius is about 1,000 miles, so the tier that can best fit a 1,000 mile radius is tier 9, and grid reference -3.004 and -3.005 contain all the items I need, you now simply retrieve by 2 terms in Lucene to find the corresponding items. Two term retrievals versus 1 million iterations is a major cost and time savings.

Listing 8.XXX shows how to index documents with spatial lucene. We use `CartesianTierPlotter` to create grid boxes for tiers 5 through 15.

Listing 8.XXX Indexing a document for spatial search

```
public class SpatialLuceneExample {

  String latField = "lat";
  String lngField = "lon";
  String tierPrefix = "_localTier";

  private Directory directory;
  private IndexWriter writer;

  SpatialLuceneExample() throws IOException {
    directory = new RAMDirectory();
    writer = new IndexWriter(directory, new WhitespaceAnalyzer(),
                             MaxFieldLength.UNLIMITED);
  }

  private void addLocation(IndexWriter writer, String name, double lat,
                           double lng) throws IOException {

    Document doc = new Document();
    doc.add(new Field("name", name, Field.Store.YES,
                      Field.Index.ANALYZED));

    doc.add(new Field(latField, NumberUtils.double2sortableStr(lat),  // #1
                      Field.Store.YES, Field.Index.NOT_ANALYZED));    // #1
    doc.add(new Field(lngField, NumberUtils.double2sortableStr(lng),  // #1
                      Field.Store.YES, Field.Index.NOT_ANALYZED));    // #1

    doc.add(new Field("metafile", "doc", Field.Store.YES,             // #2
                      Field.Index.ANALYZED));

    IProjector projector = new SinusoidalProjector();      // #3

    int startTier = 5;                                     // #4
    int endTier = 15;                                      // #5

    for (; startTier <= endTier; startTier++) {
      CartesianTierPlotter ctp;
      ctp = new CartesianTierPlotter(startTier,               // #6
                                     projector, tierPrefix);  // #6

      double boxId = ctp.getTierBoxId(lat, lng);              // #7
      System.out.println("Adding field " + ctp.getTierFieldName() + ":"
                         + boxId);
      doc.add(new Field(ctp.getTierFieldName(), NumberUtils   // #8
                        .double2sortableStr(boxId), Field.Store.YES,
```

```
                        Field.Index.NO_NORMS));

    }

    writer.addDocument(doc);
    System.out.println("===== Added Doc to index ====");
  }
```

#1 Convert the lat / long to lucene fields to lucene sortable doubles
#2 Add a default meta field to make searching all documents easy
#3 Sinusoidal projection
#4 About 1000 mile bestFit
#5 about 1 mile bestFit
#6 Create a Tier plotter for each level
#7 Compute bounding box ID
#8 Add the tier field

The most important part is the loop that creates the tiers for each location to be indexed. You start by creating a `CartesianTierPlotter` for the current tier:

```
ctp = new CartesianTierPlotter(startTier, projector, tierPrefix);
```

The parameters are:

- `tierLevel`, in our case starting at 5 and going to 15.
- `projector` is the `SinusoidalProjector` which is the method to project latitude and longitude to a flat surface
- `tierPrefix`, the string used as the prefix of the field name in our case "`_localTier`"

We then call `ctp.getTierBoxId(lat, lng)` with the latitude and longitude values. This returns the id of the grid box that will contain the latitude longitude values at this tier level, which is a double representing x,y coordinates. For example, adding `field _localTier11:-12.0016` would mean at zoom level 11 box -12.0016 contains the location you've added, at grid position x=-12, y=16. This provides a very rapid method for looking up values in an area, and finding its nearest neighbors. The method `addLocation` is very simple to use:

```
addLocation(writer,"TGIFriday",38.8725000,-77.3829000);
```

will add somewhere called "TGIFriday" with its latitude and longitude coordinates to a lucene spatial index. Let's now see how to search the spatial index.

8.16.2 Searching spatial data

Once you have your data indexed, you'll need to retrieve it. Listing 8.XXX shows how. Below we will create a method to perform a normal text search that filters and sorts according to distance from a specific origin. This is the basis of a standard local search.

Listing 8.XXX Sorting and filtering by spatial criteria

```
public void findNear(String what, double latitude, double longitude,
                     double radius) throws CorruptIndexException, IOException {
  IndexSearcher searcher = new IndexSearcher(directory);

  DistanceQueryBuilder dq;
  dq = new DistanceQueryBuilder(latitude,   // #1
                                longitude,
                                radius,
                                latField,   // #2
                                lngField,   // #3
                                tierPrefix, // #4
                                true);      // #5

  Query tq;
  if (what == null)
    tq = new TermQuery(new Term("metafile", "doc"));   // #6
  else
    tq = new TermQuery(new Term("name", what));

  DistanceSortSource dsort;                            // #7
  dsort = new DistanceSortSource(dq.distanceFilter);   // #7
  Sort sort = new Sort(new SortField("foo", dsort));   // #7

  TopDocs hits = searcher.search(tq, dq.getFilter(), 10, sort);

  int numResults = hits.totalHits;

  Map<Integer,Double> distances = dq.distanceFilter.getDistances();  // #8

  System.out.println("Number of results: " + numResults);
  System.out.println("Found:");
  for(int i =0 ; i < numResults; i++) {
    int docID = hits.scoreDocs[i].doc;
    Document d = searcher.doc(docID);

    String name = d.get("name");
    double rsLat = NumberUtils.SortableStr2double(d.get(latField));
    double rsLng = NumberUtils.SortableStr2double(d.get(lngField));
    Double geo_distance = distances.get(docID);

    System.out.printf(name +": %.2f Miles\n", geo_distance);
    System.out.println("\t\t("+ rsLat +","+ rsLng +")");
  }
}
```

#1 Create a distance query
#2 Name of latitude field in index
#3 Name of longitude field in index
#4 Prefix of tier fields in index
#5 Filter by radius

#6 Matches all documents
#7 Create a distance sort
#8 Get distances from the distance filter

#7 As the radius filter has performed the distance calculations already, pass in the filter to our `DistanceSortSource` to reuse the results.

The key component during searching is `DistanceQueryBuilder`. The parameters are:

- `latitude` and `longitude` of the center location (origin) for the search
- `radius` of your search
- `latField` and `lngField`, the names of the latitude and longitude fields in the index
- `tierPrefix` the prefix of the spatial tiers in the index, which must match the `tierPrefix` used during indexing
- `needPrecise`, true if you intend to filter precisely by distance

Probably the only parameter that is non-obvious is `needPrecise`. To ensure that all results fit in a radius, the distance from the center location of your search may be calculated for every potential result. Sometimes that precision is not needed. For instance, to filter for all locations on the West Coast of the US, which is a fairly arbitrary request, a minimal bounding box could suffice in which case you would leave `needPrecise` as `false`. If you need precisely filtered results, or you intend to sort by distance, you must specify `true`.

As distance is a dynamic field, and not part of the index, we must use spatial Lucene's `DistanceSortSource`, which takes the `distanceFilter` from the `DistanceQueryBuilder`, as it contains all the distances for the query. Note that the field name ("foo" in our example) is completely unused because `DistanceSortSource` provides the sorting information. Section 6.XXX describes custom sorting. Let's finish our example.

FINDING THE NEAREST RESTAURANT

We've seen how to populate an index with the necessary information for spatial searching and how to construct a query that filters and sorts by distance. Let's put the finishing touches on it, combining what we've done so far with some actual spatial data, seen in Listing 8.XXX. We've added an `addData` method, to enroll a bunch of bars, clubs and restaurants into the index, plus main function that creates the index and then does a search for the nearest restaurant.

Listing 8.XXX Putting it all together

```
public static void main(String[] args) throws IOException {
  SpatialLuceneExample spatial = new SpatialLuceneExample();
  spatial.addData();
```

```
    spatial.findNear("Restaurant", 38.8725000, -77.3829000, 8);
  }

  private void addData() throws IOException {
    addLocation(writer, "McCormick & Schmick's Seafood Restaurant",
             38.9579000, -77.3572000);
    addLocation(writer, "Jimmy's Old Town Tavern", 38.9690000, -77.3862000);
    addLocation(writer, "Ned Devine's", 38.9510000, -77.4107000);
    addLocation(writer, "Old Brogue Irish Pub", 38.9955000, -77.2884000);
    addLocation(writer, "Alf Laylah Wa Laylah", 38.8956000, -77.4258000);
    addLocation(writer, "Sully's Restaurant & Supper", 38.9003000, -77.4467000);
    addLocation(writer, "TGIFriday", 38.8725000, -77.3829000);
    addLocation(writer, "Potomac Swing Dance Club", 38.9027000, -77.2639000);
    addLocation(writer, "White Tiger Restaurant", 38.9027000, -77.2638000);
    addLocation(writer, "Jammin' Java", 38.9039000, -77.2622000);
    addLocation(writer, "Potomac Swing Dance Club", 38.9027000, -77.2639000);
    addLocation(writer, "WiseAcres Comedy Club", 38.9248000, -77.2344000);
    addLocation(writer, "Glen Echo Spanish Ballroom", 38.9691000, -77.1400000);
    addLocation(writer, "Whitlow's on Wilson", 38.8889000, -77.0926000);
    addLocation(writer, "Iota Club and Cafe", 38.8890000, -77.0923000);
    addLocation(writer, "Hilton Washington Embassy Row", 38.9103000,
             -77.0451000);
    addLocation(writer, "HorseFeathers, Bar & Grill", 39.01220000000001,
             -77.3942);
    writer.close();
  }
```

We add a list of named locations using `addData`. Then, we search for the word "Restaurant" in our index within 8 miles from location (38.8725000, -77.3829000). You can run this by entering "ant SpatialLucene" at the command prompt. You should see the following result:

```
Number of results: 3
Found:
Sully's Restaurant & Supper: 3.94 Miles
        (38.9003,-77.4467)
McCormick & Schmick's Seafood Restaurant: 6.07 Miles
        (38.9579,-77.3572)
White Tiger Restaurant: 6.74 Miles
        (38.9027,-77.2638)
```

As our final topic, let's look at the performance of spatial Lucene.

8.16.3 Performance characteristics of spatial

Unlike standard text search, which relies heavily on an inverted index where duplication in words actually reduces the size of an index and improves retrieval time, spatial locations have a tendency to be very unique. The introduction of a Cartesian grid with tiers provides the ability to bucketize the locations into non-unique grids of different size, thus improving retrieval time. However calculating distance still relies on visiting individual locations in the index. This presents several problems

1. Memory consumption can be high storing unique fields for both latitude and longitude.

2. Results can have varying density.

3. Distance calculations are by nature complex and slow.

MEMORY

Memory can be reduced by using the `org.apache.lucene.spatial.geohash` methods, which condense the latitude and longitude fields into a single hash field[14]. The `DistanceQueryBuilder` supports geohash with its constructor:

```
DistanceQueryBuilder (double lat, double lng, double miles,
                      String geoHashFieldPrefix,
                      String tierFieldPrefix,
                      boolean needPrecision)
```

There is a trade off in the additional processing overhead though for encoding and decoding the geohash fields.

DENSITY OF RESULTS

As you can imagine, a search for a pizza restaurant in Death Valley versus New York City will have different characteristics. The more results you have the more distance calculations you will need to perform. Distribution and multithreading helps here; the more concurrent work you can spread across threads and cpus, the quicker the response.

Caching really doesn't help here, although spatial Lucene does cache overlapping locations, as usually the center location of your search can change more frequently than your search term. Thus the key is spread the load as evenly as possible.

> **NOTE**
>
> Do not index all of your data by regions as you will find an uneven distribution of load. Cities will generally have more data than suburbs, thus taking more processing time. Furthermore, more people will search for results in cities versus suburbs.

PERFORMANCE NUMBERS

As a rough performance test, we evaluated a textual query that filters and sorts by distance. A single thread was used, running on a 3.06GHz, 1.5 Java virtual machine with 500 MB heap. The searcher was first warmed with 5 queries, and time was average of 5 requests for all documents with varying radii. There were 647,860 total documents in the index.

Table 8.XXX shows the results: Number of results is the number of documents that are returned by the query; Time to find the results is the amount of time for the boundary box calculation without the precise distance calculation; Time to calculate distance is the additional time required to get the precise result.

[14] See http://en.wikipedia.org/wiki/Geohash for a good description of what a geohash is.

Number of Results	Time to find results	Time to filter by distance
9,959	7 ms	520 ms
14,019	10 ms	807 ms
80,900	12 ms	1,650 ms

Table 8.XXX Performance results

It's clear from the above table that large sets of spatial data can be retrieved from the index rapidly: 12 ms for 80,900 items in a Cartesian boundary box is quite fast. However, a significant amount of time is consumed calculating all the precise result distances to filter out any that might exist outside of the radius and to enable sorting.

NOTE

If your main concern is the search score, and a rough bounding box will suffice for precision, e.g. all documents in the West Coast of the US vs. all documents precisely within 1,000 miles sorted by distance, then use the `DistanceQueryBuilder` with `needPrecise` set to false. Distances can be calculated at display time with `DistanceUtils.getInstance().getDistanceMi(search_lat, search_long, result_lat, result_lng);`

We'll now finish up this chapter by showing you to build the sandbox packages.

8.15 Building the Sandbox

Many of the packages from the sandbox repository are included in the standard Lucene releases, under the contrib directory. Each package generally has its own jar files for the classes and the javadocs.

Still, some packages are not part of the build and release process. Further, there may be recent improvements not yet released that you'd like to use. To handle these cases you'll need to access the source code and build the packages yourself. Fortunately, this is straightforward! You can easily obtain Lucene's source code directly via Apache's SVN access and either build the JAR files and incorporate the binaries into your project or copy the desired source code into your project and build it directly into your own binaries.

8.15.1 Check it out

Using a Subversion client (see http://subversion.tigris.org), follow the instructions provided at the Apache site: http://wiki.apache.org/lucene-java/SourceRepository. Specifically, this involves executing the following command from the command line:

```
% svn checkout http://svn.apache.org/repos/asf/lucene/java/trunk lucene-trunk
```

This is read-only access to the repository. In your current directory, you'll now have a subdirectory named lucene-trunk. Under that directory is a contrib directory where all the goodies discussed here, and more, reside. Let's build the JARs.

8.15.2 Ant in the Sandbox

Next, let's build the components. You'll need Ant 1.6.x or later in order to run the Sandbox build files. At the root of the lucene-trunk directory is a build.xml file. From the command line, with the current directory lucene-trunk, execute:

```
ant build-contrib
```

Most of the components will build and create a distributable JAR file in the build subdirectory. Now is also a good time to execute ant test, which runs all core and contrib unit tests, to confirm all of Lucene's tests are passing.

Some components, such as javascript, aren't currently integrated into this build process, so you need to copy the necessary files into your project. Some outdated contributions are still there as well (these are the ones we didn't mention in this chapter), and additional contributions will probably arrive after we've written this.

Each contrib subdirectory, such as analyzers and ant, has its own build.xml file. To build a single component, set your current working directory to the desired component's directory and execute `ant`. This is still a fairly crude way of getting your hands on these add-ons to Lucene, but it's useful to have direct access to the source. You may want to use the Sandbox for ideas and inspiration, not necessarily for the exact code.

8.16 Summary

Don't reinvent the wheel. Someone has no doubt encountered the same situation you're struggling with. The Sandbox and the other resources listed on the Lucene web site should be your first stops.

One widely used package is highlighter, enabling you to extract summaries for each hit, and highlight the terms that matched the user's query. This functionality is incredibly important and difficult to implement yourself!

Another very important package is spellchecker, for detecting mis-spelled words, generating possible suggestions, and sorting the suggestions to select the best one.

Several useful tools (Limo, lucli and Luke) let you peek into and index and see all sorts of details.

Spatial lucene is a delightful package allowing you to add geographic distance filters and sorting to your search application.

Rounding out our coverage are a number of interesting packages. WordNet's synonyms can be easily incorporated into your indexing process. XmlQueryParser aims to simplify creation of a rich search user-interface. The surround QueryParser enables a rich query language for span queries. The Javascript package gives browser control over query construction validation. Fast in-memory indices can be created using either `MemoryIndex` or `InstantiatedIndex`.

We saw a number of interesting new `Query` implementations, including a generic `MoreLikeThis` class for finding documents similar to a provided original. A great many analyzers offer support for many languages, shingles and ngrams. We saw a nice ant integration, enabling you to build a Lucene index as part of your build process. You can easily store your index in a Berkeley DB (BDB) directory, giving you all the features of BDB such as full transactions.

If you end up rolling up your sleeves and creating something new and generally useful, please consider donating it to the Sandbox or making it available to the Lucene community. We're all more than grateful for Doug Cutting's generosity for open-sourcing Lucene itself. By also contributing, you benefit from a large number of skilled developers who can help review, debug, and maintain it; and, most important, you can rest easy knowing you have made the world a better place!

9

Using Lucene from other languages

This chapter covers

- Accessing Lucene from other programming languages
- Different styles of ports
- Comparing ports' APIs, features, and performance

Today, Lucene is the de facto standard open-source IR library. Although Java is certainly a very popular programming language, not everyone uses it. Many people prefer dynamic languages (Python, Ruby, Perl, PHP, etc.). What do you do if you love Lucene but not Java? Fear not: you are in good company! Luckily, a number of options are available for accessing Lucene functionality from different languages.

In this chapter we discuss different options for accessing Lucene's functionality from programming languages other than Java. We'll provide brief examples of the ports' use, but keep in mind that each port is an independent project with its own mailing lists, documentation, tutorials, user, and developer community that will be able to provide more detailed information. We begin with our rather loose interpretation of what a *port* is.

9.1 What is a Port?

In this chapter we use the term "port" somewhat loosely. In fact there are 4 very different approaches to access Lucene from other programming languages, summarized in Table 9.1.

Table 9.1: Approaches for Lucene's "ports".

Approach	Description	Ports	Pros	Cons
Native port	All of Lucene's sources are ported to the target environment	Lucene.Net CLucene KinoSearch Ferret Lucy ZendFramework	Lightweight runtime	Port is costly, so high release delay. Possibly higher chance of bugs Likely to be less compatible with Lucene java
Reverse native port	The target language runs on a JVM	Jython, JRuby	Lightweight runtime 100% compatibility with Lucene	Target language may lose some features, eg native extensions
Local Wrapper	A JVM is embedded into the native language's runtime, and a wrapper is used to expose Lucene's API	PyLucene	Port is fast, so lower release delay, since only Lucene's APIs need to be exposed 100% compatibility with Lucene	Heavier, since two runtime environments are running side by side
Client/Server	A separate process, perhaps on a separate machine, runs Lucene Java and exposes a standard protocol for access. Clients in the target language are then created.	Solr + clients PHP Bridge Beagle	Clients are very fast to build Solr provides functionality beyond Lucene and is very actively developed 100% compatibility with Lucene	Much heavier weight since you now have a whole server to manage

A *native port* translates all of Lucene's sources into the target runtime environment. Lucene.Net, which rewrites all of Lucene in C#, is a good example. Another example is KinoSearch, which provides Lucene-like functionality with a C core and Perl bindings. Since C or C++ is the accepted extensions language for many dynamic languages, such as Perl and Python, we count this as a native port.

A reverse native port is the mirror image of a native port: the target runtime environment has been ported to run on a JVM. You write programs in your target language, such as Ruby, but the environment that runs your programs runs on a JVM and therefore has full access to any Java APIs, including Lucene. JRuby and Jython are good examples of this approach.

The local wrapper approach runs a JVM under the hood, side by side with the "normal" runtime for the target language, and then only the APIs that need exposing are wrapped to the target environment. PyLucene is a good example of this approach.

Finally, in the client/server approach, Lucene is running in a separate process, perhaps on another computer, and is accessible using a standard network-based protocol. The server could be just the JVM, as is the case with the PHP Bridge, or it may be a full server like Solr, which implements an XML over HTTP API for accessing Lucene and provides additional functionality beyond Lucene such as distributed search and faceted navigation. Clients are then developed, in multiple programming languages, to interact with the server over the network, using the target language.

There are numerous differences between these approaches, which we visit next.

9.1.1 Tradeoffs

Each approach has important tradeoffs, also summarized in Table 9.1. The native port has the advantage of running only code for the target environment, within a single process. It's perhaps the cleanest, technically, and most lightweight approach, because a single runtime environment is running all code. But the downside is the cost of maintaining this port as Lucene's sources improve with time, which means longer release delay, higher chances that the port will differ from Lucene in API and index file format and a higher risk that the project will be abandoned, as the efforts to continuously port source changes are significant. The native port is also likely to have substantially different performance characteristics, depending on whether the target environment is faster or slower than the JVM.

The reverse native port is a compelling option, assuming the runtime environment itself does not have problems running the target language. By using JRuby, you write ruby code that has access to any Java code, but will generally lose access to Ruby extensions that are implemented in C. This option is also lightweight at runtime, since it runs in a single process and with a single (JVM) runtime environment.

The wrapper approach is similarly a single process, but it embeds a JVM (to run the compiled Java bytecode from Lucene) as well as running the target environment's runtime, side by side, so it's somewhat heavier weight. The important tradeoff is that much less work is required to stay current with Lucene's releases: only the API changes need to be ported, and not Lucene's entire implementation, so the work is in proportion to the net API "surface area" and the release delay can be much less. With PyLucene in particular, which auto-generates the wrapper code using JCC, the delay is essentially zero because the computer does all the work! If only other wrappers could use JCC.

Finally, the client/server approach is the most strongly decoupled. Because a separate server runs and exposes Lucene's APIs via a standard network protocol, you can now share this server amongst multiple clients, possibly with different programming languages. But one potential downside is you now must manage a new standalone process or server, entirely different from your main application.

9.1.2 Choosing the right port

Having so many different approaches for ports seem daunting at first, but in reality this gives a lot of flexibility to people who create the ports, which in turn gives you more options to choose from. If your application is already server-centric, and you're in love with PHP, then the client/server model (Solr as server and SolrPHP client) is a no-brainer. In fact, server based applications often require a client/server search architecture so that multiple front-end computers can share access to the search server(s). At the other end of the spectrum, if you're coding up a C++ desktop application and you can't afford a separate server let alone a separate process, choose a native port like CLucene.

Ports have a tendency to come and go. Often it's one person driving the port, and if they lose interest or can't afford the ongoing time, the port slowly dies. New ports, with new approaches, may surface and attract more interest. This is the natural evolution in the open-source world. While we do our best to describe all of Lucene's ports, today, very likely by the time you read this there will be other compelling options. Be sure to do your due diligence, by searching the Web, an asking questions on the user's lists, etc, before making your final decision.

Although each port tries to remain in sync with the latest Lucene version, they all necessarily lag behind Lucene's releases. Furthermore, most of the ports are relatively young, and from what we could gather, there are little developer community overlaps. Each port takes some and omits some of the concepts from Lucene, but they all mimic its architecture. Each port has its own web site, mailing lists, and everything else that typically goes along with open-source projects. Each port also has its own group of founders and developers. There is also little communication between the ports' developers and Lucene's developers, although we're all aware of each project's existence. With this said, let's look at each port, starting with Solr.

9.2 Solr and its numerous clients

Solr, which is covered in detail in Chapter XXX, is a client/server architecture exposing access from many programming languages. Table 9.2 summarizes Solr's current status. In a nutshell, Solr is a server wrapper around Lucene. It provides a standard XML over HTTP interface for interacting with Lucene's APIs, and also layers on further functionality not normally available in Lucene, such as distributed search, faceted navigation and a field schema. Because Solr "translates" Lucene's Java-only API into a very friendly network protocol, it's very easy to

create clients in different programming languages that then speak this network protocol. For this reason, of all approaches for accessing Lucene from other languages, Solr offers the least porting effort.

Table 9.2 Solr summary

Port type	Client/server
Programming Languages	Java + many client wrappers
Web site	http://lucene.apache.org/solr/
Development Status	Stable
Activity	Active development, active users
Last stable release	1.3
Matching Lucene release	3.0
Compatible index format	Yes, 3.0
Compatible APIs	No
License	Apache License 2.0

Table 9.3 The many Solr clients currently available.

Name	Language/Environment
SolRuby, acts_as_solr	Ruby/Rails
SolPHP	PHP
SolJava	Java (yes, Java)
SolPython	Python
SolPerl, Solr.pm	Perl (http://search.cpan.org/perldoc?Solr)
SolJSON	JavaScript
SolrJS	JavaScript (http://solrjs.solrstuff.org/)
SolForrest	Apache Forrest/Cocoon
SolrSharp	C#
Solrnet	http://code.google.com/p/solrnet/
SolColdFusion	ColdFusion plugin

Solr has a delightful diversity of clients, shown in Table 9.3. Be sure to check http://wiki.apache.org/solr/IntegratingSolr for the latest complete list. If you need to access Lucene from an exotic language, chances are there is already at least one Solr client. And if there isn't, it's very easy to create one! Solr is very actively developed and of course has excellent compatibility with Lucene since it uses Lucene under-the-hood. If your application can accept the addition of a standalone server, Solr is likely a very good fit. Next let's look at CLucene.

9.3 Clucene (C++)

CLucene is Ben van Klinken's open-source port of Lucene to C++. Table 9.4 shows its current status. Ben is an Australian pursuing a Masters Degree in International Relations and Asian Politics. Although his studies aren't in a technology-related field, he has strong interest in Information Retrieval. While Ben was the original creator, many other active developers now participate in this port. Ben was kind enough to provide this overview of CLucene.

Table 9.4 Clucene summary

Port type	Native port
Programming Languages	C++
Web site	http://sourceforge.net/projects/clucene/
Development Status	Stable
Activity	Active development, active users

Last stable release	0.9.21b
Matching Lucene release	1.9
Compatible index format	Yes, 1.9
Compatible APIs	Yes
License	LGPL or Apache License 2.0

CLucene is a native port to C++: the file format and API are exactly the same as Lucene 1.9, in its latest stable release. The unstable release is actively working towards compatibility with Lucene's 2.3.1 release, and development is proceeding on a source code branch towards making this a stable release. Despite being officially marked unstable, the 2.3.1 branch seems to be quite stable and is commonly used, although the APIs are still changing as of this writing.

The distribution package of CLucene includes many of the same components as Lucene, such as tests and demo examples. CLucene contains wrappers that allow it to be used with other programming languages. Currently there are wrappers for PHP, .NET (read-only), and a Dynamic Link Library (DLL) that can be shared between different programs, and separately developed wrappers for Python and Perl.

9.3.1 Supported platforms

CLucene was initially developed in Microsoft Visual Studio, but now it also compiles in GCC, MinGW32, and (reportedly) the Borland C++ compiler (although no build scripts are currently being distributed). In addition to the MS Windows platform, CLucene has also been successfully built on Red Hat 9, Mac OS X, and Debian. The CLucene team is making use of SourceForge's multiplatform compile farm to ensure that CLucene compiles and runs on as many platforms as possible. The activity on the CLucene developers' mailing lists indicates that support for AMD64 architecture and FreeBSD is being added.

9.3.2 API compatibility

The CLucene API is similar to Lucene's. This means that code written in Java can be converted to C++ fairly easily. The drawback is that CLucene doesn't follow the generally accepted C++ coding standards. However, due to the number of classes that would have to redesigned, CLucene continues to follow a "Javaesque" coding standard. This approach also allows much of the code to be converted using macros and scripts. The CLucene wrappers for other languages, which are included in the distribution, all have different APIs.

Listing 9.1 shows a command-line program that illustrates the indexing and searching API and its use. This program first indexes several documents with a single contents field. Following that, it runs a few searches against the generated index and prints out the search results for each query.

Listing 9.1 Using CLucene's `IndexWriter` and `IndexSearcher` API

```
int main( int argc, char** argv){

try {
  SimpleAnalyzer* analyzer = new SimpleAnalyzer();
  IndexWriter writer( _T("testIndex"), *analyzer, true);

 wchar_t* docs[] = {
  _T("a b c d e"),
  _T("a b c d e a b c d e"),
  _T("a b c d e f g h i j"),
  _T("a c e"),
  _T("e c a"),
  _T("a c e a c e"),
  _T("a c e a b c")
 };

 for (int j = 0; j < 7; j++) {
  Document* d = new Document();
  Field& f = Field::Text(_T("contents"), docs[j]);
  d->add(f);

  writer.addDocument(*d);
  // no need to delete fields - document takes ownership
```

```
    delete d;
  }
  writer.close();

  IndexSearcher searcher(_T("testIndex"));
  wchar_t* queries[] = {
   _T("a b"),
   _T("\"a b\""),
   _T("\"a b c\""),
   _T("a c"),
   _T("\"a c\""),
   _T("\"a c e\""),
  };
  Hits* hits = NULL;
  QueryParser parser(_T("contents"), *analyzer);

  parser.PhraseSlop = 4;
  for (int j = 0; j < 6; j++) {

    Query* query = &parser.Parse(queries[j]);
    const wchar_t* qryInfo = query->toString(_T("contents"));
    _cout << _T("Query: ") << qryInfo << endl;
    delete qryInfo;

    Hits* hits = &searcher.search(*query);
    _cout << hits->Length() << _T(" total results") << endl;

    for (int i=0; i<hits->Length() && i<10; i++) {
      Document* d = &hits->doc(i);
      cout << i << _T(" ") << hits->score(i) <<
      _T(" ") << d->get(_T("contents")) << endl;
    }
    delete hits;
    delete query;
  }

  searcher.close();
  if ( analyzer )
    delete analyzer;
  } catch (THROW_TYPE e) {
   _cout << _T(" caught a exception: ") <<
     e.what() << _T("\n");
  } catch (...){
   _cout << _T(" caught an unknown exception\n");
  }
```

Many applications have to deal with characters outside the ASCII range. Let's look at some Unicode-related issues we mentioned earlier.

9.3.3 Unicode support

CLucene was originally written to be as fast and lightweight as possible. In the interest of speed, the decision was made not to incorporate any external libraries for string handling and reference counting. However, there are some drawbacks to this. Linux suffers from a lack of good Unicode support, and since CLucene doesn't use external libraries, Linux builds had to be built without Unicode. This led to CLucene using the _UNICODE pre-processor directive: When it's specified, the Unicode characters are used; otherwise, non-Unicode (narrow) characters are used. However, support for Unicode is included in CLucene and can be enabled at compile-time. Future version may also solve this problem by optionally including a Unicode library.

9.3.4 Performance

According to a couple of reports captured in the archives of the Lucene Developers mailing list, CLucene indexes documents faster than Lucene. We haven't done any benchmarks ourselves because doing so would require going back to version 1.2 of Lucene (not something a new Lucene user would do).

9.3.5 Users

Although the CLucene port has been around for a while and has an active user mailing list, we haven't been able to locate many actual CLucene users to list here. This could be due to the fact that the CLucene development team is

small and has a hard time keeping up with features being added to Lucene. We did find out about Awasu, a personal knowledge–management tool that uses CLucene under the covers (http://www.awasu.com/). Our next port is based on Microsoft's .NET framework.

9.4 Lucene.Net

Lucene.Net is a native port of Lucene to C#. It's current status is summarized in Table 9.5. While the last actual release is 2.0, the trunk of Lucene.Net subversion repository matches Lucene 2.3.1 and appears to be quite stable. The distribution package of Lucene.Net consists of the same components as the distribution package of Lucene. It includes the source code, tests, and a few demo examples.

Table 9.5 Lucene.Net summary

Port type	Native port
Programming Languages	C#
Web site	http://incubator.apache.org/lucene.net/
Development Status	Stable
Activity	Active development, active users
Last stable release	2.0
Matching Lucene release	2.0
Compatible index format	Yes, 2.0
Compatible APIs	Yes
License	Apache License 2.0

9.4.1 API compatibility

Although it's written in C#, Lucene.Net exposes an API that is nearly identical to that of Lucene. Consequently, code written for Lucene can be ported to C# with minimal effort. This compatibility also allows .NET developers to use documentation for the Java version, such as this book.

The difference is limited to the Java and C# naming styles. Whereas Java's method names begin with lowercase letters, the .NET version uses the C# naming style in which method names typically begin with uppercase letters.

9.4.2 Index compatibility

Lucene.Net is compatible with Lucene at the index level. That is to say, an index created by Lucene can be read by Lucene.Net and vice versa. Of course, as Lucene evolves, indexes between versions of Lucene itself may not be portable, so this compatibility is currently limited to Lucene version 1.4.

9.4.3 Performance

The developers of Lucene.Net don't have any performance numbers at this time, and they're focused on adding features to their port to ensure it stays as close to Lucene as possible. However, it would be safe to assume that Lucene.Net's performance is similar to that of its precursor; according to Lucene.Net's author, its performance was comparable to that of Lucene.

9.4.4 Users

One interesting user of Lucene.Net is Beagle (http://beagle-project.org/Main_Page), a search tool for searching your personal information space, including local files, email, images, calendar entries, addressbook entries, etc. Beagle is a large project in itself. Its design is just like Solr: there is a dedicated daemon process that exposes a network API, and then clients are available in various programming languages (currently at least C#, C and Python). Beagle seems to be well adopted by Linux desktop environments as their standard local search implementation, whereby Beagle runs under Mono.

9.5 KinoSearch and Lucy (Perl)

Perl is a popular programming language. Larry Wall has stated one of his goals in Perl is to offer many ways to accomplish a given task. Larry would be proud, as there are quite a few choices for accessing Lucene's functionality from Perl.

We'll first visit the most popular choice, KinoSearch. After that we touch on Lucy, which is still under active development and hasn't had any releases yet but is nevertheless interesting. We finish with Solr's two Perl clients and CLucene's Perl bindings.

9.5.1 KinoSearch

KinoSearch, created and actively maintained by Marvin Humphrey, is a C and Perl "loose port" of Lucene. This means its approach, at a high level, is similar to Lucene, but the architecture, APIs and index file format are not identical. The summary of its current status is shown in Table 9.6. Marvin took the time to introduce interesting innovations to KinoSearch while porting to Perl and C; some of these innovations have inspired corresponding improvements back to Lucene, which is one of the delightful and natural "cross fertilization" effects of open source development.

Table 9.6 KinoSearch summary

Port type	Native port
Programming Languages	C, Perl
Web site	http://www.rectangular.com/kinosearch/
Development Status	Alpha (though widely used and quite stable)
Activity	Active development, active users
Last stable release	0.163
Matching Lucene release	N/A (loose port)
Compatible index format	No
Compatible APIs	No
License	Custom

KinoSearch is technically in the alpha stage of its development, but in practice is nevertheless extremely stable, bug free and widely used in the Perl community. Development and users lists are active, and developers (mostly Marvin) are working toward the 1.0 first stable release. It's hard to gauge usage, but at least two well-known web sites, Slashdot.org and Eventful.com, use it. When users find issues and post questions to the mailing lists, Marvin is always very responsive.

KinoSearch also learned important lessons from an earlier port of Lucene to Perl, PLucene. PLucene, which has stopped development, suffered from performance problems, likely because it was implemented entirely in Perl; KinoSearch instead wraps Perl bindings around a C core. This allows the C core to do all the "heavy lifting", which results in much better performance. Early testing of KinoSearch showed its indexing performance to be close to Lucene's 1.9.1 release. However both KinoSearch and Lucene have changed quite a bit since then, so it's not clear how they compare today.

Probably the largest architectural difference is that KinoSearch requires you to specify field definitions up front when you first create the index (similarly to how you create a database table). The fields in documents then must match this pre-set schema. This allows KinoSearch to make internal simplifications, which gain performance, but at the cost of full document flexibility that is available in Lucene.

There are also a number of API differences. For example, there is only one class, `InvIndexer`, for making changes to an index (whereas Lucene has two classes for doing so, IndexWriter and, somewhat confusingly, IndexReader). The index file format is also different, though similar. Listings 9.2 and 9.3 show examples for creating and search an index.

Listing 9.2: Creating an index with KinoSearch

```
use KinoSearch::InvIndexer;
use KinoSearch::Analysis::PolyAnalyzer;

my $analyzer
    = KinoSearch::Analysis::PolyAnalyzer->new( language => 'en' );

my $invindexer = KinoSearch::InvIndexer->new(
    invindex => '/path/to/invindex',
    create   => 1,
    analyzer => $analyzer,
);

$invindexer->spec_field(
    name  => 'title',
    boost => 3,
);
$invindexer->spec_field( name => 'bodytext' );

while ( my ( $title, $bodytext ) = each %source_documents ) {
    my $doc = $invindexer->new_doc;

    $doc->set_value( title    => $title );
    $doc->set_value( bodytext => $bodytext );

    $invindexer->add_doc($doc);
}

$invindexer->finish;
```

Listing 9.3: Searching an index using KinoSearch

```
use KinoSearch::Searcher;
use KinoSearch::Analysis::PolyAnalyzer;

my $analyzer
    = KinoSearch::Analysis::PolyAnalyzer->new( language => 'en' );

my $searcher = KinoSearch::Searcher->new(
    invindex => '/path/to/invindex',
    analyzer => $analyzer,
);

my $hits = $searcher->search( query => "foo bar" );
while ( my $hit = $hits->fetch_hit_hashref ) {
    print "$hit->{title}\n";
}
```

Next we look at Lucy, something of a followon to KinoSearch.

9.5.2 Lucy

Lucy, at http://lucene.apache.org/lucy, is a new Lucene port. It plans to be a loose native port of Lucene to C, with a design that makes it simple to wrap the C code with APIs in different dynamic languages, with the initial focus on Perl and Ruby. Table 9.7 shows the summary of Lucy's current status.

Table 9.7 Lucy summary

Port type	Native port
Programming Languages	C with Perl, Ruby (and eventually others) bindings
Web site	http://lucene.apache.org/lucy/
Development Status	Design (no code/releases yet)
Activity	Active development
Last stable release	N/A
Matching Lucene release	N/A (loose port)
Compatible index format	No
Compatible APIs	No
License	unknown

Lucy was started by the creator of KinoSearch, Marvin Humphrey, and the creator of Ferret (see section XXX below), David Balmain. Unfortunately, David became unavailable and Marvin has now folded Lucy's approach back into KinoSearch main development repository. It may very well be that Lucy is realized as the 2.0 release of KinoSearch. Like Ferret and KinoSearch, Lucy is inspired by Lucene and derives much of its design from those two projects, aiming to achieve the best of both. Eventually other programming languages should be able to wrap Lucy's C core. Perl still offers more options.

9.5.3 Other Perl options

There are other ways to access Lucene's functionality from Perl. There are at least 2 clients for Solr: Solr.pm (available at http://search.cpan.org/perldoc?Solr and separately developed from the Solr effort), and SolPerl which is developed and distributed with Solr. Of course, Solr is a client/server approach. If you have a strong preference for API and index compatible ports, and don't like that KinoSearch is a "loose" port, have a look at CLucene's Perl bindings, which is also a native port of Lucene but with matching APIs and index file formats. Let's move next to another popular dynamic language starting with the letter P.

9.6 PyLucene (Python)

Python, preferring to have one obvious way to do something, in fact has one obvious choice for Lucene functionality: PyLucene. This section was gratefully contributed by the creator of PyLucene, Andi Vajda. Table 9.8 shows PyLucene's current status.

Table 9.8 PyLucene summary

Port type	Local wrapper
Programming Languages	Python, C++, Java
Web site	http://lucene.apache.org/pylucene/
Development Status	Stable
Activity	Active development, active users
Last stable release	3.0
Matching Lucene release	3.0
Compatible index format	Yes
Compatible APIs	Yes
License	Apache Version 2.0

PyLucene takes the "local wrapper" approach, by adding Python bindings to the actual Lucene source code. PyLucene embeds a Java VM with Lucene into a Python process. The PyLucene Python extension, a Python module called "lucene", is machine-generated by a package called JCC, also included with the PyLucene sources. JCC is fascinating in its own right: it is written in Python and C++, and uses Java's reflection API, accessed via an

embedded JVM, to peek at the public API for all classes in a JAR. Once it knows that API, it generates the appropriate C++ code that enables access to that API from Python through JNI (Java's Native Interface), using C++ as the common "bridge" language. Because JCC auto-generates all wrappers by inspecting Lucene's JAR file, the release latency is near zero.

Both PyLucene and JCC are released under the Apache 2.0 license and led by Andi Vajda, who also contributed Berkeley DbDirectory (see section 8.9) to the Lucene codebase. PyLucene began as an indexing and searching component of Chandler (described briefly in section 8.9), an extensible open-source PIM, but it was split into a separate project in June 2004. It was recently (Jan 2009) folded into Apache as a sub-project of Lucene.

9.6.1 API compatibility

The source code for PyLucene is machine-generated by JCC. Hence all public APIs in all public classes available from Lucene are available from PyLucene. JCC exposes iterator and mapping access in pythonic ways making for a true Python experience while using Lucene. Warning: once you've used Lucene from Python, it can be very hard to go back to using Java!

As far as its structure is concerned, the API is virtually the same, which makes it easy for users of Lucene to learn how to use PyLucene. Another convenient side effect is that all existing Lucene documentation can be used for programming with PyLucene.

PyLucene closely tracks the Lucene releases. The latest and greatest from Lucene is usually available via PyLucene a few days after a release.

9.6.2 Performance

The performance of PyLucene should be very similar to that of Lucene since the actual Lucene code is running in an embedded Java VM in-process. The Python/Java barrier is crossed via the Java Native Interface (JNI) and is reasonably fast. Virtually all of the source code generated by JCC for PyLucene is C++. That code uses the Python VM for exposing Lucene objects to the Python interpreter but none of the PyLucene code itself is interpreted Python.

9.6.3 Users

PyLucene was first released in 2004. It has had a number of users over the years. Some Linux distributions, such as Debian, are now beginning to distribute PyLucene and JCC. Currently, the PyLucene developer/user mailing list has about 160 members. Traffic is moderate and usually touches build issues. Lucene issues while using PyLucene are usually handled on the Lucene user mailing list.

9.6.4 Other Python options

While PyLucene is our favorite option for using Lucene from Python, there are other choices with different tradeoffs:

- Solr, a client/server approach, includes the SolPython client.
- If you prefer a native port, CLucene offers Python bindings.
- Beagle, described briefly in section 9.4.4, also includes Python bindings. Like Solr, Beagle is a client/server solution, but the server runs in a .NET environment instead of a JVM.
- If you prefer a reverse port, you could simply use Jython, a port of the Python language to run on a JVM, which has full access to any Java APIs including all releases of Lucene.

As you've seen, there are a number of ways to access Lucene from Python, the most popular being PyLucene. Let's switch to another dynamic language, Ruby.

9.7 Ferret (Ruby)

The Ruby programming language, another dynamic language, has become quite popular recently. Fortunately, you can access Lucene from Ruby in various ways. The most popular port is Ferret, summarized in Table 9.9.

Table 9.9 Ferret summary

Port type	NativeLocal wrapper
Programming Languages	C, Ruby

Web site	http://ferret.davebalmain.com/
Development Status	Stable, though some serious bugs remain
Activity	Development stopped but active users
Last stable release	0.11.6
Matching Lucene release	N/A (loose port)
Compatible index format	No
Compatible APIs	No
License	MIT-style License

Although independently developed, Ferret takes the same approach as KinoSearch, as a loose port of Lucene to C and Ruby. The C core does the heavy lifting, while the Ruby API exposes access to that core. Ferret was created by David Balmain, who has written a dedicated book about Ferret. There is also an acts_as_ferret plugin for Ruby on Rails. Unfortunately, ongoing development on Ferret has ended.

User reports have shown Ferret's performance to be quite good, comparable at least to Lucene's 1.9 release. Even though development appears to have ended, usage of Ferret still appears to be strong, especially for acts_as_ferret, although there are reports of still open serious issues on the most recent release, so you should tread carefully.

9.6.4 Other Ruby options

SolRuby is Solr's Ruby client, allowing you to add, update and delete documents, as well as issue queries. Just install it with "gem install solr-rub". Here's a quick example:

```
require 'solr'

# connect to the solr instance
conn = Solr::Connection.new('http://localhost:8983/solr', :autocommit => :on)

# add a document to the index
conn.add(:id => 123, :title_text => 'Lucene in Action')

# update the document
conn.update(:id => 123, :title_text => 'Solr in Action')

# print out the first hit in a query for 'action'
response = conn.query('action')
print response.hits[0]

# iterate through all the hits for 'action'
conn.query('action') do |hit|
  puts hit.inspect
end

# delete document by id
conn.delete(123)
```

Solr also provides a modified JSON response format that produces valid Ruby source code as the string response, which can be directly eval'd in Ruby even without the SolRuby client. This enables a very compact search solution. There is also an independently developed Rails plugin, acts_as_solr. Finally, Erik has developed Solr Flare, which is a feature rich Rails plugin that provides even more functionality than acts_as_solr.

NOTE

There is even a Common Lisp port called Montezuma, at http://code.google.com/p/montezuma. Development seems to have stopped, after an initial burst of activity. In fact, Montezuma is a port of Ferret, which in turn is a port of Lucene to Ruby (described in section 9.7).

Finally, another compelling option is to use JRuby, which is a reverse port of the Ruby language to run on a JVM. You still write Ruby code, but it's a JVM that's running your Ruby code, and thus any JAR, including Lucene,

is accessible from Ruby. The one downside to JRuby is that it cannot run a Ruby extension that's implemented in C. Let's switch gears now to another popular Web application dynamic language, PHP.

9.8 PHP

There are several interesting options if you'd like to use PHP. The first option is to use Solr with its PHP client, SolPHP, which is a client/server solution. As is the case for Ruby, Solr has a response format that produces valid PHP code, which can simply be eval'd in PHP.

The second option is CLucene's PHP bindings, which are included with CLucene's release, which is a pure native port. Another pure native port is Zend Framework.

9.8.1 Zend Framework

Zend Framework, summarized in table 9.10, is far more than a port of Lucene: it's a full open-source object-oriented web application framework, implemented entirely in PHP 5. It includes a pure native port of Lucene to PHP 5, described at http://framework.zend.com/manual/en/zend.search.lucene.html, enabling you to easily add full search to your web application.

Table 9.10 Zend Framework summary

Port type	Pure native port
Programming Languages	PHP 5
Web site	http://framework.zend.com/
Development Status	Stable
Activity	Active development and active users
Last stable release	1.7.3
Matching Lucene release	2.1
Compatible index format	Yes
Compatible APIs	Yes
License	BSD-style License

There are some reports of slow performance during indexing, though this may have been resolved by more recent releases so you should certainly test for yourself. Earlier releases did not support Unicode content, but this has since been fixed.

Zend Framework may be a good fit for your application if you want a pure PHP solution, but if you don't require a native port and you'd like a lighter weight solution instead, then PHP Bridge may be a good option.

9.8.2 PHP Bridge

The PHP/Java Bridge, hosted at http://php-java-bridge.sourceforge.net/pjb/index.php, is technically a client/server solution. Normal Java Lucene runs in a standalone process, possibly on a different computer, and then the PHP runtime can invoke methods on Java classes through the PHP Bridge. It can also bridge to a running .NET process, so you could also use PHP to access Lucene.Net, for example. The release WAR that you download from the above site even includes examples of indexing and searching with Lucene. For example, this is how you create an IndexWriter:

```
/* create the index files in the tmp dir */
$tmp = create_index_dir();
$analyzer = new java("org.apache.lucene.analysis.standard.StandardAnalyzer");
$writer = new java("org.apache.lucene.index.IndexWriter", $tmp, $analyzer, true);
```

Since this is just a client/server wrapper around Lucene, you can tap directly into the latest release of Lucene. Performance should be close to Lucene's performance, except for the overhead of invoking methods over the bridge. Likely this affects indexing performance moreso than searching performance.

9.10 Summary

In this chapter, we discussed the four different approaches that Lucene's ports use, and we visited all existing Lucene ports known to us: CLucene, Lucene.Net, Pylucene, Solr and its many clients, KinoSearch, Ferret, the upcoming Lucy, and numerous PHP options. We looked at their APIs, supported features, Lucene compatibility, development and user activity and performance as compared to Lucene, as well as some of the users of each port. The future may bring additional Lucene ports; the Lucene developers keep a list on the Lucene Wiki at http://wiki.apache.org/lucene-java/LuceneImplementations. As you can see, there are a great many ways to access Lucene from environments other than Java, each with their own tradeoffs. While this may seem daunting, if you are trying to decide which of these to use, it's actually a great sign of Lucene's popularity and maturity that so many people have created all these different options.

By covering the Lucene ports, we have stepped outside the boundaries of core Lucene. In the next chapter we'll go even further by examining several interesting Lucene case studies.

10

LUCENE PERFORMANCE TUNING AND ADMINISTRATION

In this chapter:

- Tuning for performance
- Effectively using threads
- Predicting, understanding and managing disk, file descriptors and memory usage
- Backing up and restoring your index
- Checking an index for corruption and repairing it
- Understanding common errors

You've seen diverse examples of how to use Lucene for indexing and searching, including many advanced use cases. Here we change gears and cover practical, hands-on administrative aspects of Lucene. Some say administrative details are a mundane and necessary evil, but at least one of your beloved authors would beg to differ! A well tuned Lucene application is like a well maintained car: it will operate for years without problems, requiring only a small, informed investment on your part. You can take pride in that! This chapter gives you all the tools you need to keep your Lucene application in tip top shape.

Lucene has great out-of-the-box performance, but for some demanding applications, this is still not good enough. Fear not! There are many fun ways to tune for performance. Adding threads to your application is often very effective, but the added complexity can make it tricky. We'll show you some simple drop-in classes that hide this complexity. Most likely you can tune Lucene to get the performance you need.

Beyond performance, people are often baffled Lucene's consumption of resources like disk space, file descriptors and memory. Keeping tabs on this consumption, over time, as your index grows and application evolves, is necessary to prevent sudden catastrophic problems. Fortunately, Lucene's use of these resources is simple to predict once you understand how. Armed with this, you can easily prevent many problems.

Of course, what good is great performance if you have no more search index? Despite all your preventative efforts, things will eventually go very wrong (thank you Murphy's Law), and restoring from backup will be your only option. As of 2.3, properly backing up your index, even while you are still adding documents to it, is simple. You have no excuse to delay! Just a little bit of planning ahead will save you a lot of trouble later.

So, roll your sleeves up: it's time to get your hands dirty! Let's jump right in with performance tuning.

10.1 Performance tuning

Many applications achieve awesome performance with Lucene, out of the box. But, you may find that as your index grows larger, and as you add new features to your application, or even as your web site gains popularity and must handle higher and higher traffic, performance could eventually become an issue. Tuning Lucene for better performance is really quite simple.

Before jumping into specific metrics, there are some best practices that you should always follow regardless of what specific tuning you want to do:

- Run a Java profiler, or collect your own rough timing using `System.nanoTime`, to verify your performance problem is in fact Lucene and not your application. For many applications, loading the document from a database or file system, filtering the raw document into plain text, and tokenizing that text, is time consuming. During searching, rendering the results from Lucene might be time consuming. You might be surprised!
- Run your JVM with the `-server` switch, which generally configures the JVM for faster net throughput over time but a possibly higher startup cost.
- Upgrade to the latest release of Lucene. Lucene is always getting better: performance is improved, bugs are fixed, and new features are added. In 2.3 in particular there were numerous optimizations to indexing. The Lucene development community has a clear commitment to backwards compatibility: it is strictly kept across minor releases (2.x) but not necessarily across major releases. A new minor release should just be a drop-in, so go ahead and try it!
- Use a local file system for your index. Local file systems are generally quite a bit faster than remote ones. If you are concerned about local hard drives crashing, use a `RAID` array with redundancy. In any event, be sure to make backups of your index (see 10.4): someday, something will inevitably go horribly wrong.
- Don't re-open `IndexWriter` or `IndexReader`/`IndexSearcher` any more frequently than required. Share a single instance for a long time and re-open only when necessary.
- Use multiple threads. Modern computers have amazing concurrency in CPU, IO and RAM, and that concurrency is only increasing with time. 10.2 covers the tricky details for using threads.
- Use faster hardware: fast CPU and fast IO system (for large indices) will always help.
- Put as much physical memory as you can in your computers, and configure Lucene to take advantage of all of it (see 10.3.3). Be sure Lucene is not using so much memory that your computer is forced to constantly swap or the JVM is forced to constantly GC.
- Budget enough memory, CPU and file descriptors for your peak usage. This is typically when you are opening a new searcher during peak traffic perhaps while indexing a batch of documents.
- Turn off any fields or features that your application is not actually using. Be ruthless!
- Group multiple text fields into a single text field and search only that one.

These best practices will take you a long ways towards better performance. It could be, after following these steps, you are done: congratulations! If not, don't fear: there are still many options to try. In this section we describe the overall testing process and describe some important metrics, including index-to-search-delay (how "realtime" your search application is), indexing throughput and search throughput. For each of these we show how to tune Lucene.

But first, be sure your application really does need faster performance from Lucene. Performance tuning can be a time consuming and, frankly, rather addictive affair. It can also add complexity to your application, which may introduce bugs, making your application more costly to maintain. Ask yourself, honestly (use a mirror, if necessary): could your time be better spent improving the user interface or tuning relevance? You can always improve performance by simply rolling out more or faster hardware, so always consider that option first. Never sacrifice user experience in exchange for performance: keeping users happy, by providing the best experience humanly and computerly possible, should always be your top priority. These are the costs of performance tuning so before you even start make sure you do need really better performance.

So you still have your heart set on tuning performance? No problem: read on!

10.1.1 Testing Process

First, set up a simple repeatable test that allows you to measure the specific metrics you want to improve. Without this you can't know if you're actually improving things. The test should accurately reflect your application. Try to use true documents and searches from your search logs, if available. Next, establish a baseline of your metric. If you see high variance on each run you may want to run the test 3 or more times and discard the outliers (keeping the middle result). Finally, take an open minded iterative approach: performance tuning is empirical and often

surprising. Let the computer tell you what works and what doesn't. Make one change at a time, test it, and keep it only if the metric really improved. Some changes will unexpectedly degrade performance, so don't keep those ones! Make a list of ideas to try, and sort them according to your best estimate of "bang for the buck": those changes that are quick to test and could be the biggest win should be tested first. Once you've improved your metric enough, stop and move onto other important things! You can always come back to your list later and keep iterating.

If all else fails, take your challenge to the Lucene java users list (`java-user@lucene.apache.org`). More than likely someone has already encountered and solved something similar to your problem and your question can lead to healthy discussion on how Lucene could be improved.

For our testing throughout this chapter we will use the framework in `contrib/benchmark`, described in more detail in Appendix D. This is an excellent tool for creating and running repeatable performance tests. It already has support for multiple runs of each test, changing Lucene configuration parameters, measuring metrics, and printing summary reports of the full test run. There are a large set of built-in tasks and document sources to choose from. Extending the framework with your own task is straightforward. You simply write algorithm (`.alg`) file, using a simple custom scripting language, to describe the test. Then run it like this:

```
cd contrib/benchmark
ant run-task -Dtask-alg=<file.alg> -Dtask.mem=XXXM
```

That prints great details about the metrics for each step of your test. Algorithm files also make it simple for others to reproduce your test results: you just send it to them and they run it! Let's look at specific metrics that you may need to tune.

APPLES AND ORANGES

When running indexing tests, there are a couple things to watch out for. First, because Lucene periodically merges segments, when you run two indexing tests with different settings it's quite possible for each resulting index to end in a different merge state. Maybe the first index has only 3 segments in it, because it just completed a large merge, and the other index has 17 segments. It's not really fair to compare metrics from these two tests because in the first case Lucene did more work to make the index more compact. You're really comparing apples and oranges.

To work around this, you could set mergeFactor to an enormous number, to turn off merging entirely. This will make the tests at least comparable, but just remember that the resulting numbers are not accurate in an absolute sense, because in a real application you cannot turn off merging. Of course, this is only meaningful if you are not trying to compare the cost of merging in the first place.

The second issue is to make sure your tests include the time it takes to call `close` on the `IndexWriter`. During `close`, `IndexWriter` flushes documents, may start new merges, and waits for any background merges to finish. Try to write your algorithm files so that the `CloseIndex` task is included in the report.

10.1.2 Metrics

It's important to understand which metric you need to improve, and which are less important, because optimizing one metric is often at the expense of others. Here are some common metrics:

- Index-to-search delay: the time from when a document is added to the index until your users can actually see it in their search results.
- Indexing throughput: how many documents per second can be indexed.
- Search latency: the time it takes for a search to display results to the user. It's best to measure this from the actual end user's search interface (for example a Web browser). You want this number to be no more than 1 second. There are many cumulative delays in a Web search application, so be sure to measure all steps before and after the Lucene search is actually executed to be sure it's really Lucene that needs tuning.
- Search throughput: how many searches per second can your application can handle.

Which metric is important depends on your application, and can vary with time. Often, indexing throughput is crucial while you are first building your index but then once the initial index is complete, index to search latency becomes more important.

10.1.2.1 INDEX-TO-SEARCH DELAY

Index-to-search delay is the elapsed time from when you add or update a document in the index until users can actually search that document. Because a reader always presents the index as of the "point in time" when it was opened, the only way to reduce index-to-search delay is to close your writer and reopen your reader. Unfortunately, these operations are fundamentally costly: they consume IO, CPU, and memory. The larger your index, and the more fields accessed through the `FieldCache` API or used for sorting, the more costly reopening the reader is. As a result, frequently reopening your reader necessarily degrades other metrics like indexing and search throughput.

On the bright side, many applications only require high indexing throughput while creating the initial index or doing bulk updates. During this time, the index-to-search latency does not matter because no searching is happening. But then once the index is built and in-use in the application, the rate of document turnover is often low, while index-to-search latency becomes important.

As of 2.3, reopening a reader is somewhat more resource efficient: the new `IndexReader.reopen` method will only internally create new readers for those segments that are new. While this makes reopen faster, the creation of the `IndexSearcher` is still a very costly operation. Here are the steps to follow to reopen a reader:

1. Use the `reopen` method to get a new `IndexReader`. While not strictly necessary, it's best to do this after closing your writer, or at a time when you're certain the writer is idle (not performing segment merges, eg during optimize). Otherwise, opening a reader during this time will hold open segment files for segments that otherwise may have been deleted, thus consuming unexpectedly more disk space.
2. Create the `IndexSearcher` from the new reader.
3. If necessary, warm this searcher by running carefully chosen initial searches. These initial searches should exercise the slow one-time operations, such as sorting on a field for the first time, and loading any specific fields into the `FieldCache`. While this is happening, keep your old searcher alive to answer incoming searches.
4. Once the new searcher is ready, direct new searches to it, but follow-on searches (e.g., another page of results for a previously run search) back to the original searcher. This is to ensure the user does not see results suddenly shift while paging through search results. However, keeping readers open consumes more resources (file descriptors, RAM, disk space).
5. Once all search sessions have completed, or a preset timeout periods has passed, on the old searcher, close it.

When your have multiple threads doing searches, this re-opening sequence is tricky since you cannot close the old searcher until all threads are done. See 10.2.2 for a useful utility class to handle this tracking for you.

Measure the cost of re-opening a new searcher and use this to determine how frequently your application should do so. This will decide your index-to-search delay.

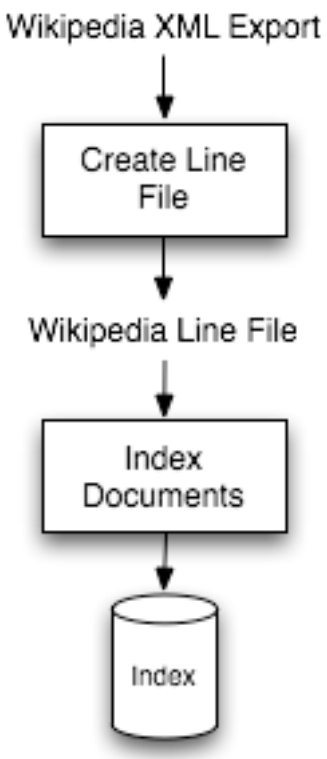


Figure 10.1 Steps to test indexing throughput on Wikipedia articles

10.1.2.2 INDEXING THROUGHPUT

Indexing throughput measures how many documents per second you are able to add to your index, which determines how much time it will take to build and update your index. In the benchmark framework there are several built-in document sources we could choose from, including the Reuter's corpus (`ReutersDocMaker`), Wikipedia articles (`EnwikiDocMaker`) and a simple document source that recursively finds all `*.txt` files under a directory (`DirDocMaker`). We'll use Wikipedia as the document source for all of our tests. This is a large and diverse collection so it makes for a good real-world test. For your own tests, create a document source implementing the `DocMaker` interface, and then use it for all of your testing.

To minimize the cost of document construction, let's first pre-process the Wikipedia XML content into a single large text file that contains one article per line. We will be following the steps shown in Figure 10.1. There is a built-in `WriteLineDoc` task for exactly this purpose, which works for simple `DocMakers` that produce only `title`, `date` and `body` fields (this includes both `EnwikiDocMaker` and `ReutersDocMaker`). Download the latest Wikipedia export from `http://wikipedia.org`, and decompress it `bunzip2 enwiki-latest-pages-articles.xml.bz2`. You can also download a snapshot of this file from here:

```
http://people.apache.org/~gsingers/wikipedia/enwiki-20070527-pages-articles.xml.bz2
```

Next, save the following algorithm to `createLineFile.alg`:

```
doc.maker=org.apache.lucene.benchmark.byTask.feeds.EnwikiDocMaker

docs.file=/lucene/enwiki-latest-pages-articles.xml
line.file.out=wikipedia.lines.txt

doc.maker.forever=false
{WriteLineDoc()}: *
```

This algorithm uses the built-in `EnwikiDocMaker`, which knows how to parse the XML format from Wikipedia, to produce one document at a time. Then, it runs the `WriteLineDoc` task over and over until there are no more documents, saving each document line by line to the file `wikipedia.lines.txt`.

Execute this by running `ant run-task -Dtask.alg=createLineFile.alg` in a shell. It will take some time to run. Sit back and enjoy the sound of hard drives seeking away doing all the hard work for you! That is, if you are not using a solid-state drive. It will print out how many documents have been processed as it's running, and at the end will produce a large file, `wikipedia.lines.txt`, with one document per line. Great! Wasn't that easy?

Now that we're done with the one-time setup, let's run a real test, using the efficient `LineDocMaker` as our document maker. For the tests below, it's best to store `wikipedia.lines.txt` on a separate drive from the `contrib/benchmark/work/index` directory where the index is created so the IO for reading the articles does not interfere with the IO for writing the index. Run this algorithm:

```
analyzer=org.apache.lucene.analysis.standard.StandardAnalyzer
doc.maker=org.apache.lucene.benchmark.byTask.feeds.LineDocMaker
directory=FSDirectory

doc.stored = true                                                    #A
doc.term.vectors = true
docs.file=/lucene/wikipedia.lines.txt

{ "Rounds"                                                           #B
  ResetSystemErase
  { "BuildIndex"
   -CreateIndex()
   { "AddDocs" AddDoc > : 200000                                     #C
   -CloseIndex()
  }
  NewRound
} : 3

RepSumByPrefRound BuildIndex                                         #D
```

#A Use stored fields and term vectors
#B Run test 3 times
#C Add first 200K docs
#D Report results

This algorithm builds an index with the first 200,000 Wikipedia articles, three times, using `StandardAnalyzer`. At the end it prints a one-line summary of each run. If you were building a real index for Wikipedia, you should use an analyzer base on the Wikipedia tokenizer under `contrib/wikipedia`. This tokenizer understands the custom elements of Wikipedia's document format such as `[[Category:…]]`. Since we are only measuring indexing throughput here, `StandardAnalyzer` is fine for our purposes. You should see something like this as your final report:

```
Operation   round   runCnt   recsPerRun        rec/s  elapsedSec    avgUsedMem    avgTotalMem
BuildIndex      0        1       200000        550.7      363.19    33,967,816     39,915,520
BuildIndex -  - 1 -  -   1 -  -  200000 -  -   557.3 -  - 358.85 -  24,595,904 -   41,435,136
BuildIndex      2        1       200000        558.4      358.17    23,531,816     41,435,136
```

Discarding the slowest and fastest run, our baseline indexing throughput is 557.3 documents/second. Not too shabby! As of Lucene 2.3, the "out of the box" default indexing throughput has improved substantially. Here are some specific things to try to further improve your application's indexing throughput:

- Use many threads. This could be the single biggest impact change you can make, especially if your computer's hardware has a lot of concurrency. See 10.2.1 for a drop-in threaded replacement for `IndexWriter`.
- Set `IndexWriter` to flush by memory usage and not document count. This is the default as of 2.3, but if your application still calls `setMaxBufferedDocs` then change it to `setRAMBufferSizeMB` instead. Test different RAM buffer sizes. Typically larger is better, to a point. Make sure you don't go so high such that the JVM is forced to GC too frequently, or the computer is forced to start swapping (see 10.3.3). Use the option `ram.flush.mb` in your algorithm to change the size of `IndexWriter`'s RAM buffer.
- Turn off compound file format (`IndexWriter.setUseCompoundFile(false)`). Creating a compound file takes some time during indexing. You'll also see a small performance gain during searching. But note that

this will require many more file descriptors to be opened by your readers (see 10.3.2), so you may have to decrease `mergeFactor` to avoid hitting file descriptor limits. Set `compound=false` in your algorithm to turn off compound file format.

- Use `autoCommit=false`. If it's not necessary for a reader to see the changes to the index as you go, using `autoCommit=false` may improve throughput especially if you use large stored fields and term vectors. Add the line `autocommit=false` to your algorithm.
- Re-use `Document` and `Field` instances. As of 2.3, a `Field` allows you to change its value. If your documents are highly regular (most are), then create a single `Document` instance and hold onto its `Field` instances. Change only the `Field` values, and then call `addDocument` with the same `Document` instance. The `LineDocMaker` source is already doing this, but you can turn it off by adding `docs.reuse.fields=false` to your algorithm.
- Make sure your own analyzers and filters are re-using a single `Token` instance by defining the `nextToken(Token)` API. For that `Token`, use the `termBuffer` API to get the internal `char[]` and change that array directly with your term text. As of 2.3, all of the core analyzers use these APIs.
- Test different values of `mergeFactor`. Higher values mean less merging cost while indexing, but slower searching since the index will have more segments. Beware: if you make this too high, and if compound file format is turned off, you can hit file descriptor limits on your OS (see 10.3.2). As of 2.3, segment merging is done in the background during indexing, so this is an automatic way to take advantage of concurrency. You may actually see faster performance with a low `mergeFactor`, especially if you optimize the index in the end. Test high and low values in your application and let the computer tell you which is best: you might be surprised!
- Use `optimize` sparingly. Use the new (as of 2.3) `optimize(maxNumSegments)` method instead. This method optimizes your index down to `maxNumSegments` (instead of always 1 segment) which can greatly reduce the cost of optimizing, while still making your searches quite a bit faster. Optimizing takes a very long time. If your searching performance is acceptable without optimizing then consider never optimizing.
- Index into separate indices, perhaps using different computers, and then merge them with `IndexWriter.addIndexesNoOptimize`. Do not use the older addIndexes methods as these make extra calls to `optimize`.
- Test the speed of creating the documents and just tokenizing them by using the `ReadTokens` task in your algorithm. This task steps through each field of the document, and tokenizes it using the specified analyzer. The document is not indexed. This is an excellent way to measure the document construction and tokenization cost, alone. Run this algorithm to tokenize this first 200K docs from Wikipedia using `StandardAnalyzer`:

```
analyzer=org.apache.lucene.analysis.standard.StandardAnalyzer

doc.maker=org.apache.lucene.benchmark.byTask.feeds.LineDocMaker

docs.file=wikipedia.lines.txt

{ "Rounds"
  ResetSystemErase
  { ReadTokens > : 200000
  NewRound
} : 3

RepSumByPrefRound ReadTokens
```

Which produces output like this:

```
Operation         round   runCnt   recsPerRun        rec/s  elapsedSec    avgUsedMem
avgTotalMem
ReadTokens_200000     0        1    161783312  1,239,927.9      130.48     2,774,040
2,924,544
ReadTokens_200000 -   1 -  -   1 -  161783312  1,259,857.2 -  - 128.41 -   2,774,112 -  -
2,924,544
ReadTokens_200000     2        1    161783312  1,253,862.0      129.03     2,774,184
2,924,544
```

Discarding the fastest and slowest runs, we see that simply retrieving and tokenizing the documents takes 129.03 seconds, which is ~27% of the total indexing time from our baseline. This number is actually very low, because we are using `LineDocMaker` as our document source. In a real application, creating, filtering, and tokenizing the document would be much more costly. Try it with your own `DocMaker`!

Let's combine the suggestions above. We'll index the same 200,000 documents from Wikipedia, but changing the settings to try to improve indexing throughput. Well turn off `autoCommit` and `compound`, increase `mergeFactor` to 30 and `ram.flush.mb` to 128, and, use 5 threads to do the indexing. Here is the alg:

```
analyzer=org.apache.lucene.analysis.standard.StandardAnalyzer
doc.maker=org.apache.lucene.benchmark.byTask.feeds.LineDocMaker
directory=FSDirectory

docs.file=/lucene/wikipedia.lines.txt

doc.stored = true
doc.term.vector = true
ram.flush.mb = 128
compound = false
autocommit = false
merge.factor = 30

doc.add.log.step=1000

{ "Rounds"
  ResetSystemErase
  { "BuildIndex"
    -CreateIndex
    [ { "AddDocs" AddDoc > : 40000] : 5         #1
    -CloseIndex
  }
  NewRound
} : 3

RepSumByPrefRound BuildIndex
```

#1 Use 5 threads in parallel, each doing 40K documents

This will produce output like this:

```
Operation   round   runCnt   recsPerRun        rec/s  elapsedSec    avgUsedMem    avgTotalMem
BuildIndex      0        1       200000        879.5      227.40   166,013,008    205,398,016
BuildIndex -  - 1 -  -   1 -  -  200000 -  -   899.7 -  - 222.29 - 167,390,016 -  255,639,552
BuildIndex      2        1       200000        916.8      218.15   174,252,944    276,684,800
```

Wow, the performance is even better: 899.7 documents per second! Of course in your testing you should test each of these changes above, one at a time, and keep only those that help.

Well there you have it! As we've seen, Lucene's "out of the box" indexing throughput is excellent. But with some simple tuning ideas, you can make it even better. Let's switch gears now and talk about search performance.

10.1.2.3 SEARCH LATENCY AND THROUGHPUT

Search latency and throughput are two sides of one coin: changes that improve search latency will also improve your search throughput, on the same hardware. This is assuming you are running enough threads to take full advantage of the computer's concurrency (which you should!).

The best way to measure your search latency and throughput is with a standalone load testing tool, such as `Grinder`. These tools do a great job simulating multiple users and reporting latency and throughput results. They also test your application end-to-end, which is what a real user experiences when using your web site.

Try to use real searches from real users when running search performance tests. If possible, cull search logs to get all searches, and run them in the same order that they came from the search logs. Use multiple threads to simulate multiple users, and verify you are fully utilizing the computer's concurrency. Include follow-on searches, like clicking through pages, in the test. The more "real world" your test is, the more accurate your test results are. For example, if you create your own small set of hand-crafted searches for testing, and run these over and over, you can easily see unexpectedly excellent performance because the OS has loaded all required bytes from disk into its IO cache. To fix this, you may be tempted to flush the IO cache before each test, which is possible. But then you're going too far in the other direction, by penalizing your results too heavily, since in your real application the IO cache would help your performance.

Here are some steps to improve search performance:

- Use enough threads to fully utilize your computer's concurrency. See 10.2.2 for details.
- Use a read-only IndexReader, by calling `IndexReader.open(dir, true)`. Read-only `IndexReaders` have better concurrency because they can avoid synchronizing on certain internal data structures.
- If you are not on Windows, use `NIOFSDirectory`, which has better concurrency, instead of FSDirectory
- Make sure each step between the user and Lucene is not adding unnecessary latency. For example, make sure your request queue is first-in-first-out, and all threads pull from this queue, so searches are answered in the order they came. Verify that rendering the results returned by Lucene is fast.
- Be sure you are using enough threads to fully utilize the computer's hardware. Increase the thread count until throughput no longer improves before search latency starts getting worse.
- If you have enough RAM and enough file descriptors, consider using more than one instance of `IndexSearcher`. There are some places where `IndexReader` is not fully concurrent, and many threads sharing a single instance could bottleneck.
- Warm up your searchers before using them on real searches. The first time a certain sort is used, it must populate the `FieldCache`. Pre-warm the searching by issuing one search for each of the sort fields that may be used (see 10.1.2).
- Use `FieldCache` instead of stored fields, if you can afford the RAM. `FieldCache` pulls all stored fields into RAM, whereas stored fields must go back to disk for every document. Populating a `FieldCache` is resource consuming (CPU and IO), but done only once per field the first time it's accessed. Once it is populated, accessing it is very fast.
- Decrease `mergeFactor` so there are fewer segments in the index.
- Turn off compound file format.
- Limit your use of term vectors: retrieving them is quite slow. If you must, then do so only for those hits that require it. Use `TermVectorMapper` to carefully select only the parts of the term vectors that you actually need.
- If you must load stored fields, use `FieldSelector` to restrict to exactly those fields that you need. Use lazy field loading for large fields so that the contents of the field are only loaded when actually requested.
- Run `optimize` or `optimize(maxNumSegments)` periodically on your index.
- Don't iterate over more hits than needed.
- Only re-open the `IndexReader` when it's really necessary.
- Call `query.rewrite().toString()` and print the result. This is the actual query Lucene runs. You might be surprised to see how queries like `FuzzyQuery` and `RangeQuery` rewrite themselves!
- If you are using `FuzzyQuery`, set the minimum prefix length to a value greater than zero (e.g., 3).

Note that quite a few of these options are in fact detrimental to indexing throughput: they are automatically at odds with one another. You have to find the right balance for your application.

10.2 Threads & concurrency

Modern computers have highly concurrent hardware. Moore's law lives on, but instead of giving us faster clock speeds, we get more CPU cores. It's not just the CPU. Hard drives accept many IO requests at once and re-order them to make more efficient use of the disk heads. Even solid state disks do the same, and then go further by using multiple channels to concurrently access the raw storage. The interface to system RAM uses multiple channels. Then, there is concurrency across these resources: when one thread is stuck waiting for an IO request to complete, another thread can use the CPU, and you will gain concurrency.

Therefore, it's critical to use threads for indexing and searching. Otherwise, you are simply not fully utilizing the computer. It's like buying a Corvette and driving it no faster than 20 mph! Likely, switching to using threads is the single change you can make that will increase performance the most. You'll have to empirically test, to find the right number of threads for your application and tradeoff search or indexing latency and throughput. Generally, at first, as you add more threads, you'll see latency stay about the same, but throughput will improve.

Then when you hit the right number of threads, adding more will not improve throughput, and may hurt it somewhat due to context switching costs, but will increase the latency.

Unfortunately, there is the dark side to threads, which if you've explored them in the past you've discovered: they add substantial complexity to your application. Suddenly you must take care to make the right methods synchronized (but not too many!), change your performance testing to use threads, manage thread pools, spawn and join threads at the right times, etc. Entirely new kinds of intermittent bugs become possible, such as deadlock if locks are not acquired in the same order by different threads or `ConcurrentModificationException` and other problems if you are missing synchronization. Testing is difficult because the threads are scheduled at different times by the JVM every time you run a test. Are they really worth all this hassle?

Yes, they are! Lucene has been carefully designed to work very well with many threads. Lucene is thread safe: sharing `IndexSearcher`, `IndexReader`, `IndexWriter`, etc, across multiple threads is perfectly fine. Lucene is also thread friendly: synchronized code is minimized such that multiple threads can make full use of the hardware's concurrency. In fact, as of 2.3, Lucene already makes use of concurrency right out of the box: `ConcurrentMergeScheduler` merges segments using background threads. You can choose a merge scheduler in your algorithm by setting the `merge.scheduler` property. For example, to test indexing with the `SerialMergeScheduler`, which matches how segment merges were done before 2.3, add `merge.scheduler = org.apache.lucene.index.SerialMergeScheduler` to your algorithm. In this section we show you how to leverage threads during indexing and searching, and provide a couple of drop-in classes to make it simple to gain concurrency.

10.2.1 Using threads for indexing

Listing 10.1 shows a simple utility class that extends `IndexWriter` and uses `java.util.concurrent` to manage multiple threads adding and updated documents. The class simplifies multithreaded indexing because all details of these threads are hidden from you. It's also a drop-in for anywhere you are currently using the `IndexWriter` class, though you may need to modify it if you need to use one of `IndexWriter`'s expert constructors. Just specify how many threads to use, and the size of the queue, when you instantiate the class. Test different values to find the sweet spot for your application, but a good rule of thumb for `numThreads` is one plus the number of CPU cores in your computer that you'd like to consume on indexing, and then `4*numThreads` for `maxQueueSize`. As you use more threads for indexing, you'll find that a larger RAM buffer size should help even more, so be sure to test different combinations of number of threads and RAM buffer size to reach your best performance. Check process monitor tools, like top or `ps` on `unix`, or Task Manager on Windows, to verify that CPU utilization is near 100%.

Listing 10.1 Utility class to use multiple threads to index documents. This is a drop-in replacement wherever you use IndexWriter.

```
public class ThreadedIndexWriter extends IndexWriter {

  private ExecutorService threadPool;

  private class Job implements Runnable {                      //A
    Document doc;
    Analyzer analyzer;
    Term delTerm;
    public Job(Document doc, Term delTerm, Analyzer analyzer) {
      this.doc = doc;
      this.analyzer = analyzer;
      this.delTerm = delTerm;
    }
    public void run() {                                        //B
      try {
        if (analyzer != null) {
          if (delTerm != null) {
            ThreadedIndexWriter.super.updateDocument(delTerm, doc, analyzer);
          } else
            ThreadedIndexWriter.super.addDocument(doc, analyzer);
        } else {
          if (delTerm != null) {
            ThreadedIndexWriter.super.updateDocument(delTerm, doc);
          } else
```

```
            ThreadedIndexWriter.super.addDocument(doc);
          }
        } catch (IOException ioe) {
          // TODO: you must log & escalate this error
          ioe.printStackTrace(System.err);
        }
      }
    }

    public ThreadedIndexWriter(Directory dir, Analyzer a, boolean create, int numThreads, int
maxQueueSize, IndexWriter.MaxFieldLength mfl)
        throws CorruptIndexException, IOException {
      super(dir, a, create, mfl);
      threadPool = new ThreadPoolExecutor(                        //C
            numThreads, numThreads,
            Long.MAX_VALUE, TimeUnit.NANOSECONDS,
            new ArrayBlockingQueue<Runnable>(maxQueueSize, false),
            new ThreadPoolExecutor.CallerRunsPolicy());
    }

    public void addDocument(Document doc) {                       //D
      threadPool.execute(new Job(doc, null, null));               //D
    }                                                             //D
                                                                  //D
    public void addDocument(Document doc, Analyzer a) {           //D
      threadPool.execute(new Job(doc, null, a));                  //D
    }                                                             //D
                                                                  //D
    public void updateDocument(Term term, Document doc) {         //D
      threadPool.execute(new Job(doc, term, null));               //D
    }                                                             //D
                                                                  //D
    public void updateDocument(Term term, Document doc, Analyzer a) { //D
      threadPool.execute(new Job(doc, term, a));                  //D
    }                                                             //D

    public void close() throws CorruptIndexException, IOException {
      finish();
      super.close();
    }

    public void close(boolean doWait) throws CorruptIndexException, IOException {
      finish();
      super.close(doWait);
    }

    public void rollback() throws CorruptIndexException, IOException {
      finish();
      super.rollback();
    }

    private void finish() {                                       //E
      threadPool.shutdown();
      while(true) {
        try {
          if (threadPool.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS))
            break;
        } catch (InterruptedException ie) {
          Thread.currentThread().interrupt();
        }
      }
    }
  }
```

#A Holds one document to be added
#B Does real work to add or update document
#C Create thread pool
#D Have thread pool execute job
#E Shuts down thread pool

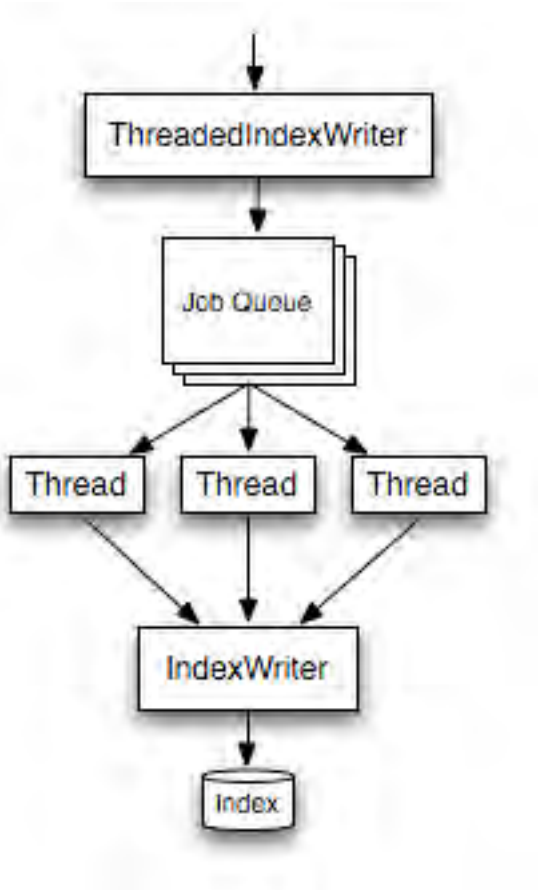


Figure 10.2 ThreadedIndexWriter manages multiple threads for you.

Figure 10.2 shows how the class works. The class overrides the `addDocument` and `updateDocument` methods: when one of these is called, a `Job` is created and added to the work queue in the thread pool. If the queue in the thread pool is not full, then control immediately returns back to the caller. Otherwise, the caller's thread is used to immediately execute the `Job`. In the background, a worker thread wakes up, takes jobs from the front of the work queue, and does the real work. When you use this class, you cannot re-use `Document` or `Field` instances, because you can't control precisely when a `Document` is done being indexed. The class overrides `close` and `rollback` methods, to first shutdown the thread pool to ensure all adds and updates in the queue have completed.

Let's test `ThreadedIndexWriter` by using it in the benchmark framework, which makes it wonderfully trivial to extend with a new task. Make a `CreateThreadedIndexTask.java`, like this:

```
public class CreateThreadedIndexTask extends CreateIndexTask {

  public CreateThreadedIndexTask(PerfRunData runData) {
    super(runData);
  }

  public int doLogic() throws IOException {
    PerfRunData runData = getRunData();
    Config config = runData.getConfig();
    IndexWriter writer = new ThreadedIndexWriter(runData.getDirectory(),
                                                 runData.getAnalyzer(),
                                                 true,
                                                 config.get("writer.num.threads", 4),
                                                 config.get("writer.max.thread.queue.size",
20),
                                                 IndexWriter.MaxFieldLength.LIMITED);
    CreateIndexTask.setIndexWriterConfig(writer, config);
```

```
      runData.setIndexWriter(writer);
      return 1;
    }
  }
```

Create a new algorithm, derived from the baseline algorithm from 10.2.3, with only these changes:

- Replace `CreateIndex` with `CreateThreadedIndex`
- Add `docs.reuse.fields=false`, which tells `LineDocMaker` to not reuse fields
- Optionally set `writer.num.threads` and `writer.max.thread.queue.size` to test different values

Compile your `CreateThreadedIndexTask.java`, and then run your algorithm like this so it knows where to find your new task:

```
ant run-task -Dtask.alg=indexWikiLine.alg -Dbenchmark.ext.classpath=/path/to/my/classes
```

You should see it finish quite a bit faster than the original baseline! Now you can just drop this class in wherever you now use `IndexWriter` and take advantage of concurrency. Let's look next at using threads during searching.

10.2.2 Using threads for searching

Fortunately, a modern web or application server handles most of the threading issues for you: it maintains a first-in-first-out request queue, and a thread pool to service requests from the queue. This means much of the hard work is already done! All you have to do is create a `Query` based on the details in the user's request, invoke your `IndexSearcher`, and render the results. It's so easy! If you are not running Lucene in a Web application, then the thread pool support in `java.util.concurrent` should help you.

Be sure you tune the size of the thread pool to make full use of the computer's concurrency. Also, tune the maximum allowed size of the request queue for searching: when your web site is suddenly popular and far too many searches per second are arriving, you want new requests to quickly receive an `HTTP 500 Server Too Busy` error, instead of waiting in the request queue forever. This also ensures that your application gracefully recovers once the traffic settles down again. Run a redline stress test to verify this!

There is one tricky aspect that the application server will not handle for you: reopening your searcher when your index has changed. Since an `IndexReader` only sees the index as of the point in time when it was opened, once there are changes to the index you must reopen your `IndexReader` to search them. Unfortunately, this is frequently a costly operation, consuming CPU and IO resources. Yet, for some applications, minimizing index-to-search delay is worth that cost, which means you'll have to reopen your searcher frequently.

Threads make reopening your searcher challenging, because you cannot close the old searcher until all searches are done with it, including iterating through the hits after `IndexSearcher.search` has returned. Beyond that, you may want to keep the old searcher around for long enough for all search sessions (original search plus all follow on actions like clicking through pages) to finish or expire. For example, consider a user who is stepping through page after page of results, where each page is a new search on your server. If you suddenly swap in a new searcher in between pages then the documents assigned to each page could shift, causing the user to see duplicate results across pages, or, to miss some results. This erodes your user's trust which is pretty much the kiss of death for your application. Prevent this by sending new pages for a previous search back to the original searcher, when possible.

Listing 10.2 shows a useful utility class, `SearcherManager`, that hides the tricky details of re-opening your searcher in the presence of multiple threads.

Listing 10.2 Utility class to manage reopening IndexSearcher in a multithreaded world.

```
public class SearcherManager {

  private IndexSearcher currentSearcher;                          //A
  private Directory dir;

  public SearcherManager(Directory dir) throws IOException {
    this.dir = dir;
    currentSearcher = new IndexSearcher(IndexReader.open(dir));  //B
  }
```

```
  public void warm(IndexSearcher searcher) {}                    //C

  private boolean reopening;

  private synchronized void startReopen()                        //D
    throws InterruptedException {
    while (reopening) {
      wait();
    }
    reopening = true;
  }

  private synchronized void doneReopen() {                       //E
    reopening = false;
    notifyAll();
  }

  public void maybeReopen() throws InterruptedException, IOException { //F

    startReopen();

    try {
      final IndexSearcher searcher = get();
      try {
        long currentVersion = currentSearcher.getIndexReader().getVersion();  //G
        if (IndexReader.getCurrentVersion(dir) != currentVersion) {           //G
          IndexReader newReader = currentSearcher.getIndexReader().reopen();  //G
          assert newReader != currentSearcher.getIndexReader();               //G
          IndexSearcher newSearcher = new IndexSearcher(newReader);           //G
          warm(newSearcher);                                                  //G
          swapSearcher(newSearcher);                                          //G
        }
      } finally {
        release(searcher);
      }
    } finally {
      doneReopen();
    }
  }

  public synchronized IndexSearcher get() {                      //H
    currentSearcher.getIndexReader().incRef();
    return currentSearcher;
  }

  public synchronized void release(IndexSearcher searcher)       //I
    throws IOException {
    searcher.getIndexReader().decRef();
  }

  private synchronized void swapSearcher(IndexSearcher newSearcher) //J
      throws IOException {
    release(currentSearcher);
    currentSearcher = newSearcher;
  }
}
```

#A Current IndexSearcher
#B Create initial searcher
#C Implement in subclass to warm new searcher
#D Pauses until no other thread is reopening
#E Finish reopen and notify other threads
#F Reopen searcher if there are changes
#G Check index version and reopen, warm, swap if needed
#H Returns current searcher; must be matched with a call to release
#I Release searcher returned from get()
#J Swaps currentSearcher to new searcher

This class uses the `IndexReader.reopen` API to efficiently open a new `IndexReader` that may share some `SegmentReaders` internally with the previous one. Instantiate this class once in your application, naming it `searcherManager`. Then, whenever you need a searcher, do this:

```
IndexSearcher searcher = searcherManager.get()
Try {
    <do searching & rendering here…>
} finally {
    searcherManager.release(searcher);
}
```

Every call to `get` must be matched with a corresponding call to `release`, ideally using a `try/finally` clause.

Note that this class does not actually do any re-opening on its own. Instead, you must call `maybeReopen` every so often according to your application's needs. For example, a good time to call this is after you've closed the `IndexWriter`. Reopening your reader while an `IndexWriter` is still open could tie up extra disk space because segment merges might be in progress. Once optimize is done, you should definitely reopen to free up the disk space and get faster searches. If your index is not too large, it's fine to call `maybeReopen` during a search request. The call will reopen the searcher, if necessary, using the foreground thread, thus adding latency to the unlucky search that hits it. If your index is large, then it's better to call `maybeReopen` from a dedicated background thread. You should also create a subclass that implements the `warm` method to run the targetted initial searches against the new searcher before it's made available for general searching (see 10.1.2.3).

10.3 Managing resource consumption

Like all software, Lucene requires certain precious resources to get its job done. A computer has a limited supply of things like disk storage, file descriptors and memory. Often Lucene must share these resources with other applications. Understanding how Lucene uses resources and what you can do to control this lets you keep your search application healthy. You might assume Lucene's disk usage is simply proportional to the total size of all documents you've added, but you'll be surprised to see that often, this is far from the truth. Similarly, Lucene's usage of simultaneous open file descriptors is unexpected: changes to a few Lucene configuration options can drastically change the number of open files. Finally, to manage Lucene's memory consumption, you'll see why it's not always best to give Lucene access to all memory on the computer.

We start with everyone's favorite: how much disk space does Lucene require? Next we describe Lucene's open file descriptor usage, and finally, memory usage.

10.3.1 Disk space

Lucene's disk usage depends on many factors. An index with only a single pure indexed, typical text field will be about 1/3rd of the total size of the original text. But at the other extreme, an index that has stored fields and term vectors with offsets and positions, with numerous deleted documents plus an open reader on the index, with an optimize running, can easily consume 10X the total size of the original text! This wide range and seeming unpredictability makes it exciting to manage disk usage for a Lucene index.

Figure 10.3 shows the disk usage over time while indexing all documents from Wikipedia, finishing with an `optimize` call. The final disk usage was 14.2 GB, but the peak disk usage was 32.4 GB, which was reached while several large concurrent merges were running. You can immediately see how erratic it is. Rather than increasing gradually with time, as you add documents to the index, disk usage will suddenly ramp up during a merge and then quickly fall back again once the merge has finished, creating a saw tooth pattern. The size of this jump corresponds to how large the merge was (the net size of all segments being merged). Furthermore, with `ConcurrentMergeScheduler`, several large merges can be running at once and this will cause an even larger increase of temporary disk usage.

How can you manage disk space when it has such wild swings? Fortunately, there is a method to this madness. Once you understand what's happening under the hood, you can predict and understand Lucene's disk usage. It's important to differentiate transient disk usage, while the index is being built (Figure 10.3), versus final disk usage, when the index is completely built and optimized to one. It's easiest to start with the final size. Here is a coarse formula to estimate the final size based on the size of all text from the documents:

1/3 x indexed fields +

1 x stored fields +
1 x term vectors fields (2 x if offsets & positions are stored)

For example if your documents have a single field that is indexed, with stored and has term vectors turned on, you should expect the index size to be around 2 1/3 x the total size of all text across all docs. Note that this formula is very approximate. For example, documents with unusually diverse or unique terms, like a large spreadsheet that contains many unique product SKUs, will use more disk space.

You can reduce disk usage somewhat by turning off norms (section 2.XXX), turning off term frequency information for fields that don't need it (section 2.XXX) and indexing and storing fewer fields per document

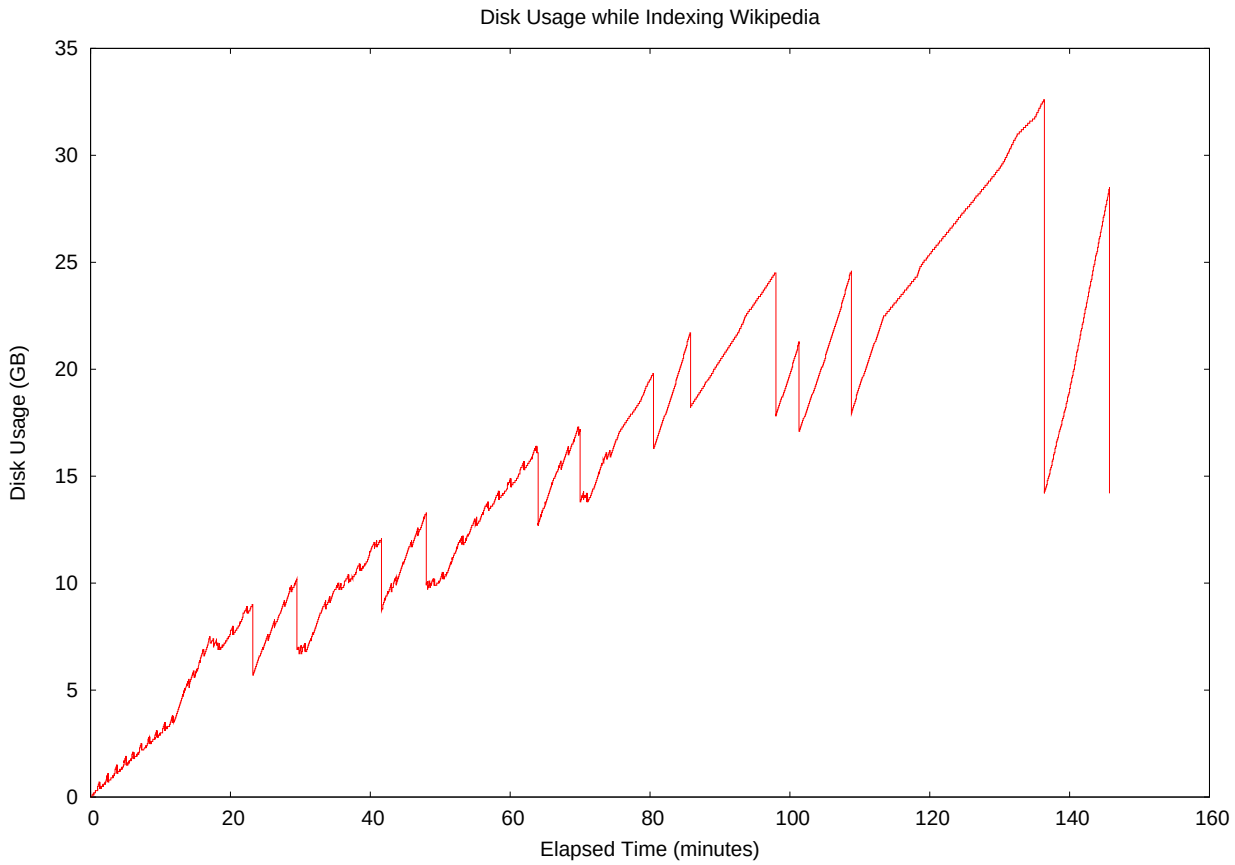


Figure 10.3 Disk usage while building an index of all Wikipedia documents, with optimize called in the end

The transient disk usage depends on many factors. As the index gets larger, the size of each saw tooth will get larger as bigger merges are being done. Large merges also take longer to complete and will therefore tie up disk space for more time. When you optimize the index, down to one segment, the final merge is the largest merge possible and will require 1X the final size of your index in temporary disk space. Here are other things that will affect transient disk usage:

- Open readers prevent deletion of those files they are using. You should have only one open reader at a time, except when you are re-opening it. Be sure to close the old reader!
- With `autoCommit=false`, disk usage will be higher because segments referenced by the starting commit point (when the writer was opened), as well as those referenced by the current (in memory) commit point are not deleted.
- If you frequently replace documents but do not run optimize then the space used by old copies of the deleted documents won't be reclaimed until those segments are merged.

- The more segments in your index, the more disk space will be used than if those segments were merged. This means a high `mergeFactor` will result in more disk space being used.
- Given the same net amount of text, many small documents will result in a larger index than fewer large documents.
- Do not open a new reader while `optimize`, or any other merges, are running as this will hold references to segments that would otherwise be deleted. Instead, open after you have closed your `IndexWriter`.
- Do open a new reader after making changes with `IndexWriter`. If you don't, then the reader is holding references to files that `IndexWriter` may have deleted, due to merging.
- If you are running a backup (see 10.4) then the files in the snapshot being copied will also consume disk space, until you release the snapshot.

Note that on UNIX you may think disk space has been freed because the writer has deleted the old segments files, but in fact the files still consume disk space as long as those files are held open by an `IndexReader`. Windows does not allow deletion of open files so you'll still see the files when you look at the directory. Don't be fooled!

So how can you make sense of all of this? A good rule of thumb is to measure the total size of your index, called that X. Then, make sure at all times you have 2X free disk space, or 3X if you have a reader open, on the file system where the index is stored, at all times. Let's consider file descriptor usage, next.

10.3.2 File Descriptors

Suppose you're happily tuning your application to maximize indexing throughput. You turned off compound file format. You cranked up `mergeFactor` and got awesome speedups, so you want to push it even higher. Unfortunately, there is a secret cost to these changes: you are drastically increasing how many files Lucene must hold open at once. At first you're ecstatic about your changes, since everything seems fine. Then, as you add more documents index grows, Lucene will need more and more open files when one day -- BOOM! -- you hit the dreaded "Too many open files" `IOException`, and the OS stops you dead in your tracks. Faced with this silent and sudden risk, how can you possibly tune for the best indexing performance, while staying under this limit?

Fortunately, there is hope! With a few simple steps you can take control of the situation. Start by running the test below:

```
public class TestOpenFileLimit extends LuceneTestCase {
  public void testLimit() throws IOException {
    int count = 0;
    List files = new ArrayList();
    try {
      while(true) {
        files.add(new RandomAccessFile("tmp" + count, "rw"));
        count++;
      }
    } catch (IOException ioe) {
      System.out.println("IOException after  " + count + " open files:");
      ioe.printStackTrace(System.out);
      for(int i=0;i<count;i++)
        new File("tmp" + i).delete();
    }
  }
}
```

When you run the test, it will always fail and then tell you how many files it was able to open before the OS cut it off. There is tremendous variation across operating systems and JVMs. Running this under OS X 10.4 and Java 1.5 shows that the limit is 10,233. Java 1.6 on Windows Server 2003 shows a limit of 9,994 open files. Java 1.5 on Debian Linux with Kernel 2.6.22 shows a limit of 1,018 open files.

Next, increase the limit to the maximum allowed by the OS. The exact command for doing so varies according to OS and shell (hello Google my old friend). Run the test again to make sure you've actually increased the limit.

Finally, monitor how many open files your JVM is actually using. There are OS level tools to do this. On UNIX, use the `lsof`. On Windows, use `Task Manager`. You'll have to add `File Handles` as a column, using the `View`

-> Select Columns… menu. The sysinternals tools from Microsoft also include useful utilities like Process Monitor to see which specific files are held open by which processes.

Listing 10.3: Drop-in replacement for FSDirectory to track open files

```
public class TrackingFSDirectory extends FSDirectory {

  private HashSet openOutputs = new HashSet();              #A
  private HashSet openInputs = new HashSet();

  synchronized public int getFileDescriptorCount() {        #B
    return openOutputs.size() + openInputs.size();
  }

  synchronized private void report(String message) {
    System.out.println(System.currentTimeMillis() + ": " +
                       message + "; total " + getFileDescriptorCount());
  }

  synchronized public IndexInput openInput(String name)     #C
    throws IOException {
    return openInput(name, BufferedIndexInput.BUFFER_SIZE);
  }

  synchronized public IndexInput openInput(String name, int bufferSize)
    throws IOException {                                    #C
    openInputs.add(name);
    report("Open Input: " + name);
    return new TrackingFSIndexInput(name, bufferSize);
  }

  synchronized public IndexOutput createOutput(String name)
    throws IOException {                                    #D
    openOutputs.add(name);
    report("Open Output: " + name);
    File file = new File(getFile(), name);
    if (file.exists() && !file.delete())          // delete existing, if any
      throw new IOException("Cannot overwrite: " + file);
    return new TrackingFSIndexOutput(name);
  }

  protected class TrackingFSIndexInput extends FSIndexInput { #E
    String name;
    public TrackingFSIndexInput(String name, int bufferSize) throws IOException {
      super(new File(getFile(), name), bufferSize);
      this.name = name;
    }

    boolean cloned;

    public Object clone() {
      TrackingFSIndexInput clone = (TrackingFSIndexInput)super.clone();
      clone.cloned = true;
      return clone;
    }

    public void close() throws IOException {
      super.close();
      if (!cloned) {
        synchronized(TrackingFSDirectory.this) {
          openInputs.remove(name);
        }
      }
      report("Close Input: " + name);
    }
  }

  protected class TrackingFSIndexOutput extends FSIndexOutput { #F
    String name;
    public TrackingFSIndexOutput(String name) throws IOException {
      super(new File(getFile(), name));
      this.name = name;
```

```
      }
      public void close() throws IOException {
        super.close();
        synchronized(TrackingFSDirectory.this) {
          openOutputs.remove(name);
        }
        report("Close Output: " + name);
      }
    }
  }
  #A Holds all open file names
  #B Returns total open file count
  #C Opens tracking input
  #D Opens tracking output
  #E Tracks eventual close
```

To get more specifics about which files Lucene is opening, and when, use the class in Listing 10.3. This class is a drop-in replacement for `FSDirectory` that adds tracking of open files. It reports whenever a file is opened or closed, for reading or writing, and lets you retrieve the current total count of open files. Compile the code and ensure the class file is on your CLASSPATH. Then, set the Java system property `org.apache.lucene.FSDirectory.class` to the name of your class, and run your application. To use this class in your benchmark algorithm, add the following line under the `run-task` ant task in the file `contrib/benchmark/build.xml`:

```
          <sysproperty key="org.apache.lucene.FSDirectory.class"
  value="org.apache.lucene.store.TrackingFSDirectory"/>
```

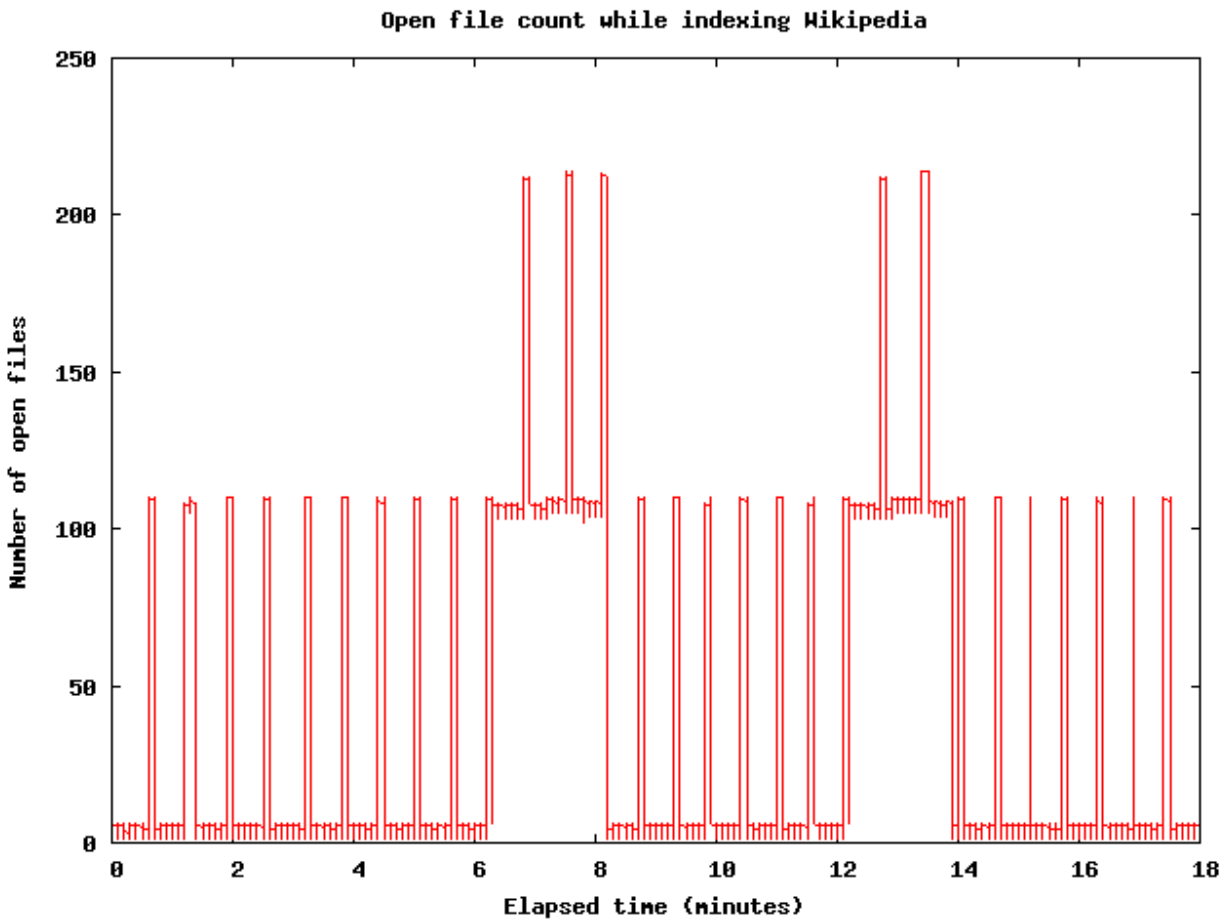


Figure 10.4 File descriptor consumption while building an index of Wikipedia articles

Figure 10.4 shows the open file count while building a Wikipedia index, with compound file format turned off and `mergeFactor` left at its default (10). You can see that it follows a peaky pattern, with very low usage when flushing segments and rather high usage while merges are running since the writer holds open files for all segments being merged plus the new segment being created. This means `mergeFactor`, which sets the number of segments to merge at a time, directly controls the open file count during idnexing. When two merges are running at once, which happens for 3 small merges starting around 7 minutes and then again for 2 small merges starting around 13 minutes, you'll see twice the file descriptor consumption.

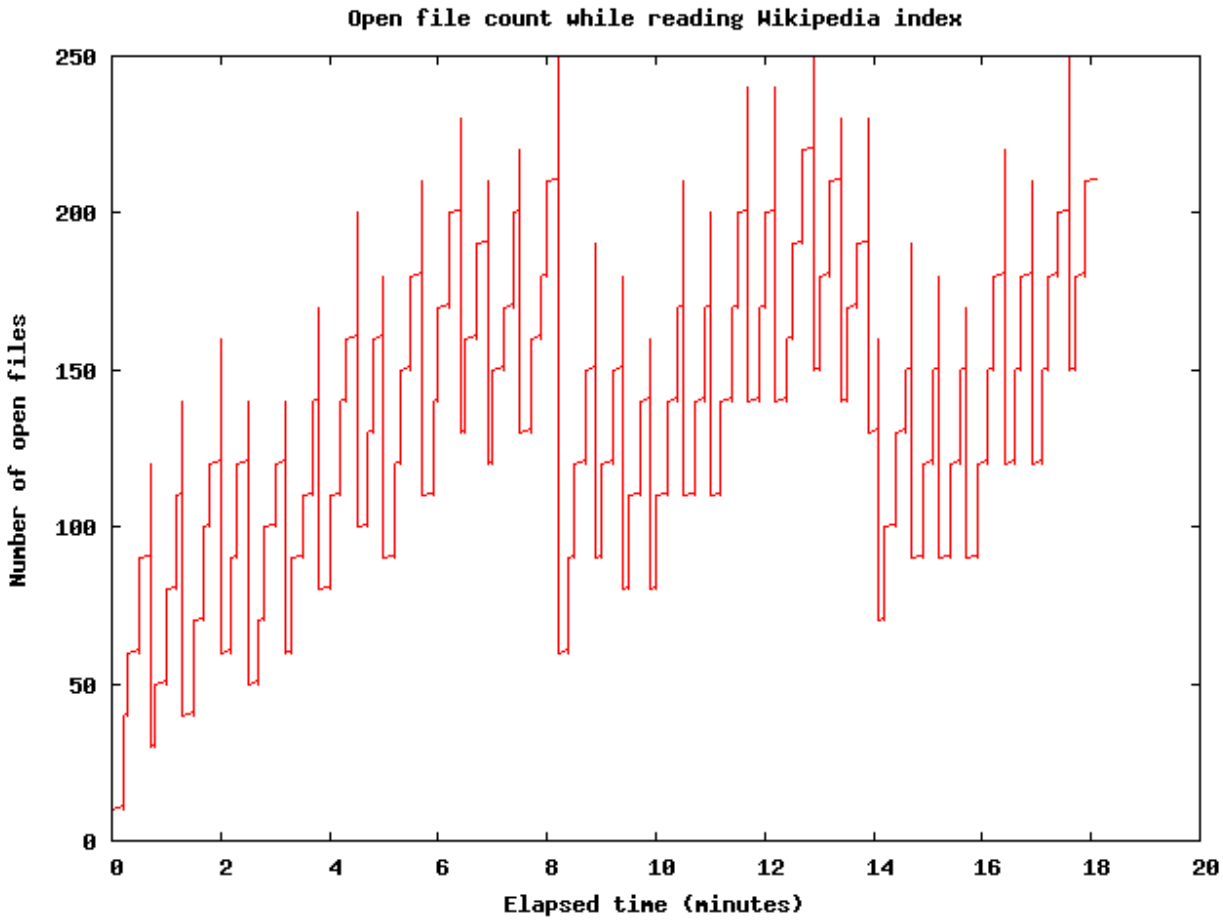


Figure 10.5 File descriptor usage by an IndexReader reopening every 30 seconds while Wikipedia articles are indexed.

Unlike indexing, where peak open file count is a simple multiple of `mergeFactor`, searching can require many more open files. For each segment in the index, the reader must hold open all files for that segment. If you're not using compound file format, that's 7 files if there are no term vectors indexed, or 10 files if there are. For a quickly changging and growing index, this count can really add up! Figure 10.5 shows open file count for an `IndexReader` reading the same index from Figure 10.5, while it is being built, reopening the reader every 10 seconds. During reopen, if the index has changed substantially because a merge has completed, the open file count will at first peak very high, because during this time both the old and new reader are in fact open. Once the old reader is closed, the usage drops down, in proportion to how many segments are in the index. When you use the new `IndexReader.reopen` method in 2.3, this spike is quite a bit smaller than if you open a new reader, because the file descriptors for segments that have not changed are shared. As the index gets larger, the usage increases, though it is not a straight line because sometimes the reader catches the index soon after a large merge has finished. Armed with your new knowledge about open file consumption, here are some simple tips to keep them under control while still enjoying your indexing performance gains:

- Increase the `IndexWriter` buffer (`setRAMBufferSizeMB`). The less often the writer flushes a segment, the fewer segments there will be in the index.
- Use `IndexReader.reopen` instead of opening a whole new reader. This is a very big reduction on peak open file count.
- Reduce `mergeFactor`, but not so much that it substantially hurts indexing throughput.
- Consider reducing the maximum number of simultaneous merge threads by calling `ConcurrentMergeScheduler.setMaxThreadCount`.
- Optimize the index. A partial optimize, using the new `IndexWriter.optimize(int maxNumSegments)` method, is a good compromise for minimizing the time it takes to optimize while still substantially reducing the number of segments in the index.
- Always measure your peak usage. This is often when you are opening and warming a new reader, before you've closed the old one.
- If you run indexing and searching from a single JVM, you must add up the peak open file count for both. The peak often occurs when several concurrent merges are running and you are reopening your reader. If possible, close your writer before reopening your reader to prevent this "perfect storm".
- Double check that all other code also running in the same JVM is not using too many open files. If it is, consider running a separate JVM for it.
- If you find you are still running out of file descriptors far earlier than you'd expect, make sure you are really closing your old `IndexReader` instances.

Striking the right balance between performance, and the dreaded open file limit, feels like quite an art. But now that you understand how Lucene uses open files, how to test and increase the limit on your OS, and how to measure exactly which files are held open by Lucene, you have all the tools you need to strike that perfect balance. It's now more science than art! We'll move next to another challenging resource: memory usage.

10.3.3 Memory

You've surely hit `OutOfMemoryError` in your Lucene application in the past? If you haven't, then you will, especially when many of the ways to tune Lucene for performance also increase its memory usage. So you thought: no problem, just increase the JVMs' heap size, and move on. Nothing to see here. You do that, and things seem fine, but little do you know you actually really hurt the performance of your application because the computer has started swapping memory to disk. And perhaps a few weeks later you hit the same error again. What's going on? How can you control this devious error and still keep your performance gains?

Managing memory usage is especially exciting, because there are two very different levels of memory. First, you must control how the JVM uses memory from the OS. Second, you must control how Lucene uses memory from the JVM. And the two must be properly tuned together. Once you understand these levels you'll have no trouble preventing memory errors and maximizing your performance, at the same time.

You manage the JVM by telling it how large its heap should be. The option `-Xms size` sets the starting size of the heap and the option `-Xmx size` sets the maximum allowed size of the heap. In a production server environment, you should set both of these sizes to the same value, so the JVM does not spend time growing and shrinking the heap. Also, if there will be problems reaching the max (e.g. the computer must swap excessively) you can see these problems quickly on starting the JVM instead of hours later (at 2 AM) when your application finally needs to use all the memory. The heap size should be large enough to give Lucene the RAM that it needs, but not so large that you force the computer to swap excessively. Generally you shouldn't just give all RAM to the JVM: it's beneficial to leave excess RAM free to allow the OS to use as its IO cache.

How can you tell if the computer is excessively swapping? Here are some clues:

- Listen to your hard drives, if your computer is nearby: they will be noticeably grinding away.
- On UNIX, run `vmstat 1` to print virtual memory statistics, once per second. Then look for the columns for pages swapped in (typically `si`) and pages swapped out (typically `so`). On Windows, use `Task Manager`, and add the column `Page Faults Delta`, using the `View -> Select Columns...` menu. Check for high numbers in these columns (say greater than 50).
- Ordinary interactive processes, like a shell or command prompt, or a text editor, or Windows explorer, are

not responsive to your actions.

- Using `top` on UNIX, check the `Mem:` line. Check if the free number and the buffers number are both near 0. On Windows, use `Task Manager` and switch to the `Performance` tab. Check if the `Available` and `System Cache` numbers, under `Physical Memory`, are both near 0. The numbers tell you how much RAM the computer is using for its IO cache.
- CPU usage of your process is unexpectedly low.

Note that modern OSs happily swap out processes that seem idle in order to use the RAM for the IO cache. Some amount of swapping is normal. But, excessive swapping, especially while you are indexing or searching, is not good.

In order to manage how Lucene, in turn, uses memory from the JVM, you first need to measure how much memory Lucene needs. There are various ways, but the simplest is to specify the `-verbose:gc` and `-XX:+PrintGCDetails` options when you run `java`, and then look for the size of the total heap after collection. This is useful because it excludes the memory consumed by garbage objects that are not yet collected. If your Lucene application needs to use up nearly all of the memory allocated for the JVM's maximum heap size, it may cause excessive GC, which will slow things down. If you use even more memory than that, you'll eventually hit `OutOfMemoryError`.

During indexing, the RAM required by Lucene is entirely dictated by the size of the buffer used by `IndexWriter`, which you can control with `setRAMBufferMB`. Don't set this too low as it will slow down indexing throughput. While a segment merge is running, a small amount of additional RAM is required. Searching is more RAM intensive. Here are some tips to reduce RAM usage during searching:

- Optimize your index to purge deleted documents.
- Limit how many fields you directly load into the `FieldCache`, which is entirely memory resident and time consuming to load. Try not to load `String` fields as these are far more memory consuming than `int`, `long` and `float`.
- Limit how many fields you sort by. The first time a search is sorted by a given field, its values are loaded into the `FieldCache`. Similarly, try not to sort on `String` fields.
- Turn off field norms. Norms encode index-time boosting, which combines field boost, doc boost and length boost, into a single byte per document. Even documents without this field consume one byte because the norms are stored as a single contiguous array. This quickly works out to a lot of RAM if you have many indexed fields. Often norms are not actually a big contributor to relevance scoring. For example, if your field values are all similar in length (e.g., a title field), and you are not using field or document boosting, then norms are not necessary. Turn them off by calling `Field.setOmitNorms(true)` whenever you add this field to a document, or use `Field.Index.NO_NORMS` when creating the field. Be sure to fully rebuild your index once you turn them off because if even a single document in your index had norms enabled for that field then all documents in the same segment will still have norms allocated.
- Use a single text field: it's better to combine multiple text fields into a single indexed, tokenized field and search only that field. This will also make searching faster and may improve your search relevance.
- Make sure your analyzer is producing reasonable terms. Use Luke to look at the terms in your index and verify these are legitimate terms that users may search on. It's easy to accidentally index binary documents, which can produce a great many bogus binary terms that would never be used for searching. These terms cause all sorts of problems once they get into your index, so it's best to catch them early by skipping or properly filtering the binary content. If your index has an unusually large number of legitimate terms, e.g. if you are searching a large number of product SKUs, then try calling `IndexReader.setTermIndexInterval` to reduce how many index terms are loaded into RAM. But note that this may slow down searching. There are so many tradeoffs!
- Double check that you're closing and releasing all previous `IndexSearcher`/`IndexReader` instances. Accidentally keeping a reference to past instances can very quickly exhaust RAM and file descriptors.
- Use a Java memory profiler to see what's using so much RAM.

Be sure to test your RAM requirements during searching while you are re-opening a new reader because this will be your peak usage.

Let's go back and re-run our fastest Wikipedia indexing algorithm, intentionally using a heap size that's too small to see what happens if you don't tune memory usage appropriately. Last time we ran it with a 512MB heap size, and we achieved 899.7 doc/s throughput. This time let's give it only 145 MB heap size (anything below this will likely hit `OutOfMemoryError`). Run the algorithm, adding `-Dtask.mem=145M`, and you should see something like this:

```
Operation   round   runCnt   recsPerRun        rec/s  elapsedSec    avgUsedMem    avgTotalMem
BuildIndex      0        1       200002        323.4      618.41   150,899,008    151,977,984
BuildIndex -  - 1 -  -   1 -  -  200002 -  -   344.0 -  - 581.36 - 150,898,992 -  151,977,984
BuildIndex      2        1       200002        334.4      598.05   150,898,976    151,977,984
```

Whoa, that's 334.4 documents per second, which is 2.7 times slower than before! That slowdown is due to excessive GC that the JVM must do to keep memory usage under 145MB. You can see the importance of giving Lucene enough RAM!

10.4 Backing up an index

So, it's 2 AM, and you're having a pleasant dream about all the users who love your Lucene search application when, suddenly, you wake up to the phone. It's an emergency call saying your search index is corrupt. No problem, you answer: restore from the last backup! You do back up your search index, right?

Things will inevitably go wrong: a power supply fails, a hard drive crashes, your RAM becomes random. These events can suddenly render your index completely unusable, almost certainly at the worst possible time. Your final line of protection against such failures is a periodic backup of the index along with simple, accessible steps to restore it.

The simplest way to backup an index is to close your writer and make a copy of all files in the index directory. But this approach has some problems. During the copy, which could take a very long time for a large index, you cannot have a writer open. Many applications cannot accept such a long downtime in their indexing. When a reader is open on the index, you will copy more files than needed, if the reader is holding some files open that are no longer referenced. Finally, the IO load of the copy can slow down searching. You might be tempted to throttle the copy rate to compensate for this, but that will increase your downtime. No wonder so many people just skip backups entirely, cross their fingers, and hope for the best!

Fortunately, as of Lucene 2.3, there is now a simple answer: you can easily make a "live backup" of the index, such that you create a consistent backup image, with just the files referenced by the most recent commit point, without closing your writer. No matter how long the copying takes, you can still make updates to the index. The approach is to use the new `SnapshotDeletionPolicy`, which keeps a commit point alive for as long it takes to complete the backup. Your backup program can take as long as it needs to copy the files. You could throttle its IO or set it to low process priority to make sure it does not interfere with ongoing searching or indexing. You can spawn a sub-process to run `rsync`, `tar`, `robocopy` or your favorite backup utility, giving it the list of files to copy. This can also be used to mirror a snapshot of the index to other computers.

The backup must be initiated by the JVM that has your writer and you must create your writer using the `SnapshotDeletionPolicy`, like this:

```
IndexDeletionPolicy policy = new KeepOnlyLastCommitDeletionPolicy();
SnapshotDeletionPolicy snapshotter = new SnapshotDeletionPolicy(policy);
IndexWriter writer = new IndexWriter(dir, autoCommit, analyzer, snapshotter);
```

Note that you can pass any existing deletion policy into `SnapshotDeletionPolicy` (it does not have to be `KeepOnlyLastCommitDeletionPolicy`).

When you want to do a backup, just do this:

```
try {
  IndexCommitPoint commit = snapshotter.snapshot();
  Collection fileNames = commit.getFileNames();
  <iterate over & copy files from fileNames>
} finally {
  snapshotter.release();
}
```

Inside the `try` block, all files referenced by the commit point will not be deleted by the writer, even if the writer is still making changes, optimizing, etc., as long as the writer is not closed. It's fine if this copy takes a long time

because it's still copying a single point-in-time snapshot of the index. While this snapshot is kept alive, the files that belong to it will hold space on disk. So while a backup is running your index will use more disk space than it normally would (assuming the writer is continuing to commit changes to the index). Once you're done, call `release` to allow the writer to delete these files the next time it flushes or is closed.

As of 2.1 Lucene is "write once". This means you can do an incremental backup by simply comparing file names. You do not have to look at the contents of each file, not its last modified timestamp, because, once a file is written, it will not be changed. The only exception to this is the file `segments.gen`, which is overwritten on every commit, and so you should always copy this file. You should not copy the write lock file (`write.lock`).

`SnapshotDeletionPolicy` has a couple small limitations:

- It only keeps one snapshot alive at a time. You could fix this by making a similar deletion policy that keeps track of more than one snapshot at a time.
- The current snapshot is not persisted to disk. This means if you close your writer and open a new one, the snapshot will be deleted. So you cannot close your writer until the backup has completed. This is also easy to fix: you could store and load the current snapshot on disk and then protect it on opening a new writer. This would allow the backup to keep running even if the original writer is closed and new one opened.

Believe it or not, that's all there is to it! Now we move onto restoring your index.

10.4.2 Restoring the index

In addition to doing periodic backups, you should have simple steps on hand to quickly restore the index from backup and restart your application. You should periodically test both backups and restore. 2 AM is the worst time to find out you had a tiny bug in your backup process!

Here are the steps to follow when restoring an index:

1. Close any existing readers and writers on the index directory, so the file copies will succeed. On Windows, if there are still processes using those files, you won't be able to overwrite them.
6. Remove all existing files from the index directory. If you hit an "Access is denied" error, double check step 1.
7. Copy all files from your backup into the index directory. Be certain this copy does not hit any errors, like a disk full, because that is a sure way to corrupt your index.

Speaking of corruption, let's talk next about common errors you may hit with Lucene.

10.5 Common Errors

Lucene is wonderfully resilient to most common errors. If you fill up your disk, or hit an `OutOfMemoryException`, you will lose only the documents buffered in memory at the time. Documents already committed to the index will be intact, and the index will be consistent. The same is true if the JVM crashes, or hits an unhandled exception, or is explicitly killed.

If you hit a `LockObtainFailedException`, that's likely because there's a leftover `write.lock` file in your index directory which was not properly released before your application or JVM shut down or crashed. Consider switching to `NativeFSLockFactory`, which uses the OS provided locking (through the `java.nio.*` APIs) and will properly release the lock whenever the JVM exits normally or abnormally. You can safely remove the `write.lock` file, or use the `IndexReader.unlock` static method to do so. But first be certain there is in fact no writer writing to that directory!

If you hit `AlreadyClosedException`, double check your code: this means you are closing the writer or reader but then continuing to use it. One common pitfall is to iterate through the hits of a search after the underlying reader has been closed. This is not allowed because those hits need the reader to remain open in order to load documents.

10.5.1 Index Corruption

So maybe you've seen an odd, unexpected exception in your logs, or, maybe the computer crashed while indexing. Nervously, you bring your Lucene application back up, and all seems to be fine, so you just shrug and move onto the next crisis. But you can't escape the sinking sensation and burning question deep in your mind: is it possible my index is now corrupt? Suddenly a month or two later, more strange exceptions start appearing. Corruption is

insidious: it may silently enter your index but take quite a long time to be discovered, perhaps when the corrupt segment is next merged, or, when a certain search term happens to hit on a corrupt posting list. How can you manage this risk?

Unfortunately, there are certain known situations that can lead to index corruption. If this happens to you, try to get to the root cause of the corruption. Look through your logs and explain all exceptions. Otherwise it may simply re-occur. Here are some typical causes of index corruption:

- Hardware problems: bad power supply, slowly failing hard drive, or bad RAM.
- The OS or computer crashes, or the power cord is pulled (this is a known issue in Lucene and should be fixed in future releases).
- Accidentally allowing two writers to write to the same index at the same time. Lucene's locking normally prevents this. But if you use a different `LockFactory` inappropriately, or if you incorrectly removed a `write.lock` that was in fact a real lock, that could allow two writers at once.
- Errors when copying. If you have a step in your indexing process where an index is copied from one place to another, an error in that copying can easily corrupt the target index.
- It's even possible you've hit a previously undiscovered bug in Lucene! Take your case to the lists, or open an issue with as much detail as possible about what led to the corruption. The Lucene developers will jump on it!

While you can't eliminate these risks, you can be proactive in detecting index corruption. If you hit a `CorruptIndexException`, then you know your index is corrupted. But all sorts of other unexplained exceptions are also possible. To proactively test your index for corruption, here are two things to try:

- Run Lucene with assertions enabled (`java -ea:org.apache.lucene`, when launching `java` at the command line). This causes Lucene to do more thorough checking at many points during indexing and searching, which could catch corruption sooner than you would otherwise.
- Run the `org.apache.lucene.index.CheckIndex` tool, new as of 2.3, providing the path to your index directory as the only command-line argument. This tool runs a thorough check of every segment in the index, and reports detailed statistics, and any corruption, for each. It produces output like this:

```
Opening index @ /tango/lucene/work/index

Segments file=segments_2 numSegments=1 version=FORMAT_SHARED_DOC_STORE [Lucene 2.3]
  1 of 1: name=_8 docCount=36845
    compound=false
    numFiles=11
    size (MB)=103.619
    docStoreOffset=0
    docStoreSegment=_0
    docStoreIsCompoundFile=false
    no deletions
    test: open reader.........OK
    test: fields, norms.......OK [4 fields]
    test: terms, freq, prox...OK [612173 terms; 20052335 terms/docs pairs; 42702159 tokens]
    test: stored fields.......OK [147380 total field count; avg 4 fields per doc]
    test: term vectors........OK [110509 total vector count; avg 2.999 term/freq vector fields
per doc]
No problems were detected with this index.
```

If you find your index is corrupt, first try to restore from backups. But what if all of your backups are corrupt? This can easily happen since corruption may take a long time to detect. What can you do, besides rebuilding your full index from scratch? Fortunately, there is one final resort: use the `CheckIndex` tool to repair it.

10.5.2 Repairing an index

When all else fails, your final resort is the `CheckIndex` tool. In addition to printing details of your index, this tool can repair your index if you add the `-fix` command-line option:

```
java org.apache.lucene.index.CheckIndex <pathToIndex> -fix
```

That will forcefully remove those segments that hit problems. Note that this completely removes all documents that were contained in that segment, so use this option with caution and make a backup copy of your index first! You should use this tool just to get your search operational again, on an emergency basis. Once you are back up, then you should rebuild your index to recover the lost documents.

10.6 Summary

We've covered many important hands-on topics in this chapter! Think of this chapter like your faithful swiss-army knife: you now have the necessary tools under your belt to deal with all the important, real-world aspects of Lucene administration.

Lucene has great out-of-the-box performance, and now you know how to further tune that performance for specific metrics that are important to your application, using the powerful and extensible `contrib/benchmark` framework. Unfortunately, tuning for one metric often comes at the expense of others, so you should decide up front which metric is most important to you.

You've seen how crucial it is to use threads during indexing and searching to take advantage of the concurrency in modern computers, and now you have a couple drop-in classes that make this painless. Backing up and index is a surprisingly simple operation that no longer requires stopping your `IndexWriter`.

Lucene's consumption of disk, file descriptors, and memory is no longer a mystery and is well within your control. Index corruption is not something to fear, since you know what might cause it and you know how to detect and repair a corrupted index. The common errors that happen are easy to understand and fix. Really, that's it! Go forth and build!

B

Lucene index format

So far, we have treated the Lucene index more or less as a black box and have concerned ourselves only with its logical view. Although you don't need to understand index structure details in order to use Lucene, you may be curious about the "magic." Lucene's index structure is a case study in itself of highly efficient data structures and clever arrangement to maximize performance and minimize resource usage. You may see it as a purely technical achievement, or you can view it as a masterful work of art. There is something innately beautiful about representing rich structure in the most efficient manner possible. (Consider the information represented by fractal formulas or DNA as nature's proof.)

In this appendix, we'll look at the logical view of a Lucene index, where we've fed documents into Lucene and retrieved them during searches. Then, we'll expose the inner structure of Lucene's inverted index.

B.1 Logical index view

Let's first take a step back and start with a quick review of what you already know about Lucene's index. Consider figure B.1. From the perspective of a software developer using Lucene API, an index can be considered a black box represented by the abstract `Directory` class. When indexing, you create instances of the Lucene `Document` class and populate it with `Field`s that consist of name and value pairs. Such a `Document` is then indexed by passing it to `IndexWriter.addDocument(Document)`. When searching, you again use the abstract `Directory` class to represent the index. You pass that `Directory` to the `IndexSearcher` class and then find `Document`s that match a given query by passing search terms encapsulated in the `Query` object to one of `IndexSearcher`'s search methods. The results are matching `Document`s represented by the `Hits` object.

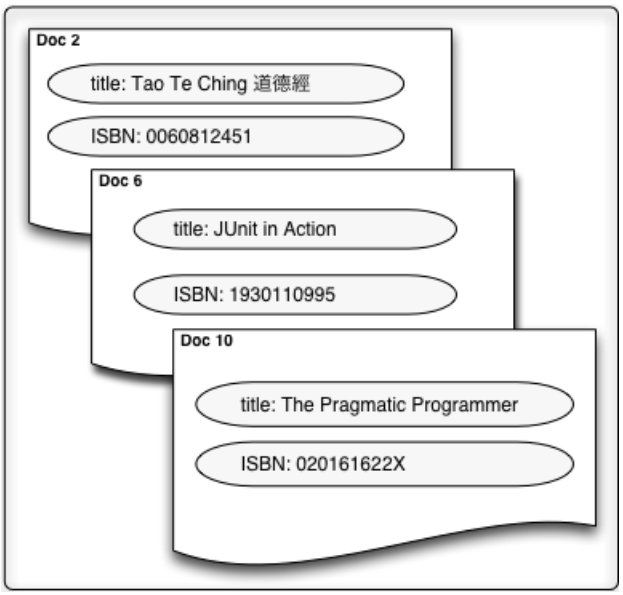


Figure B.1 The logical, black-box view of a Lucene index

B.2 About index structure

When we described Lucene's `Directory` class in section 1.5, we pointed out that one of its concrete subclasses, `FSDirectory`, stores the index in a file-system directory. We have also used `Indexer`, a program for indexing text files, shown in listing 1.1. Recall that we specified several arguments when we invoked `Indexer` from the command line and that one of those arguments was the directory in which we wanted `Indexer` to create a Lucene index. What does that directory look like once `Indexer` is done running? What does it contain? In this section, we'll peek into a Lucene index and explain its structure.

Before we start, though, you should know that the index file format often changes between releases. It's free to change without breaking backwards compatibility because the classes that access the index can detect when they are interacting with an older format index and act accordingly. The current format is documented here:

http://lucene.apache.org/java/docs/fileformats.html

Lucene supports two index structures: multifile indexes and compound indexes. Multifile indexes use quite a few files to represent the index, while compound indexes use a special file, much like an archive such as a zip file, to hold multiple index files in a single file. Let's look at each type of index structure, starting with multifile.

B.2.1 Understanding the multifile index structure

If you look at the index directory created by our `Indexer`, you'll see a number of files whose names may seem random at first. These are *index files*, and they look similar to those shown here:

```
-rw-rw-rw- 1 mike users 12327579 Feb 29 05:29 _2.fdt
-rw-rw-rw- 1 mike users     6400 Feb 29 05:29 _2.fdx
-rw-rw-rw- 1 mike users       33 Feb 29 05:29 _2.fnm
-rw-rw-rw- 1 mike users  1036074 Feb 29 05:29 _2.frq
-rw-rw-rw- 1 mike users     2404 Feb 29 05:29 _2.nrm
-rw-rw-rw- 1 mike users  2128366 Feb 29 05:29 _2.prx
-rw-rw-rw- 1 mike users    14055 Feb 29 05:29 _2.tii
-rw-rw-rw- 1 mike users  1034353 Feb 29 05:29 _2.tis
-rw-rw-rw- 1 mike users     5829 Feb 29 05:29 _2.tvd
-rw-rw-rw- 1 mike users 10227627 Feb 29 05:29 _2.tvf
-rw-rw-rw- 1 mike users    12804 Feb 29 05:29 _2.tvx
-rw-rw-rw- 1 mike users       20 Feb 29 05:29 segments.gen
-rw-rw-rw- 1 mike users       53 Feb 29 05:29 segments_3
```

Notice that some files share the same prefix. In this example index, a number of files start with the prefix *_2*, followed by various extensions. This leads us to the notion of *segments*.

INDEX SEGMENTS

A Lucene index consists of one or more segments, and each segment is made up of several index files. Index files that belong to the same segment share a common prefix and differ in the suffix. In the previous example index, the index consisted of a single segment whose files started with *_2*:

The following example shows an index with two segments, _0 and _1:

```
-rw-rw-rw- 1 mike users 7743790 Feb 29 05:28 _0.fdt
-rw-rw-rw- 1 mike users    3200 Feb 29 05:28 _0.fdx
-rw-rw-rw- 1 mike users      33 Feb 29 05:28 _0.fnm
-rw-rw-rw- 1 mike users  602012 Feb 29 05:28 _0.frq
-rw-rw-rw- 1 mike users    1204 Feb 29 05:28 _0.nrm
-rw-rw-rw- 1 mike users 1337462 Feb 29 05:28 _0.prx
-rw-rw-rw- 1 mike users   10094 Feb 29 05:28 _0.tii
-rw-rw-rw- 1 mike users  737331 Feb 29 05:28 _0.tis
-rw-rw-rw- 1 mike users    2949 Feb 29 05:28 _0.tvd
-rw-rw-rw- 1 mike users 6294227 Feb 29 05:28 _0.tvf
-rw-rw-rw- 1 mike users    6404 Feb 29 05:28 _0.tvx
-rw-rw-rw- 1 mike users 4583789 Feb 29 05:28 _1.fdt
-rw-rw-rw- 1 mike users    3200 Feb 29 05:28 _1.fdx
-rw-rw-rw- 1 mike users      33 Feb 29 05:28 _1.fnm
-rw-rw-rw- 1 mike users  405527 Feb 29 05:28 _1.frq
-rw-rw-rw- 1 mike users    1204 Feb 29 05:28 _1.nrm
-rw-rw-rw- 1 mike users  790904 Feb 29 05:28 _1.prx
-rw-rw-rw- 1 mike users    7499 Feb 29 05:28 _1.tii
-rw-rw-rw- 1 mike users  548646 Feb 29 05:28 _1.tis
-rw-rw-rw- 1 mike users    2884 Feb 29 05:28 _1.tvd
-rw-rw-rw- 1 mike users 3933404 Feb 29 05:28 _1.tvf
-rw-rw-rw- 1 mike users    6404 Feb 29 05:28 _1.tvx
-rw-rw-rw- 1 mike users      20 Feb 29 05:28 segments.gen
```

```
-rw-rw-rw- 1 mike users      78 Feb 29 05:28 segments_3
```

You can think of a segment as a subindex, although each segment isn't a fully independent index.

As you can see in figure B.2, each segment contains one or more Lucene `Documents`, the same ones we add to the index with the `addDocument(Document)` method in the `IndexWriter` class. By now you may be wondering what function segments serve in a Lucene index; what follows is the answer to that question.

INCREMENTAL INDEXING

Using segments lets you quickly add new `Documents` to the index by adding them to newly created index segments and only periodically merging them with other, existing segments. This process makes additions efficient because it minimizes physical index modifications. Figure B.2 shows an index that holds 34 `Documents`. This figure shows an unoptimized index—it contains multiple segments. If this index were to be optimized using the default Lucene indexing parameters, all 34 of its documents would be merged in a single segment.

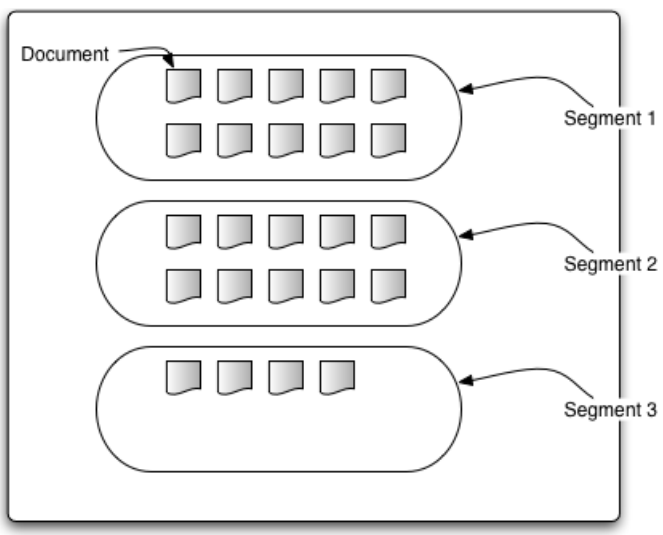


Figure B.2 Unoptimized index with 3 segments, holding 34 documents

One of Lucene's strengths is that it supports incremental indexing, which isn't something every IR library is capable of. Whereas some IR libraries need to reindex the whole corpus when new data is added to their index, Lucene does not. After a document has been added to an index, its content is immediately made searchable. In IR terminology, this important feature is called *incremental indexing*. The fact that

Lucene supports incremental indexing makes Lucene suitable for environments that deal with large bodies of information where complete reindexing would be unwieldy.

Because new segments are created as new `Documents` are indexed, the number of segments, and hence index files, varies while indexing is in progress. Once an index is fully built, the number of index files and segments remains steady.

A CLOSER LOOK AT INDEX FILES

Each index file carries a certain type of information essential to Lucene. If any index file is modified or removed by anything other than Lucene itself, the index becomes corrupt, and the only option is a complete reindexing of the original data. On the other hand, you can add random files to a Lucene index directory without corrupting the index. For instance, if we add a file called random.txt to the index directory, as shown here, Lucene ignores that file, and the index doesn't become corrupt:

```
-rw-rw-rw- 1 mike users 12327579 Feb 29 05:29 _2.fdt
-rw-rw-rw- 1 mike users     6400 Feb 29 05:29 _2.fdx
-rw-rw-rw- 1 mike users       33 Feb 29 05:29 _2.fnm
-rw-rw-rw- 1 mike users  1036074 Feb 29 05:29 _2.frq
-rw-rw-rw- 1 mike users     2404 Feb 29 05:29 _2.nrm
-rw-rw-rw- 1 mike users  2128366 Feb 29 05:29 _2.prx
-rw-rw-rw- 1 mike users    14055 Feb 29 05:29 _2.tii
-rw-rw-rw- 1 mike users  1034353 Feb 29 05:29 _2.tis
-rw-rw-rw- 1 mike users     5829 Feb 29 05:29 _2.tvd
-rw-rw-rw- 1 mike users 10227627 Feb 29 05:29 _2.tvf
-rw-rw-rw- 1 mike users    12804 Feb 29 05:29 _2.tvx
-rw-rw-rw- 1 mike users       17 Mar 30 03:34 random.txt
-rw-rw-rw- 1 mike users       20 Feb 29 05:29 segments.gen
-rw-rw-rw- 1 mike users       53 Feb 29 05:29 segments_3
```

The secret to this is the `segments` file (`segments_3`). As you may have guessed from its name, the `segments` file stores the name and certain details of all existing index segments. Every time an `IndexWriter` commits a change to the index, the generation (the _3 in the above listing) of the segments file is incremented. For example, a commit to this index would write `segments_4` and remove `segments_3` as well as any now unreferenced files. Before accessing any files in the index directory, Lucene consults this file to figure out which index files to open and read. Our example index has a single segment, _2, whose name is stored in this segments file, so Lucene knows to look only for files with the *_2* prefix. Lucene also limits itself to files with known extensions, such as .fdt, .fdx, and other extensions shown in our example, so even saving a file with a segment prefix, such as _2.txt, won't throw Lucene off. Of course, polluting an index directory with non-Lucene files is strongly discouraged.

The exact number of files that constitute a Lucene index and each segment varies from index to index and depends on the number of fields the index contains. However, every index contains a single segments file and a single segments.gen file. The segments.gen file is always 20 bytes and contains the suffix (generation) of the current segments as a redundant way for Lucene to determine the most recent commit.

CREATING A MULTIFILE INDEX

By now you should have a good grasp of the multifile index structure; but how do you use the API to instruct Lucene to create a multifile index and not the default compound-file index? Let's look back at our faithful `Indexer` from listing 1.1. In that listing, you'll spot the following:

```
IndexWriter writer = new IndexWriter(indexDir,
  new StandardAnalyzer(), true, IndexWriter.MaxFieldLength.UNLIMITED);
writer.setUseCompoundFile(false);
```

Because the compound-file index structure is the default, we disable it and switch to a multifile index by calling `setUseCompoundFile(false)` on an `IndexWriter` instance.

B.2.2 UNDERSTANDING THE COMPOUND INDEX STRUCTURE

When we described multifile indexes, we said that the number of index files depends on the number of indexed fields present in the index. We also mentioned that new segments are created as documents are added to an index; since a segment consists of a set of index files, this results in a variable and possibly large number of files in an index directory. Although the multifile index structure is straightforward and works for most scenarios, it isn't suitable for environments with large number of indexes and other environment where using Lucene results in a large number of index files.

Most, if not all, contemporary operating systems limit the number of files in the system that can be opened at one time. Recall that Lucene creates new segments as new documents are added, and every so often it merges them to reduce the number of index files. However, while the merge procedure is executing, the number of index files temporarily increases. If Lucene is used in an environment with lots of indexes that are being searched or indexed simultaneously, it's possible to reach the limit of open files set by the operating system. This can also happen with a single Lucene index if the index isn't optimized or if other applications are running simultaneously and keeping many files open. Lucene's use of open file handles depends on the structure and state of an index. Section 10.XXX presents formulas for calculating the number of open files, as well as ideas to reduce that number.

COMPOUND INDEX FILES

The only visible difference between the compound and multifile indexes is the contents of an index directory. Here's an example of a compound index:

```
-rw-rw-rw-    1 mike     users   12441314 Mar 30 04:27 _2.cfs
-rw-rw-rw-    1 mike     users         15 Mar 30 04:27 segments_4
-rw-rw-rw-    1 mike     users         20 Mar 30 04:27 segments.gen
```

Instead of having to open and read 10 files from the index, as in the multifile index, Lucene must open only three files when accessing this compound index, thereby consuming fewer system resources. The compound index reduces the number of index files, but the concept of segments, documents, fields, and terms still applies. The difference is that a compound index contains a single .cfs file per segment, whereas each segment in a multifile index contains consists of seven different files. The compound structure encapsulates individual index files in a single .cfs file.

CREATING A COMPOUND INDEX

Because the compound index structure is the default, you don't have to do anything to specify it. However, if you like explicit code, you can call the `setUse-Compound(boolean)` method, passing it a `true` value:

```
IndexWriter writer = new IndexWriter(indexDir,
  new StandardAnalyzer(), true, IndexWriter.MaxFieldLength.UNLIMITED);
writer.setUseCompoundFile(true);
```

Pleasantly, you aren't locked into the multifile or compound format. After indexing, you can still switch from one format to another, although this will only affect newly written segments. But there is a trick!

B.2.3 CONVERTING FROM ONE INDEX STRUCTURE TO THE OTHER

It's important to note that you can switch between the two described index structures at any point during indexing. All you have to do is call the IndexWriter's `set-UseCompoundFiles(boolean)` method at any time during indexing; the next time Lucene merges index segments, it will write the new segment in the format you specified.

Similarly, you can convert the structure of an existing index without adding more documents to it. For example, you may have a multifile index that you want to convert to a compound one, to reduce the number of open files used by Lucene. To do so, open your index with `IndexWriter`, specify the compound structure, optimize the index, and close it:

```
IndexWriter writer = new IndexWriter(indexDir,
  new StandardAnalyzer(), false, IndexWriter.MaxFieldLength.UNLIMITED);
writer.setUseCompoundFile(true);
writer.optimize();
writer.close();
```

Note that the third `IndexWriter` parameter is `false` to ensure that the existing index isn't destroyed. We discussed optimizing indexes in section 2.8. Optimizing forces Lucene to merge index segments, thereby giving it a chance to write them in a new format specified via the `setUseCompoundFile(boolean)` method.

B.3 Choosing the index structure

Although switching between the two index structures is simple, you may want to know beforehand how many open files resources Lucene will require when accessing your index. If you're designing a system with multiple simultaneously indexed and searched indexes, you'll most definitely want to take out a pen and a piece of paper and do some simple math with us now.

B.3.1 Calculating the number of open files

Let's consider a multifile index first. A multifile index contains seven index files for each segment (10 if any fields have term vectors indexed) and a single segments plus a segments.gen file for the whole index.

Imagine a system that contains 100 Lucene indexes. Also assume that term vectors are used, these indexes aren't optimized and that each has nine segments that haven't been merged into a single segment yet, as is often the case during indexing. If all 100 indexes are open for searching at the same time, this will result in 9,000 open files. Here is how we got this number:

```
100 indexes * (9 segments per index * 10 files per segment)
= 100 * 9 * 10
= 9000 open files
```

Although today's computers can usually handle this many open files, most come with a preconfigured limit that is much lower. In section 10.3.2, we go into more detail about how to measure and manage file descriptor.

Next, let's consider the same 100 indexes, but this time using the compound structure. Only a single file with a .cfs extension is created per segment, in addition to a single segments and segments.gen file for the whole index. Therefore, if we use the compound index instead of the multifile one, the number of open files is reduced to 900:

```
100 indexes * (9 segments per index * (1 file per segment))
= 100 * 9 * 1
= 900 open files
```

The lesson here is that if you need to develop Lucene-based software that will run in environments with a large number of Lucene indexes with a number of indexed fields, you should consider using a compound index. Of course, you can use a compound index even if you're writing a simple application that deals with a single Lucene index.

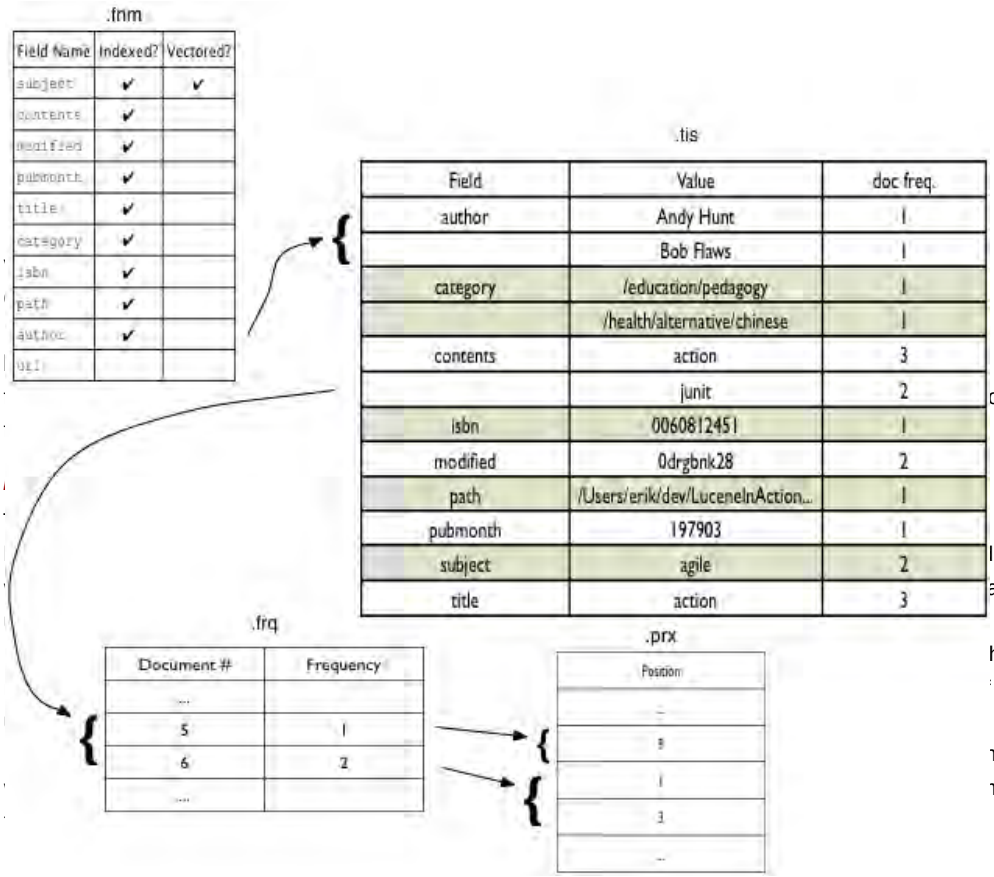
B.3.2 INDEXING AND SEARCHING PERFORMANCE

Performance is another factor you should consider when choosing the index structure. Creating an index with a compound structure is generally 5–10% slower than creating an equivalent multifile index. This is because when writing a new segment the IndexWriter first writes the multifile format and then creates the compound file. This extra steps adds time to the indexing process. Section 10.1.2 goes into more detail about tuning and measuring indexing performance. Here's our advice: If you need to squeeze every bit of indexing performance out of Lucene, use the multifile index structure, but first try tuning compound structure indexing by manipulating the indexing parameters covered in sections 2.7 and 10.1.2. This performance difference and the difference in the amount of system resources the two index structures use are their only notable differences. All Lucene's features work equally well with either type of index.

B.4 Inverted index

Lucene uses a well-known index structure called an *inverted index*. Quite simply, and probably unsurprisingly, an inverted index is an inside-out arrangement of documents such that terms take center stage. Each term refers to the documents that contain it. Let's dissect our sample book data index to get a deeper glimpse at the files in an index `Directory`.

Regardless of whether you're working with a `RAMDirectory`, an `FSDirectory`, or any other `Directory` implementation, the internal structure is a group of files. In a `RAMDirectory`, the files are



a file-system

the files in a
to a single .cfs
ompound index

web site at
lious for you, if
all file structure

he actual data
getting caught

nt (in this case,
me prefix (_*c* in

Figure B.3 Detailed look inside the Lucene index format

The following sections describe each of the files shown in figure B.3 in more detail.

FIELD NAMES (.FNM)

The .fnm file contains all the field names used by documents in the associated segment. Each field is flagged to indicate options that were used while indexing:

- Is it indexed?
- Does it have term vectors enabled?
- Does it store norms?
- Does it have payloads?

The order of the field names in the .fnm file is determined during indexing and isn't necessarily alphabetical. Each field is assigned a unique integer, the field number, according to the order in this file. That field number, instead of the string name, is used in other Lucene files to save space.

In our sample index, only the `subject` field is vectored. The `url` field was added as a `Field.Index.NO` field, which is neither indexed nor vectored.

TERM DICTIONARY (.TIS)

All terms (tuples of field name and value) in a segment are stored in the .tis file. Terms are ordered first alphabetically, according to the UTF16 java character, by field name and then by value within a field. Each term entry contains its *document frequency:* the number of documents that contain this term within the segment.

Figure B.3 shows only a sampling of the terms in our index, one or more from each field. Note that the `url` field is missing because it was added as an unindexed field, which is stored only and not available as terms. Not shown is the .tii file, which is a cross-section of the .tis file designed to be kept in physical memory for random access to the .tis file. For each term in the .tis file, the .frq file contains entries for each document containing the term.

In our sample index, two books have the value "junit" in the `contents` field: *JUnit in Action* (document ID 6), and *Java Development with Ant* (document ID 5).

TERM FREQUENCIES

Term frequencies in each document are listed in the .frq file. In our sample index, *Java Development with Ant* (document ID 5) has the value "junit" once in the `contents` field. *JUnit in Action* has the value "junit" twice, provided once by the title and once by the subject. Our `contents` field is an aggregation of title, subject, and author. The frequency of a term in a document factors into the score calculation (see section 3.3) and typically boosts a document's relevance for higher frequencies.

For each document listed in the .frq file, the positions (.prx) file contains entries for each occurrence of the term within a document.

TERM POSITIONS

The .prx file lists the position of each term within a document. The position information is used when queries demand it, such as phrase queries and span queries. Position information for tokenized fields comes directly from the token position increments designated during analysis. This file also contains the payloads, if any.

Figure B.3 shows three positions, for each occurrence of the term *junit*. The first occurrence is in document 5 (*Java Development with Ant*) in position 9. In the case of document 5, the field value (after analysis) is "java development ant apache jakarta ant build tool ***junit*** java development erik hatcher steve loughran". We used the `StandardAnalyzer`; thus stop words (*with* in *Java Development with Ant*, for example) are removed and aren't accounted for in positional information (see section 4.7.3 for more on stop word removal and positional information). Document 6, *JUnit in Action*, has a `contents` field containing the value "junit" twice, once in position 1 and again in position 3: "***junit*** action ***junit*** unit testing mock objects vincent massol ted husted".[1]

STORED FIELDS

When you request that a field be stored (Field.Store.YES), it is written into two files: the .fdx file and the .fdt file. The .fdx file contains simple index information, which is uses to resolve document number to

[1]We're indebted to Luke, the fantastic index inspector, for allowing us to easily gather some of the data provided about the index structure.

exact position in the .fdt file for that document's stored fields. The .fdt file actually contains the contents of the fields that were stored.

TERM VECTORS

Term vectors are stored in three files. The .tvf file is the largest and actually stores the specific terms, sorted alphabetically, and their frequencies, plus the optional offsets and positions for the terms. The .tvd file lists which fields had term vectors for a given document and indexes byte offsets into the .tvf file so specific fields can be retrieved. Finally, the .tvx file has index information, which resolves document number into the byte positions in the .tvf and .tvd files.

NORMS

The .nrm file contains normalization factors that represent the boost information gathered during indexing. Each document has one byte in this file, which encodes the combination of the document's boost, that field's boost, and a normalization factor based on the overall length of the content in that field.

B.5 Summary

The rationale for the index structure is two-fold: maximum performance and minimum resource utilization. For example, if a field isn't indexed it's a very quick operation to dismiss it entirely from queries based on the indexed flag of the .fnm file. The .tii file, cached in RAM, allows for rapid random access into the term dictionary .tis file. Phrase and span queries need not look for positional information if the term itself isn't present. Streamlining the information most often needed, and minimizing the number of file accesses during searches is of critical concern. These are just some examples of how well thought out the index structure design is. If this sort of low-level optimization is of interest, please refer to the Lucene index file format details on the Lucene web site, where details we have glossed over here can be found.

D

Lucene Contrib Benchmark

The contrib/benchmark package is a very useful framework for running repeatable performance tests. By creating a short script, called an "algorithm" (file.alg), you tell benchmark what test to run and how to report its results. We briefly used benchmark in Chapter 10 to measure Lucene's indexing performance. In this appendix we go into more detail. Benchmark is quite new and will improve over time, so always check the javadocs. The `byTask` sub-package has a good overview.

You might be tempted to just create your own testing framework. Likely you've done so already many times in the past. But there are some important reasons to use benchmark instead:

- Since an algorithm file is a simple text file, it's easily and quickly shared with others so they can reproduce your results. This is vitally important in cases where you are trying to track down a performance anomaly and you'd like to understand the source. Whereas for your own testing framework often there are numerous dependencies and perhaps access to a database that holds that would have to be transferred for someone else to run the test.
- The benchmark framework already has builtin support for common standard sources of documents (eg, Reuters, Wikipedia, Trec).
- With your own test, it's easy to accidentally create performance overhead in the test code itself, or even a bug, which skews results. The benchmark package, thanks to being open source, is well-debugged and well-tuned so it's less likely to hit this. And it only gets better over time!
- Thanks to a great many builtin tasks, you can create many algorithms without writing any Java code. By writing a few lines (the algorithm file) you can craft any test you'd like. You only have to change your script and re-run if you want to test something else. No compilation required!
- Benchmark has multiple extension points, to easily customize the source of documents, source of queries, how the metrics are reported in the end. You can also create your own tasks, as we did in Section 10.XXX.
- Benchmark already gathers important metrics, like run time, documents per second, and memory usage, saving you from having to instrument these in your custom test code.

Let's get our feet wet now and run a basic algorithm.

D.1 Running an algorithm

Save the following lines to a file, `test.alg`:

```
# The analyzer to use
analyzer=org.apache.lucene.analysis.standard.StandardAnalyzer

# Document source
doc.maker=org.apache.lucene.benchmark.byTask.feeds.ReutersDocMaker

# Directory
directory=FSDirectory
```

```
# Turn on stored fields
doc.stored = true

# Turn on term vectors
doc.term.vectors = true

# Use autoCommit=false when creating IndexWriter
autocommit = false

# Don't use compound file format
compound = false

# Make one pass through all documents
doc.maker.forever = false

# Repeat 3 times
{ "Rounds"

  # Clear the index
  ResetSystemErase

  # Name the following tasks "BuildIndex"
  { "BuildIndex"

   # Create a new index
   -CreateIndex

   # Add all docs
   { "AddDocs" AddDoc > : * }

   # Close the index
   -CloseIndex
  }

  NewRound

} : 3

# Report on the BuildInex tasks
RepSumByPrefRound BuildIndex
```

As you can probably guess, this algorithm indexes the entire Reuter's corpus, 3 times, and then reports the performance of the `BuildIndex` step separately for each round. Those steps include creating a new index, adding all Reuter's documents, and closing the index. Remember, when testing indexing performance it's important to include the time to close the index since necessary time consuming things happen during `close()`. For example, Lucene waits for any still-running background merges to finish, and then syncs all newly written files in the index.

To run your algorithm, do this:

```
ant run-task -Dtask-alg=<file.alg> -Dtask.mem=512M
```

If you've implemented any custom tasks, you'll need to include the classpath to your compiled sources by adding this to the ant command line:

```
-Dbenchmark.ext.classpath=/path/to/classes
```

Ant first runs a series of dependency targets, for example making sure all sources are compiled and downloading, and unpacking the Reuters corpus. Finally it runs your task and produces something like this under the "run-task" output:

```
Working Directory: work
Running algorithm from: /mnt2/mike/src/lucene.lia/eg1.alg
------------> config properties:
analyzer = org.apache.lucene.analysis.standard.StandardAnalyzer
autocommit = false
```

```
    compound = false
    directory = FSDirectory
    doc.maker = org.apache.lucene.benchmark.byTask.feeds.ReutersDocMaker
    doc.maker.forever = false
    doc.stored = true
    doc.term.vectors = true
    work.dir = work
    ------------------------------
    ------------> ReutersDocMaker statistics (0):
    total count of unique texts:             21,578
    total bytes of unique texts:         17,550,748


    ------------> algorithm:
    Seq {
        Rounds_3 {
            ResetSystemErase
            BuildIndex {
                -CreateIndex
                AddDocs_Exhaust {
                    AddDoc
                > * EXHAUST
                -CloseIndex
            }
            NewRound
        } * 3
        RepSumByPrefRound BuildIndex
    }

    ------------> starting task: Seq
    --> 1.14 sec: main processed (add) 500 docs
    --> 1.96 sec: main processed (add) 1000 docs
   yada yada yada...
    --> 31.06 sec: main processed (add) 21000 docs
    --> 31.86 sec: main processed (add) 21500 docs

    --> Round 0-->1

    ------------> ReutersDocMaker statistics (1):
    num docs added since last inputs reset:              21,578
    total bytes added since last inputs reset:         17,550,748


    --> 0.74 sec: main processed (add) 22000 docs
    --> 1.5 sec: main processed (add) 22500 docs
    yada yada yada....
    --> 33.2 sec: main processed (add) 42500 docs
    --> 33.95 sec: main processed (add) 43000 docs

    --> Round 1-->2

    ------------> ReutersDocMaker statistics (2):
    num docs added since last inputs reset:              21,578
    total bytes added since last inputs reset:         17,550,748


    --> 0.62 sec: main processed (add) 43500 docs
    --> 1.4 sec: main processed (add) 44000 docs
   yada yada yada...
    --> 28.5 sec: main processed (add) 64000 docs
    --> 29.16 sec: main processed (add) 64500 docs

    --> Round 2-->3


    ------------> Report sum by Prefix (BuildIndex) and Round (3 about 3 out of 14)
    Operation   round   runCnt   recsPerRun        rec/s  elapsedSec    avgUsedMem
avgTotalMem
    BuildIndex      0        1        21578        671.8       32.12    24,821,656
34,082,816
    BuildIndex -  - 1 -  -   1 -  -   21578 -  -   627.7 -  -  34.38 -  29,382,656 -
```

```
34,463,744
     BuildIndex      2        1        21578        729.1       29.60    34,173,632
36,777,984

     ####################
     ###  D O N E !!! ###
     ####################
```

First, benchmark prints all the settings you are running with. It's best to look this over and verify the settings are what you intended. Next it "pretty prints" the steps of the algorithm. You should also verify this algorithm is what you expected. If you put a closing } in the wrong place then this is where you will spot it! Finally, it runs the algorithm and prints the status output, which usually consists of 1) the documents source periodically printing how many documents it has produced, and 2) the "Rounds" task printing whenever it finishes a new round. When this finishes, and assuming you have reporting tasks in your algorithm, the report is generated detailing the metrics from each round.

The final report shows one line per round, because we are using a report task (`RepSumByPrefRound`) that breaks out results by round. For each round, it includes the number of records (added documents in this case), records per second, elapsed seconds, and memory usage. The average total memory is obtained by calling `java.lang.Runtime.getRuntime().totalMemory()`. The average used memory is computed by subtracting `freeMemory()` from `totalMemory()`.

What exactly is a "record"? In general, most tasks count as +1 on the record count. For example, every call to `AddDoc` adds 1. Task sequences aggregate all records counts of their children. To prevent the record count from incrementing, you prefix the task with a "-" as we have done above for `CreateIndex` and `CloseIndex`. This allows us to include the cost (time & memory usage) of creating and closing the index yet correctly amortize that total cost across all added documents.

So that was pretty simple, right? From this you could probably poke around and make your own algorithms. But to really shine, you'll need to know the full list of settings and tasks that are available.

D.2 Parts of an algorithm file

Let's dig into the different parts of an algorithm file. This file is a simple text file, where comments begin with the "#" character, and whitespace is generally not significant. Usually the settings, which bind global names to their values, appear at the top. Next comes the heart of the algorithm, which expresses the series of tasks to run, and in what order. Finally, there is usually one or more reporting tasks at the very end to generate the final summary. Let's look first at the settings.

Settings are lines that match this form:

```
name = value
```

where name is a known setting (full list is below). For example `compound=false` will open the `IndexWriter` with `setUseCompoundFile` set to `false`. Often you want to run a series of rounds where each round uses different combination of settings. One example would be to measure the performance impact of changing RAM buffer sizes during index. You can do this like so:

```
name = header:value1:value2:value3
```

For example `ram.flush.mb = MB:2:4:8:16` would use a 2.0, 4.0, 8.0 and 16.0 MB ram buffer size in each round of the test, and label the corresponding column in the report as "MB". Table D.1 shows the general settings; table D.2 shows settings that affect logging, and table D.3 shows settings that affect `IndexWriter`. Be sure to consult the online documentation for an up to date list. Also, your own tasks can define their own settings!

Table D.1 General settings

Name	Default value	Description

work.dir	System property "benchmark.work.dir" or "work".	Root directory for data and indexes.
analyzer	not set	Class name to instantiate as the analyzer for indexing and parsing queries.
doc.maker.forever	true	Boolean. If true, the doc.maker will reset itself upon running out of documents and just keep producing documents forever. Otherwise, it will stop when it has made one pass through its documents.
html.parser	not set	Class name to filter HTML to text. Default is null (no HTML parsing is invoked). You can use `org.apache.lucene.benchmark.byTask.feeds.DemoHTMLParser` to use a the simple HTML parser included in Lucene's demo package.
doc.stored	false	Boolean. If true, all fields added to the document by the builtin document source are created with Field.Store.YES.
doc.tokenized	true	Boolean. If true, all fields added to the document by the builtin document source are created with Field.Index.ANALYZED.
doc.term.vector	false	Boolean. If true then fields are indexed with term vectors enabled.
doc.term.vector.positions	false	Boolean. If true then term vectors positions are indexed.
doc.term.vector.offsets	false	Boolean. If true then term vector offsets are indexed.
doc.store.body.bytes	false	Boolean. If true, the builtin document sources will additionally store the body as UTF-8 encoded bytes.
docs.dir	not set	String directory name. Used by certain document sources as the root directory for finding document files in the filesystem.
docs.file	not set	String file name. Used by `LineDocMaker` and `WriteLineFile` as the file for single line documents.
docs.reuse.field	true	Boolean. Used only by LineDocMaker. If true then a single shared instance of Document, and a single shared instance of Field for each field in the document, is re-used. This gains performance by avoiding allocation and GC costs, however if you create a custom tasks that adds documents to an index using private threads you will need to turn this off. The normal parallel task sequence, which also uses threads, may leave this at true because the single instance is per thread.
query.maker	not set	String class name for the source of queries. See Query makers section below for details.
file.query.maker.file	not set	String path to file name used by FileBasedQueryMaker. This is the file that contains one text query per line.
file.query.maker.default.field	body	The field that FileBasedQueryMaker will issue is queries against.
doc.delete.step	8	When deleting documents in steps, this is the step that's added in between deletions. See the DeleteDoc task for more detail.

Table D.2 Settings that affect logging.

Name	Default value	Description
doc.add.log.step	500	Integer. How often to print the progress line, as measured by number of docs created.

doc.delete.log.step	500	Integer. How often to print the progress line, as measured by number of docs deleted.
log.queries	false	Boolean. If true, the queries returned by the query maker are printed.
task.max.depth.log	0	Integer. Controls which nested tasks should do any logging. Set this to a lower number to limit how many tasks log.
doc.tokenize.log.step	500	Integer. How often to print the progress line, as measured in number of docs analyzed. This is used only with the ReadTokens task.

Table D.3 Settings that affect IndexWriter

Setting	Default	Description
compound	true	Boolean. True if compound file format should be used
merge.factor	10	Merge factor
max.buffered	-1 (don't flush by doc count)	Max buffered docs
max.field.length	10000	Max field length
directory	RAMDirectory	Directory
ram.flush.mb	16.0	RAM buffer size
autocommit	true	Auto commit
merge.scheduler	org.apache.lucene.index.ConcurrentMergeScheduler	Merge scheduler
merge.policy	org.apache.lucene.index.LogByteSizeMergePolicy	Merge policy

D.2.1 Document Maker

The setting `doc.maker` defines the class to use for generating documents, for algorithms that consume documents with the `AddDoc` task. This is a string class name, and must be a class that subclasses the `benchmark.byTask.feeds.DocMaker` class. This class is instantiated once, globally, and then all tasks will pull documents from this source. Table D.2 describes the builtin `DocMaker` classes.

You can also create your own document source by subclassing `DocMaker`. However, take care to make your class thread safe since multiple threads will share a single instance of your `DocMaker`.

Table D.4 Built-in document sources

Name	Description
DirDocMaker	Recursively walks a root directory (specified as docs.dir setting), opening any file ending with extension .txt and yields the

	contents as a document. The first line of each file should contain the date; the second line should contain the title and the rest of the document is the body.
LineDocMaker	Opens a single file, specified as the setting docs.file, and reads one document per line. Generally this DocMaker has far less overhead in creating documents than the others since it minimizes the IO cost by only working with a single file. Unlike other DocMakers, this DocMaker does not accept a parameter specifying the max size of the body field. Instead, you should pre-create a line file, using the `WriteLineDoc` task, with the target document size.
EnWikiDocMaker	Generates documents directly from the large XML export provided by http://wikipedia.org. The setting `keep.image.only.docs`, a Boolean setting that defaults to true, decides whether image-only (no text) documents are kept. Use `docs.file` to specify the XML file.
ReutersDocMaker	Generates documents unpacked from the Reuters corpus. The ant task "get-files" retrieves and unpacks the Reuters corpus. Documents are created as *.txt files under the output directory `work/reuters-out`. The setting docs.dir, defaulting to `work/reuters-out`, specifies the root location of the unpacked corpus.
TrecDocMaker	Generates documents from the Trec corpus. This assumes you have already unpacked the Trec into the directory set by docs.dir.
SimpleDocMaker	Trivial doc maker to be used only for testing. This generates a single document with a small fixed English text.

D.2.2 Query maker

The `query.maker` setting determines which class to use for generating queries. Table D.5 describes the built-in query makers.

Table D.5 Builtin query makers

FileBasedQueryMaker	Reads queries from a text file one per. Set file.query.maker.default.field (defaults to "body") to specify which index field the parsed queries should be issued against. Set file.query.maker.file to specify the file containing the queries.

ReutersQueryMaker	Generates a small fixed set of 10 queries that roughly match the Reuters corpus.
SimpleQueryMaker	Used only for testing. Generates a fixed set of 10 synthetic queries.
SimpleSloppyPhraseQueryMaker	Takes the fixed document text from SimpleDocMaker and programmatically generates a number of queries with varying degrees of slop (from 0 to 7) that would match the single document from SimpleDocMaker.

D.3 Control structures

We've finished talking about settings. Now we'll talk about the available control structures in an algorithm, which is really the "glue" that allows you to take builtin tasks and combine them in interesting ways. Here are the building blocks:

- Serial sequences are created with `{ ... }`. The enclosed tasks are run one after another, by a single thread. For example:

```
{CreateIndex AddDoc CloseIndex}
```

  creates a new index, adds a single document pulled from the doc maker, and then closes the index.

- Parallel sequences are created with `[ ... ]`. A parallel sequence runs the enclosed tasks with as many threads as there are tasks, with each task running in its own thread. For example:

```
[AddDoc AddDoc AddDoc AddDoc]
```

  will create 4 threads, each of which adds a single document, and then stops.

- Repeating a task multiple times is achieved by appending `:N` to the end. For example:

```
{AddDoc}: 1000
```

  adds the next 1000 documents from the document source. Use `*` to pull all documents from the doc maker. For example:

```
{AddDoc}: *
```

  adds all documents from the doc maker. When you use this, you must also set doc.maker.forever to false.

- Repeating a task for a specified amount of time is achieved by appending `:Xs` to the end. For example:

```
{AddDoc}: 10.0s
```

  Runs the `AddDoc` task for 10 seconds.

- Name a sequence like this:

```
{"My Name" AddDoc} : 1000
```

  This defines a single AddDoc call "My Name", and then runs that task 1000 times. The double-quotes surrounding the name are required. Your name will then be used in the reports.

- Command parameters: some tasks optionally take a single parameter. If you try to pass a parameter to a task that does not take one, or the type is not correct, you'll hit a "Cannot understand algorithm" error. The following tasks take parameters:

AddDoc takes a numeric parameter indicing the size of the added document, in characters. The body of each document from the docmaker will be truncated to this size, with the leftover being prepended to the next document. This requires that the doc maker supports changing the document size.

DeleteDocs takes a numeric parameter indicating the document number to be deleted.

SetProp takes a name,value, and changed the named property to the specified value.

SearchTravRetTAsk and SearchTravTask take a numeric parameter which is the required traversal size

SearchTravRetLoadFieldSelectoorTask takes a sring parameter containing a comma separated list of fields to load

- Turning off statistics of child tasks the > character, instead of } or]. This is useful for avoiding the overhead of gathering statistics when you don't require that level of detail. For example:

```
{ "ManyAdds" AddDoc > : 10000
```

  adds 10000 docs and will not individually track statistics of each AddDoc call (but the 10000 added docs is tracked by the outer sequence containing "ManyAdds").

- Rate limiting: in addition to specifying how many times a task or task sequence should be repeated, you can also specify the target rate in count per second (default) or count per minute. Do this by adding : N : R after the task. For example:

```
{ AddDoc } : 10000 : 100/sec
```

  adds 10000 documents at a rate of 100 documents per second. Or:

```
[ AddDoc ]:  10000: 3
```

  will add 10000 docs in parallel, spawning one thread for each added document, at a rate of 3 new threads

per second.

- Disable counting for a task: each tasks contributes to the records count that is used for reporting at the end. For example, AddDoc returns 1. Most tasks return count 1, some return count 0 and some return count greater than 1. Sometimes you do not want to include the count of a task in your final report. To do that, simply prepend "-" into your task. For example:

```
{"BuildIndex"
  -CreateIndex
  {AddDoc}:10000
  -CloseIndex
}
```

The report would record exactly 10000 records, but if you left the - out it would report 10002.

D.4 Builtin tasks

We've discussed the settings and the control structures, or glue, that allows you to combine tasks into larger sequences of tasks. Now, finally, let's review the builtin tasks. Table D.6 describes the builtin administration tasks, and Table D.7 describes the tasks for indexing and searching

If the commands available for use in the algorithm do not meet your needs, you can add commands by adding a new task under `org.apache.lucene.benchmark.byTask.tasks` - you should extend the `PerfTask` abstract class. Make sure that your new task class name is suffixed by `Task`. For example, once you compile the class `SliceBreadTask.java`, and ensure it's on the classpath that you specify to ant, then you can invoke this task by using `SliceBread` in your algorithm.

Table D.6 Administration tasks

ClearStats	Clears all statistics. Report tasks run after this point will only include statistics from tasks run after this task.
NewRound	Begin a new round of a test. This command makes most sense at the end of an outermost sequence. This increments a global "round counter". All tasks that start will record this new round count and their statistics would be aggregated under that new round count. For example, see the RepSumByNameRound reporting task. In addition, NewRound moves to the next setting if the setting specified different settings per round. For example, with setting merge.factor=mrg:10:100:10:100, merge.factor would change to the next value after each round. Note that if you have more rounds than number of settings, it simply wraps around to the first setting again.
ResetInputs	Re-initializes the document and query sources back to the start. For example, it's a good idea to insert this call after NewRound to make sure your document source feeds the exact same documents for each round. This is only necessary when you are not running your document source to exhaustion.
ResetSystemErase	Reset all index and input data, and call System.gc(). This does not reset statistics. This also calls ResetInputs. All writers and readers are closed, nulled and deleted. The index and directory are erased. You must call CreateIndex to create a new index after this call, if you intend to add documents to an index.

ResetSystemSoft	Just like ResetSystemErase, except the index and Directory are not erased. This is useful for testing performance of opening an existing index for searching or updating. You can use the OpenIndex task after this reset.

Table D.7 Builtin tasks for indexing and searching.

CreateIndex	Create a new index with IndexWriter. You can then use the AddDoc task to add documents to the index.
OpenIndex	Open an existing index with IndexWriter. You can then use the AddDoc task to add documents to the index.
OptimizeIndex	Optimize the index. This task optionally takes an integer parameter, which is the maximum number of segments to optimize. This calls the IndexWriter.optimize(int maxNumSegments) method. If there is no parameter, it defaults to 1.
CloseIndex	Close the open index. This task takes an optional Boolean parameter specifying whether Lucene should wait for running merges to complete. The default is true.
OpenReader	Create an IndexReader and IndexSearcher, available for the search tasks. If a Read task is invoked, it will use the currently open reader. If no reader is open, it will open its own reader, perform its task, and then close the reader. This enables testing of various scenarios: sharing a reader, searching with a "cold" reader, with a "warm" reader, etc. The read tasks affected by this are: Warm, Search, SearchTrav (search and traverse) and SearchTravRet (search, traverse and retrieve). Note that each of the 3 search tasks maintains its own queryMaker instance.
CloseReader	closes the previously opened reader.
NewAnalyzer	switches to a new analyzer. This task takes a single parameter, which is a comma separated list of class names. Each class name can be shorted to just the class name, if it falls under org.apache.lucene.analysis package; otherwise it must be fully qualified. Each time this task is executed, it will switch to the next analyzer in its list, rotating back to the start if it his the end.
Search	search an index. If the reader is already opened (with the OpenReader task), it's searched. Otherwise a new reader is opened, searched, and hten closed. This task simply issues the search but does not traverse the results.
SearchTrav	search an index and traverse the results. Like Search, except the Hits are visited. This task takes an optional integer parameter, which is the number of Hits to visit. If no parameter is specified, the full result set is visited. This task returns as its count the number of documents visited.
SearchTravRet	search an index and traverse and retrieve the results. Like SearchTrav, except for each hit visited the corresponding document is also retrieved from the index.

SearchTravRetLoadFieldSelector	search an index and traverse and retrieve only specific fields in the results, using FieldSelector. Like SearchTrav, excep this task takes an optional comma-separated string parameter specifying which fields of the document should be retrieved.
SearchTravRetHighlight	search an index, and traverse and retrieve and highlight certain fields from the results. This task taskes a comma-seperated parameter list to control details of the highlighting. Please consult its javadocs for the details.
SetProp	change an algorithm setting. Settings are normally globally set in your algorithm file, but this task can be used to change a setting at a specific point. All tasks executed after this point will use the new setting.
Warm	warms up a previously opened searcher by retrieving all documents in the index. Note that in a real application, this is not sufficient as you would also want to pre-populate the FieldCache if you are using it, and possibly issue initial searches for commonly searched for terms.
DeleteDoc	delete a document by document ID, or by incrementing step size to compute the document ID to be deleted. This takes takes an integer parameter. If the parameter is negative, deletions are done by the doc.delete.step setting. For example, if the step size is 10, then each time this task is executed it will delete the document IDs in the sequence 0, 10, 20, 30, etc. If the parameter is non-negative then this is a fixed document ID to delete.
ReadTokens	this tasks tests the performance of just the analyzer's tokenizer. It simply reads the next document from the document maker and fully tokenizes all of its fields. As the count this task returns the number of tokens encountered. This is a useful task to measure cost of document retrieval and tokenization. By subtracting this cost from the time spent building an index you can get a rough measure of what the actual indexing cost is.
WriteLineDocTask	used to create a line file that can then be used by LineDocMaker. See section “Creating and using line files” below.

D.4.1 Creating and using line files

Line files are simple text files that contain one document per line. Indexing documents from a line file incurs quite a bit less overhead than other approaches such as one file per document, or pulling files from a database, etc. This is important if you are trying to measure performance of just the core indexing process. If instead you are trying to measure indexing performance from your specific document source then you should not use a line file!

The benchmark framework provides a simple task, `WriteLineDocTask`, to create line files. Using this you can translate any document source into a line file. However, the one limitation is that each document only has a date, title, and body field. The `line.file.out` setting specifies the file that is created. For example, use this algorithm to translate the Reuter's corpus into a single line file.

```
# Where to get documents from:
doc.maker=org.apache.lucene.benchmark.byTask.feeds.ReutersDocMaker

# Stop after processing the document feed once:
doc.maker.forever=false
```

```
# Where to write the line file output:
line.file.out=work/reuters.lines.txt

# Process all documents, appending each one to the line file:
{WriteLineDoc()}: *
```

Once you've done this you can then use reuters.lines.txt, and `LineDocMaker`, like this:

```
# Feed that knows how to process the line file format:
doc.maker=org.apache.lucene.benchmark.byTask.feeds.LineDocMaker

# File that contains one document per line:
docs.file=work/reuters.lines.txt

# Process documents only once:
doc.maker.forever=false

# Create a new index, index all docs from the line file, close the
# index, produce a report.
CreateIndex
{AddDoc}: *
CloseIndex

RepSumByPref AddDoc
```

D.4.2 Builtin reporting tasks

Report tasks generate a summary report at the end of the algorithm, showing how many records per second were achieved, how much memory was used, one line per task or task sequence that gathered statistics. The Report tasks themselves are not measured and not reported. Table D.8 describes the builtin reporting tasks. If needed, additional reports can be added by extending the abstract class `ReportTask`, and by manipulating the statistics data in Points and `TaskStats`.

Table D.8 Reporting tasks

Task name	Description
RepAll	All (completed) task runs.
RepSumByName name	All statistics, aggregated by name. So, if AddDoc was executed 2000 times, only 1 report line would be created for it, aggregating all those 2000 statistic records.
RepSelectByPref prefix	all records for tasks whose name start with prefix
RepSumByPref prefix	all records for tasks whose name start with prefixWord aggregated by their full task name.
RepSumByNameRound name	all statistics, aggregated by name and by round. So, if AddDoc was executed 2000 times in each of 3 rounds, 3 report lines would be created for it, aggregating all those 2000 statistic records in each round. See more about rounds in the NewRound task description below.
RepSumByPrefRound prefix	similar to RepSumByNameRound, except only tasks whose name starts with prefixWord are included

D.5 Evaluating search quality

How do you test the relevance or quality of your search application? Relevance testing is crucial because, at the end of the day, your users will not be satisfied if they don't get relevant results. Many small changes to how you use Lucene, from the analyzer chain, to which fields you index, to how you build up a Query, to how you customize scoring, can have large impacts on relevance. Being able to properly measure such effects allows you to make changes that improve your relevance.

Yet, despite being the most important aspect of a search application, quality is devilishly difficult to pin down. There are certainly many subjective approaches. You could run a controlled user trial, or you can play with the application yourself. What do you look for? Besides checking if the returned documents are relevant, there are many other things to check: are the excerpts accurate? Is the right metadata presented? Is the UI easily consumed on quick glance? No wonder so few applications are tuned for their relevance!

That said, if you'd like to objectively measure the relevance of returned documents, you're in luck: recent additions to the benchmark framework, under the `quality` package, allow you to do so. These classes provide concrete implementations based on the formats from the TREC corpus, but you can also implement your own. You'll need a "ground truth" transcribed set of queries, where each query lists the documents that are relevant to it. This approach is entirely binary: a given document from the index is either relevant or not. Still, how can we objectively measure relevance?

D.5.1 Precision and recall

Precision and recall are standard metrics in the information retrieval community for objectively measuring relevance of search results. Precision measures what subset of the documents returned for each query were relevant. For example, if a query has 20 hits and only 1 is relevant, precision is 0.05. If only 1 hit was returned and it was relevant, precision is 1.0. Recall measures what percentage of the relevant documents for that query was actually returned. So if the query listed 8 documents as being relevant, but 6 were in the result set, that's a recall of 0.75.

In a properly configured search application, these two measures are naturally at odds with one another. Let's say, on one extreme, you only show the user the very best (top 1) document matching their query. With such an approach, your precision will typically be high, because the first result has a good chance of being relevant, while your recall would be very low, because if there are many relevant documents for a given query you have only returned one of them. If we increase top 1 to top 10, then suddenly we will be returning many documents for each query. The precision will necessarily drop because most likely you are now allowing some non-relevant documents into the result set. But recall should increase because each query should return a larger subset of its relevant documents.

Still, you'd like the relevant documents to be higher up in the ranking. To measure this, average precision is computed. This measure computes precision at each of the N cutoffs, where N ranges from 1 to a maximum value, and then takes the average. So this measure is higher if your search application generally returns relevant documents earlier in the result set. Mean average precision, or MAP, then measures the mean of average precision across a set of queries. A related measure, mean reciprocal rank or MRR, measures 1/M where M is the first rank that had a relevant document. You want both of these numbers to be as high as possible!

Listing D.1 Example code to compute precision and recall statistics for your IndexSearcher

```
/* This code was extracted from the Lucene contrib/benchmark sources */

public class PrecisionRecall {

  public static void main(String[] args) throws Throwable {

    File topicsFile = new File("src/lia/benchmark/topics.txt");
    File qrelsFile = new File("src/lia/benchmark/qrels.txt");
    Searcher searcher = new IndexSearcher("indexes/MeetLucene");

    String docNameField = "filename";

    PrintWriter logger = new PrintWriter(System.out, true);
```

```
    TrecTopicsReader qReader = new TrecTopicsReader();   //#1
    QualityQuery qqs[] = qReader.readQueries(            //#1
        new BufferedReader(new FileReader(topicsFile))); //#1

    Judge judge = new TrecJudge(new BufferedReader(      //#2
        new FileReader(qrelsFile)));                     //#2

    judge.validateData(qqs, logger);                     //#3

    QualityQueryParser qqParser = new SimpleQQParser("title", "contents");  //#4

    QualityBenchmark qrun = new QualityBenchmark(qqs, qqParser, searcher, docNameField);
    SubmissionReport submitLog = null;
    QualityStats stats[] = qrun.execute(judge,           //#5
            submitLog, logger);

    QualityStats avg = QualityStats.average(stats);      //#6
    avg.log("SUMMARY",2,logger, "  ");
  }
}
```

#1 Read TREC topics as QualityQuery[]
#2 Create Judge from TREC Qrel file
#3 Verify query and Judge match
#4 Create parser to translate queries into Lucene queries
#5 Run benchmark
#6 Print precision and recall measure

Listing D.1 shows how to use the `quality` package to compute precision and recall. Currently, in order to measure search quality, you must write your own Java code (ie, there are no builtin tasks for doing so, that would allow you to solely use an algorithm file). The queries to be tested are represented as an array of `QualityQuery` instances. The `TrecTopicsReader` knows how to read the TREC topic format, into `QualityQuery` instances, but you could also implement your own. Next, the ground truth is represented with the simple `Judge` interface. The `TrecJudge` class loads TREC's Qrel format and implements `Judge`. `QualityQueryParser` translates each `QualityQuery` into a real Lucene query. Finally, `QualityBenchmark` tests the queries by running them against a provided `IndexSearcher`. It returns and array of `QualityStats`, one each for each of the queries. The `QualityStats.average` method computes and reports precision and recall.

When you run the code in Listing D.1, by entering `ant PrecisionRecall` at the command line within the book's source code directory, it will produce something like this:

```
SUMMARY
  Search Seconds:         0.015
  DocName Seconds:        0.006
  Num Points:            15.000
  Num Good Points:        3.000
  Max Good Points:        3.000
  Average Precision:      1.000
  MRR:                    1.000
  Recall:                 1.000
  Precision At 1:         1.000
  Precision At 2:         1.000
  Precision At 3:         1.000
  Precision At 4:         0.750
  Precision At 5:         0.600
  Precision At 6:         0.500
  Precision At 7:         0.429
  Precision At 8:         0.375
  Precision At 9:         0.333
  Precision At 10:        0.300
  Precision At 11:        0.273
  Precision At 12:        0.250
  Precision At 13:        0.231
  Precision At 14:        0.214
```

Note that this test uses the MeetLucene index, so you'll need to run `ant MeetLucene` if you skipped over that in chapter 1. This was a trivial test, since we ran on a single query that has exactly three correct documents (see the source files `src/lia/benchmark/topics.txt` for the queries and `src/lia/benchmark/qrels.txt` for the correct documents). You can see that the precision was perfect (1.0) for the top 3 results, meaning the top 3 results were in fact the correct answer to this query. Precision then gets worse beyond the top 3 results because any further document is incorrect. Recall is perfect (1.0) because all three correct documents were returned. In a real test you won't see perfect scores!

For ideas on how to improve relevance, have a look at this recently added page on the Lucene wiki:

http://wiki.apache.org/lucene-java/TREC_2007_Million_Queries_Track_-_IBM_Haifa_Team

And here's the full paper describing the approach:
http://elvis.slis.indiana.edu/irpub/TREC/TREC2007_NOTEBOOK/NOTEBOOK.PAPERS/ibm_haifa_mq.pdf

D.5 Errors

If you make a mistake in writing your algorithm, which is in fact very easy to do, you'll see a somewhat cryptic exception like this:

```
java.lang.Exception: Error: cannot understand algorithm!
      at org.apache.lucene.benchmark.byTask.Benchmark.<init>(Benchmark.java:63)
      at org.apache.lucene.benchmark.byTask.Benchmark.main(Benchmark.java:98)
Caused by: java.lang.Exception: colon unexpexted: - Token[':'], line 6
      at org.apache.lucene.benchmark.byTask.utils.Algorithm.<init>(Algorithm.java:120)
      at org.apache.lucene.benchmark.byTask.Benchmark.<init>(Benchmark.java:61)
```

When this happens, simply scrutinize your algorithm. One common error is a mis-balanced { or }. Try iteratively simplifying your algorithm to a smaller part and run that, to isolate the error.

D.6 Summary

As we've seen, the benchmark package is a powerful framework for quickly creating performance tests and for evaluating your search application for precision and recall. It saves you tons of time because all of the normal overhead in creating a performance test is handled for you. Combine this with the large library of built-in tasks for common indexing and searching operations, plus extensibility to add your own report, task or a document or query source, and you've got one very useful tool under your belt.