Community Experience Distilled

Apache Solr Essentials

Leverage the power of Apache Solr to create efficient search applications

Andrea Gazzarini

[PACKT] open source* PUBLISHING
community experience distilled

Apache Solr Essentials

Leverage the power of Apache Solr to create efficient search applications

Andrea Gazzarini

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Apache Solr Essentials

First published: February 2015

Production reference: 1210215

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-964-1

www.packtpub.com

Credits

Author
Andrea Gazzarini

Reviewers
Ahmad Maher Abdelwhab
Markus Klose
Julian Lam
Puneet Singh Ludu

Commissioning Editor
Usha Iyer

Acquisition Editor
Larissa Pinto

Content Development Editor
Kirti Patil

Technical Editor
Ankur Ghiye

Copy Editor
Vikrant Phadke

Project Coordinator
Nidhi J. Joshi

Proofreaders
Stephen Copestake
Maria Gould
Bernadette Watkins

Indexer
Priya Sane

Graphics
Abhinash Sahu

Production Coordinator
Shantanu N. Zagade

Cover Work
Shantanu N. Zagade

About the Author

Andrea Gazzarini is a software engineer. He has mainly focused on the Java technology. Although often involved in analysis and design, he strongly loves coding and definitely likes to be considered a developer.

Andrea has more than 15 years of experience in various software branches, from telecom to banking software. He has worked for several medium- and large-scale companies, such as IBM and Orga Systems.

Andrea has several certifications in the Java programming language (programmer, developer, web component developer, business component developer, and JEE architect), BEA products (build and portal solutions), and Apache Solr (Lucid Apache Solr/Lucene Certified Developer).

In 2009, Andrea stepped into the wonderful world of open source projects, and in the same year, he became a committer for the Apache Qpid project. His adventure with Solr began in 2010, when he joined @Cult, an Italian company that mainly focuses its projects on library management systems, online access public catalogs, and linked data.

He's currently involved in several (too many!) projects, always thinking about a "big" idea that will change his (developer) life.

Acknowledgments

I'd like to begin by thanking the people who made this book what it is. Writing a book is not a single person's work, and help from experienced people that guide you along the path is crucial. Many thanks to Larissa, Kirti, Ankur, and Vikrant for supporting me in this process.

I am also grateful to the technical reviewers of the book, Ahmad Maher Abdelwhab, Markus Klose, Puneet Singh Ludu, and Julian Lam, for carefully reading my drafts and spotting (hopefully) most of my mistakes. This book would not have been so good without their help and input.

In general, I want to thank everyone who directly or indirectly helped me in creating this book, except for a long-sighted teacher who once told me when I was in university, "Hey, guy with all those earrings! You won't go anywhere!"

Finally, a special thought to my family; to my girls, the actual supporters of the book; my wonderful wife, Nicoletta (to whom I promise not to write another book), my pride and joy, Sofia and Caterina, and my first actual teacher—my mom, Lina. They are the people who really made sacrifices while I was writing and who definitely deserve the credits for the book.

Once again, thank you!

About the Reviewers

Ahmad Maher Abdelwhab is currently working at Knowledgeware Technologies as an open source developer. He has over 10 years of experience, with special development skills in PHP, Drupal , Perl, Ruby On Rails , Java, XML, XSL, MySQL, PostgreSQL, MongoDB, SQL, and Linux. He graduated in computer science from Mansoura University in 2005.

I would like to thank my father, mother, and sincere wife for their continuous support while reviewing this book.

Markus Klose is a search and big data consultant at SHI GmbH & Co.KG in Germany. He is in charge of project management and supervision, project analysis, and delivering consulting and training services.

Most of Markus' daily business is related to Apache Solr, Elasticsearch, and Fast ESP. He travels across Germany, Switzerland, and Austria to provide his services and knowledge.

On a regular basis, you can find him at meets, user groups, or conferences such as Berlin Buzzword oder Solr Revolution, where he speaks about Apache Solr.

Besides search-related training and consulting, he is currently establishing additional areas of work. He uses tools such as Logstash and Kibana to fulfill customer requirements in monitoring and analytics.

Thanks to the experience gained from his daily work, Markus wrote the first German book on Apache Solr (*Einführung in Apache Solr*) with his colleague, Daniel Wrigley. It was published by O'Reilly in February 2014.

Besides writing, Markus spends a lot of his free time using his knowledge and programming skills to work on and contribute to open source projects such as Latin stemmer and number converter for Solr (`https://issues.apache.org/jira/browse/LUCENE-4229`) and SolrAppender for log4j 2 (`https://issues.apache.org/jira/browse/LOG4J2-618`) .

Julian Lam is a cofounder and core maintainer of NodeBB, a type of free and open source forum software built upon modern web tools, such as Node.js and Redis. He has spoken several times on topics related to Javascript in the workplace and best practices for hiring. Julian is an advocate of client-side rendering, which can be used to build highly performant web applications.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Hi Dad, when you bought me my first computer, you had no idea what was coming next...

Table of Contents

Preface

As you may have guessed from the title, this is a book about Apache Solr—specifically about Solr essentials. What do I mean by essentials? Nice question! Such a term can be seen from so many perspectives. Solr, mainly from 2010 onwards, witnessed exponential growth in terms of popularity, stakeholders, community, and the capabilities it offers. This rapid growth reflects the rich portfolio of the things that have been developed in these years and are nowadays available. So, strictly speaking, it's not so easy to define the "essentials" of Solr.

The perspective that I will use to explain the term "essentials" is quite simple and pragmatic. I will describe the building blocks of Apache Solr, and at the same time, I will try to put my personal experience on those topics. In recent years, I've worked with Solr in several projects. As a user, I had to learn how to install, configure, tune, troubleshoot, and monitor Solr. As a developer, things were different for me. If you're working in the IT domain and you're reading this book (I guess you are), you probably know that each time you try to implement a solution, there's something in the project that a specific tool doesn't cover. So, after spending a lot of time analyzing, reading documentation, searching on the Internet, reading Wikis, and so on, you realize that you need to add a custom piece of code somewhere. That's because "the product covers the 99.9999 percent of the possible scenarios but..." For this specific case, if this happens or that happens, you always fall under that 0.0001 percent. I don't know about you, but for me, this has always been so. No matter what the project, the company, or the team is, this has been an implicit constant of every project, always.

That's the reason I will try as much as possible to explain things throughout the book using real-world examples directly coming from my personal experience. I hope this additional perspective will be useful for better understanding of what is considered the most popular open source search platform.

What this book covers

Chapter 1, *Get Me Up and Running*, introduces the basic concepts of Solr and it provides you with all the necessary steps to quickly get it up and running.

Chapter 2, *Indexing Your Data*, begins our first detailed discussion on Solr. In this chapter, we look at the data indexing process and see how it can be configured, tuned, and customized. This is also where we encounter the first line of code.

Chapter 3, *Searching Your Data*, explores the other specular side of Solr. First, we stored our data; now we explore all that Solr offers in terms of search services.

Chapter 4, *Client API*, covers client-side usage of Solr libraries, providing a description of the main use cases from a client's perspective.

Chapter 5, *Administering and Tuning Solr*, takes you through the available tools for configuring, managing, and tuning Solr.

Chapter 6, *Deployment Scenarios*, illustrates the various ways in which you can deploy Solr, from a standalone instance to a distributed cluster.

Chapter 7, *Solr Extensions*, describes several available Solr extensions and how they can be useful in solving common concrete use cases.

Chapter 8, *Contributing to Solr*, explains the wonderful world of open source software by illustrating the compounding pieces of the process of participation and contribution.

What you need for this book

In order to be able to run the code examples in the book, you will need the Java Development Kit (JDK) 1.7 and Apache Maven.

Alternatively, you will need an Integrated Development Environment (IDE). Eclipse is strongly recommended as it is the same environment I used to capture the screenshots. However, even if you want to use another IDE, the steps should be quite similar.

The difference between the two alternatives mainly resides in the role that you want to assume during the reading. While you may want to only start and execute the examples as a user, you would surely want to see the working code in a usable environment as a developer. That's the reason an IDE is strongly recommended in the second case.

The first chapter will provide the instructions necessary for installing all that you'll need through the book.

Who this book is for

This book is targeted at people—users and developers—who are new to Apache Solr or are experienced with a similar product. The book will gradually help you to understand the focal concepts of Solr with the help of practical tips and real-world use cases. Although all the examples associated with the book can be executed with a few simple commands, a familiarity with the Java programming language is required for a good understanding.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and explanations of their meanings.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Each folder has a subfolder called `conf` where the configuration for that specific core resides."

A block of code is set as follows:

```
{
  { "id": 1, "title":"The Birthday Concert" },
  { "id": 2, "title":"Live in Italy" },
  { "id": 3, "title":"Live in Paderborn" },
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.StopFilterFactory" words="stopwords.txt"
  ignoreCase="true"/>
```

Any command-line input or output is written as follows:

```
# mvn cargo:run -P fieldAnalysis
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Choose a field type or a field. Then press the **Analyse Values** button."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

Alternatively, you can also download the examples from **GitHub**, on `https://github.com/agazzarini/apache-solr-essentials`. There, you can download the whole content as a zip file from `https://github.com/agazzarini/apache-solr-essentials/archive/master.zip` or, if you have `git` installed on your machine, you can clone the repository by issuing the following command:

```
# git clone https://github.com/agazzarini/apache-solr-essentials.git
<path-to-your-work-dir>
```

Where, `<path-to-your-work-dir>` is the destination folder where the project will be cloned.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Get Me Up and Running

This chapter describes how to install Solr and focuses on all the required steps to get a complete study and development environment that will guide us through the book.

Specifically, according to the double perspective previously described, I will illustrate two kinds of installations. The first is the installation of a standalone Solr instance (this is very quick). This is a simple task because the download bundle is preconfigured with all that you need to get your first taste of the product. As a developer, the second perspective is what I really need every day in my ordinary job—a working integrated development environment where I can run and debug Solr with my configurations and customizations, without having to manage an external server. In general, such an environment will have all that I need in one place for developing, debugging, and running unit and integration tests.

By the end of the chapter, you will have a running Solr instance on your machine, a ready-to-use **Integrated Development Environment** (**IDE**), and a good understanding of some basic concepts.

This chapter will cover the following topics:

- Installation of a simple, standalone Solr instance from scratch
- Setting up of an Integrated Development Environment
- A quick overview about what we installed
- Troubleshooting

Installing a standalone Solr instance

Solr is available for download as an archive that, once uncompressed, contains a fully working instance within a Jetty servlet engine. So the steps here should be pretty easy.

Prerequisites

In this section, we will describe a couple of prerequisites for the machine where Solr needs to be installed.

First of all, Java 6 or 7 is required: the exact choice depends on which version of Solr you want to install. In general, regardless of the version, make sure you have the latest update of your **Java Virtual Machine** (**JVM**). The following table describes the association between the latest Solr and Java versions:

Solr version	Java version
4.7.x	Java 6 or greater
4.8.x	Java 7 (update 55) or greater; Java 8 is verified to be compatible
4.9.x	Java 7 (update 55) or greater; Java 8 is verified to be compatible
4.10.x	Java 7 (update 55) or greater

Java can be downloaded from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

Other factors such as CPU, RAM, and disk space strongly depend on what you are going to do with this Solr installation. Nowadays, it shouldn't be hard to have a couple of GB available on your workstation. However, bear in mind that at this moment I'm playing on Solr 4.9.0 installed on a Raspberry PI (its RAM is 512 MB). I gave Solr a maximum heap (-Xmx) of 256 MB, indexed about 500 documents, and executed some queries without any problem. But again, those factors really depend on what you want to do: we could say that, assuming you're using a modern PC for a study instance, hardware resources shouldn't be a problem.

Instead, if you are planning a Solr installation in a test or in a production environment, you can find a useful spreadsheet at `https://svn.apache.org/repos/asf/lucene/dev/trunk/dev-tools/size-estimator-lucene-solr.xls`.

Although it cannot encompass all the peculiarities of your environment, it is definitely a good starting point for RAM and disk space estimation.

Downloading the right version

The latest version of Solr at the time of writing is 4.10.3, but a lot of things we will discuss in the book are valid for previous versions as well.

You might already have Solr somewhere and might not want to redownload another instance, your customer might already have a previous version, or, in general, you might not want the latest version. Therefore, I will try to refer to several versions in the book—from 4.7.x to 4.10.x—as often as possible. Each time a feature is described, I will indicate the version where it appeared first.

The download bundle is usually available as a tgz or zip archive. You can find that at `https://lucene.apache.org/solr/downloads.html`.

Setting up and running the server

Once the Solr bundle has been downloaded, extract it in a folder. We will refer to that folder as `$INSTALL_DIR`. Type the following command to extract the Solr bundle:

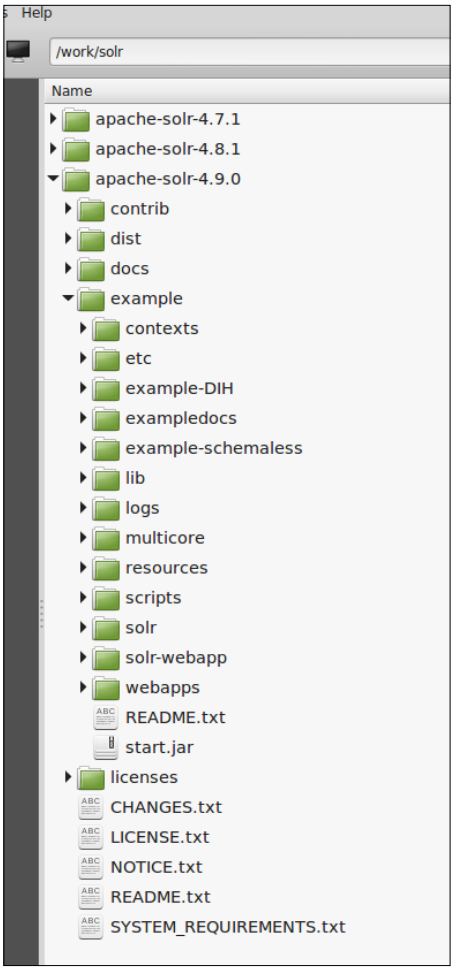
```
# tar -xvf $DOWNLOAD_DIR/solr-x.y.z.tar.gz -C $INSTALL_DIR
```

or

```
# unzip $DOWNLOAD_DIR/solr-x.y.z.zip -d $INSTALL_DIR
```

depending on the format of the bundle.

At the end, you will find a new `solr-x.y.z` folder in your `$INSTALL_DIR` folder. This folder will act as a container for all Solr instances you may want to play with. Here is a screenshot of the `solr-x.y.z` folder on my machine, where you can see I have three Solr versions:



The `solr-x.y.z` directory contains Jetty, a fast and small servlet engine, with Solr already deployed inside. So, in order to start Solr, we need to start Jetty. Open a new shell and type the following commands:

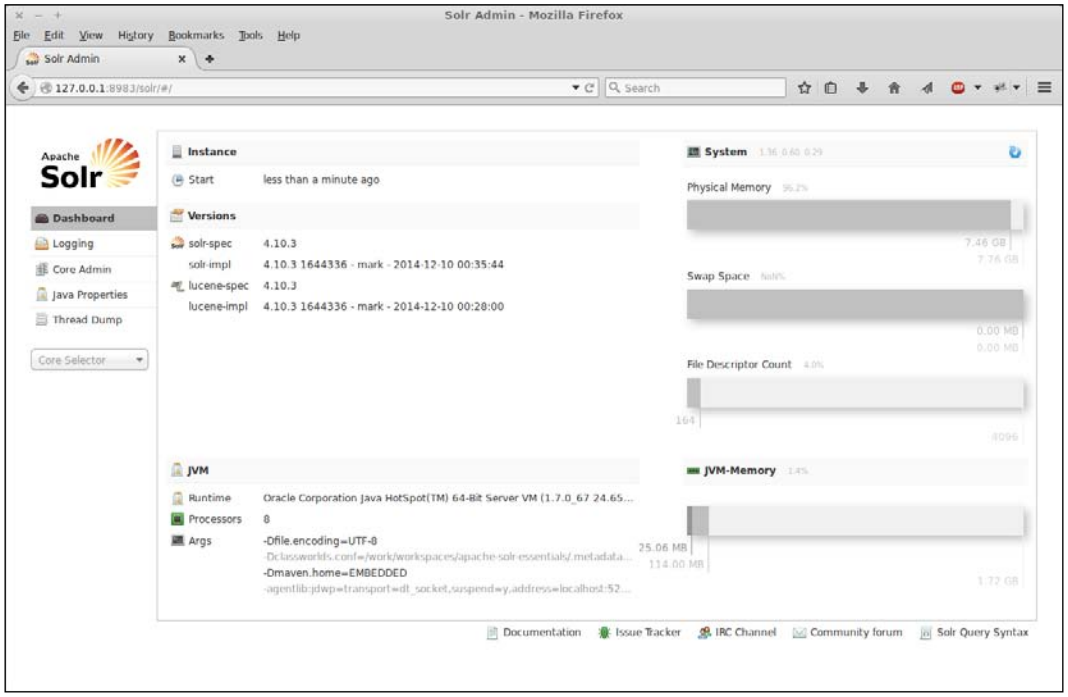
```
# cd $INSTALL_DIR/solr-x.y.z/example
# java -jar start.jar
```

You should see a lot of log messages ending with something like this:

```
...
[INFO]  org.eclipse.jetty.server.AbstractConnector  - Started
SocketConnector@0.0.0.0:8983
...
[INFO] org.apache.solr.core.SolrCore  - [collection1] Registered new
searcher Searcher@66b664d7[collection1] main{StandardDirectoryReader(segm
ents_2:3:nrt _0(4.9):C32)}
```

These messages tell you Solr is up-and-running! Open a web browser and type `http://127.0.0.1:8983/solr`.

You should see the following page:



This is the Solr administration console.

Setting up a Solr development environment

This section will guide you through the necessary steps to have a working development environment that allows you to have a place to write and execute your code or configurations against a running and debuggable Solr instance.

If you aren't interested in such a perspective because, for instance, your usage scenario falls within the previous section, you can safely skip this and proceed with the next section.

The source code included with this book contains a ready-to-use project for this section. I will later explain how to get it into your workspace in one shot.

Prerequisites

The development workstation needs to have some software. As you can see, I kept the list small and minimal.

Firstly, you need the **Java Development Kit** 7 (**JDK**), of which I recommend the latest update, although the older version of Solr covered by this book (4.7.x) is able to run with Java 6. Java 7 is supported from 4.7.x to 4.10.x, so it is definitely a recommended choice.

Lastly, we need an IDE. Specifically, I will use Eclipse to illustrate and describe the developer perspective, so you should download a recent JSE version (that is, Eclipse IDE for Java Developers) from `https://www.eclipse.org/downloads`.

Do not download the EE version of Eclipse because it contains a lot of things we don't need in this book.

Starting from Eclipse Juno, all the required plugins are already included. However, if you love an older version of Eclipse (such as Indigo) like I do, then Maven integration for Eclipse—also known as **M2Eclipse** (**M2E**)—needs to be installed. You can find this in the Eclipse marketplace (go to **Help** | **Eclipse Marketplace**, then search for `m2e`, and click on the **Install** button).

Importing the sample project of this chapter

It's time to see some code, in order to touch things with your hands. We will guide you through the necessary steps to have your Eclipse configured with a sample project, where you will be able to start, stop, and debug Solr with your code.

First, you have to import to Eclipse the sample project in your local `ch1` folder. I assume you already got the source code from the publisher's website or from Github, as described in the Preface. Open Eclipse, create a new workspace, and go to **File** | **Import** | **Maven** | **Existing Maven Projects**.

Downloading the example code

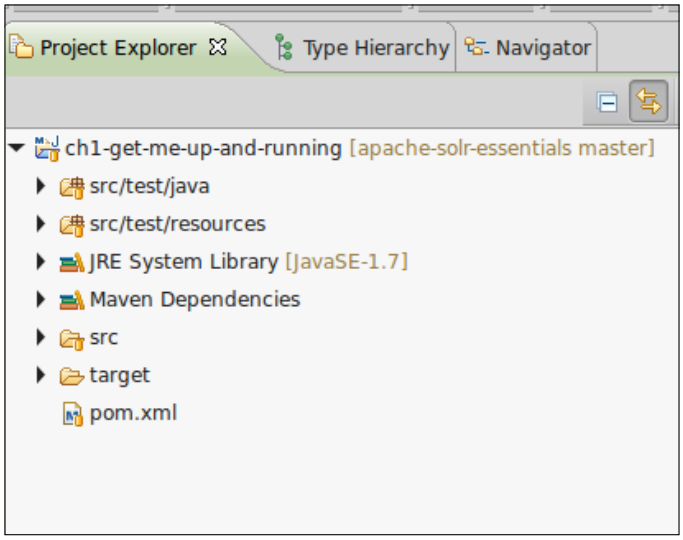
You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Alternatively, you can also download the examples from GitHub, on `https://github.com/agazzarini/apache-solr-essentials`. There, you can download the whole content as a zip file from `https://github.com/agazzarini/apache-solr-essentials/archive/master.zip` or, if you have git installed on your machine, you can clone the repository by issuing the following command:

```
# git clone https://github.com/agazzarini/apache-solr-essentials.git <path-to-your-work-dir>
```

Where `<path-to-your-work-dir>` is the destination folder where the project will be cloned.

In the dialog box that appears, select the `ch1` folder and click on the **Finish** button. Eclipse will detect the Maven layout of that folder and will create a new project on your workspace, as illustrated in the following screenshot (**Project Explorer** view):



Understanding the project structure

The project you've imported is very simple and contains just few lines of code, but it is useful for introducing some common concepts that will guide us through the book (the other chapters use examples with a similar structure).

The following table shows the structure of the project:

Folder or File	Description
`src/main/java`	The main source folder. It is empty at the moment, but it will contain the Solr extensions (and dependent classes) you want to implement. You won't find this directory in this first project because we don't have the source files yet.
`src/main/resources`	This contains project resources such as properties and configuration files. You won't find this directory in this first project because we don't have any resources yet.
`src/test/java`	This source folder contains Unit and Integration tests. For this first project, you will find a single integration test here.
`src/test/resources`	This contains test resources such as properties and configuration files. It includes a sample logging configuration (`log4j.xml`).
`src/dev/eclipse`	Preconfigured Eclipse launchers used to run Solr and the examples in the project.
`src/solr-home`	This contains the Solr configuration files. We will describe the content of this directory later.
`pom.xml`	This is the Maven Project definition. Here, you can configure any feature of your project, including dependencies, properties, and so on.

Within the Maven project definition (that is, `pom.xml`), you can do a lot of things. For our purposes right now, it is important to underline the plugin section, where you can see the **Maven Cargo Plugin** (`http://cargo.codehaus.org/Maven2+plugin`) configured to run an embedded Jetty 7 container and deploy Solr. Here's a screenshot that shows the Cargo Plugin configuration section:

```
<plugin>
    <groupId>org.codehaus.cargo</groupId>
    <artifactId>cargo-maven2-plugin</artifactId>
    <version>1.4.9</version>
    <configuration>
        <container>
            <systemProperties>
                <cargo.servlet.port>8983</cargo.servlet.port>
                <solr.solr.home>${basedir}/src/solr-home</solr.solr.home>
                <solr.data.dir>${project.build.directory}/solr</solr.data.dir>
                <log4j.configuration>file://${basedir}/src/test/resources/log4j.xml</log4j.configuration>
            </systemProperties>
            <containerId>jetty7x</containerId>
            <type>embedded</type>
            <dependencies>
        </container>
        <deployables>
            <deployable>
                <groupId>org.apache.solr</groupId>
                <artifactId>solr</artifactId>
                <type>war</type>
                <properties>
                    <context>/solr</context>
                </properties>
            </deployable>
        </deployables>
        <wait>false</wait>
    </configuration>
    <executions>
```

If you have the **Build automatically** flag set (the default behavior in Eclipse), most probably Eclipse has already downloaded all the required dependencies. This is one of the great things about Apache Maven.

So, assuming that you have no errors, it's now time to start Solr. But where is Solr?

The first question that probably comes to mind is: "I didn't download Solr! Where is it?" The answer is still Apache Maven, which is definitely a great open source tool for software management and something that simplifies your life.

Maven is already included in your Eclipse (by means of the `m2e` plugin), and the project you previously imported is a fully compliant Maven project.

So don't worry! When we start a Maven build, Solr will be downloaded automatically. But where? In your local Maven repository, and you don't need to concern yourself with that.

Within the `pom.xml` file, you will find a property, `<solr.version>`, with a specific value. If you want to use a different version, just change the value of this property.

Different ways to run Solr

It's time to start Solr in your IDE for the first time but, prior to that, it's important to distinguish the two ways to run Solr:

- **Background server**: As a background server, so that you can start and stop Solr for debugging purposes
- **Integration test server**: As an integration test server so that you can have a dedicated Solr instance to run your integration tests suite

Background server

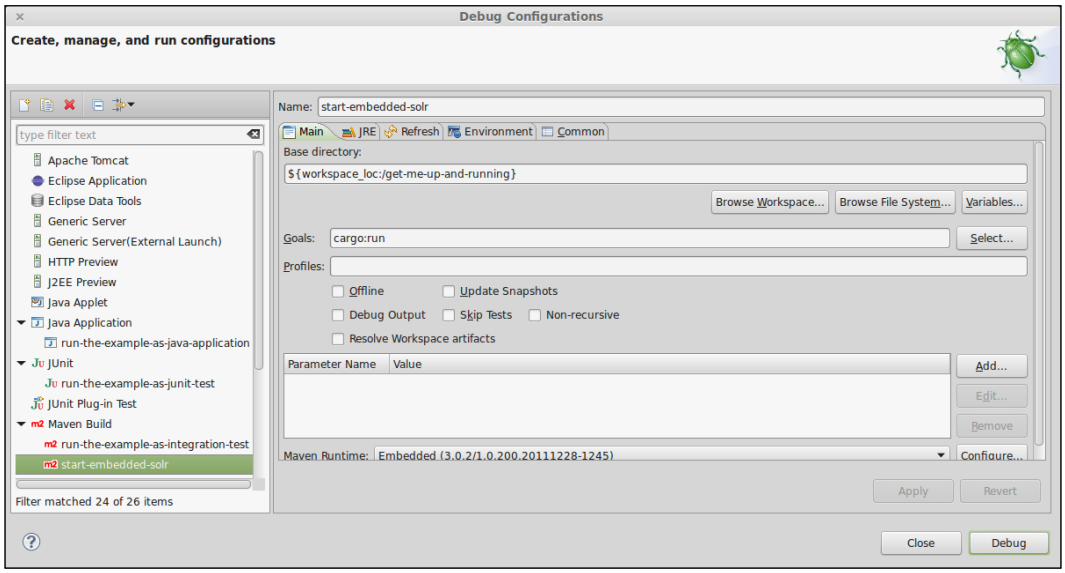
The first thing you will need in your IDE is a server instance that you can start, stop, and (in general) manage with a few simple commands.

In this way, you will be able to have Solr running with your configurations. You can index your data and execute queries in order to (manually) ensure that things are working as expected.

To get this type of server, follow these instructions:

1. Right-click on the project and create a new Maven (Debug) launch configuration (**Debug As | Maven build...**).
2. In the dialog, type `cargo:run` in the **Goals** text field.

3. Next, click on the **Debug** button as shown in the following screenshot:



The very first time you run this command, Maven will download all the required dependencies and plugins, including Solr. At the end, it will start an embedded Jetty instance.

Why a Debug instead of a Run configuration?

You must use a **Debug** configuration so that you will be able to stop the server by simply pressing the red button on the Eclipse console. **Run** configurations have an annoying habit: Eclipse will say the process is stopped, but Jetty will be still running, often leaving an orphan process.

You should see the following output in the Eclipse console:

```
[INFO] ------------------------------------------------------------
[INFO] Building Chapter 1 Project 1.0
[INFO] ------------------------------------------------------------
Downloading: http://repo1.maven.org/maven2/org/apache/solr/solr/4.9.0/
solr-4.9.0.war
Downloaded: http://repo1.maven.org/maven2/org/apache/solr/solr/4.8.0/
solr-4.9.0.war (28585 KB at 432.5 KB/sec)
...
[INFO] Jetty 7.6.15.v20140411 Embedded started on port [8983]
```

This means that Solr is up and running and it is listening on port `8983`. Now open your web browser and type `http://127.0.0.1:8983/solr`. You should see the Solr administration console.

In the project, and specifically in the `src/dev/eclipse` folder, there are some useful, ready-to-use Eclipse launchers. Instead of following the manual steps illustrated previously, just right-click on the `start-embedded-solr.launch` file and go to **Debug As** | **run-ch1-example-server.launch**.

Integration test server

Another important thing you could (or should, in my opinion) do in your project is to have an integration test suite. Integration tests are classes that, as the name suggests, run verifications against a running server.

When you're working on a project with Solr and you want to implement an extension, a search component, or a plugin, you will obviously want to ensure that it is working properly. If you're running an external Solr server, you need to pack your classes in a jar, copy that bundle somewhere (later, we will see where), start the server, and execute your checks.

There are a lot of drawbacks with this approach. Each time you get something wrong, you need to repeat the whole process: fix, pack, copy, restart the server, prepare your data, and run the check again. Also, you cannot easily debug your classes (or Solr classes) during that iterative check. All of this will most probably end with a lot of statements in your code as follows:

```
System.out.println("BLABLABLA");
```

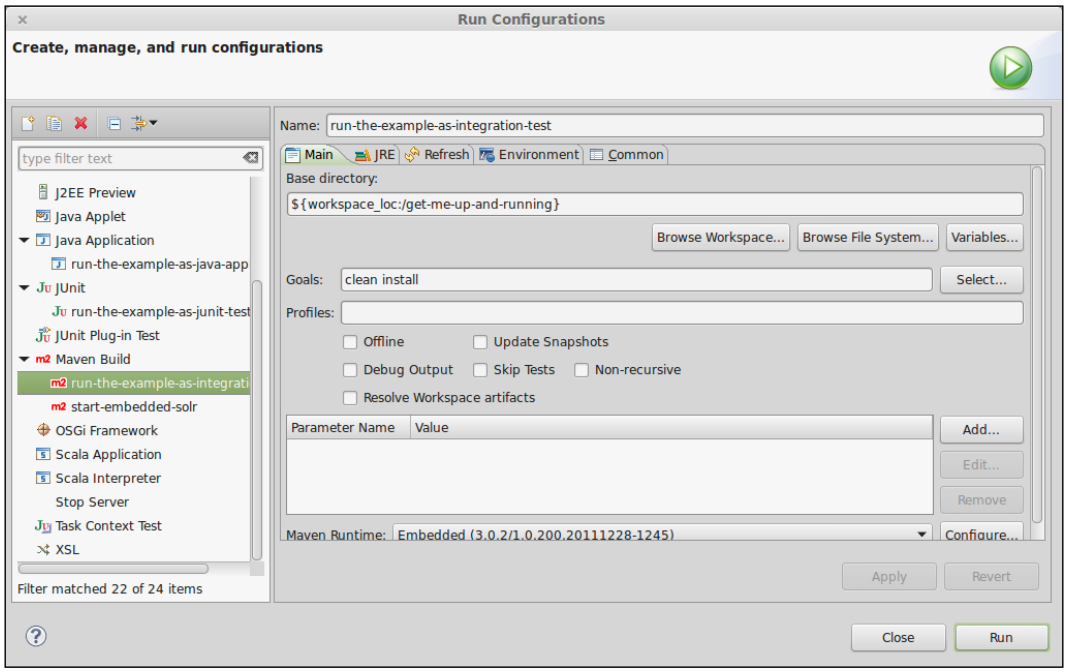
I suppose you know what I'm talking about.

This is where integration tests become very helpful. You can code your checks and your assertions as normal Java classes, and have an automated test suite that does the following each time it is executed:

- Starts an embedded Solr instance
- Executes your tests against that instance
- Stops the Solr instance
- Produces useful reports

The project we set up previously has that capability already, and there's a very basic integration test in the `src/test/java` folder to simply add and query some data.

In order to run the integration test suite, create a new Maven run configuration (right-click on the project and go to **Run As | Maven build...**), and, in the dialog box, type `clean install` in the **Goals** text field:



After clicking on the **Run** button, you should see something like this:

```
...
[INFO]  Jetty 7.6.15.v20140411 Embedded starting...
...
[INFO]  Reading Solr Schema from schema.xml
...
[INFO] Jetty 7.6.15.v20140411 Embedded started on port [8983]
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.gazzax.labs.solr.ase.ch1.it.FirstQueryITCase
...
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

As before, under the `src/dev/eclipse` folder, there is already a preconfigured Eclipse launcher for this scenario. Right-click on the `start-embedded-solr.launch` file and go to **Debug As | run-the-example-as-integration-test**.

From the Eclipse log, you can see that a test (specifically, an integration test) has been successfully executed. You can find the source code of that test in the project we checked out before. The name of the class that is reported in the log is `FirstQueryITCase` (IT stands for Integration Test), and it is in the `org.gazzax.labs.solr.ase.ch1.it` package.

The `FirstQueryITCase.java` class demonstrates a basic interaction flow we can have with Solr:

```
// This is the (input) Data Transfer Object between your
client and SOLR.
final SolrInputDocument input = new SolrInputDocument();

// 1. Populates with (at least required) fields
input.setField("id", 1);
input.setField("title", "Apache SOLR Essentials");
input.setField("author", "Andrea Gazzarini");
```

```
input.setField("isbn", "972-2-5A619-12A-X");

// 2. Adds the document
client.add(input);

// 3. Commit changes
client.commit();

// 4. Builds a new query object with a "select all" query.
final SolrQuery query = new SolrQuery("*:*");

// 5. Executes the query
final QueryResponse response = client.query(query);

// 6. Gets the (output) Data Transfer Object.
final SolrDocument output = response.getResults().iterator().next();

final String id = (String) output.getFieldValue("id");
final String title = (String) output.getFieldValue("title");
final String author = (String) output.getFieldValue("author");
final String isbn = (String) output.getFieldValue("isbn");

// 7.1 In case we are running as a Java application print
out the query results.
System.out.println("It works! I found the following book: ");
System.out.println("-------------------------------------");
System.out.println("ID: " + id);
System.out.println("Title: " + title);
System.out.println("Author: " + author);
System.out.println("ISBN: " + isbn);

// 7. Otherwise asserts the query results using standard
JUnit procedures.
assertEquals("1", id);
assertEquals("Apache SOLR Essentials", title);
assertEquals("Andrea Gazzarini", author);
assertEquals("972-2-5A619-12A-X", isbn);
```

`FirstQueryITCase` is an integration test and a main class at the same time. This means that you can run it in three ways: as described earlier, as a main class, and as a `JUnit` test. If you prefer the second or the third option, remember to start Solr before (using the `run-ch1-example-server.launch`). You can find the launchers under the `src/dev/eclipse` folder. Just right-click on one of them and run the example in one way or an other.

What do we have installed?

Regardless of the kind of installation, you should now have a Solr instance up and running, so it's time to have a quick overview of its structure.

Solr is a standard JEE web application, packaged as a .war archive. If you downloaded the bundle from the website, you can find it under the `webapps` folder of Jetty, usually under:

```
$INSTALL_DIR/solr-x.y.z/example/webapps
```

Instead, if you followed the developer way, Maven downloaded that war file for you, and it is now in your local repository (usually a folder called `.m2` under your `home` directory).

Solr home

In any case, Solr has been installed and you don't need to concern yourself with where it is physically located, mainly because all that you have to provide to Solr must reside in an external folder, usually referred to as the **Solr home**.

In the download bundle, there's a preconfigured Solr home folder that corresponds to the `$INSTALL_DIR/solr-x.y.z/example/solr` folder. Within your Eclipse project, you can find that under the `src` folder; it is called (not surprisingly) `solr-home`.

In a Solr home folder, you will typically find a file called `solr.xml`, and one or more folders that correspond to your Solr cores (we will see what a core is, in *Chapter 2*, *Indexing Your Data*). Each folder has a subfolder called `conf` where the configuration for that specific core resides.

solr.xml

The first file you will find within the Solr home directory is `solr.xml`. It declares some configuration parameters about the instance.

Previously (in Solr 4.4), you had to declare all the cores of your instance in this file. Now there's a more intelligent autodiscovery mechanism that helps you avoid explicit declarations about the cores that are part of your configuration.

In the download bundle, you will find an example of a Solr home with only one core:

```
$INSTALL_DIR/solr-x.y.z/example/solr
```

There is also an example with two cores:

```
$INSTALL_DIR/solr-x.y.z/example/multicore
```

This directory is built using the old style we mentioned previously, with all the cores explicitly declared. In the Eclipse project, you can find the single core example in a directory called `solr-home`. The multicore example is in the `example-solr-home-with-multicore` folder.

schema.xml

Although the `schema.xml` file will be described in detail later, it is important to briefly mention it because this is the place where you can declare how your index (of a specific core) is composed, in terms of fields, types, and analysis, both at index time and query time. In other words, this is the schema of your index and (most probably) the first thing you have to design as part of your Solr project.

In the download bundle you can find the `schema.xml` sample under the `$INSTALL_DIR/solr-x.y.z/example/solr/collection1/conf` folder, which is huge and full of comments. It basically illustrates all the predefined fields and types you can use in Solr (you can create your own type, but that's definitely an advanced topic).

If you want to see something simpler for now, the Eclipse project under the `solr-home/conf` directory has a very simple schema, with a few fields and only one field type.

solrconfig.xml

The `solrconfig.xml` file is where the configuration of a Solr core is defined. It can contain a lot of directives and sections but, fortunately for most of them, Solr's creators have set default values to be automatically applied if you don't declare them.

Default values are good for a lot of scenarios. When I was in Barcelona at the Apache Lucene Eurocon in 2011, the speaker asked during a presentation, "How many of you have ever changed default values in solrconfig.xml?" In a large room (200 people), only five or six guys raised their hands.

This is most probably the second file you will have to configure. Once the schema has been defined, you can fine-tune the index chain and search behavior of your Solr instance here.

Other resources

Schema and Solr configurations can make use of other files for several purposes. Think about stop words, synonyms, or other configuration files specific to some component. Those files are usually put in the `conf` directory of the Solr core.

Troubleshooting

If you have problems related to what we described previously, the following tips should help you get things working.

UnsupportedClassVersionError

You can install more than one version of Java on your machine but, when running a command (for example, java or javac), the system will pick up the java interpreter/compiler that is declared in your path. So if you get the `UnsupportedClassVersionError` error, it means that you're using a wrong JVM (most probably Java 6 or older). In the *Prerequisites* section earlier in this chapter, there's a table that will help you. However, this is the short version: Solr 4.7.x allows Java 6 or 7, but Solr 4.8 or greater runs only with (at least) Java 7.

If you're starting Solr from the command line, just type this:

```
# java -version
```

The output of this command will show the version of Java your system is actually using. So make sure you're running the right JVM, and also check your `JAVA_HOME` environment variable; it must point to the right JVM.

If you're running Solr in Eclipse, after checking what is described previously (that is, the JVM that starts Eclipse), make sure you're using a correct JVM by navigating to **Window** | **Preferences** | **Java** | **Installed JREs**.

The "Failed to read artifact descriptor" message

When running a command for the first time (for example, clean, install, or test), Apache Maven will have to download all the required libraries. In order to do that, your system must have a valid Internet connection.

So if you get this kind of message, it means that Maven wasn't able to download a required dependency. The name of the dependency should be in the message. The reason for failure could be a network issue, either permanent or transient.

In the first case, you should simply check your connection. In the second scenario (that is, a transient network failure during the download), there are some manual steps that need to be done. Assume that the dependency is `org.apache.solr:solr-solrj:jar:4.8.0`. You should go to your local Maven repository and remove the content of the folder that hosts that dependency, like this:

```
# rm -rf $HOME/.m2/repository/org/apache/solr/solr-solrj/4.8.0
```

On the next build, Maven will download that dependency again.

Summary

In this chapter, we began our Solr tour with a quick overview, including the steps that must be performed when installing Solr. We illustrated the installation process from both a user's and a developer's perspective. Regardless of the path you followed, you should have a working Solr installed on your machine.

In the next chapter, we will continue our conversation by digging further into the Solr indexing process.

2
Indexing Your Data

Although the final motive behind getting a Solr instance is to enable fast and efficient searches, we need to populate that instance with some data in the first (and mandatory) step. This operation is usually referred to as the indexing phase. The term **index** plays an important role in the Solr domain because its underlying structure is an index itself. This chapter focuses on the indexing process.

By the end of this chapter, you will be reasonably conversant with how the indexing process works in Solr, how to index data, and how to configure and customize the process.

This chapter will cover the following topics:

- The Solr data model: inverted index, document, fields, types, analyzers, and tokenizers
- Index and indexing configuration
- The Solr write path
- How to extend and customize the indexing process
- Troubleshooting

Understanding the Solr data model

Whenever I start to learn something that is not simple, I strongly believe the key to controlling its complexity is a good understanding of its domain model. This section describes the underlying building blocks of Solr. It starts with the simplest piece of information, the document, and then walks though the other fundamental concepts, describing how they form the Solr data model.

The document

A **document** represents the basic and atomic unit of information in Solr. It is a container of fields and values that belong to a given entity of your domain model (for example, a book, car, or person).

If you're familiar with relational databases, you can think of a document as a record. The two concepts have some similarities:

- A document could have a primary key, which is the logical identity of data it represents.
- A document has a structure consisting of one or more attributes. Each attribute has a name, type, and value.

However, a Solr document differs in the following ways from a database record:

- Attributes can have more than one value, whereas a row in a database table can have only one value (including NULL).
- Attributes either have a value or don't exist at all. There's no notion of NULL value in Solr.
- Attribute names can be static or dynamic, but table columns in a database must be explicitly declared in advance.
- Attribute types are, in general, more articulated and flexible because they must define how Solr interprets data both at index and query time.
- Attribute types can be defined and configured. This can be done by using, mixing, and configuring a rich set of built-in classes or creating new types (this is actually an advanced scenario).

A simple way to represent a Solr document is a map—a general data structure that maps unique keys (attribute names) to values, where each key (that is, attribute) can have one or more values. The following JSON data represents two documents:

```
{
  {
    "id":27302038,
    "title":"A book about something",
    "author": ["Ashler, Frank","York, Lye"],
    "subject": ["Generalities", "Social Sciences"],
    "language": "English"
  },
  {
    "id":2830002,
    "title": "Another book about something",
    "author": "Ypsy, Lea",
    "subject: "Geography & History",
    "publisher": "Vignanello: Edikin, 2010"
  }
}
```

Although the earlier documents represent books and have some common attributes as you can see, the first has two subjects and a language, while the second doesn't have a publication language. It has only one subject and an additional publisher attribute.

From a document's perspective, there's no constraint about which and how many attributes a document can have. Those constraints are instead declared within the Solr schema, which we will see later.

The `src/solr/example-data` folder of the project associated with this chapter contains some example data where the same documents are represented in several formats.

The inverted index

Solr uses an underlying, persistent structure called **inverted index**. It is designed and optimized to allow fast searches at retrieval time. To gain the speed benefits of such a structure, it has to be built in advance.

An inverted index consists of an ordered list of all the terms that appear in a set of documents. Beside each term, the index includes a list of the documents where that term appears.

For example, let's consider three documents:

```
{
  { "id": 1, "title":"The Birthday Concert" },
  { "id": 2, "title":"Live in Italy" },
  { "id": 3, "title":"Live in Paderborn" },
}
```

The corresponding inverted index would be something like this:

Terms	**Document Ids**		
	1	**2**	**3**
Birthday	X		
Concert	X		
Italy		X	
Live		X	X
Paderborn			X
The	X		
In		X	X

Like the index of a book (here, I mean the index that you usually find at the end of a book), if you want to search documents that contain a given term, an inverted index help you with that efficiently and quickly.

In Solr, index files are hosted in a so-called Solr data directory. This directory can be configured in `solrconfig.xml`, the main configuration file.

After running any example in the project associated with this book, you will find the Solr index under the subfolders located in `target/solr`. The name of the subfolder actually depends on the name of the core used in the example.

The Solr core

The index configuration of a given Solr instance resides in a Solr core, which is a container for a specific inverted index. On the disk, Solr cores are directories, each of them with some configuration files that define features and characteristics of the core.

In a core directory, you will typically find the following content:

- A `core.properties` file that describes the core.
- A `conf` directory that contains configuration files: a `schema.xml` file, a `solrconfig.xml` file, and a set of additional files, depending on components in use for a specific instance (for example, `stopwords.txt` and `synonyms.txt`).
- A `lib` directory. Every JAR file placed in this directory is automatically loaded and can be used by that specific core.

In a Solr installation you can have one or more cores, each of them with a different configuration, that will therefore result in different inverted indexes.

The concept of the Solr core has been expanded in Solr 4, specifically in SolrCloud. We will discuss this in *Chapter 6*, *Deployment Scenarios*.

The Solr schema

Returning to the comparison with databases, another important difference is that, in relational databases, data is organized in tables. You can create one or more tables depending on how you want to organize the persistence of the entities belonging to your domain model.

In Solr, things behave differently. There's no notion of tables; in a Solr schema, you must declare attributes, a primary key, and a set of constraints and features of the entity represented by the incoming documents. Although this doesn't strictly mean you must have only one entity in your schema, let's think in this way at the moment (for simplicity): a Solr schema is like the definition of a single table that describes the structure and the constraints of the incoming data (that is, documents).

The Solr schema is defined in a file called (not surprisingly) `schema.xml`. It contains several concepts, but the most important are certainly those related to types and fields. Before Solr 4.8, types and fields were declared within a `<types>` and a `<fields>` tag, respectively. Now their declarations can be mixed, which allows better grouping of fields with their corresponding types.

You can find a sample schema within the download bundle we set up in the previous chapter, specifically under `$INSTALL_DIR/solr-x.y.z/example/solr/collection1/conf/schema.xml`. It is huge and contains a lot of examples about predefined and built-in types and fields, with many useful comments.

Field types

Field types are one of the top-level entities declared in Solr schemas. A field type is declared using the `<fieldType>` element. As you can see in the example schema, you can have a simple type, such as this:

```
<fieldType name="string" class="solr.StrField"
sortMissingLast="true"/>
```

You can also have types with a lot of information, as shown here:

```
<fieldType name="text-general" class="solr.TextField"
positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.SynonymFilterFactory"
      synonyms="synonyms.txt"/>
  </analyzer>
</fieldType>
```

All types share a set of common attributes that are described in the following table:

Attribute	Description
`name`	The name of the field type. This is required.
`type`	The fully qualified name of the class that implements the field type behavior. This is required.
`sortMissingFirst` `sortMissingLast`	Optional attributes that are valid only for sortable fields. They define the sort position of the documents that have no values for a given field.
`indexed`	If this is `true`, fields associated with this type will be searchable, sortables and facetable.
`stored`	If this is `true`, fields associated with this type are retrievable. Briefly, stored fields are what Solr returns in search responses.
`multiValued`	If this is `true`, fields associated with this type can have multiple values.
`omitNorms`	Norms are values consisting of one byte per field where Solr records index time boost and length normalization data. Index time boost allows one field to be boosted higher than other. Length normalization allows shorter fields to be boosted more than longer fields. If you don't use index time boost and don't want to use length normalization, then this attribute can be set to true.
`omitTermsAndFrequencyPositions`	Tokens produced by text analysis during the index process are not simply text. They also have metadata such as offsets, term frequency, and optional payloads. If this attribute is set to `true`, then Solr won't record term frequencies and positions.
`omitPositions`	Omits the positions in indexed tokens.
`positionsIncrementGap`	When a field has multiple values, this attribute specifies the distance between each value. This is used to prevent unwanted phrase matches.
`autogeneratePhraseQueries`	Only valid for text fields. If this is set to `true`, then Solr will automatically generate phrase queries for adjacent terms.
`compressed`	In order to decrease the index size, stored values of fields can be compressed.
`compressThreshold`	Whenever the field is compressed, this is the associated compression threshold.

Besides all of this, each specific type can declare its own attributes, depending on the characteristic of the type itself.

The text analysis process

Before talking about fields, which are the top-level building blocks of the Solr schema, let's introduce a fundamental concept—text analysis.

The text analysis process converts an incoming value in tokens by means of a dedicated transformation chain that is in charge of manipulating the original input value. Each resulting token is then posted to the index with the following metadata:

- **Position increment**: The position of the token relative to the previous token in the input stream
- **Start and end offset**: The starting and ending indexes of the token within the input stream
- **Payload**: An optional byte array used for several purposes, such as boosting

A token with its metadata is usually referred to as a **term**.

In Solr, text analysis happens at two different moments: index and search time. In the first case, the value is the content of a given field of a given document that a client sent for indexing. In the second case, the incoming value typically contains search terms within a query.

In both cases, you must tell Solr how to handle those values. You can do that in the schema, in the field types section.

For field types, the following general rules always apply:

- If the field type implementation class is `solr.TextField` or it extends `solr.TextField`, then Solr allows you to configure one or two analyzer sections in order to customize the index and/or the query text analysis process
- In other cases, no analyzers can be defined, and the configuration of the type is done using the available attributes of the type itself

This is an example of a field type definition:

```
<fieldType name="text-general" class="solr.TextField"
positionIncrementGap="100">
  <analyzer type="index">
    …
  </analyzer>
```

```
    <analyzer type="query">
      …
    </analyzer>
  </fieldType>
```

Here, you can see two different **analyzer** sections. In the first section, you will declare what happens at index time for a given field associated with that field type. The second section has the same purpose, but it is valid for query time.

If you have the same analysis at index and query times, you can define just one `<analyzer>` section with no name attribute. That will be supposed to be valid for both phases.

Within each analyzer definition, you define the text analysis process by means of character filters, tokenizers, and token filters.

Char filters

Char filters are optional components that can be set at the beginning of the analysis chain in order to preprocess field values. They can manipulate a character stream by adding, removing, or replacing characters while preserving the original character position.

In the following example, two char filters are used to replace diacritics (that is, letters with glyphs such as à, ü) and remove some text:

```
<analyzer type="index">
  <charFilter class="solr.MappingCharFilterFactory"
    mapping="mapping-FoldToASCII.txt"/>
  <charFilter class="solr.PatternReplaceCharFilterFactory"
    pattern="\\(Author\\)" replacement=""/>
</analizer>
```

You must never declare the implementation class. Instead, declare its factory.

Using the preceding chain, the `Millöcker, Carl` text (name of author) will become `Millocker, Carl`.

A complete list of available char filters can be found at `http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters#CharFilterFactories`.

Tokenizers

A **tokenizer** breaks an incoming character stream into one or more tokens depending on specific criteria. The resulting set of tokens is usually referred to as a token stream. An analyzer chain allows only one tokenizer.

Suppose we have "I'm writing a simple text" as the input text. The following table shows how two sample tokenizers work:

Tokenizer	Description	Tokens
`WhitespaceTokenizer`	Splits by white spaces	"I'm", "writing", "a", "simple", "text"
`KeywordTokenizer`	Doesn't split at all	"I'm writing a simple text"

A complete list of available tokenizers can be found at `http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters#TokenizerFactories`.

Token filters

Token filters work on an input token stream, contributing some kind of transformation to it. Analyzing token after token, a filter can apply its logic in order to add, remove, or replace tokens, and can thus produce a new output token stream.

Token filters can be chained together in order to produce complex analysis chains. The order in which those filters are declared is important because the chain itself is not commutative. Two chains with the same filters in a different order could produce a different output stream.

This is an extract of a sample filter chain:

```
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.StopFilterFactory" words="stopwords.txt"
  ignoreCase="true"/>
```

A filter declaration includes the name of the implementation factory class and a set of attributes that are specific to each filter. In the preceding chain, this is what happens for each token in the input stream:

- The token is made into lowercase, so "Happy" will become "happy"

- If the token is a stopword, that is, one of the words declared in a file called stopwords.txt, it gets filtered from the outgoing stream

A complete list of available token filters is available at `http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters#TokenFilterFactories`.

Putting it all together

The following code illustrates a complete field type definition:

```
<fieldType name="my-text-type" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer type="index">
    <charFilter class="solr.MappingCharFilterFactory"
      mapping="mapping-FoldToASCII.txt"/>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

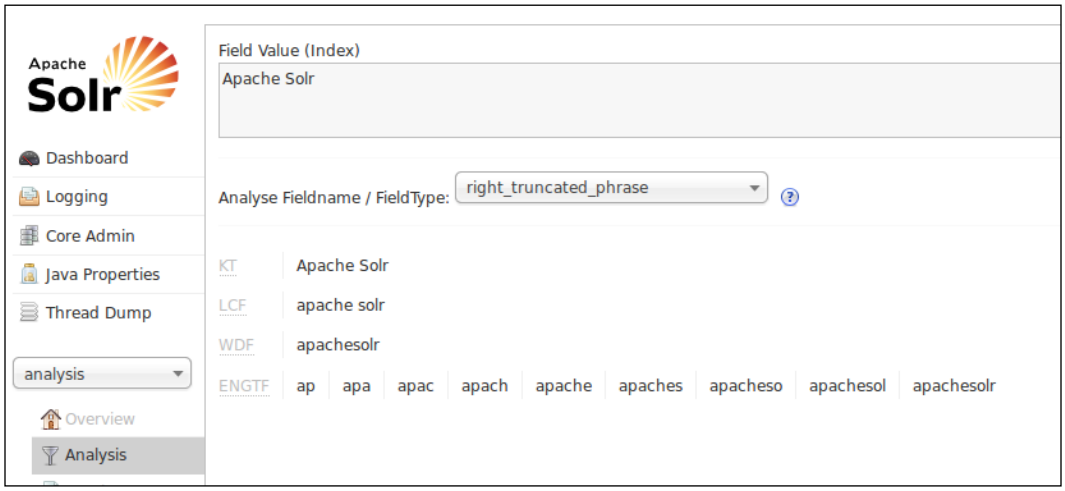
In order to get a concrete view of what happens during the index phase of a given field, open a shell in the top-level directory of the project associated with this chapter. Next, type the following command:

```
# mvn cargo:run –P fieldAnalysis
```

You can do the same with Eclipse by creating a new Maven Debug launch configuration. On the launch dialog, you must fill the **Goals** input field with `cargo:run` and the **Profile** input field with `fieldAnalysis`.

That will start a Solr instance with an example schema that contains several types. Once Solr has been started, open your browser and type `http://127.0.0.1:8983/solr/#/analysis/analysis`. The page that appears lets you simulate the index phase of a given value (the content of the left text area) for a given field or field type (the content of the drop-down menu at the bottom of the page).

Type some text in the **Field Value (Index)** text area, choose a field type or a field, and press the **Analyse Values** button. The page will show the input and the output values of each member of the index chain. The following screenshot illustrates the resulting page after analyzing the "Apache Solr" text with a `right_truncated_phrase` field type:



Some example field types

This section lists and describes some important field types and their main features in a non-exhaustive way. The `schema.xml` file in the download bundle contains a lot of examples with all the available types.

In addition, a list of all field types is available at `https://cwiki.apache.org/confluence/display/solr/Field+Types+Included+with+Solr`.

String

The string type retains the incoming value as a single token.

That doesn't mean the field cannot be indexed. It only means that the field cannot have a user-defined analysis chain.

This type is usually associated with the following:

- **Indexed fields**: Fields that represent codes, classifications, and identifiers, such as A340, 853.92, SKU#22383, 3919928832, 292381, and en-US
- **Sort fields**: Fields that can be used as sort criteria, such as authors, titles, and publication dates

Numbers

There are several numeric types defined in Solr. They can be classified into three groups:

- Basic types such as `IntField`, `FloatField`, and `LongField`. These are the legacy types that encode numeric values as strings.
- Sortable fields types such as `SortableDoubleField`, `SortableIntField`, and `SortableLongField`. These are the legacy types that encode numeric values as strings in order to match their natural numeric order (this is different from the string's lexicographic order).
- Trie fields types such as `TrieIntField`, `TrieFloatField`, and `TrieLongField`. These are the types that index numeric values using various and tunable levels of precision in order to enable efficient range queries and sorting. Those levels are configured using a `precisionStep` attribute in the field type definition.

The first two groups, basic and sortable types, are deprecated and will soon be removed (most probably in Solr 5.0). This is because their features and characteristics are already included in Trie types, which are more efficient and provide a unified way of dealing with numbers.

Boolean

Boolean fields can have a value of `true` or `false`. Values of `1`, `t`, or `T` are interpreted as true.

Date

The format that Solr uses for dates is a restricted version of the ISO 8601 Date and Time format and is of the `YYYY-MM-DDThh:mm:ss.SSSZ` form. Here are some examples of this field type:

```
2005-09-27T14:43:11Z
2011-08-23T02:43:00.992Z
```

The `Z` character is a literal, trailing constant that indicates the UTC method of the date representation. Only the milliseconds are optional. If they are missing, the dot (`.`) after the seconds must be removed.

As with numbers, there are two available types to represent dates in Solr:

- A basic `DateField` type, which is a deprecated legacy type
- `TrieDateField`, which is the recommended date type

A useful feature of date types is a simple expression language that can be used to form dynamic date expressions, like this:

```
NOW + 2YEARS
NOW + 3YEARS – 3DAYS
2005-09-27T14:43:00 + 1YEAR
```

The expression language allows the following keywords:

Keyword	Description
`YEAR/YEARS`	One or more years. These are basically synonyms; the difference is just to make the expressions more readable (for example, `2YEARS` is better than `2YEAR`).
`MONTH/MONTHS`	One or more months (for example, `NOW + 4MONTHS`, `NOW – 1MONTH`).
`DAY/DAYS/DATE`	A day or a certain number of days (for example, `NOW + 1DAY`).
`HOUR/HOURS`	An hour or a certain number of hours.
`MINUTE/MINUTES`	One or more minutes.

`MILLI/MILLIS`
`MILLISECOND`
`MILLISECONDS`
	One or more milliseconds.

Text

Text is the basic type for fields that can have a configurable text analysis. This is the only type that accepts analyzer chains in configurations.

Other types

The following list briefly describes some other interesting types:

- **Currency**: This type provides support for monetary values with a dedicated type. It also includes the capability to plug in several providers for determining exchange rates between currencies.
- **Binary**: This type is used to handle binary data. Data is sent and retrieved in Base64-encoded strings.
- **Geospatial types**: Two types are available for support to geospatial searches. The first is `LatLonType`, from Solr 3.x onwards. The second type, `SpatialRecursivePrefixTreeFieldType`, is a new type introduced in Solr 4, and it supports polygon shapes.
- **Random**: This is used to generate random sequences. It is useful if you want pseudorandom sort ordering of indexed documents.

Fields

Fields are containers of values associated with a specific type. They represent the structure and the composition of the entity of your domain model.

In simple words, fields are the attributes of the documents you're going to manage with Solr. So, for example, if Solr serves a library **Online Public Application Catalogue** (**OPAC**), the entities in the schema will most probably represent books, and they could have fields such as title, author, ISBN, cover, and so on.

Fields are declared in the schema. Each field declaration includes a name, type, and set of attributes. This is an example of field declaration:

```
<field name="title" type="string" indexed="false" stored="true"
required="true" multiValued="false"/>
```

The following table lists the attributes that can be specified for each field:

Keyword	Description
`name`	The name of the field must be unique in the schema and must consist only of alphanumeric and underscore characters. It must not start with an underscore, and it must not have both a leading and a trailing underscore because those kinds of names are reserved.
`type`	This is the type associated with the field.
`indexed`	If this is `true`, fields associated with this type will be searchable, sortable, and facetable. It overrides the same setting on the associated type.
`stored`	If this is `true`, it makes the fields associated with this type retrievable. It overrides the same setting on the associated type.
`required`	This marks the field as mandatory in input documents.
`default`	A default value that will be used at index time, if the field in the input document doesn't have a valid value.
`sortMissingFirst` `sortMissingLast`	These are optional attributes defining the sort position of the documents that have no values for that field. They override the same settings on the associated type.
`omitNorms`	Omits the norms associated with this field. Overrides the same attribute on the field type.
`omitPositions`	Omits the term positions associated with this field. Overrides the same attribute on the field type.
`omitTermFreqAndPositions`	Omits the term frequency and positions associated with this field. Overrides the same attribute on the field type.
`termVectors`	Stores the term vectors. A term vector is a list of the document's terms and their number of occurrences in that document.
`docValues`	Only available for the String, Trie, and UUID fields. This attribute enhances the index by adding column-oriented fields to a document-to-value mapping.

Static fields

The first category of fields contains those statically declared in the schema. In this context, **static** simply means that the name of the field is explicitly known in advance. This is an example of a static field:

```
<field name="isbn" (other attributes follow) />
```

Dynamic fields

There are certain situations where you don't know in advance the name of some fields in the incoming documents. Although this may sound strange, it is rather a frequent scenario.

Think about a document that represents a book and is the result of some kind of cataloguing. In general, a bibliographic record has a lot of fields. Some of them represent text that can be expressed by cataloguers in several languages. For example, you can have a book with these abstracts:

```
{
"id":92902893,
  "abstract_en": "This is the English summary",
  "abstract_es": "Éste es el resumen en español",
  (other fields follow)
}
```

You can have another book with the following definition:

```
{
"id":92902893,
  "abstract_it": "L'automazione della biblioteca digitale"
  (other fields follow)
}
```

So the question here is, how can we define the *abstract* field (or fields) in our schema? The first approach could be to declare several static fields—one for each language—but this will be valid only if we know all the input languages in advance. Moreover, this is not very extensible because adding a new language (for example, `abstract_ru`) will require a change in the schema. Dynamic fields are the alternative.

A field is dynamic when its name includes a leading or a trailing wildcard, therefore allowing a dynamic match with incoming input fields. A dynamic field is declared using the `<dynamicField>` element, as follows:

```
<dynamicField name="abstract_*" (other attributes follow) />
```

The field will catch all fields that have a prefix equal to `abstract`. Hence, it avoids the need to statically define fields one by one, but most importantly, it will catch any `abstract` field regardless of its language suffix.

Copy fields

In the Solr schema, you can use a special `copyField` directive to copy one field to another. This is useful when a document has a given field, and starting from its value, you want to have other fields in your schema populated with the same value but with a different text analysis.

Let's suppose your documents represent books that can contain two different kinds of authors:

- persons (for example, Dante Alighieri and Leonardo Da Vinci)
- corporates (for example, Association for Childhood Education International)

You must show those authors separately in the user interface, as part of customer requirements. You can give them dedicated labels, for example. At the same time, the customer wants to have an author search feature on the user interface that triggers a search for all kinds of authors. The following screenshot shows a GUI widget that is often used in these scenarios—a search toolbar with a drop-down menu that allows the user to constrain the scope of the search within a given context (for example, authors, subjects, and titles):

A first approach could be to have two stored and indexed fields. When the user searches for an author by typing a name or a surname, such terms will be searched within those two fields. The schema in this case should be as follows:

```
<field name="author_person" type="text" indexed="true"
stored="true" …/>
<field name="author_corporate" type="text" indexed="true"
stored="true" …/>
```

A second choice could be to have a more cohesive design by separating search and view responsibilities. In this case, we will have two stored (but not indexed) fields representing the two kinds of authors, and a generic indexed (but not stored) `author_search` field containing all the authors of a document, regardless of its type. In this way, the user interface will use the stored fields for visualization, while Solr will use the catch-all `author_search` field for searches. This design introduces the `copyField` directive; here is the corresponding schema:

```
<field name="author_person" type="string" indexed="false"
stored="true" required="false" multiValued="true"/>
<field name="author_corporate" type="string" indexed="false"
stored="true" required="false" multiValued="true"/>
<field name="author_search" type="text" indexed="true" stored="false"
required="false" multiValued="true"/>

<copyField source="author_person" dest="author_search"/>
<copyField source="author_corporate" dest="author_search"/>
```

The `copyField` directive copies the incoming value of the `source` field in the `dest` field; thus, at the end, the `author_search` field will contain all kinds of authors.

In both the `source` and `dest` attributes, it's possible to use a trailing or a leading wildcard, therefore avoiding repetitive code. In the preceding example, we could have just one `copyField` declaration:

```
<copyField source="author_*" dest="author_search"/>
```

Other schema sections

Other than fields and field types, the Solr schema contains some other things as well. This section briefly illustrates them.

Unique key

This field uniquely identifies your document. This is not strictly required but strongly recommended if you want to update your documents, avoid duplicates, and (last but not least) use Solr distributed features.

Default similarity

This element allows you to declare the factory of the class used by Solr to determine the score of documents while searching.

Solr indexing configuration

Once the schema has been defined, it's time to configure and tune the indexing process by means of another file that resides in the same directory of the schema—`solrconfig.xml`.

The file contains a lot of sections, but fortunately, there are a lot of optional parts with default values that usually work well in most scenarios. We will try to underline the most important of them with respect to this chapter.

As a general note, it's possible to use system properties and default values within this file. Therefore, we are able to create a dynamic expression, like this:

```
<dataDir>${my.data.dir:/var/data/defaultDataDir}</dataDir>
```

The value of the `dataDir` element will be replaced at runtime with the value of the `my.data.dir` system property, or with the default value of `/var/data/defaultDataDir` if that property doesn't exist.

General settings

The heading part of the `solrconfig.xml` file contains general settings that are not strictly related to the index phase.

The first is the Lucene match version:

```
<luceneMatchVersion>LUCENE_47</luceneMatchVersion>
```

This allows you to control which version of Lucene will be internally used by Solr. This is useful to manage migration phases towards the newer versions of Solr, thus allowing backward compatibility with indexes built with previous versions.

A second piece of information you can set here is the data directory, that is, the directory where Solr will create and manage the index. It defaults to a directory called `data` under `$SOLR_HOME`.

```
<dataDir>/var/data/defaultDataDir</dataDir>
```

Index configuration

The section within the `<indexConfig>` tag contains a lot of things that you can configure in order to fine-tune the Solr index phase.

A curious thing you can see in this section, in the `solrconfig.xml` file of the example core, is that most things are commented. This is very important, because it means that Solr provides good default values for those settings.

The following table summarizes the settings you will find within the `<indexConfig>` section:

Attribute	Description
`writeLockTimeout`	The maximum allowed time to wait for a write lock on an `IndexWriter`.
`maxIndexingThreads`	The maximum allowed number of threads that index documents in parallel. Once this threshold has been reached, incoming requests will wait until there's an available slot.
`useCompoundFile`	If this is set to `true`, Solr will use a single compound file to represent the index. The default value is `false`.
`ramBufferSizeMB`	When accumulated document updates exceed this memory threshold, all pending updates are flushed.
`ramBufferSizeDocs`	This has the same behavior as that of the previous attribute, but the threshold is defined as the count of document updates.
`mergePolicy`	The names of the class, along with settings, that defines and implements the merge strategy.

Attribute	Description
`mergeFactor`	A threshold indicating how many segments an index is allowed to have before they are merged into one segment. Each time an update is made, it is added to the most recent index segment. When that segment fills up (that is, when the `maxBufferedDocs` and `ramBufferSizeMB` thresholds are reached), a new segment is created and subsequent updates are inserted there. Once the number of segments reaches this threshold, Solr will merge all of them into one segment.
`mergeScheduler`	The class that is responsible for controlling how merges are executed.
`lockType`	The lock type used by Solr to indicate that a given index is already owned by `IndexWriter`.

Update handler and autocommit feature

The `<UpdateHandlerSection>` configures the component that is responsible for handling requests to update the index.

This is where it's possible to tell Solr to periodically run unsolicited commits so that clients won't need to do that explicitly while indexing. Declaring two different thresholds can trigger auto-commits:

- **maxDocs**: The maximum number of documents to add since the last commit
- **maxTime**: The maximum amount of time (in milliseconds) to pass for a document being added to index

They are not exclusive, so it's perfectly legal to have settings such as these:

```
<autoCommit>
  <maxDocs>5000</maxDocs>
  <maxTime>300000</maxTime>
</autoCommit>
```

Starting from Solr 4.0, there are two kinds of commit. A **hard commit** flushes the uncommitted documents to the index, therefore creating and changing segments and data files on the disk. The other type is called **soft commit**, which doesn't actually write uncommitted changes but just reopens the internal Solr searcher in order to make uncommitted data in the memory available for searches.

Hard commits are expensive, but after their execution, data is permanently part of the index. Soft commits are fast but transient, so in case of a system crash, changes are lost.

Hard and soft commits can coexist in a Solr configuration. The following is an example that shows this:

```
<autoCommit>
  <maxTime>900000</maxTime>
</autoCommit>
<autoSoftCommit>
  <maxTime>1000</maxTime>
</autoSoftCommit>
```

Here, a soft commit will be triggered every second (`1000` milliseconds), and a hard commit will run every 15 minutes (`900000` milliseconds).

RequestHandler

A `RequestHandler` instance is a pluggable component that handles incoming requests. It is configured in `solrconfig.xml` as a specific endpoint by means of its name attribute.

Requests sent to Solr can belong to several categories: search, update, administration, and stats. In this context, we are interested in those handlers that are in charge of handling index update requests. Although not mandatory, those handlers are usually associated with a name starting with the `/update` prefix, for example, the default handler you will find in the configuration:

```
<requestHandler name="/update" class="solr.UpdateRequestHandler"/>
```

Prior to Solr 4, each kind of input format (for example, JSON, XML, and so on) required a dedicated handler to be configured. Now the general-purpose update handler, that is, the `/update` handler uses the content type of the incoming request in order to detect the format of the input data. The following table lists the built-in content types:

Mime-type	Description
application/xml **text/xml**	XML messages
application/json **text/json**	JSON messages
application/csv **text/csv**	Comma-separated values
application/javabin	Java-serialized objects (Java clients only)

Each format has its own way of encoding the kind of update operation (for example, `add`, `delete`, and `commit`) and the input documents. This is a sample `add` command in XML:

```
<add>
  <doc>
    <field name="id">12020</field>
    <field name="title">Round around midnight</field>
  </doc>
  …
</add>
```

Later, we will index some data using different techniques and different formats.

UpdateRequestProcessor

The write path of the index process has been conceived by Solr developers with modularity and extensibility in mind. Specifically, the index process has been structured as a chain of responsibilities, where each set of components adds its own contribution to the whole index process.

The `UpdateRequestProcessor` chain is an important configurable aspect of the index process. If you want to declare your custom chain, you need to add a corresponding section within the configuration. This is an example of a custom chain:

```
<updateRequestProcessorChain name="my-index-chain">
  <processor class="…"/>
  <processor class="…">
    <str name="aParameterName">aParameterValue</str>
  </processor>
  <processor name="solr.RunUpdateProcessorFactory"/>
  <processor name="solr.LogUpdateProcessorFactory"/>
</updateRequestProcessorChain>
```

Defining a new chain requires a name and a set of `UpdateRequestProcessorFactory` components that are in charge of creating processor instances for that chain.

Actually, the definition of the chain is not enough. It must be enabled, (that is, associated with `RequestHandler`) in the following way:

```
<requestHandler name="/myReqHandler"
  class="solr.UpdateRequestHandler">
  <lst name="defaults">
    <str name="update.chain">chain.name</str>
  </lst>
</requestHandler>
```

There are a lot of already implemented `UpdateRequestProcessor` components that you can use in your chain, but in general, it's absolutely easy to create your own processor and customize the index chain.

The example project with this chapter contains several examples of `UpdateRequestProcessor` within the `org.gazzax.labs.solr.ase.ch2.urp` package.

Index operations

This section shows you the basic commands needed for updating an index, by adding or removing documents. As a general note, each command we will see can be issued in at least two ways: using the command line, through the cURL tool, for example (a built-in tool in a lot of Linux distributions and available for all platforms); and using code (that is, SolrJ or some other client API). When you want to add documents, it's also possible to run those commands from the administration console.

SolrJ and client APIs will be covered later in a dedicated chapter.

Another common aspect of these interactions is the Solr response, which always contains a `status` and a `QTime` attribute. The status is a returned code of the executed command, which is always 0 if the operation succeeds. The `QTime` attribute is the elapsed time of the execution. This is an example of the response in XML format:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">97</int>
  </lst>
</response>
```

Add

The command sends one or more documents to add to Solr. The documents that are added are not visible until a `commit` or an `optimize` command is issued.

We already saw that documents are the unit of information in Solr. Here, depending on the format of the data, one or more documents are sent using the proper representation.

Since the attributes and the content of the message will be the same regardless of the format, the formal description of the message structure will be given once. The following is an `add` command in XML format:

```
<add commitWithin="10000" overwrite="true">
  <doc boost="1.9">
    <field name="id">12020</field>
    <field name="title" boost="2.2">Round around midnight</field>
    <field name="subject">Music</field>
    <field name="subject">Jazz</field>
  </doc>
  …
</add>
```

Let's discuss the preceding command in detail:

- `<add>`: This is the root tag of the XML document and indicates the operation.
- `commitWithin`: This is an alternative to the autocommit features we saw previously. Using this optional attribute, the requestor asks Solr to ensure that the documents will be committed within a given period of time.
- `overwrite`: This tells Solr to check out and eventually overwrite documents with the same `uniqueKey`. If you don't have a uniqueKey, or you're confident that you won't ever add the same document twice, you can get some index performance improvements by explicitly setting this flag to `false`.
- `<doc>`: This represent the document to be added.
- `boost`: This is an optional attribute that specifies the boost for the whole document (that is, for each field). It defaults to 1.0.
- `<field>`: This is a field of the document with just one value. If the field is multivalued, there will be several fields with the same name and different values.
- `boost`: This is an optional attribute that specifies the boost for the specific field. It defaults to 1.0.

The same data can be expressed in JSON as follows:

```
{
  "add": {
    "commitWithin": 10000,
    "overwrite": true,
    "doc": {
      "boost": 1.9,
      "id": 12020,
      "title": {
        "value": "Round around midnight",
        "boost": 2.2
        },
      "subject": ["Music", "Jazz"]
      }
    }
}
```

As you can see, the information is the same as in the previous example. The difference is in the encoding of the information according to the JSON format.

Sending add commands

We can issue an `add` command in several ways: using cURL, the administration console, and a client API such as SolrJ.

The cURL tool is a command-line tool used to transfer data with URL syntax. Among other protocols, it supports HTTP and HTTPS, so it's perfect for sending commands to Solr. These are some examples of `add` commands sent using cURL:

```
# curl http://127.0.0.1:8983/solr/update -H "Content-type: text/xml"
--data-binary @datafile.xml


# curl http://127.0.0.1:8983/solr/update -H "Content-type: text/xml"
--data-binary
'<add commitWithin="10000" overwrite="true">
  <doc boost="1.9">
    <field name="id">12020</field>
    …
    <field name="subject">Jazz</field>
  </doc>
  …
</add>'
```

The first example uses data contained in a file. The second (useful for short requests) directly embeds the documents in the data-binary parameter. The preceding examples are perfectly valid for JSON and CSV documents as well (obviously, the data format and the content type will change).

Delete

A `delete` command will mark one or more documents as deleted. This means the target documents are not immediately removed from the index. Instead, a kind of tombstone is placed on them; when the next commit event happens, that data will be removed. Commits and optimizes are commands that make the update changes visible and available. In other words, they make those changes effectively part of the Solr index. We will see both of them later.

Solr allows us to identify the target documents in two different ways: by specifying a set of identifiers or by deleting all documents matched by a query. In the same way as we sent `add` commands, we can use cURL to issue `delete` commands:

```
# curl http://127.0.0.1:8983/solr/update -H "Content-type: text/xml"
--data-binary @datafile_with_deletes.xml


# curl http://127.0.0.1:8983/solr/update -H "Content-type: text/xml"
--data-binary
'<delete>
  <id>92392</id>
  <query>publisher:"Ashler"</query>
</delete>'
```

In the second example, we issued a command to delete:

- The document with `92392` as `uniqueKey`
- All documents that have a publisher attribute with the `Ashler` value

Commit, optimize, and rollback

Changes resulting from add and delete operations are not immediately visible. They must be committed first; that is, a commit command has to be sent.

We already explored hard and soft unsolicited commits in the *Index configuration* section. The same command can be explicitly sent to Solr by clients.

Although we previously described the difference between hard and soft commits, it's important to remember that a hard commit is an expensive operation, causing changes to be permanently flushed to disk. Soft commits operate exclusively in memory, and are therefore very fast but transient; so, in the event of a JVM crash, softly committed data is lost.

In a prototype I'm working on, we index data coming from traffic sensors in Solr. As you can imagine, the input flow is continuous; it can happen several times in a second. A control system needs to execute a given set of queries at short periodic intervals, for example, every few seconds. In order to make the most updated data available to that system, we issue a soft commit every second and a hard commit every 20 minutes. At the moment, this seems to be a good compromise between the availability of fresh data and the risk of data loss (it could still happen during those 20 minutes).

For those interested, the Solr extension we will use in that project is available on GitHub, at `https://github.com/agazzarini/SolRDF`. It allows Solr to index RDF data, and it is a good example of the capabilities of Solr in the realm of customization.

A third kind of commit, which is actually a hard commit, is the so-called **optimize**. With optimize, other than producing the same results as those of a hard commit, Solr will merge the current index segments into a single segment, resulting in a set of intensive I/O operations. The merge usually occurs in the background and is controlled by parameters such as merge scheduler, merge policy, and merge factor. Like the hard commit, optimize is a very expensive operation in terms of I/O because, apart from costing the same as a hard commit, it must have some temporary space available on the disk to perform the merge.

It is possible to send the commit or the optimize command together with the data to be indexed:

```
# curl http://127.0.0.1:8983/solr/update?commit=true -H "Content-
type: text/xml" --data-binary @datafile.xml

# curl http://127.0.0.1:8983/solr/update?optimize=true -H "Content-
type: text/xml" --data-binary @datafile.xml
```

The message payload can also be a `commit` command:

```
# curl http://127.0.0.1:8983/solr/update -H "Content-type: text/xml"
--data-binary '<commit/>'
```

A commit has a few additional Boolean parameters that can be specified to customize the service behavior:

Parameter	Description
`waitSearcher`	The command won't return until a new searcher is opened and registered as the main searcher
`waitFlush`	The command won't return until uncommitted changes are flushed to disk
`softCommit`	If this is `true`, a soft commit will be executed

Before committing any pending change, it's possible to issue a **rollback** to remove uncommitted add and delete operations. The following are examples of rollback requests:

```
# curl http://127.0.0.1:8983/solr/update?rollback=true
# curl http://127.0.0.1:8983/solr/update -H "Content-type: text/xml"
--data-binary '<rollback/>'
```

Extending and customizing the index process

As we saw before, the Solr index chain is highly customizable at different points. This section will give you some hints and examples to create your own extension in order to customize the indexing phase.

Changing the stored value of fields

One of the most frequent needs that I encounter while I'm indexing bibliographic data is to correct or change the headings (labels) belonging to the incoming records (documents).

This has nothing to do with the text analysis we have previously seen. Here, we are dealing with unwanted (wrong) values, diacritics that need to be replaced, or in general, labels in the original record that we want to change and show to the end users. In Solr terms, we want to change the stored value of a field before it gets indexed.

Suppose a library has a lot of records and wants to publish them in an OPAC. Unfortunately, many of those records have titles with a trailing underscore, which has a special meaning for librarians. While this is not a problem for the cataloguing software (because librarians are aware of that convention), it is not acceptable to end users, and it will surely be seen as a typo. So if we have records with titles such as "A good old story_" or "This is another title_" in our application, we want to show "A good old story" and "This is another title" without underscores when the user searches for those records.

Remember that analyzers and tokenizers declared in your schema only act on the *indexed* value of a given field. The stored value is copied verbatim as it arrives, so there's no chance to modify it once it is indexed.

In these cases, an `UpdateRequestProcessor` perfectly fits our needs. The example project associated with this chapter contains several examples of custom `UpdateRequestProcessors`. Here, we are interested in `RemoveTrailingUnderscoreProcessor`, which can be found in the `src/main/java` within the `org.gazzax.labs.solr.ase.chr.urp` package.

As you can see, writing an `UpdateRequestProcessor` requires two classes to be implemented:

- **Factory**: A class that extends `org.apache.solr.update.processor.UpdateRequestProcessorFactory`
- **Processor**: A class that extends `org.apache.solr.update.processor.UpdateRequestProcessor`

The first is a factory that creates concrete instances of your processor and can be configured with a set of custom parameters in `solrconfig.xml`:

```
<processor class="org.gazzax.labs.solr.ase.chr.urp.
RemoveTrailingUnderscoreProcessorFactory">
  <arr name="fields">
    <str name="fields">title</str>
    <str name="fields">author</str>
  </arr>
</processor>
```

In this case, instead of hardcoding the name of the fields that we want to check, we define an array parameter called fields. That parameter is retrieved in the factory, specifically in the `init()` method, which will be called by Solr when the factory is instantiated:

```
private String [] fields;

@Override
public void init (NamedList args) {
  SolrParams parameters = SolrParams.toSolrParams(args);
  this.fields = parameters.getParams("fields");
}
```

The other relevant section of the factory is in the `getInstance` method, where a new instance of the processor is created:

```
@Override
public void getInstance(SolrQueryRequest req, SolrQueryReponse
res, UpdateRequestProcessor next) {
 return new RemoveTrailingUpdateRequestProcessor(next, fields);
}
```

A new processor instance is created with the next processor in the chain and the list of target fields we configured. Now the processor receives those parameters and can add its contribution to the index phase. In this case, we want to put some logic before the add phase:

```
@Override
public void processAdd(final AddUpdateCommand command) {
  // 1. Retrieve the Solr (Input) Document
  SolrInputDocument document = command.getSolrInputDocument();

  // 2. Loop thorugh target fields
  for (String name : fields) {
    // 3. Get the field value
    // we assume target fields are monovalued for simplicity
    String value = document.getFieldValue(name);

    // 4. Check and eventually change the value
    if (value != null && value.endsWith("_")) {
      String newValue = value.substring(0, value.length()-1);
      document.setFieldValue(name, newValue);
    }
```

```
    }

    // 5. IMPORTANT: forward to the next processor in the chain
    super.processAdd(command);
  }
```

You can find the source code of the whole example under the `org.gazzax.labs.solr.ase.ch2.urp` package of the source folder in the project associated with this chapter. The package contains additional examples of `UpdateRequestProcessor`.

Indexing custom data

The default `UpdateRequestHandler` is very powerful because it covers the most popular formats of data. However, there are some cases where data is available in a legacy format. Hence, we need to do something in order to have Solr working with that.

In this example, I will use a flat file, that is, a simple text file that typically describes records with fields of data defined by fixed positions. They are very popular in integration projects between banks and ERP systems (just to give you a concrete context).

In the example project associated with this chapter, you can find an example of such a file describing books under the `src/solr/solr-homes/flatIndexer/example-input-data` folder.

Here, each line has a fixed length of 107 characters and represents a book, with the following format:

Parameter	Position
Id	0 to 8
ISBN	8 to 22
Title	22 to 67
Author	67 to 106

There are two approaches in this scenario: the first moves the responsibility on the client side, thus creating a custom indexer client that gets the data in any format and carries out some manipulation to convert it into one of the supported formats. We won't cover this scenario right now, as we will discuss client APIs in a next chapter.

Another approach could be a custom extension of the `UpdateRequestHandler`. In this case, we want to have a new content type (`text/plain`) and a corresponding custom handler to load that kind of data. There are two things we need to implement. The first is a subclass of the existing `UpdateRequestHandler`:

```
public class FlatDataUpdate extends UpdateRequestHandler {
  @Override
  protected Map<String, ContentStreamLoader>
  createDefaultLoaders(NamedList n) {
    Map<String, ContentStreamLoader> registry = new
    HashMap<String, ContentStreamLoader>();
    registry.put("text/plain", new FlatDataLoader());
    return registry;
  }
}
```

Here, we are simply overriding the content type registry (the registry in the superclass cannot be modified) to add our content type, with a corresponding handler called `FlatDataLoader`. This class extends `ContentStreamLoader` and implements the parsing logic of the flat data:

```
public class FlatDataLoader extends ContentStreamLoader
```

The custom loader must provide a `load(…)` method to implement the stream parsing logic:

```
@Override
public void load(
SolrQueryRequest req,
SolrQueryResponse rsp,
ContentStream stream,
UpdateRequestProcessor processor) throws Exception {

  // 1. get a reader associated with the content stream
  BufferedReader reader = null;
  try {
    reader = new BufferedReader(stream.getReader());
    String actLine = null;
    while ((actLine = reader.readLine()) != null) {

    // 2. Sanity check: check line length
    if (actLine.length() != 107) {
      continue;
```

```
        }

        // 3. parse and create the document
        SolrInputDocument doc = new SolrInputDocument();
        doc.setField("id", actLine.substring(0, 8));
        doc.setField("isbn", actLine.substring(8,22));
        doc.setField("title", actLine.substring(22, 67));
        doc.setField("author", actLine.substring(67));
        AddUpdateCommand command = getAddCommand(req);
        command.solrDoc = document;
        processor.processAdd(command);
      } finally {
      // Close the reader
      …
      }
    }
```

If you want to view this example, just open the command line in the folder of the project associated with this chapter, and run the following command:

```
# mvn cargo:run -P flatIndexer
```

You can do the same with Eclipse by creating a new Maven launch as previously described. In that case, you will also be able to put debug breakpoints in the source code (your source code and the Solr source code) and proceed step by step in the Solr index process.

Once Solr has started, open another shell, change the directory to go to the project folder, and run the following command:

```
# curl http://127.0.0.1:8983/solr/flatIndexer/update?commit=true -H
"Content-type:text/plain" --data-binary @src/solr/solr-
homes/flatIndexer/example-input-data/books.flat
```

You should see something like this in the console:

```
[UpdateHandler] start
commit{,optimize=false,openSearcher=true,waitSearcher=true,
expungeDeletes=false,softCommit=false,prepareCommit=false}
…
[SolrCore] SolrDeletionPolicy.onCommit: commits: num=2
[SolrCore] newest commit generation = 4
[SolrIndexSearcher] Opening Searcher@77ee04bb[flatIndexer] main
[UpdateHandler] end_commit_flush
```

Now open the administration console at `http://127.0.0.1:8983/solr/#/flatIndexer/query`, and click on the **Execute Query** button. You should see three documents on the right pane.

You can find the source code of the entire example under the `org.gazzax.labs.solr.ase.ch2.handler` package of the source folder in the project associated with this chapter.

Troubleshooting

This section provides suggestions and tips on how to resolve some common problems encountered when dealing with indexing operations.

Multivalued fields and the copyField directive

The cardinality of a field can be tricky, especially when used in conjunction with `copyField` directives, where two or more single-valued fields are copied to another field, like this:

```
<field name="author_person" … required="true"/>
<field name="author_corporate" … required="true"/>
<field name="author_search" … multiValued="true"/>

<copyField source="author_person" dest="author_search"/>
<copyField source="author_corporate" dest="author_search"/>
```

In this case, the destination field must be multivalued. Otherwise, there will be two values for two different source fields, and Solr will refuse to index the whole document, showing **ERROR multiple values encountered for non multiValued field author_search**.

The copyField input value

A common misunderstanding with the copyField directive is related to the value that is being copied from the source to the dest field. Suppose you define field `A`, field `B`, and a `copyField` directive from `A` to `B`:

```
<field name="A" type="text_without_stopwords" … />
<field name="B" type="light_stemmed_text" … />
<copyField source="A" dest="B"/>
```

Irrespective of the text analysis we defined for field `A` and field `B`. Field `B` will get the stored value of field `A`, without any text analysis applied. In other words, the incoming value for the field `A` is copied verbatim to field `B` before any analysis text can be associated with that field.

So, if we have a value of "one and two" for field A, "and" is considered as a stop word. The "one and two" value is injected into field A, which will trigger the text analysis for the text_without_stopwords type, therefore resulting in an indexed value (for field A) composed of two tokens: "one", "two" ("and" has been removed).

Next, the value original value of field A ("one and two") is copied to field B, triggering the text analysis associated with that field.

Required fields and the copyField directive

A required attribute on a static field denotes that an incoming document must contain a valid value for that field. If a field is the target or destination of a `copyField` directive the required attribute means that in some way, there should be a value for that field coming from its sources. See the following example:

```
<field name="A" … required="false"/>
<field name="B" … required="false"/>
<field name="C" … required="true" multiValued="true"/>
<copyField src="A" dest="C"/>
<copyField src="B" dest="C"/>
```

Fields `A` and `B` are not required and they are copied in field `C`. Since the field `C` is mandatory, you have to make sure that, for each input document, at least `A` or `B` will have a valid value, otherwise Solr will complain about a missing value for field `C`.

Stored text is immutable!

A stored field value is the text that comes from the Solr (Input) document. It will be copied verbatim because it arrives without any changes. Any text analysis configured in the schema for a given field type won't affect that value.

In other words, the stored value won't be changed at all by Solr during the index phase.

Data not indexed

The design of `UpdateRequestProcessor` follows the decorator pattern, consisting of a nested chain of responsibility where each ring is executed one after the other. Your custom `UpdateRequestProcessor` will get a reference to the next processor in the chain during its life cycle. Once its work has been done, it is crucial to forward the execution flow to the next processor. Otherwise, the chain will be interrupted and no data will be indexed.

Summary

In this chapter, we saw the main concepts of the indexing phase in Solr. Being an inverted-index-based search engine, Solr strongly relies on the indexing phase by allowing a customizable and tunable index chain.

The Solr write path is a chain of responsibility consisting of several actors, each of them with a precise role in the overall process. While you must know, configure, and control those components as a user, you must also be aware of their high level of extensibility (as a developer). This allows you to adapt and eventually customize a Solr instance according to your specific needs.

We addressed the concepts that form the Solr data model, such as documents, core, schema, fields, and types. We also looked at the indexing configuration and the involved components such as update request processors, update chains, and request handlers. We finally described how to configure these components and write extensions on top of them.

The purpose of the indexing phase and the index itself is to optimize speed and performance in finding relevant documents during searches. Hence, the whole process is not useful without the search phase, which is the subject of the next chapter.

3
Searching Your Data

Once data has been properly indexed, it's definitely time to search! The indexing phase makes no sense if things end there. Data is indexed mainly to speed up and facilitate searches.

This chapter focuses on search capabilities offered by Solr and illustrates the several components that contribute to its read path.

The chapter will cover the following topics:

- Querying
- Search configuration
- The Solr read path: query parsers, search components, request handlers, and response writers
- Extending Solr
- Troubleshooting

The sample project

Throughout this chapter, we will use a sample Solr instance with a configuration that includes all the topics we will gradually describe. This instance will have a set of simple documents representing music albums. These are the first three documents:

```
<doc>
  <field name="id">1</field>
  <field name="title">A Modern Jazz Symposium of Music and
    Poetry</field>
  <field name="composer">Charles Mingus</field>
  …
</doc>
<doc>
```

```
    <field name="id">2</field>
    <field name="title">Where Jazz meets Poetry</field>
    <field name="artist">Raphael Austin</field>
    …
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">I'm In The Mood For Love</field>
    <field name="composer">Charlie Parker</field>
    <field name="genre">Jazz</field>
    …
  </doc>
```

The source code of the sample project associated with this chapter contains the entire Maven project, which can be either loaded in Eclipse or used via the command line. As a preliminary step, open a shell (or run the following command within Eclipse) in the project folder and type this:

```
# mvn clean cargo:run -P querying
```

The preceding command will start a new Solr instance, with sample data preloaded.

The sample data is automatically loaded at startup by means of a custom `SolrEventListener`. You can find the source code under the `org.gazzax.labs.solr.ase.ch3.listener` package.

You can use the page located at `http://127.0.0.1:8983/solr/#/example/query` to try and experiment by yourself the several things we will discuss.

If you loaded the project in Eclipse, under `/src/dev/eclipse` you will find the launch configuration used to start Solr.

Querying

Solr can be seen as a tell-and-ask system; that is, you first put in (index) some data, then it can answer questions you ask (query) about that data. Since the actors involved in these interactions are not humans, Solr provides a formal and systematic way to execute both index and query operations. Specifically, from a query perspective, that requires a specialized language that can be interpreted by Solr in order to produce the expected answers. Such a language is usually called a **query language**.

Search-related configuration

The `solrconfig.xml` file has a `<query>` section that contains several search settings. Most of them are related to caches, a critical topic that will be described in *Chapter 5*, *Administering and Tuning Solr*.

As we already said for the index section, all those parameters have good defaults that work well in a lot of scenarios. This list describes the relevant settings (cache settings are not included):

- **Searcher lifecycle listeners**: Whenever a searcher is opened, it's possible to configure one or more queries that will be automatically executed in order to prepopulate caches.
- **Use cold searcher**: If a search is issued and there isn't a registered searcher, the current warming searcher is immediately used. If this attribute is set to `false`, the incoming request will wait until the warming completes.
- **Max warming searchers**: This is the maximum number of searchers that are warming in parallel. The example configuration contains a value of 2, which is good for pure searcher instances. For indexers (which could be also searchers), a higher value could be needed.

Query analyzers

In the previous chapter, we discussed analyzers. Their meaning here is the same, and the difference resides only in their input value. When we index data, that value is the content of the fields that make up the input documents. At query time, the analyzer processes a value, term, or phrase coming from a query parser and representing a compounding piece of the user-entered query.

In the previous chapter, we used the analysis page to see how text analysis works at index time. That very page has an additional section that can be used to see the same process but using the query analyzer.

Common query parameters

A query to Solr, other than a search string, includes several parameters that are passed using standard HTTP procedures, that is, name/value pairs in the query string, like this:

```
http://127.0.0.1:8080/solr/ch3/search?q=history&start=10&rows=10&sort
=title asc
```

While some of them strictly depend on the component that will be in charge of handling the request, there are sets of common parameters. The following table describes them:

Parameter	Description
`q`	The search string that indicates what we are asking to Solr according to a given syntax.
`start`	The start offset within search results. This is used to paginate search results.
`rows`	The maximum size (that is, number of documents) of the returned page.
`sort`	A comma-separated list of (indexed) fields that will be used to sort search results. Each field must be followed by the keyword `asc` (for ascending order) or `desc` (descending order).
`defType`	Indicates the query parser that will interpret the specific search string. Each query parser has different features and different rules and accepts a different syntax in queries.
`fl`	A comma- or space-separated list of fields that will be returned as part of the matched documents.
`fq`	A filter query. The parameter can be repeated.
`wt`	The response output writer that will determine the response output format.
`debugQuery`	If this is true, an additional section will be appended to the response with an explanation of the current read path.
`explainOther`	The unique key of a document that is not part of search results for a given query. Solr will add a section to the response explaining why the document associated with that identifier has been excluded from search results.
`timeAllowed`	A constraint on the maximum amount of time allowed for query execution. If the timeout expires, Solr will return only partial results.
`cache`	Enables or disables query caching.
`omitHeader`	By default, the response contains an information header that contains some metadata about the query execution (for example, input parameters or query execution time). If this parameter is set to true, then the header is omitted in the response.

The following are some examples queries:

```
http://localhost:8983/solr/example/query?q=charles&fq=genre:jazz&rows
=5&omitHeader=tue&debugQuery=true
```

```
http://localhost:8983/solr/example/
query?q=charles&rows=10&omitHeader=tue&debugQuery=true&explainOther=2
```

```
http://localhost:8983/solr/example/query?q=*:*&start=5&rows=5
```

As you can imagine, the `q` parameter, which contains the query, will be very important in this chapter. Besides this, there are two other parameters—`fl` (field list) and `fq` (filter queries)—that will be described in the next sections, because they have some interesting aspects.

Field lists

The `fl` parameter indicates which fields (among fields that have been marked as **stored**) will be returned in documents within a query response. Think of these two scenarios:

- A schema that contains a lot of fields, probably defining multiple entities (that is, books and authors). I'm looking for books so I don't want to see any author attributes (and vice versa).
- A schema that contains stored fields with a lot of text, used for the highlighting component, for example (it requires that highlight snippets come from a stored field). When I execute queries I don't want those fields to be returned as part of the matching documents. In other words: I want to exclude those fields from search results.

The `fl` parameter specifies the list of fields that will compound each matched document, thus filtering out unwanted attributes. The parameter accepts a space- or comma-separated list of values, where each value can be any of the following:

- A field name (for example, title, artist, released, and so on).
- The literal `score`, which is a virtual field indicating the computed score for each document.
- A glob, which is an expression that dynamically matches one or more fields by means of the `*` and `?` wildcard characters (for example, `art*`, `r?leas?d`, and `re?leas*`).

- The asterisk (`*`) character, which matches all available (that is, stored) fields.
- A function that, when evaluated, will produce a value for a virtual field that will be added to documents.
- A transformer. Like a function, this is another way to create virtual fields in documents, with additional data such as the Lucene document ID, shard identifier, or the query execution explanation.

Explicit fields, score, functions, and transformers can be aliased by prefixing them with a name that will be used in place of the real name of that member.

SOLR-3191 tracks the activity related to a so-called field exclusion feature. Once this patch has been applied, it will be possible to explicitly indicate which fields must not be part of the returned documents.

The following table lists some examples of the `fl` parameter:

Example	Description
`*, score`	All stored fields and the `score` virtual field
`t*,*d`	All fields starting with `t` and ending with `d`
`max(old_price, new_price)`	Maximum value between `old_price` and `new_price`
`max_price:max(p1, p2)`	A function alias
`title, t_alias:title, [docid]`	Title, aliased title, and a transformer

The difference between the third and fourth examples in the preceding table is in the name of the field that will hold the function value. In the first case, it will be the function itself; in the other, it will be a virtual field called `max_price`.

With the sample instance running, you can try these examples by issuing a request such as `http://127.0.0.1:8983/solr/example/query?q=id:1&fl=`, replacing the value of the `fl` parameter.

A complete list of available functions can be accessed at `http://wiki.apache.org/solr/FunctionQuery#Available_Functions`.

A complete list of available transformers can be read at `https://cwiki.apache.org/confluence/display/solr/Transforming+Result+Documents`.

Filter queries

Filter queries operate a kind of intersection on top of documents, resulting from the execution of the main query. A filter query is like having a required condition in your main query (that is, an additional clause concatenated with the `AND` operator), but with some important differences:

- It is executed separately and before the main query
- The filter and the intersection are applied on top of the main query results
- It doesn't influence the score of the documents, which is computed in the execution of the main query
- The results of filter queries are cached separately so that they can be reused for further executions

There can be more than one `fq` parameter in a search query. In this case, the result of the overall execution will take into account all filter clauses, therefore resulting in documents that satisfy the intersection between the main results and the results of each filter query.

Filter query caching is one of the most crucial features of Solr. A filter query's design should reflect the access pattern of requestors as much as possible. Consider this filter query:

```
fq=genre:Jazz AND released:1981
```

The preceding query will cache the results of those two clauses together. So, if your application provides two separate filters (for the end users), `genre` and `released`, the following filter queries won't benefit from this cache, and they will be cached (again) separately:

```
fq=genre:Jazz
fq=released:1981
```

In this situation, the first query should be rewritten in the following way, allowing reuse of the cache associated with each filter query:

```
fq=genre:Jazz&fq=released:1981
```

Query parsers

A query parser is a component responsible for translating a search string or expression into specific instructions for Solr. Every query parser understands a given syntax for expressing queries.

Solr comes with several query parsers, giving the requestors a wide range of ways of asking what they need.

The Solr query parser

The Solr query parser, often mistakenly called Lucene query parser, is implemented in `org.apache.solr.search.LuceneQParserPlugin`. It is rather a schema-driven superset of the default Lucene query parser.

Note the `Plugin` suffix of the class name. Solr provides an extensible framework for creating and plugging in your own query parser.

The following sections will describe the relevant aspects of this parser.

Terms, fields, and operators

You've already met terms. They are atomic units of information resulting from an analysis applied to given text. At index time, that text is the value of a field belonging to a given (input) document. At query time, terms come from the user-entered query string. Specifically, a query string is broken into terms, fields, and operators.

Terms can be simple or compound terms; for example, they can be single words such as CM, Standard, and 1959 or phrases such as "Goodbye Pork Pie Hat." Phrases are two or more words surrounded by double quotes.

Fields are what we declared in the `schema.xml` file. Their use within a search string allows a requestor to express instructions such as "search `x` in field `y`" where x is a term or a phrase and y is the field name. Here are some examples of the use of fields:

```
title: "Where Jazz meets Poetry"
composer: Mingus
```

Operators are keywords or symbols used as conjunctions between several field-value criteria in order to create complex expressions, such as this:

```
title:Jazz OR composer:Charlie AND released:1959
genre:Jazz AND NOT released:1959
```

The following table describes the available operators:

Operator	Description
AND	A conjunction between two criteria, both of which must be satisfied
OR	A conjunction between two criteria where at least one must be satisfied
+	Marks a term as required
-/NOT	Marks a term as prohibited

It's also possible to use a pair of parentheses to group several fields or values criteria, like this:

```
(released:1957 AND composer:Mingus) OR (released:1976 AND NOT
genre:Jazz) OR released: (1988 OR 1959)
```

Boosts

Boosting allows you to control the relevance of a given matching document, thus offering a way to give to some query results more importance than others; for example, if you are mainly interested in Jazz and less in Fusion albums, you could use this:

```
+genre:Fusion +genre:Jazz^2
```

The boost factor is inserted after a field value criterion and prefixed with a caret symbol. It has to be greater than 0, and since it is a factor, a value between 0 and 1 represents a negative boost. If it is absent, a default boost factor of 1 will be applied.

Wildcards

The wildcard characters, `*` and `?`, can be used within terms, with zero or more occurrences. They cannot be applied to compound terms (that is, search phrases) or numeric and date types. The `?` wildcard matches a single character, while the `*` matches zero or more sequential characters. Here are some examples of wildcards:

```
(title:moder* AND artist:Min*) OR artist:(Yngw?e AND M?lm*)
```

Fuzzy

The tilde symbol (~) at the end of a term enables a so-called fuzzy query, allowing you to match terms that are similar to that term. Fuzzy logic is based on the **Damerau-Levenshtein** distance algorithm. After the tilde, you can put a value between 0 and 2, indicating the required similarity (2 means high similarity is required). The default value that is used if the parameter is not given is 0.5.

With the example Solr instance running, open the query page in the admin console and type the following query:

```
artist:Charles~0.7
```

The query response will contain two results. The first is an album of Charles Mingus, that is a perfect match with the search term entered. The second artist is *Charlie* Parker, whose name is similar but not equal to `Charles`.

Proximity

The same symbol that is used for a fuzzy query has a different meaning when used in conjunction with phrase queries. Now run the following query:

```
title:"Jazz Poetry"
```

You won't get any result because there's no record with those two consecutive terms in the title. Using a tilde followed by a number, which expresses a distance between terms, you can enable a proximity search, allowing matches of documents that have those two terms within a specific distance from one another.

This query will match the document that has *Where Jazz meets Poetry* as its title:

```
title:"Jazz Poetry"~2
```

The following query will also match the document that has *A Modern Jazz Symposium of Music and Poetry* as the title:

```
title:"Jazz Poetry"~4
```

Ranges

Range searches allow us to specify for a given field a set of matching values that fall between a lower and a higher bound, inclusive or exclusive of those bounds. Here are some examples of ranges:

```
released:[1957 TO 1988]
released:[1957 TO *]
released: [* TO 1988]
released:{1957 TO 1988}
released:[1957 TO 1988}
genre: [Jazz TO NewAge]
```

You can see that the lower and higher bounds can be literal values, as shown in the first example, where we are searching for albums released between `1957` and `1988`. The bounds can also be wildcards, as shown in the second and third examples. Square and curly brackets are used to denote an included or an excluded bound, respectively. So, in the first example, both `1957` and `1988` are included; in the fourth example they are excluded.

Keep in mind that, for non-numeric fields (as shown in the fifth example in the preceding code snippet) sorting is done lexicographically. Therefore, a sequence such as `1`, `02`, `14`, `100` will result in `02`, `1`, `100`, `14` using the lexicographic order, which is very different from a numeric sort.

The Disjunction Maximum query parser

The Solr query parser is powerful when it comes to building complex expressions. However, those are quite far from what the user usually types in a search field.

Think about the Google search page. What do you type in the search text field? Not an expression, but just one, two, or more terms associated with what you're looking for.

The **Disjunction Max** (**DisMax**) query parser directly processes those user-entered terms and searches for each of them across a set of configurable target fields, with a configurable weight for each field.

The DisMax parser is enabled by setting the `defType` parameter to `dismax`.

The example Solr instance has a request handler listening to `/glike1` that uses the DisMax parser.

Other than search terms, this query parser supports some features of the Solr query parser, such as quotes, that can be used to indicate phrases, and the `+` and `-` operands to mark mandatory and prohibited terms, respectively. All other term modifiers we saw for the Solr query parser are escaped, so they will be interpreted as search terms.

The name of the parser comes from its behavior:

- **Dis**: This stands for disjunction, which means that, for each word in the query string, the parser builds a new subquery across fields and boosts specified in the `qf` parameter. The resulting queries are subjected to the first (required) constraint defined with the `mm` parameter, and a set of optional clauses defined with other parameters, which we will see later.

- **Max**: This means maximum, and it pertains to the scoring computation. The DisMax parser scores a given document by getting the maximum score value among all matching subqueries.

The following sections describe the several parameters that the parser accepts.

Query Fields

The `qf` parameter indicates a set of target fields with their corresponding (optional) boosts. Fields are separated by spaces, and each of them can have an optional boost associated with it, hence resulting in expressions such as this:

```
qf = title^3.5 artists^2.0 genre^1.5 released
```

Here, we want to search across four fields, each of them with a different importance, which will affect the score assigned to each matching document. The `qf` parameter is one of the main places where we define our search strategy, depending on customer requirements.

In OPACs, there's a never-ending debate about which is the more relevant attribute among titles and subjects. A title, as you can imagine, is important, but couldn't contain terms that are representatives of a work. A subject is a kind of controlled classification assigned by a professional user (that is, a librarian). As a search service provider, you can use the `qf` parameter to configure boosts, depending on customer needs, and avoid entering that debate!

The DisMax query parser has another interesting feature when searching fields declared in the `qf` parameter: when those fields are numeric or dates, inappropriate terms are dropped. Returning to the `qf` expression, consider searching for this:

```
Mingus 1962
```

For the `title`, `artist` and `genre` fields, Solr will build two queries. But for the `released` field, it will create just one query using the `1962` word, thus resulting in a total of 7 queries:

```
title:Mingus^3.5, artist:Mingus^2.0, genre:Mingus^1.5,
title:1962^3.5, artist:1962^2.5, genre:1962^1.5, released:1962
```

As you can see, the `released:Mingus` query has been dropped because `released` is a numeric field.

Alternative query

The `q.alt` optional parameter defines a query that will be used in the absence of the main query.

The `q.alt` query is parsed by default using the Solr query parser, so it accepts the syntax we described in the previous paragraph. Using `LocalParams`, you can change the `q.alt` parser.

Minimum should match

Every word or phrase that is a part of the search string, unless it is constrained by the `+` or `-` operators (and therefore, marked as required or prohibited), is considered as optional. For those optional parts, the `mm` parameter defines the minimum number of matches that satisfy the query execution. The interesting point here is that other than accepting a quantity or a number, this parameter also allows complex expressions. The following table illustrates some examples of `mm`:

Value	Description
An integer (for example, `3`)	At least the given number of optional clauses must match.
A percentage (for example, `66%`)	At least the given percentage of optional clauses must match.
A negative number or a negative percentage	The number of optional clauses that must match is the result of subtracting the given value from the total number of optional clauses (absolute or 100 percent depending on the parameter value).
One or more expressions with the *X* < | > *Y* format	If there are less than X optional clauses, they must match. If clauses are greater than X, then Y must be used as the `mm` value. Y can be a positive or negative integer or a percentage value. It is also possible to concatenate several expressions, like this: 3< 75% 6<-1 This means that, with three optional clauses, all of them are required. Between 4 and 6 optional clauses, we require a match of 75 percent. Finally, for more than six clauses, we require a match of all clauses but one.

The several subqueries resulting from search terms parsing are constrained with the `mm` parameter (specifically, an additional Boolean query acting as a constraint is concatenated with the `AND` operator), so matching documents that don't satisfy the `mm` constraint won't be part of the search results.

Phrase fields

Once the list of matching documents has been populated according to the search criteria and constraints (for example, `mm` or filter queries), the `pf` parameter raises the score of documents that have search terms in proximity.

As the `qf` parameter, `pf` can declare a list of fields with an optional boost factor.

Query phrase slop

The `qs` parameter indicates a proximity factor to be used in those phrase queries that are eventually included in the search string.

Phrase slop

The `ps` parameter indicates a proximity factor to be used in phrase queries built for `pf` fields. Note that such queries will be executed only to boost results (see the previous section), so this parameter doesn't affect matching but only boosting.

Boost queries

The `bq` parameter defines a query parsed by the Solr query parser that will additionally boost search results. It can be repeated, thus allowing one or more queries.

If, for example, you want to give more importance to items with a price that falls within a given range, you can use a boost query like this:

```
price:[10.00 TO 19]
```

Additive boost functions

The `bf` parameter defines a function that will additionally boost search results by adding its value to the computed score. As with the `bq` parameter, it can be repeated in order to have multiple functions.

Tie breaker

The `tie` parameter is a float number. It has a value between 0 and 1, and it affects the strategy used by the parser to determine the final score of a given (matching) document.

The Disjunction Max parser, as said before, executes a set of subqueries on top of the fields declared in the `qf` parameter. The subquery that has the maximum score determines the score of the document. So schematically:

```
documentScore = score of matching sub query with highest score
```

However, you could end up with two documents getting the same score, because the maximum value computed by each winner subquery is the same.

The `tie` parameter lets you take fine-grained control of the final score assigned to each document, by including the score of all matching subqueries in the computation. Those additional scores are multiplied by a factor, the `tie` value. So, the preceding formula becomes the following:

```
documentScore = (score of matching sub query with highest score) +
( (tie) * (scores of other matching sub queries) )
```

With a value of 0.0, we will have a pure **disjunction max query**, where only the maximum score is included. A value of 1.0 will lead to a **disjunction sum query**, where the final score is the sum of the scores of all matching subqueries.

The Extended Disjunction Maximum query parser

This parser (**eDisMax**) is built on top of the **DisMax** parser and has some additional features such as fielded search, Boolean operators, term modifiers, and better handling of mistakes in queries.

The eDisMax parser can be enabled by setting the `defType` parameter to `edismax`.

The example Solr instance has a request handler listening to `/glike2` that uses the eDisMax parser.

The following sections describe additional parameters that this parser accepts. All parameters described in the DisMax parser section are included.

Fielded search

The eDisMax parser supports the full syntax of the Solr query parser, therefore allowing a so-called fielded search (that is, `title:Jazz`) with Boolean operators and term modifiers (for example, fuzzy and proximity).

In addition, this parser supports field aliasing and renaming. This allows you to give an interaction view to the requestor (for example, an end user, a query client, and so on) that is partially or completely decoupled from Solr's underlying data model.

Aliasing is done using the following syntax:

```
f.<alias>.qf = (one or more real fields with optional boosts)
```

Here, `<alias>` is the virtual name that will be associated with the field (or fields) declared on the right operand. As you can see, an alias can be applied to single fields or to a group of fields. When aliases are declared, requestors can use them in their queries.

We can use aliases to localize field names:

```
f.artista.qf = artist // Italian users will see an "artista" field
f.kunstler.qf = artist // for German users
```

We can also use them to create metafields that group a set of real fields:

```
f.people.qf = author, illustrator, editor, translator
f.titles.qf = title, front_cover_title, sub_title, uniform_title
```

Phrase bigram and trigram fields

Other than supporting the `pf` parameter we have already seen for DisMax, this parser adds two optional features. The `ps` parameter boosts the score of documents where input terms appear in proximity. The `pf2` and `pf3` parameters offer the same feature but by splitting the input terms in consecutive bigrams and trigrams, respectively. Therefore, the `All the things you are` input string will become the following set of (consecutive) bigrams:

```
All the, the things, things you, you are
```

For the same logic, it will become the following set of trigrams:

```
All the things, the things you, things you are
```

Phrase bigram and trigram slop

As `ps` sets the phrase slop for the `pf` parameter, `ps2` and `ps3` do the same for `pf2` and `pf3`. If they are absent, the value of `ps` is used.

Multiplicative boost function

The `boost` parameter declares one function as the `bf` parameter, as we have seen for the DisMax parser. The difference here is that the function value is multiplied (not added) by the computed score.

User fields

The `uf` parameter specifies which fields (real or virtual) the requestors are allowed to use in their queries. Used in conjunction with aliasing, it allows you to completely hide real fields and have queries with only virtual (that is, aliased) fields.

Lowercase operators

In plain Solr query parser syntax, operators need to be in uppercase (`AND`, `OR`). The `lowercaseOperators` flag parameter, which defaults to `true`, allows us to interpret as operators lowercase tokens (and, or).

At the time of writing this book, only the `and` and `or` Boolean operators are affected by this parameter. The `NOT` operator is not handled, and therefore, the lowercase word **not** is parsed as a literal term, even if `lowercaseOperators` is set to `true`. The Jira issue at `https://issues.apache.org/jira/browse/SOLR-3580` tracks the activity on this topic.

Other available parsers

There are a lot of other available parsers, as listed in the following table:

Parser	Code	Description
Lucene query parser	`lucene`	The Lucene query parser has more or less the same features as the Solr query parser. However, this is the Lucene-specific implementation.
Function query parser	`func`	Creates a function query from the input string.
Join query parser	`join`	Normalizes relationships between documents by emulating a join.
Term query parser	`term`	Creates a single-term query from the input string.
Boost query parser	`boost`	Creates a boosted query from the input string. An additional parameter, `b`, is required to indicate the boost function.

Parser	Code	Description
Raw query parser	`raw`	Creates a term query from the input string without any text analysis.
Spatial filter query parser	`geofilt`	Enables spatial queries.
Field query parser	`field`	Create a field query from the input string.
Surround query parser	`surround`	Creates a surround query. This query is used for proximity searches.

Besides all of this, the query parser framework has been conceived with extensibility in mind, so developers are free to implement, register, and use their own query parsers.

Search components

A search component is a reusable module that contributes to search results. While defining a search handler, that is, a controller for a given kind of search, you can customize its behavior by defining and configuring search components that will contribute to its output results.

Search components must be declared and used within `solrconfig.xml`, the main Solr configuration file. A component declaration requires a name, the implementation class, and a set of optional initialization parameters:

```
<searchComponent name="prices" class="a.b.c.MyComponent">
    <str name="ds-jndi">jdbc/datasource</str>
    <str name="service-uri">http://example.org#me</str>
</searchComponent>
```

Once declared, these can be used within request handlers, which are the runtime controllers of the executions of requests (we will cover request handlers later in the chapter).

There are some predefined search components that mustn't be explicitly declared in `solrconfig.xml`.

That doesn't mean they are automatically enabled. They must be explicitly activated or disabled, depending on their default state.

The default components are those components that are responsible for absolving the fundamental or common steps of a query execution flow. This is the reason there's no need to declare them explicitly, unless you want to use a different configuration. In the following sections, we will illustrate these components.

Query

The query component is responsible for parsing and executing a query. This is the component that accepts query and query parser parameters, gets a reference to the appropriate query parser, coordinates the parser in order to produce a query, executes that query, and outputs a corresponding response.

Facet

This component enables the so-called **faceted search**. It contributes to search results by adding a set of configurable aggregations called **facets**.

When you execute some search, you will get back a single page of results consisting of a certain number of matching documents. Enabling faceting allows you to get an additional perspective of the overall data, consisting of a set of aggregations. The following screenshot shows some Solr-powered facets in action on a website, on the right side:

The facet component can be activated by specifying a `facet` parameter with one of the following values: `yes`, `true`, or `on`.

Solr provides several types of facets: queries, fields, ranges, pivot, and interval. Each of them, whenever enabled, will add a dedicated section to the response.

Facet queries

The `facet.query` parameter declares a query (parsed by the Solr query parser) that will be used as a facet with the corresponding counts. The results (that is, counts) of this facet will be in a specific response section called **facets_queries**. The parameter can be repeated multiple times, allowing us to specify several queries. Using the example dataset, with Solr running, open a browser and type `http://127.0.0.1:8983/solr/example/select?q=*:*&facet=true&facet.query=genre:jazz`

In the XML response, you will see matching documents within the `<result>` tag, and an additional section dedicated to facets:

```
<lst name="facet_counts">
  <lst name="facet_queries">
    <int name="genre:Jazz">3</int>
  </lst>
  <lst name="facet_fields"/>
  <lst name="facet_dates"/>
  <lst name="facet_ranges"/>
</lst>
```

Here, you can see that three documents match the facet query. The other facet sections are empty because we didn't ask for them.

Facet fields

Facet fields are surely the most popular kind of facets. They aggregate search results using a set of given and configurable fields.

Remember that a field must be declared as indexed in the schema in order to be faceted.

Other than activating the facet feature for a given field, Solr has a rich set of parameters that can be used to tune and configure the field's faceting behavior. These settings can be specified for all fields or for a given field. For the first case, the following table illustrates the available parameters, their names, and meanings. For field-specific settings, the same parameters must be declared with the following convention:

```
f.<field>.<parameter> = <value>
```

In this way, the value associated with parameter will be valid only for the specific field.

Parameter	Description
`facet.field`	Declares a field that will be used as a facet. This parameter must be repeated for each facet field.
`facet.prefix`	Limits the terms used in faceting to values that begin with a given prefix.
`facet.sort`	The sort strategy of counts within each facet. Only two values are allowed: `count`, which means order by count, and `index`, which means lexicographic order.

Parameter	Description
`facet.limit`	The maximum number of counts that can be returned for each facet. A value of `-1` will return all available counts.
`facet.offset`	Specifies a start offset within the available counts of facets.
`facet.mincount`	The minimum count needed for a field to be included in the response.
`facet.missing`	Includes in the response the count of documents that match the query but don't have a value for a given facet.
`facet.method`	The type of algorithm that Solr will use to compute facets.
`facet.threads`	The number of parallel workers (that is, threads) that will compute the facets.

Returning to our previous example, let's remove the facet query and use some additional parameters so that facet fields will be built (for simplicity, only the query string is reported):

```
q=*:*&facet=on&facet.field=genre&facet.minCount=1
```

In the facet sections, you will see the genre facets under the `facet_fields` subsection:

```
<lst name="facet_fields">
<lst name="genre">
  <int name="Progressive Rock">10</int>
  <int name="Rock">5</int><int name="Fusion">4</int>
  <int name="Heavy Metal">4</int>
  …
  <int name="Pop metal">1</int></lst>
</lst>
```

We asked for the `genre` facet and we set `mincount` to `1`, which means that facets with no counts are excluded from the response. It is important to underline the fact that the displayed value for a facet field is its indexed value, and not the stored value (that is, the value that is copied verbatim as it arrives in input documents). In the previous example, the `genre` field is `String`, and therefore, it is not tokenized. This is the reason you see the compound term (`Progressive Rock`) as one of its values. If that field had been declared as `TextField` and tokenized with `WhiteSpaceTokenizer`, you would have seen two different values for that facet (assuming no further filtering): `Progressive` and `Rock`.

Facet ranges

Facet ranges can be applied to numeric or date fields. As the name suggests, with facet ranges, Solr creates a facet classification based on ranges. The following parameters control this kind of faceting:

Parameter	Description
`facet.range`	Declares a field that will be used as the facet range. The parameter must be repeated for each facet field.
`facet.range.start`	Declares the start of the facet interval.
`facet.range.end`	Declares the end of the facet interval.
`facet.range.gap`	The size of each step between the start and the end of the interval.

The following is a sample query that uses facet ranges for faceting albums by release date:

```
q=*:*&facet=on&facet.range=released&facet.range.start=1950&facet.
range.end=2000&facet.range.gap=10
```

That will add another section within the `facet_counts` element:

```
<lst name="facet_ranges">
  <lst name="released">
    <lst name="counts">
      <int name="1950">1</int>
      <int name="1960">1</int>
      <int name="1970">6</int>
      <int name="1980">8</int>
      <int name="1990">5</int>
    </lst>
    …
  </lst>
</lst>
```

Pivot facets

We previously described facet fields; they provide the ability to aggregate search results by one or more categories. Pivot facets go a step ahead in that direction. They allow us to analyze data in multiple dimensions, breaking down the faceted values by subsequent, nested subcategories.

This kind of faceting can be activated through a request like this:

```
q=*:*&facet=on&facet=true&facet.pivot=genre,released
```

The `facet.pivot` parameter can be repeated multiple times. For each repetition, there will be a dedicated and aggregated result within the **facet_pivot** section of the response. Here, for simplicity, we put just one parameter with two categories, `genre` and `released`. The following example is an extract of the response you will get using the sample instance associated with this chapter:

```
<lst name="facet_pivot">
<arr name="genre,released">
  <lst>
    <str name="field">genre</str>
    <str name="value">Progressive Rock</str>
    <int name="count">10</int>
    <arr name="pivot">
      <lst>
        <str name="field">released</str>
        <int name="value">1992</int>
        <int name="count">2</int>
      </lst>
      <lst>
        <str name="field">released</str>
        <int name="value">1969</int>
        <int name="count">1</int>
      </lst>
  <lst>
  <str name="field">genre</str>
  <str name="value">Rock</str>
```

```
<int name="count">5</int>
<arr name="pivot">
  <lst>
    <str name="field">released</str>
    <int name="value">1969</int>
    <int name="count">1</int>
  </lst>
  <lst>
    <str name="field">released</str>
    <int name="value">1986</int>
    <int name="count">1</int>
  </lst>
  …
```

As you can see, the `genre` facet is broken down by a nested `released` category. Note that the preceding nested structure is returned with just one request-response interaction. In order to get the same result with classic facet fields, you should query Solr several times with incremental filters. That's the reason the pivot facets feature, acting as a façade and hiding all of that interaction complexity, is very useful for navigating the hierarchy of those aggregations. However, it should be used carefully, as it could have an impact on performance.

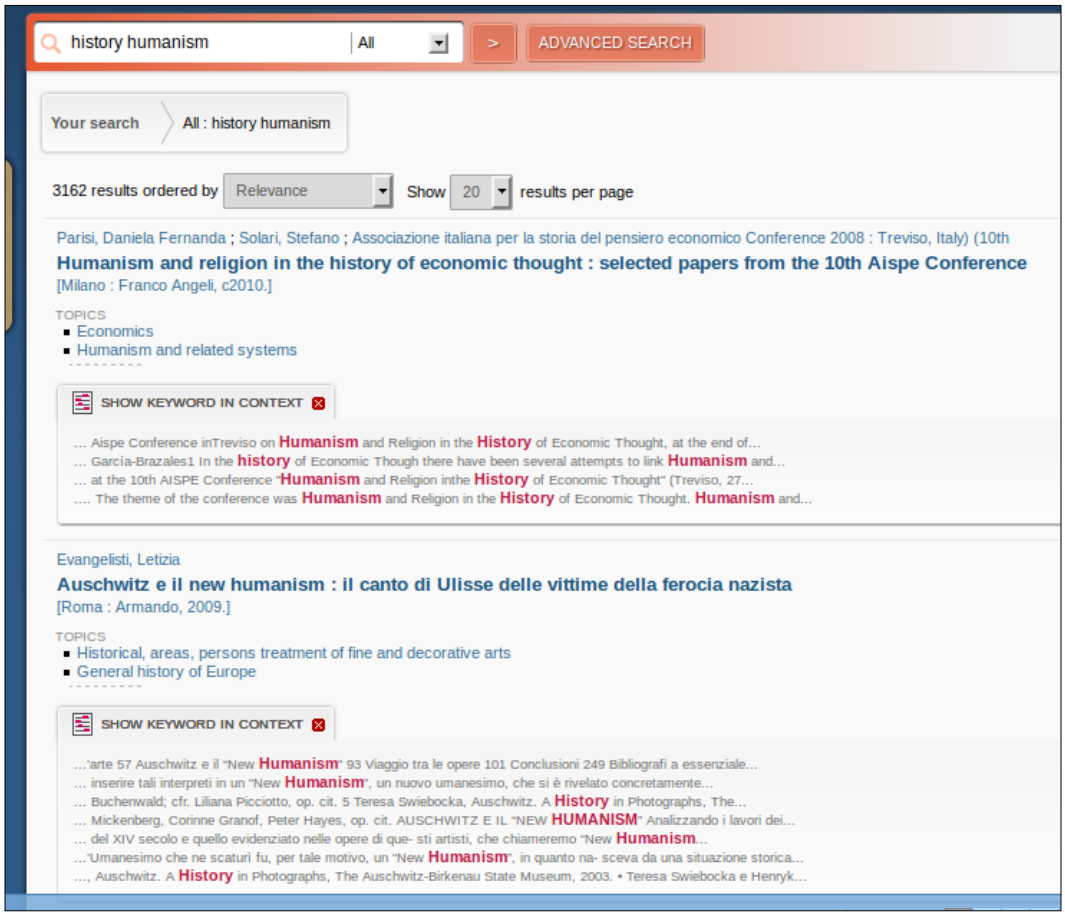
Interval facets

Interval facets were introduced in **Solr 4.10**. They can be seen as an alternative to facet (range) queries because they allow you to set interval criteria for one or more fields, and count the number of matching documents that have values within those constraints.

Although the same result can be achieved with facet range queries, this implementation could provide performance improvement in several contexts. As suggested in the Solr reference guide, it is recommended that you try both the methods.

Highlighting

The highlight component contributes to search results by adding a section that contains (for each document in the current result page) a set of snippets highlighting the search terms that are in the document content (that is, in one or more fields of the document). The following screenshot shows a web application that uses the highlighting feature:



This feature is particularly useful when your data comes from rich documents such as PDFs or Microsoft Office documents (as shown in the preceding example). Using the highlighting feature, it's possible to give the end user an approximate idea of the context where, within the document, entered terms have been found.

Within the example Solr instance associated with this chapter, there is a request handler called `/highlight` that enables this feature on `title` and `artist` fields.

The highlighting component can be tuned, or configured, with several parameters. Fortunately, the provided default values work well in many scenarios. Some of those parameters are described in the following table:

Parameter	Description
`hl`	Turns highlighting off or on. The default value is `false`.
`hl.q`	Terms to be highlighted are taken from the main query unless this parameter, which itself requires a query, is specified.
`hl.fl`	A space- or comma-separated list of fields that will be used for highlighting. Snippets will come only from these fields.
`hl.snippets`	The number of highlighting snippets that will be returned. The default value is `1`.
`hl.maxAnalyzedChar`	The maximum number of characters that will be inspected (in a given field) to compute the snippets.
`hl.simple.pre/` `hl.simple.post`	Indicates text that should appear before and after a highlighted term. They default to `<em>` and `</em>` HTML tags, respectively.

Solr comes with three different kind of highlighters, described in the following sections.

Standard highlighter

This is the first highlighter that was introduced in Solr. Solr uses it by default. It is able to work on top of a lot of query types and doesn't have any special requirement on fields to be highlighted. However, in order to speed up its work, `termVectors` should be turned on (for those fields).

Fast vector highlighter

Fast vector highlighter is the second type of highlighter introduced in Solr. It requires that `termVectors`, `termPositions`, and `termOffsets` are turned on for each field that needs to be highlighted. That allows fast and scalable execution, especially with documents containing large amounts of text, but requires a lot of extra space for the index. However, it supports few query types.

The fast vector highlighter can be enabled by setting the `hl.useFastVectorHighlighter` parameter to `true`.

Note that, if the preceding flags are not set for target fields, Solr will continue to use `StandardHighlighter`.

Postings highlighter

This highlighter doesn't use term vectors, nor does it reanalyze the text to be highlighted. It only requires the `storeOffsetsWithPositions` flag set for the fields to be highlighted. Unlike the others, this highlighter must be explicitly declared in the `solrconfig.xml` file with the following declaration:

```
<searchComponent class="solr.HighlightComponent" name="highlight">
  <highlighting
    class="org.apache.solr.highlight.PostingsSolrHighlighter"/>
</searchComponent>
```

This is a good compromise, compared with the first two highlighters, in terms of performance and index space. The information (that is, the posting offsets) required by the `storeOffsetsWithPositions` flag is cheaper than term vectors in terms of memory and disk occupation. However, it is supposed to be used to highlight simple query terms, so it could have some unexpected or unwanted results with phrase queries.

More like this

The **more like this** search component allows us to find documents that have some kind of similarity with a given document. There are several ways to use this feature in Solr:

- `MoreLikeThisHandler`: This is a front controller that is completely dedicated to "more like this" requests. It accepts a query that identifies a document, and looks for similar documents according to a configured criterion.
- `MoreLikeThisHandler`: This is similar to `MoreLikeThisHandler`, but instead of taking a document as the input (matched by a given query), the text used to compute similarity can be directly passed or fetched from a URL.
- `MoreLikeThisSearchComponent`: As a search component, it will execute the *similar* search for each document of the current result page, thus appending a **more like this** section to the Solr response, with a list of similar documents for each document. This is not really recommended because it could slow down overall query execution.

In general, the first type is the most widely used. `MoreLikeThis` doesn't have special requirements for fields that are to be used for the similarity computation. However, for best performance, `TermVectors` should be enabled for them.

The following table illustrates the parameters accepted by this component:

Parameter	Description
`mlt`	Turns highlighting off or on. It defaults to `false`.
`mlt.count`	The maximum number of similar documents that must be returned (for each document).
`mlt.fl`	The fields used for similarity. They should have `TermVectors` enabled (recommended) or they need to be stored.
`mlt.qf`	A list of space- or comma-separated fields (already declared in `mlt.fl`) with corresponding boosts.
`mlt.minwl / mlt.maxwl`	The minimum and maximum word length boundaries. Words whose length is more that these boundaries are ignored.
`mlt.boost`	A flag indicating whether the query will be boosted by the relevance of the interesting terms. It defaults to `false`.
`mlt.mintf`	This is the minimum term frequency boundary. It defaults to `2`.
`mlt.mindf`	This is the minimum document frequency boundary. It defaults to `5`.

Other components

Other than the components we saw in the previous sections, there are other built-in search components that are part of the Solr framework. Remember that, if you want to use them, they will have to be explicitly declared and configured within the Solr configuration.

The following is a short and non-exhaustive list of additional components:

- **Query elevation**: This is used to give more importance to some results using a criterion that has nothing to do with the normal Solr scoring algorithm. The component lets you associate a given query with a corresponding list of most important results.
- **Terms**: This provides access to the Lucene internal term dictionary.
- **Stats**: This provides numeric fields statistics.
- **Spellcheck**: This provides spell checking capabilities by means of n-gram analysis of indexed documents or external dictionaries. From a functional point of view, this component is used to build the so-called "Did you mean?" feature, offering alternative search suggestions in case of user mistakes.

- **Term Vector**: This adds term vectors (that is, term, frequency, position, offset, and IDF) of the matching documents to a request.
- **Debug**: This adds debuging and explanatory information about the request execution.

Search handler

We saw request handlers in the previous chapter. There, we defined a request handler as a *pluggable component that handles incoming requests*. In that chapter, we were referring to update requests, that is, requests containing index update commands.

Here, we will focus our attention on `SearchHandler`, a special front controller used to handle incoming search requests. The `SearchHandler` class, although it could be seen as the supertype layer of all search handlers, is not abstract and it defines a standard search behavior.

Standard request handler

`StandardRequestHandler` is an empty subclass of `SearchHanlder`, so at the time of writing this book, using one of them is basically the same. Request handlers are declared in the `solrconfig.xml` file, and they define search endpoints. Each instance is associated with a given name prefixed by a slash (the name must be unique), an implementation class, and a set of configuration parameters:

```
<requestHandler name="/mySeacher" class="solr.SearchHandler">
  (configuration)
</requestHandler>
```

With the sample Solr instance running, the preceding handler will answer to one of these URIs:

```
http://localhost:8983/solr/example/query

http://localhost:8983/solr/example/facets

http://localhost:8983/solr/example/jazz
```

Configuring a `SearchHandler` instance means defining configuration parameters and (optionally) search components that will participate in the query execution chain.

Search components

Most of the time, unless you have a specific need, the search components that drive the logic of the search execution can be omitted because the following list will be automatically injected:

Code	Component
query	QueryComponent
facet	FacetComponent
mlt	MoreLikeThisComponent
highlight	HighlightComponent
stats	StatsComponent
debug	DebugComponent

Only the "query" component is enabled; the others need to be explicitly activated.

If the default chain is not what you need, it is possible to define a custom chain in the following way:

```
<arr name="components">
  <str>query</str>
  <str>facet</str>
  …other components follow
</arr>
```

This will completely replace the default chain. It is also possible to leave the default chain as it is and have additional prepended or appended components:

```
<arr name="first-components">
  <str>my_custom_component</str>
  …other components follow
</arr>

<arr name="last-components">
  <str>another_custom_component</str>
  …other components follow
</arr>
```

So, in general, the order of execution for search components will be the following:

- Components declared as "first-components" (optional).
- Components declared as "components" In their absence, the default chain will be used.
- Components declared as "last-components" (optional).

The following is an example declaration of `StandardRequestHandler`:

```
<requestHandler name="/jazz" class="solr. StandardRequestHandler">
  <!-- parameters that will be always applied to the
  incoming requests -->
    <lst name="invariants">
      <int name="rows">10</int>
    </lst>

  <!-- parameters that will be always added to the
  incoming requests -->
    <lst name="appends">
      <int name="fq">genre:jazz</int>
    </lst>

  <!-- default settings that can be overridden by the
  incoming requests -->
    <lst name="defaults">
      <str name="sort">title asc</str>
      <str name="echoParams">explicit</str>
      <str name="q">*:*</str>
      <bool name="facet">false</bool>
    </lst>

  <!—This is a custom search component that will run
after the default component chain-->
    <arr name="last-components">
      <str>prices</str>
    </arr>
  </requestHandler>
```

Query parameters

The request handlers and the search components involved in the chain accept several parameters to drive their execution logic. These parameters (with corresponding values) can be declared in three different sections:

- **defaults**: Parameter values will be used unless overridden by incoming requests
- **appends**: Parameter values will appended to each request
- **invariants**: Parameter values will be always be applied and cannot be overridden by incoming requests or by the values declared in defaults and append sections

All sections are optional, so you can have no parameters configured for a given handler and allow the incoming requests to define them. This is an example of a handler configuration:

```
<lst name="defaults">
  <str name="defType">edismax</str>
</lst>
<lst name="appends">
  <str name="facet.field">artist</str>
  <str name="facet">genre</str>
</lst>
<lst name="invariants">
  <str name="wt">json</str>
  <bool name="facet">true</bool>
</lst>
```

RealTimeGetHandler

`RealTimeGetHandler` is basically a `SearchHandler` subclass that adds `RealTimeSearchComponent` to the search request execution. In this way, it's possible to retrieve the latest version of softly committed documents by specifying their identifiers.

In order to enable such a component, you must turn the update log feature on, in `solrconfig.xml`:

```
<updateHandler class="solr.DirectUpdateHandler2">
  <updateLog>
    <str name="dir">${solr.ulog.dir:}</str>
  </updateLog>
  …
</updateHandler>
```

Then the request handler can be declared and configured using the procedure that we saw in the previous section:

```
<requestHandler name="/get" class="solr.RealTimeGetHandler">
  …
</requestHandler>
```

This handler accepts an additional `id` or `ids` parameter that allows us to specify the identifiers of the documents we want to retrieve. The `id` parameter accepts one identifier and can be repeated in requests. The `ids` parameter accepts a comma-separated list of identifiers.

Once the example Solr instance is up, this handler responds to `/get` requests.

Response output writers

As a last step, query results are returned to requestors in a given format. Solr communicates with clients using the HTTP protocol. Those clients are free to start the interaction by asking for one format or another, depending on their needs.

Although a default format can be set, the client can override it by means of the `wt` parameter. The value of the `wt` parameter is a mnemonic code associated with an available response writer.

There are several built-in response writers in Solr, which are described here:

ResponseWriter	Description
`xml`	The eXtensible Markup Language response writer. This is the default writer.
`xslt`	Combines the XML results with an XSLT file in order to produce custom XML documents.
`json`	JavaScript Object Notation response writer.
`csv`	Comma-Separated Value response writer.
`velocity`	This uses Apache Velocity to directly build web pages with query results. It is very useful for fast prototyping.
`javabin`	Java clients have a privileged way to obtain results from Solr using this response writer, which directly outputs Java Objects.
`python`, `ruby`, `php`	Specialized response writers for these languages that produce a structure directly tied to the language requirements.

Extending Solr

The following sections will describe and illustrate a couple of ways of extending, and customizing searches in Solr.

Mixing real-time and indexed data

Sometimes, as a part of your search results, you may want to have data that is not managed by Solr but retrieved from a real-time source, such as a database.

Think of an e-commerce application; when you search for something, you will see two pieces of information beside each item:

- **Price**: This could be the result of some kind of frequently updated marketing policy. Non-real-time information could cause problem on the vendor side (for example, a wrong price policy could be applied).
- **Availability**: Here, wrong information could cause an invalid claim from customers; for example, "I bought that book because I saw it as available, but it isn't!"

This is a good scenario for developing a search component. We will create our search component and associate it with a given `RequestHandler`.

A search component is basically a class that extends (not surprisingly) `org.apache.solr.handler.component.SearchComponent`:

```
public class  RealTimePriceComponent extends SearchComponent
```

The initialization of the component is done in a method called `init`. Here, most probably we will get the JNDI name of the target data source from the configuration. This source is where the prices must be retrieved from:

```
public void init(NamedList args) {
  String dsName = SolrParams.toSolrParams(args).get("ds-name");
  Context ctx = new InitialContext();
  this.datasource = (DataSource) ctx.lookup(dName);
}
```

Now we are ready to process the incoming requests. This is done in the `process` method, which receives a `ResponseBuilder` instance, the object we will use to add the component contribution to the search output. Since this component will run after the query component, it will find a list containing query results in `ResponseBuilder`. For each item within those results, our component will query the database in order to find a corresponding price:

```
public void process(ResponseBuilder builder)  throws IOException {
  SolrIndexSearcher searcher = builder.req.getSearcher();

  // holds the component contribution
  NamedList contrib = new SimpleOrderedMap();
    for (DocIterator it = builder.getResults().docList.iterator();
    iterator.hasNext();) {
```

```
        // This is the Lucene internal document id
        int docId = iterator.nextDoc();
        Document ldoc = searcher.doc(docId, fieldset);

        // This is the Solr document Id
        String id = ldoc.get("id");

        // Get the price of the item
        BigDecimal price = getPrice(id);

        // Add the price of the item to the component contribution
        result.add(id, price);
    }

    // Add the component contribution to the response builder
    builder.rsp.add("prices", result);
}
```

In `solrconfig.xml`, we must declare the component in two places. First, we must declare and configure it in the following manner:

```
<searchComponent name="prices" class="a.b.c.
RealTimePriceComponent">
  <str name="ds-name">jdbc/prices</str>
</searchComponent>
```

Then it has to be enabled in request handlers (as shown in the following snippet). Since this component is supposed to contribute to a set of query results, it must be placed after the query component:

```
<requestHandler name="/xyz" …>
  …
  <arr name="last-components">
    <str>prices</str>
  </arr>
</requestHandler>
```

Done! If you run a query invoking the `/xyz` request handler you will see after query result a new section called **prices** (the name we used for the search component). This reports the document id and the corresponding price for each document in the search results.

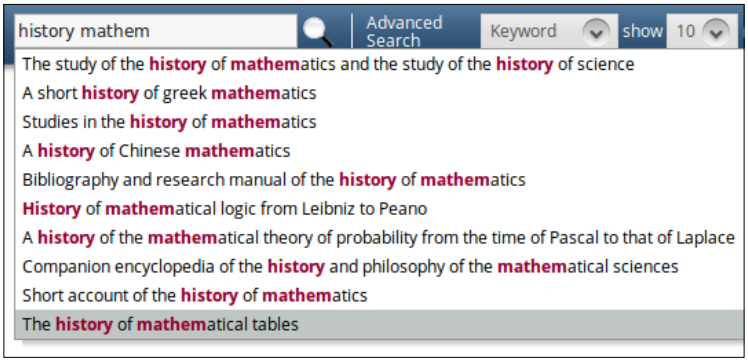
You can find the source code of the entire example in the `src` folder of the project associated with this chapter, under the `org.gazzax.labs.solr.ase.ch3.sp` package.

If you want to start Solr with that component, just run the following command from the command line or from Eclipse:

```
mvn clean install cargo:run -P custom-search-component
```

Using a custom response writer

In a project I was working on, we implemented the autocomplete feature, that is, a list of suggestions that quickly appears under the search field each time a user types a key. Thus, the search string is gradually composed. The following screenshot shows this feature:



A new response writer was implemented because the user interface widget had already been built by another company, and the exchange format between that widget and the search service had been already defined.

Doing that in Solr is very easy. A response writer is a class that extends `org.apache.solr.response.QueryResponseWriter`. Like all Solr components, it can be optionally initialized using an `init` callback method, and it provides a `write` method where the response should be serialized according to a given format:

```
public void write(
  Writer writer,
  SolrQueryRequest request,
  SolrQueryResponse response) throws IOException {

  // 1. Get a reference to values that compound the current response
  NamedList elements = response.getValues();

  // 2. Use a StringBuilder to build the output
  StringBuilder builder = new StringBuilder("{")
    .append("query:'")
    .append(request.getParams().get(CommonParams.Q))
    .append("',");

  // 3. Get a reference to the object which
  // hold the query result
  Object value = elements.getVal(1);
```

```
    if (value instanceof ResultContext)
    {
      ResultContext context = (ResultContext) value;

      // The ordered list (actually the page subset)
      // of matched documents
      DocList ids = context.docs;
      if (ids != null)
      {
        SolrIndexSearcher searcher = request.getSearcher();
        DocIterator iterator = ids.iterator();
        builder.append("suggestions:[");

        // 4. Iterate over documents
        for (int i = 0; i < ids.size(); i++)
        {
          // 5. For each document we need to get the "label" attr
          Document document = searcher.doc(iterator.nextDoc(), FIELDS);
          if (i > 0)  { builder.append(","); }

          // 6. Append the label value to writer output
          builder
            .append("'")
            .append(((String) document.get("label")))
            .append("'");
        }
        builder.append("]").append("}");
      }
    }

    // 7. and finally write out the result.
    writer.write(builder.toString());
  }
```

That's all! Now try issuing a query like this:

```
http://127.0.0.1:8983/solr/example/auto?q=ma
```

Solr will return the following response:

```
{
query:'ma',
suggestions:['Marcus Miller','Michael Manring','Got a
match','Nigerian Marketplace','The Crying machine']
}
```

You can find the source code of the entire example under the `org.gazzax.labs.solr.ase.ch3.rw` package of the source folder in the project associated with this chapter.

If you want to start Solr with that writer, run the following command from the command line or from Eclipse:

```
mvn clean install cargo:run -P custom-response-writer
```

Troubleshooting

This section will provide help, tips, and suggestions about difficulties that you could meet while you're experimenting with what we described in this chapter.

Queries don't match expected documents

There's no single answer to this big and popular question. Without any additional information, the first two things I would do are as follows:

- Retry the query by appending debug parameters (for example, `debugQuery` and `explainOther`) and analyze the explain section. There's a wonderful online tool (`http://explain.solr.pl`) that makes life easy by explaining debug information.
- Use the field analysis page, type some sample values, and see what happens at index and query time. Probably, your analyzer chains are not consistent.

Mismatch between index and query analyzer

Using different analyzer chains at index and query time sometimes causes problems because tokens produced at query time don't match, as one would expect, with the output tokens at index time. The field analysis page helps a lot in debugging these situations. Type a value for a field and see what happens at query and index time. In addition, this page provides a check for all highlighting matches between index and query tokens.

No score is returned in response

The `score` field is a virtual field that must be explicitly asked for in requests. A value of `*` in the `fl` parameter is not enough because `*` means "all real fields." A request for all real fields that also include the score must provide an `fl` parameter with the value of `*,score`. Note that this is valid in general for all virtual fields (for example, functions, transformers, and so on).

Summary

In this chapter we met the Solr search capabilities, a huge set of features that power up information retrieval on Solr. We saw a lot of tools used to improve the search experience of clients, requestors, and last but not least, end users. After examining the indexing phase, you can well imagine that search and information retrieval constitute the actual functional goals of a full-text search platform.

We met the different pieces that compound Solr's search capabilities: analyzers, tokenizers, query parsers, search components, and output writers. For all of them, Solr provides a good set of alternatives, already implemented and ready to use. For those who have specific requirements, it is always possible to create customizations and extensions.

In the next chapter, keeping in mind the big picture of crucial phases in an information retrieval system, we will take a look at client APIs. The available libraries are great examples of how to use Solr's HTTP services to work programmatically with it on the client side.

4
Client API

A search application needs to interact with Solr by issuing index and search requests. Although Solr exposes these services through HTTP, working at that (low) level is not so easy for a developer. Client APIs are façade libraries that hide the low-level details of client-server communication. They allow us to interact with Solr using client-native constructs and structures such as the so-called **Plain Old Java Object** (**POJO**) in the Java programming language.

In this chapter we will describe Solrj, the official Solr client Java library. We will also describe the structure and the main classes involved in index and search operations. The chapter will cover the following topics:

- Solrj: the official Java client library
- Other available bindings

Solrj

Solrj is the name of the official Solr Java client. It completely abstracts the underlying (HTTP) transport layer and offers a simple interface to client applications to interact with Solr.

SolrServer – the Solr façade

A client library necessarily needs a façade or a proxy, that is, an object representing the remote resource that hides and abstracts the low-level details of client-server interaction. In Solrj, this role is played by classes that implement the `org.apache.solr.client.solrj.SolrServer` abstract class. At the time of writing this book, these are the available `SolrServer` implementers:

- `EmbeddedSolrServer`: This connects to a local **SolrCore** without requiring an HTTP connection. This is not recommended in production but is definitely useful for unit tests and development.

- `HttpSolrServer`: This is a proxy that connects to a remote Solr using an HTTP connection.
- `LBHttpSolrServer`: A proxy that wraps multiple `HttpSolrServer` instances and implements client-side, round-robin load balancing between them. It also ensures it periodically checks the (running) state of each server, eventually removing or adding members to the round-robin list.
- `ConcurrentUpdateSolrServer`: This is a proxy that uses an asynchronous queue to buffer input data (that is, documents). Once a given buffer threshold is reached, data is sent to Solr using a configurable number of dequeuer threads.
- `CloudSolrServer`: A proxy used to communicate with **SolrCloud**.

Although any `SolrServer` implementers mentioned previously offer the same functionalities, `HttpSolrServer` and `LBHttpSolrServer` are better suited for issuing queries, while `ConcurrentUpdateSolrServer` is recommended for update requests.

The test case, `org.gazzax.labs.solr.ase.ch3.index.SolrServersITCase`, contains several methods that demonstrate how to index data using different types of servers.

Input and output data transfer objects

As described in the previous chapters, a **Document** is a central concept in Solr. It represents an atomic unit of information exchanged between the client and the server. The Solr API separates input documents from output documents using the `SolrInputDocument` and `SolrDocument` classes, respectively.

Although they share basic data transfer object behavior, each of them has its own specific features associated with the direction of interaction between the client and the server where they are supposed to play.

`SolrInputDocument` is a write object. You can add, change, and remove fields in it. You can also set a name, value, and optional boost for each of them:

```
public void addField(String name, Object value)
public void addField(String name, Object value, float boost)
public void setField(String name, Object value)
public void setField(String name, Object value, float boost)
```

`SolrDocument` is the output data transfer object, and it is primarily intended as a query result holder. Here, you can get field values, field names, and so on:

```
public Object getFieldValue(String name)
public Collection<Object> getFieldValues(String name)
public Object getFirstValue(String name)
```

Within an `UpdateRequestProcessor` instance, or while adding data to Solr, we will use `SolrInputDocument` instances. In `QueryResponse` (that is, the result of a query execution), we will find `SolrDocument` instances.

All the examples in the sample project associated with this chapter make extensive use of these data transfer objects.

Adds and deletes

Once a valid reference of a `SolrServer` has been created, adding data to Solr is very easy. The `SolrServer` interface defines several methods to do this:

```
void add(SolrInputDocument document)
void add(List<SolrInputDocument> document)
```

So we first create one or more `SolrInputDocument` instances filled with the appropriate data:

```
final SolrInputDocument doc1 = new SolrInputDocument();
doc1.setField("id", 1234);
doc1.setField("title", "Delicate Sound of Thunder");
doc1.addField("genre", "Rock");
doc1.addField("genre", "Progressive Rock");
```

Then, using the proxy instance, we can add that data:

```
solrServer.add(doc1);
```

Finally, we can commit:

```
solrServer.commit();
```

We can also accumulate all the documents within a list and use that as the argument of the `add` method.

Following the same logic as described in the second chapter for REST services, `SolrServer` provides the following methods to delete documents:

```
UpdateResponse deleteById(String id)
UpdateResponse deleteById(String id, int commitWithinMs)
UpdateResponse deleteById(List<String> ids)
UpdateResponse deleteById(List<String> ids, int commitWithinMs)
UpdateResponse deleteByQuery(String query)
UpdateResponse deleteByQuery(String query, int commitWithinMs)
```

The `org.gazzax.labs.solr.ase.ch3.index.SolrServersITCase` test case contains several methods that illustrate how to index and delete data.

Search

Searching with Solrj requires knowledge of (mainly) two classes: `org.apache.solr.client.solrj.SolrQuery` and `org.apache.solr.client.solrj.response.QueryResponse`. The first is an object representation of a query that can be sent to Solr. It allows us to inject all parameters we described in the previous chapter. One way of doing this is by providing dedicated methods, such as these:

```
SolrQuery setQuery(String query)
SolrQuery setRequestHandler(String qt)
SolrQuery addSort(String field, ORDER order)
SolrQuery setStart(Integer start)
SolrQuery setFacet(boolean b)
SolrQuery addFacetField(String ... fields)
SolrQuery setHighlight(boolean b)
SolrQuery setHighlightSnippets(int num)
…
```

Alternatively, generic setter methods can be provided:

```
SolrQuery setParam(String name, String … values)
SolrQuery setParam(String name, boolean value)
```

Note that all the preceding methods return the same `SolrQuery` object, thus allowing a caller to chain method calls, like this:

```
SolrQuery query = new SolrQuery()
  .setQuery("Charles Mingus")
  .setFacet(true)
  .addFacetField("genre")
```

```
    .addSort("title", Order.ASC)
    .addSort("released", Order.DESC)
    .setHighlighting(true);
```

Once a `SolrQuery` has been built, we can use the appropriate method in the `SolrServer` proxy to send the query request:

```
QueryResponse query(SolrParams params)
```

The method returns a `QueryResponse`, which is an object representation of the response that Solr sent back as a result of the query execution. With that object, we can get the list of `SolrDocuments` of the currently returned page. We can also get facets and their values, and in general, we can inspect and access any part of the response.

The `org.gazzax.labs.solr.ase.ch3.search.SearchITCase` test case contains several examples that demonstrate how to query with Solrj.

The following is an example of the use of `QueryResponse`:

```
// Executes a query and get the corresponding response
QueryResponse  res = solrServer.query(aQuery);

// Gets the request execution elapsed time
long elapsedTime = res.getElapsedTime();

// Gets the results (i.e. a page of results)
SolrDocumentList results = res.getResults();

// How many total hits for this response
int totalHits = results.getNumFound();

// Iterates over the current page
for (SolrDocument document : results) {
  // Do something with the current document
  String title = document.getFieldValue("title");
  …
}
// Gets the facet field "genre"
FacetField ff = res.getFacetField("genre");
// Iterate over the facet values
for (Count count : genre.getValues()) {
  String name = count.getName(); // e.g. Jazz
```

```
  String count = count.getCount(); // e.g. 19
}
// The Highlighting section is a bit complicated, as the
// value object is a composite map where keys are the documents
identifiers while values are maps with highlighted fields as key
and snippets (a list of snippets) as values.

Map<String, Map<String, List<String>>> hl =
response.getHighlighting();

// Iterates over highlighting sectio
for (Entry<String, Map<String, List<String>> docEntry : hl) {
  String docId = docEntry.getKey();

  // Iterates over highlighted fields
  for (Entry <String, List<String> fEntry : entry.getValue()) {
    String fEntry = field.getKey();

    // Iterates over snippets
    for (String snippet : field.getValue()) {
      // Do something with the snippet
  }
}
```

Other bindings

Solrj is a very powerful client API, but of course, it is only available for Java clients. Since Solr services are exposed using standard HTTP procedures, other client API implementations have been created for other languages. Hence, it is possible to interact with Solr using Python, Perl, Ruby, .NET, or your favorite programming language.

The following table lists some of them, together with their location (only Solrj is a part of the Solr distribution; all other client libraries are independent projects):

Project	Language	Address
sunburnt	Python	`https://pypi.python.org/pypi/sunburnt`
pysolr	Python	`https://pypi.python.org/pypi/pysolr/3.2.0`
solrcloudpy	Python	`https://pypi.python.org/pypi/solrcloudpy`
solr-ruby	Ruby	`https://github.com/erikhatcher/solr-ruby-flare/tree/master/solr-ruby`
Blacklight	Ruby	`http://projectblacklight.org`

Project	Language	Address
Solarium	PHP	`http://www.solarium-project.org/`
Solr-PHP-UI	PHP	`http://www.opensemanticsearch.org/solr-php-ui/`
PECL/Solr	PHP	`http://pecl.php.net/package/solr`
Flux	Clojure	`https://github.com/mwmitchell/flux`
solr-scala-client	Scala	`https://github.com/takezoe/solr-scala-client`
SolrNet	.NET	`https://github.com/mausch/SolrNet`

A complete and updated list of all bindings is available at `https://wiki.apache.org/solr/IntegratingSolr`.

Summary

A distributed search system, such as Solr, requires remote service invocations to send and receive data across a network. Clients without appropriate APIs will be exposed to the complexity of dealing with low-level details of the communication protocol.

Since Solr provides all core services through HTTP, a lot of client libraries have been developed to hide that complexity. Regardless of the concrete binding, a client library encapsulates the low-level details of client-server communication and provides a uniform service interface for clients.

In this chapter, we focused on the Solr client APIs, specifically on the official Java binding called Solrj, its main features, and the main classes involved in index and query operations.

We briefly described and listed some other popular bindings that have been developed on top of the Solr HTTP services.

In the next chapter, we will return to the server side to describe how to fine-tune and manage a Solr instance.

5
Administering and Tuning Solr

You can manage a Solr installation using any of the several system administration tools provided with Solr. The system administration tools include the Administration Console, the REST services, and the JMX API, with which you manage and monitor cores, hardware resources, runtime configuration, and the health of the Solr environment to ensure maximum availability and performance.

Although the topic of administration is usually outside the scope of a developer sphere, most probably you, as a provider of a solution based on Solr, will need to know something about it. Specifically, you need to know about a set of tools that let you monitor Solr, tune it, and investigate troubles.

Throughout this chapter, we will use a Solr instance preloaded with sample data. In order to have that up and running, you should check out the source code of the book, go to the `ch5` folder, and run this (using Eclipse or from the command-line):

```
# mvn clean install cargo:run
```

The `ch5` sample project has a preconfigured Eclipse launcher used to run Solr. You can find it under the `src/dev/eclipse` folder. Just right-click on `start-ch5-server.launch` and select the **Debug as** menu item.

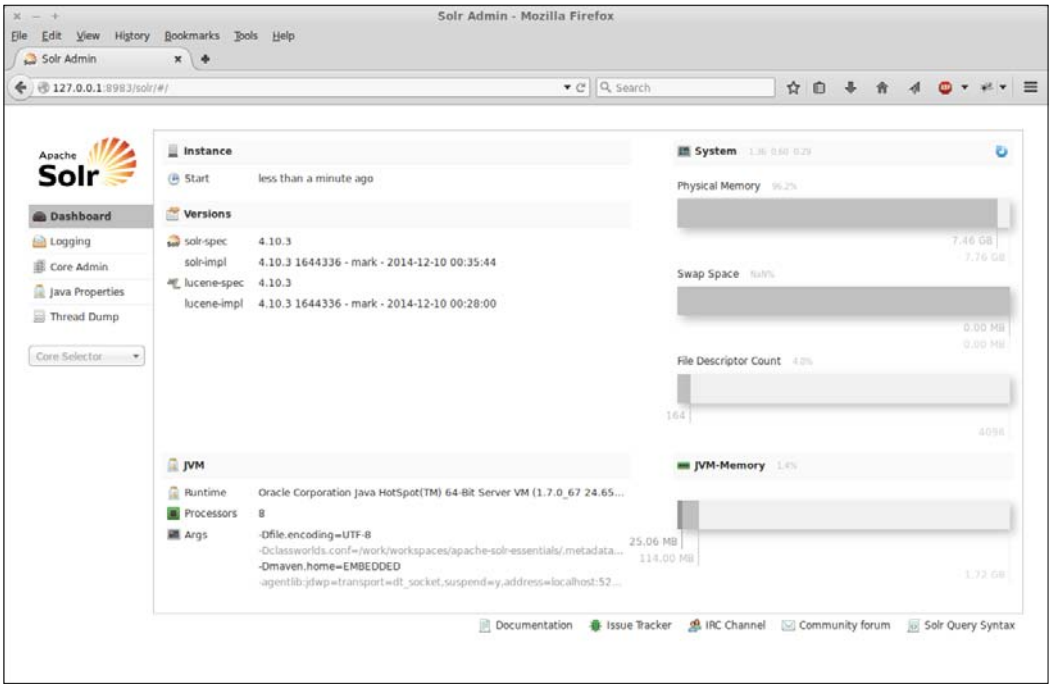
This chapter will describe the most relevant sections of the Solr administration console. We will also explore the JMX API. Each time a hardware resource is involved, we will talk about it. Specifically, this chapter will cover the following topics:

- The Solr Administration Console
- Usage of hardware resources
- JConsole and JMX

Dashboard

The Administration Console is a web application that is part of Solr. You can access the Administration Console from any machine on the local network that can communicate with Solr, through a web browser.

Type `http://127.0.0.1:8983/solr` on the web browser's address bar. The first page that appears is the dashboard, as shown in the following screenshot:



This is where you can see general information about Solr (for example, the version, startup time, and so on) and about its hosting environment (for example, JVM version, JVM args, processors, physical and JVM memory, and file descriptors).

Physical and JVM memory

The first and the last gray bars on the right side of the dashboard represent the physical and JVM memory, respectively. The first measure is the amount of the memory that is available in the hosting machine. The second measure is the amount assigned to the JVM at startup time by means of the `-Xms` and `-Xmx` options.

For a complete list of available JVM options, see `https://docs.oracle.com/cd/E22289_01/html/821-1274/configuring-the-default-jvm-and-java-arguments.html`.

Each bar reports both the available amount and used amount of memory. As you can imagine, memory is one of the crucial factors concerning Solr performance and response times.

When we think about a web application, we may consider it as a standalone container that, for example, reads data from an external database and shows some dynamic pages to the end users. Solr is not like that; it is a service. Despite its web-application-like nature, it makes extensive use of local hardware resources such as disk and memory.

Memory (here, I'm referring to the JVM memory) is used by Solr for a lot of things (for example, caches, sorting, faceting, and indexing) so understanding all those mechanisms is crucial to determine the right amount of memory one should assign to the JVM.

There's a useful spreadsheet (although we already mentioned this in the first chapter) that you can find in the Solr source repository at `https://svn.apache.org/repos/asf/lucene/dev/trunk/dev-tools/size-estimator-lucene-solr.xls`. It is a good starting point from which to estimate RAM and disk space requirements.

However, a resource that is often considered as external to the Solr domain is the system memory, that is, the remaining memory available for the operating system once the JVM memory has been deducted.

In an optimal situation, that kind of memory should be enough to:

- Let the operating system manage its resources.
- Accommodate the Solr index. Ideally, if it is able to contain the whole index, there won't be any disk seek.

The first point is quite obvious; an operating system needs a given amount of memory to manage its ordinary tasks.

The second point has to do with the so-called (OS) filesystem cache. The JVM works directly with the memory that we made available in the startup command-line by means of the `-Xms` and `-Xmx` options. This is the memory we are using in our Java application to load object instances, implement application-level caches, and so on.

However, applications such as Solr that widely use filesystem resources (to load and write index files) also rely on another important part of the memory that is available for the operating system and is used to cache files. Once a file is loaded, its content is kept in memory until the system requires that space for other purposes. Data in this filesystem cache provides quick access, without requiring disk accesses and seeks.

Remember that this type of memory has nothing to do with the memory assigned to the JVM.

As you can imagine, this aspect can dramatically improve overall performance in both index (writes) and query (reads) phases. In those cases where it's not possible to fit all of the index in the filesystem cache (the index can easily reach a size that is relatively small in terms of disk space but definitely huge in terms of memory), the system memory should be enough to allow efficient load and unload management of that filesystem cache.

Disk usage

The dashboard page reports information about the swap space, but it says nothing about disk usage. This is because that kind of information is reported in a dedicated section for every managed core. Unfortunately, there isn't a central point where it's possible to see the total disk space used by the instance.

As described in the previous section, the disk is a resource widely used by Solr, and its role is fundamental for getting optimal performance. Here, we can add additional information by mentioning **Solid State Disks** (**SSD**), which are usually a very good choice for getting fast reads and writes. But again, the most critical factor is understanding and tuning the filesystem cache; in the most extreme cases, this entirely avoids disk seeks at all. To put it in a nutshell SSDs are fast, but memory is better.

File descriptors

The third bar (shown in the previous screenshot) shows the maximum number (light gray) and the effective opened (dark gray) file descriptors associated with the Java process that runs Solr (that is, the Java process of your servlet container).

A Solr index can be composed of a lot of files that need to be opened at least once. Especially if you have many cores, frequent changes, commits, and optimizes, the incremental nature of a Solr index can lead to exhaustion of all the available file descriptors. This is usually the case where you get an IOException (too many open files).

The first place where you can manage and limit the number of files used by Solr is Solr itself. Within the `solrconfig.xml` file, you'll find a `<mergeFactor>` parameter in the `<indexConfig>` section. This parameter decides how many segments will be merged at a time.

The Solr/Lucene index is composed of multiple subindexes called segments. Each segment is an independent index composed of several files. When documents are added, updated, or deleted, Solr asynchronously persists those changes by creating new segments or merging existing segments. This is the reason the total number of files compounding the index will necessarily change (it changes gradually, following a reasonable amount of changes applied to your dataset). Hence, it needs to be monitored.

With a `mergeFactor` value set to `10` (the default value) there will be no more than nine segments at a given moment. When update thresholds (the `maxBufferedDocs` or `ramBufferSize` parameters) are reached, a new segment will be created. If the total number of segments is equal to the configured `mergeFactor`, Solr will attempt to merge all existing segments into a new segment.

Another parameter in the `solrconfig.xml` file that has an impact on the number of open files is `<useCompoundFile>`. If this is set to `true` (note that it defaults to `false`), Solr will combine the files that make up a segment into a single file. While that may produce a benefit in terms of open file descriptors, it may also lead to some performance issues because of the monolithic nature of the compound file.

On top of that, there are scenarios where a lot of files are the natural consequence of your infrastructure. Think of a system with several cores, for example. The previous settings are specific to a single core, but what if you have a lot of them?

When I use Solr for library search services, I usually create at least six cores: one for the main index, one that holds the headings used for the autocompletion feature, and one for each alphabetical index (for example, authors, titles, subjects, and publishers). There are some customers who require up to 50 alphabetical indexes (which means up to 50 cores).

In such cases, after checking out your application and seeing that it effectively requires more file descriptors than the default (usually 1024), you may want to increase that limit by using the `ulimit` command, as follows:

```
# ulimit -n 5000
```

Here, `5000` is the new limit. Note that this command requires root privileges and it applies that limit only to the current session. If you want it to be permanent, that value has to be configured in the `/etc/security/limits.conf` configuration file.

Logging

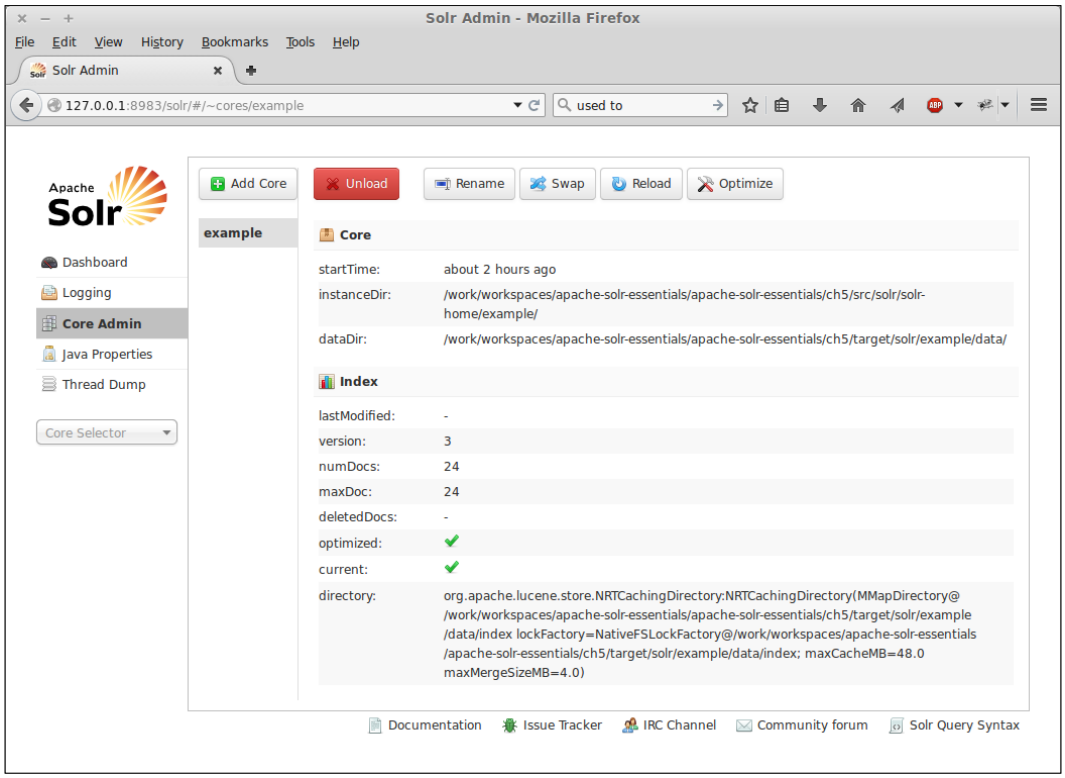
The Administration Console allows you to see log messages (also available in a log file) and change the log settings.

While the first feature is useful only if you don't have access to the log files (inspecting log files with Unix command-line tools is definitely more powerful than doing the same with the AJAX-refreshed page), managing log settings is very useful because it doesn't require manual edits or server restarts. So, if you want to limit the priority level of log messages on-the-fly, or debug the behavior of a component, this is the right place to do so.

A verbose log level can slow down index operations, so it's better to check log settings before calling the `/update` request handler. For the same reason, remember that Solr logs all query requests at the `INFO` level. Depending on how many users your application has, this could lead to a huge amount of log messages.

Core Admin

The Core Admin section is a central point where you can manage registered cores. You can create a new core on-the-fly (assuming that the core instance and data directories exist on the disk) or manage the existing cores one by one, selecting them from the list on the left. The following screenshot shows the Core Admin page of the Solr instance set up for this chapter:



The top toolbar contains these buttons:

Button	Description
Unload	Unloads the core. The core will be removed after pending requests are processed.
Rename	Changes the core name. Note that this change will affect the URI endpoints of the core services.
Swap	Swaps two active cores. This is useful for switching between two versions (that is, online and offline versions) of the same core. Note that both of them will still be alive after issuing the swap command.

Button	Description
Reload	Reloads a core. The current core instance will be available only for satisfying pending requests. This command is useful if some (backward-compatible) changes have been made to the `solrconfig.xml` or `schema.xml` configuration files or core libraries and you want to load those changes.
Optimize	Issues an optimize command to the selected core.

The central area shows the following information about the core and the corresponding index:

Attribute	Description
startTime	The core start (or reload) time.
instanceDir	The top core folder. It contains a `conf` subfolder that contains Solr configuration files (`schema.xml`, `solrconfig.xml`, and dependent files).
dataDir	The folder containing the index data files.
lastModified	The last modification date of the index.
version	A version number assigned to the IndexReader instance associated with the index.
numDocs	The number of searchable documents in the index. In other words, this is the number of documents you can get back from a `*:*` query.
maxDocs	The number of internal document identifiers actually in use. The difference between `maxDocs` and `numDocs` indicates how many documents have been deleted or replaced. The old (deleted and replaced) identifiers are gradually removed during merges or after issuing an index optimize.
deletedDocs	The number of deleted documents. It also includes replaced documents because Solr doesn't actually support updates; it simply deletes a given document and subsequently adds its new version. This is basically the difference between `maxDocs` and `numDocs` after a commit and before merging or optimizing.
optimized	Indicates whether the index has been optimized.
current	Indicates whether the index has been committed.
directory	The underlying Lucene Directory implementation.

Java properties and thread dump

Java properties form a read-only section where you can see the system properties associated with the current JVM instance.

Remember that you can use those variables in `solrconfig.xml`, so you may want to check in this page whether a specific property has the expected value.

The thread dump page shows a snapshot of what live threads in the JVM are doing at a given instant. The same information can be retrieved using the `jstack` command-line utility available in JVM.

Thread dumps are very useful for debugging high-CPU-usage scenarios and deadlocks.

Unlike log analysis, the user interface here is definitely more user-friendly than manual inspection of the `jstack` output.

Core overview

Selecting one of the available cores in the drop-down list on the left side of the Administration Console will open a core dedicated area, with several other sections. The first section is an overview of the selected core. It reports more or less the same information that we saw in the dashboard and in the core admin page.

Here, there is additional information about the health check (heartbeat information enabled only if you configured the ping request handler) and the replication status.

The replication section shows the index status of the master and slave (only if the current Solr instance acts as a slave) in terms of replicability.

The replication section is useful for monitoring master-repeater-slave instances, especially when you get some synchronization issues within the Solr ensemble. Note that the console also has a dedicated Replication section where that information is more detailed.

The master-slave replication architecture is explained in the next chapter.

Caches

To speed up query execution, Solr stores data using several types of in-memory caches. Caches transparently store filters, documents, and identifiers so that future requests for the same data can be served faster. If you run the same search twice, you will see in the Solr logs a marked difference between the first and the second query in terms of response time, as shown in the following example:

```
… params={q=history&fq=catalog:NRA} hits=17298 status=0 QTime=78
…
… params={q=history&fq=catalog:NRA} hits=17298 status=0 QTime=2
```

Solr comes with several kinds of caches. They can be configured and tuned in `solrconfig.xml`:

```
<filterCache class="solr.FastLRUCache" size="512"
initialSize="512" autowarmCount="0"/>
<queryResultCache class="solr.LRUCache" size="512"
initialSize="512" autowarmCount="0"/>
<documentCache class="solr.LRUCache" size="512"
initialSize="512" autowarmCount="0"/>
<fieldValueCache class="solr.FastLRUCache" size="512"
autowarmCount="128" showItems="32" />
```

The following table briefly describes the types of caches available in Solr:

Cache	Description
`FilterCache`	Holds the document identifiers associated with filter queries that have been executed.
`QueryResultCache`	Holds the document identifiers resulting from queries that have been executed.
`DocumentCache`	Holds Lucene document instances for quick access to their stored fields.
`FieldCache`	A low-level Lucene field cache that is not managed by Solr (in other words, it cannot be configured). It is used for sorting and faceting.
`FieldValueCache`	This is a field cache very similar to `FieldCache`, but it can be configured. It is mainly used for faceting.
`CustomCache`	Application-level caches used to hold custom user/application data.

Cache life cycles

A cache is always associated with an index searcher instance, and it follows the same lifecycle of that instance. This means that, when an index searcher is instantiated (on startup or after a commit), cache instances are created and associated with it. As a consequence of this, caches and cached objects don't have an expiry time; they will be valid as long as the owning index searcher instance is active.

When a searcher is instantiated, and if it is not the first searcher (that is, at startup time), caches can be optionally auto-warmed; that is, they can be prepopulated with some data coming from their previous colleagues (caches from the previous searcher). The `autowarmcount` attribute allows us to declare the maximum amount of data (absolute or a percentage) that can be used to prepopulate the new cache.

Data from the previous cache is not taken as it is. It has to be validated against the new searcher "view" of the index. A given object previously cached can't be valid after the new searcher has been opened; it could have been deleted. The `autowarmcount` attribute refers only to valid entries.

When a new searcher is opened, the current searcher will continue to serve pending requests. After that, it will be closed and the orphan caches will be subjected to garbage collection.

Cache sizing

Cache size can refer to two different measures: the total count of objects a cache contains at a specific moment, and the maximum number of objects a cache can hold.

Within `solrconfig.xml`, you can configure the minimum (initial) and maximum size of a cache by means of the `initialSize` and `size` attributes, respectively:

```
<FilterCache … class="…" size="512" initialSize="512"/>
```

The `initialSize` attribute is used when the cache instance is created. It preallocates a given number of seats for objects that will be cached.

The ideal dimension of a cache strictly depends on the application. Erroneously, one could think: the bigger, the better, but this is a half truth; a huge cache would have the advantage of holding all the required structures in memory, thus allowing fast access to that information. However, unless your index is completely static and it never changes, you will sooner or later add, update, or remove something, and you will need to commit those changes. A commit will open a new searcher, which in turn will create new caches, and the (old) huge caches will be discarded.

In this situation, the garbage collector will have a lot of work to do reclaiming all objects from the old caches. Worse, if you have configured auto-warming, the prepopulation of the newly created caches could take a lot of time.

In other words, this scenario requires a lot of memory to manage all of those objects. From my experience, I can tell you that this is one of the common ways of getting "Out Of Memory" error messages. Remember that garbage collection is not under your control, so most probably there will be a given interval of time during which the JVM must hold both new and old object references.

The suggestion here is to start with default sizes, and then use the Solr Administration Console to constantly monitor how things move. Cache management is not a do-once-and-forget task. Caches must be periodically monitored and eventually tuned in order to gain optimal advantage for your application.

Cached object life cycle

The `class` attribute of a cache determines primarily its implementation, but most importantly, it defines how objects are managed within the cache. In other words, it implements the logic needed to know what to do when the cache reaches its maximum size and which objects must be evicted when a new entry arrives.

Solr offers three cache implementations:

- **LRUCache**: Once the maximum size of the cache has been reached and a new object needs to be cached, this implementation will remove the oldest entry. The age of an object is determined by the last time it was requested from the cache.
- **FastLRUCache**: This implements behavior similar to `LRUCache` but uses a separate thread to (asynchronously) clean up the oldest entries.
- **LFUCache**: This policy implements an eviction based on the popularity of each object in the cache (that is, how many times a given object in the cache has been requested).

Cache stats

For each cache, the Administration Console reports (**Plugin/Stats** | **Cache**) the following attributes:

Attribute	Description
`lookups`	The total count of lookup requests.
`hits`	The number of requests that successfully found the requested object.
`hitratio`	The number of hits on top of the total number of requests. A value of `1` represents optimal usage of the cache (every requested object has been found in the cache).
`inserts`	The total number of inserted objects.
`evictions`	The total number of evictions (objects removed).
`size`	The current size of the cache.
`warmupTime`	The time needed to auto-warm the cache.

`cumulative_lookups`
`cumulative_hits`
`cumulative_hitratio`
`cumulative_inserts`
`cumulative_evictions`
	A cache instance dies when the associated searcher is discarded. The cumulative attributes retain lookups, hits, hit ratio, inserts, and evictions among all cache instances (of the same type), so the value of those attributes measures the same things we just saw but cumulatively, since Solr startup.

Types of cache

As we have briefly described, Solr comes with several kinds of caches. The following paragraphs describe them further.

Filter cache

Each time a filter query is executed, Solr places a new entry in a filter cache. A filter cache is a kind of map where the key is represented by the filter query string (for example, `catalog:NRA` or `genre:Jazz`) and the entry is a list of all matching document identifiers.

The filter cache is configured in the `solrconfig.xml` file, in the following fragment:

```
<filterCache class="solr.FastLRUCache" size="512"
initialSize="512" autowarmCount="0"/>
```

Filter queries play a crucial role in performance and response time optimization. The cached identifiers can be used and reused with subsequent queries; briefly, requests that contain cached filter queries will improve overall performance because those queries won't be actually executed again.

Auto-warming a filter cache means refreshing every cached filter query result by executing (again) all of those queries against the index view represented by the new searcher. Let's see this with a concrete example; the sample Solr instance contains 24 albums. At startup time, the filter cache is empty. Now let's suppose the following queries are executed:

`http://127.0.0.1:8983/solr/example/query?q=*:*&`**`fq=genre:Jazz`** `(3 results)`

`http://127.0.0.1:8983/solr/example/query?q=*:*&`**`fq=genre:Fusion`** `(4 results)`

`http://127.0.0.1:8983/solr/example/query?q=*:*&`**`fq=released:1986`** `(2 results)`

The three filter queries populate the filter cache as described in the following table:

Cache entries (filter queries)	Query results (Document identifiers)
`genre:Jazz`	1, 2, 3
`genre:Fusion`	4, 5, 6, 7
`released:1986`	6, 8

Now we decide to remove document `#6`. In order to do this, we send a `delete` command and then a `commit` command. Once the change has been committed, document `#6` no longer exists. A new searcher is opened, and the cache content needs to be refreshed because it still contains an invalid entry. So, the auto-warming process simply repeats each filter query in the cache (`genre:Jazz`, `genre:Fusion` and `released:1986` in this case) and refreshes the content with valid query results. After the auto-warming, the filter cache will have the following content:

Cache entries (filter queries)	Query results (Document identifiers)
`genre:Jazz`	1, 2, 3
`genre:Fusion`	4, 5, 7
`released:1986`	8

This re-execution is in general the cost of auto-warming, which is directly connected with the cache size (a huge cache in most cases will take some time to re-execute all cached queries).

Query Result cache

With this kind of cache, each time a query is executed, its results (in terms of matching document identifiers) are cached for future reuse. This is configured in the following fragment of the `solrconfig.xml` file:

```
<queryResultCache class="solr.FastLRUCache" size="512"
initialSize="512" autowarmCount="0"/>
```

The underlying reason is that popular queries (that is, queries that are often repeated) will gain a clear advantage here because they won't be actually executed again—their results are already computed.

Other than popular queries, pagination mechanisms also benefit from this cache. When the user asks for the next or the previous page of results for a given query execution, Solr will repeat the query but with a different `start` parameter.

Document cache

Both `FilterCache` and `QueryResultCache` store document identifiers. So, on top of a given query, Solr computes the matching identifiers; for each of them, it needs to query the index to retrieve its stored fields. After that, the response is populated with those documents and their corresponding (stored) fields.

`DocumentCache` caches Lucene documents, so once a query has been executed, Solr doesn't need (with regard to documents that are found in this cache) to query the index to populate the list of results.

If you have huge stored fields (for example, full-text fields used for highlighting), be aware that you cannot specify which fields must be in the cache. Therefore, huge fields may require a lot of memory.

Field value cache

The field value cache has a map structure where keys are field names and values are uninverted fields. This structure maps document identifiers with values. If it is not explicitly declared, this cache is automatically generated with an initial size of 10, a maximum size of 10000, and no auto-warming. It is primarily used for faceting.

Custom cache

Custom caches are intended for developers who write their own Solr extensions. Unlike the other types, custom caches accept a `regenerator` attribute, which declares a class that implements the auto-warming logic for the cache.

Query handlers

The page accessed by navigating to **Plugin/Stats** | **QueryHandler** shows an expandable list where each item is a query handler configured in `solrconfig.xml`. This list includes handlers that represent search endpoints (that is, `SearchHandler`) but also other handlers such as `/admin/ping`, `/admin/dump`, and `/debug`.

The configured `UpdateRequestHandler` instances (for example, `/update` and `/update/json`), being subclasses of `RequestHandler`, are also listed in this page.

For each handler, the console shows some basic attributes such as the class name, version, a short description, and a set of statistical data, as listed in the following table:

Attribute	Description
`handlerStart`	The date (in milliseconds) when the handler received its first request.
`Requests`	The total number of requests received.
`Errors`	The number of requests that raised an exception during the execution.
`timeouts`	If the query is executed with the `timeAllowed` parameter and the given timeout expires, Solr will return only partial results. This attribute counts the requests that face this scenario.
`totalTime`	The total (requests) execution time.
`avgRequestsPerSecond`	The average number of requests per second.
`5minRateReqsPerSecond` `15minRateReqsPerSecond`	The average number of requests per second over the last five and fifteen minutes, respectively.
`avgTimePerRequest`	The average (request) execution time.

`75thPcRequestTime`
`95thPcRequestTime`
`99thPcRequestTime`
`999thPcRequestTime`
	Starting from the distribution of the total request execution times, these attributes report the value at the 75th, 95th, 99th, and 999th percentile in that distribution, respectively.

So, especially for search endpoints, this page is very useful to understand and monitor the usage and the statistical behavior of your Solr instance.

Update handlers

Under the same path (**Plugin | Stats**), the **UpdateHandler** is a page containing an entry corresponding to the `org.apache.solr.update.DirectUpdateHandler2` instance.

The following table lists and describes the attributes of that handler:

Attribute	Description
`commits`	The total number of commit requests received.
`autocommit maxTime`	The maximum amount of time that is allowed to pass since a document was added before automatically triggering a new commit.
`autocommits`	The total number of hard auto-commits executed.
`soft autocommits`	The total number of soft auto-commits executed.
`optimizes`	The total number of optimize requests received.
`rollbacks`	The total number of rollback requests received.
`expungeDeletes`	The total number of hard commits with the `expungeDeletes` flag set to `true`.
`docsPending`	The total number of updates that have been processed but not committed.
`adds`	The total number of `adds` requests received.
`deletesById`	The total number of `deleteById` requests received.
`deletesByQuery`	The total number of `deleteByQuery` requests received.
`errors`	The total number of failed operations (for example, updates, commits, and rollbacks).

`cumulative_adds`
`cumulative_ deletesById`
`cumulative_ deletesByQuery`
`cumulative_errors`
	`UpdateHandler` has a life cycle associated with owning SolrCore. In other words, when SolrCore is reloaded, a new instance of `UpdateHandler` is created. The monitoring attributes prefixed with cumulative are a cumulative measure of a specific attribute (for example, additions and deletions) since the Solr startup.

Most Solr installations I've done in libraries update the index on a daily basis. Each morning, the `UpdateHandler` stats page shows a perfect summary of what happened during the previous day and cumulatively since the last startup. Clearly, in the event of errors, log files serve as my friends.

On the other hand, if I need to monitor the overall progress of an index update in real time, then I prefer the JMX way, which is described in the next section.

JMX

Java Management Extensions (**JMX**) are a powerful set of APIs used to monitor and manage a running JVM. The building blocks of JMX are the so-called **Management Beans** (**MBeans**), which are basically wrappers that decorate existing objects with a management interface. The core classes of JVM are decorated with MBeans.

More information about JMX can be found at `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

MBeans are registered with an MBeanServer that exposes those management interfaces to external clients. Applications are free to create, register, and expose the management interface of their own specific services. Solr MBeans are not automatically registered with the MBeanServer, but if you want to do that, just write (or uncomment) the following line in `solrconfig.xml`:

```
<jmx/>
```

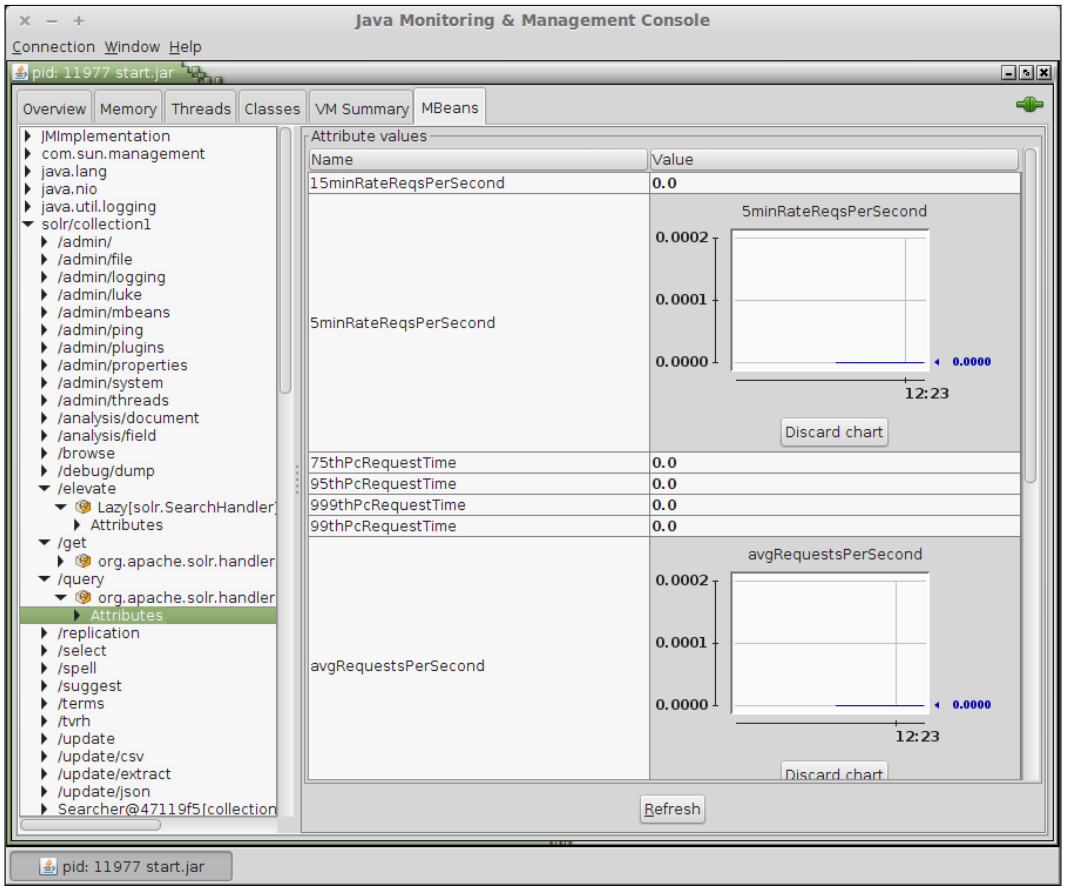
The JVM comes with two built-in JMX clients called **JConsole** and **JVisualVM**.

JVisualVM and JConsole are very similar tools. Here, we will talk only about the JConsole because JVisualVM doesn't have the MBeans perspective.

Open a shell in your PC and type the following command:

```
# $JAVA_HOME/bin/jconsole
```

A dialog pop-up will appear. This is the first screen of JConsole, which is a Java standalone application. The dialog contains a list of locally running JVMs. One of them should be the one where Solr is running. Select that entry, and you should see a screen with several tabs: **Overview**, **Memory**, **Threads**, **Classes**, **VM Summary**, and **MBeans**. At the moment, we are interested in the last tab, **MBeans**. Here you can see (the tree component on the left side) all registered **MBeans**, as depicted in the following screenshot:



For each MBean in the tree, you can see its management interface in the right pane. A management interface is composed of attributes and operations.

Operations can be invoked and attributes can be monitored by looking at their value at a given moment or for a given interval. To do this, you have to double-click on them and activate a real-time chart.

The main differences between the Solr Administration Console and JConsole are as follows:

- The Solr Administration Console, being a web application, offers static snapshots of the system. With JConsole, it's possible to activate real-time monitoring of one or more attributes. This is not limited to MBean a ttributes. In the other tabs, you can monitor threads, processors, memory, and garbage collection.

- JConsole has a finer level of granularity than the Administration Console. There, we can see all attributes and operations exposed for management.
- JConsole, being more technical, is less usable than the Administration Console.

Clearly, JConsole, JVisualVM, and the Solr Administration Console are not alternatives. They should be used together in order to get a different perspective on the system.

Summary

In this chapter, we described some concepts about Solr administration and monitoring. We introduced a few system administration tools such as the Solr Administration Console and JConsole, and we covered hardware resources.

Remember that, although the topics covered in this chapter should be relevant for an administrator nowadays, this role is spread among several people (especially in small and medium companies) who are mostly developers (a developer in a small or medium company is a like a "factotum"). This is the reason it is important for non-administrators to have at a least basic understanding of administration, management, and monitoring.

In the next chapter, you will see how Solr can be deployed in the context of development, testing, and production. We will illustrate and describe several deployment scenarios, starting from the simplest, standalone instance, continuing with a gradually growing level of complexity, and ending with SolrCloud. SolrCloud is a highly available, fault-tolerant cluster of Solr servers that provide distributed indexing and search capabilities.

6
Deployment Scenarios

This chapter contains information on the various ways in which you can deploy Solr, including key features and pros and cons for each scenario.

Solr has a wide range of deployment alternatives, from monolithic to distributed indexes and standalone to clustered instances. We will organize this chapter by deployment scenarios, with a growing level of complexity.

This chapter will cover the following topics:

- Sharding
- Replication: master, slave, and repeaters
- SolrCloud

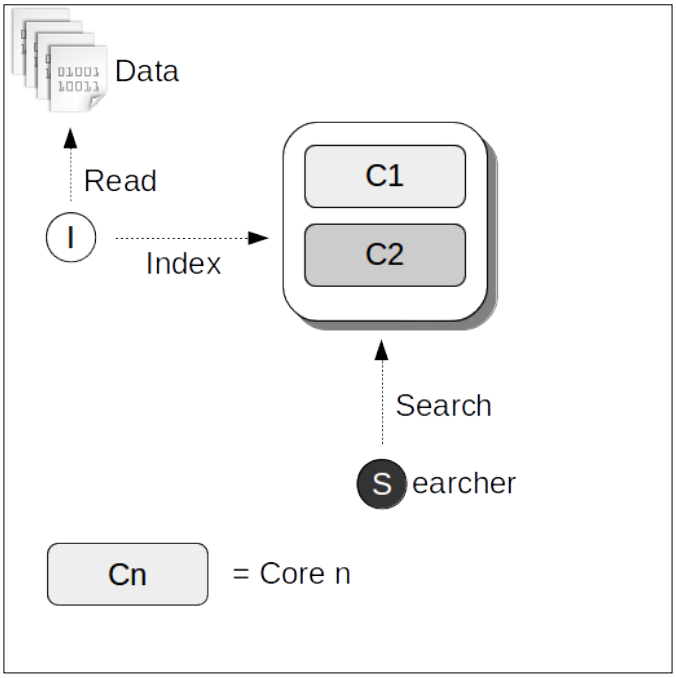
Standalone instance

All the examples we found in the previous chapters use a standalone instance of Solr, that is, one or more cores managed by a Solr deployment hosted in a standalone servlet container (for example, Jetty, Tomcat, and so on).

This kind of deployment is useful for development because, as you learned, it is very easy to start and debug. Besides, it can also be suitable for a production context if you don't have strict non-functional requirements and have a small or medium amount of data.

I have used a standalone instance to provide autocomplete services for small and medium intranet systems.

Anyway, the main features of this kind of deployment are simplicity and maintainability; one simple node acts as both an indexer and a searcher. The following diagram depicts a standalone instance with two cores:

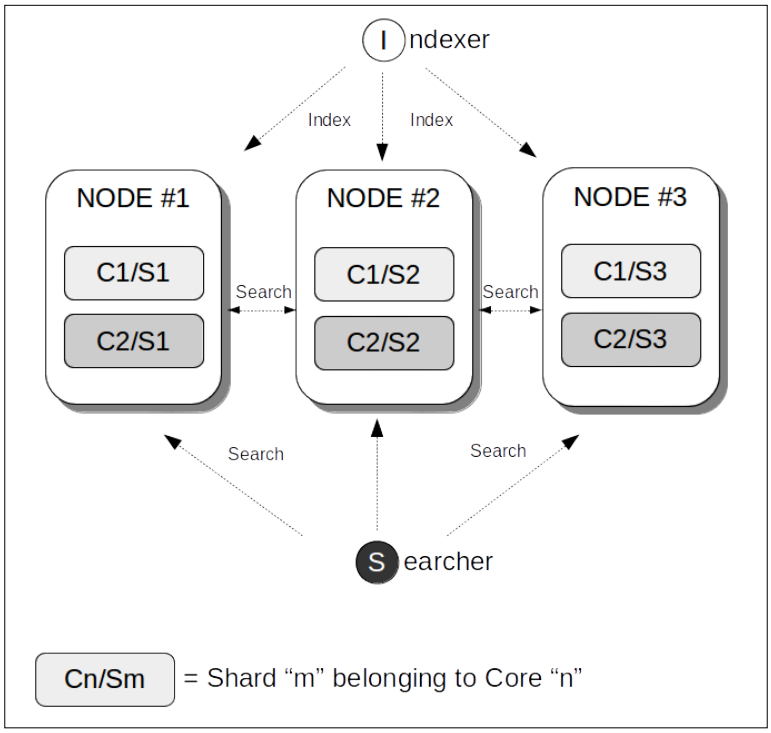


Shards

When a monolithic index becomes too large for a single node or when additions, deletions, or queries take too long to execute, the index can be split into multiple pieces called **shards**.

The previous sentence highlights a logical and theoretical evolution path of a Solr index. However, this (in general) is valid for all scenarios we will describe. It is strongly recommended that you perform a preliminary analysis of your data and the estimated growth factor in order to decide from the beginning the right configuration that suits your requirements. Although it is possible to split an existing index into shards (`https://lucene.apache.org/core/4_10_3/misc/org/apache/lucene/index/PKIndexSplitter.html`), things definitely become easier if you start directly with a distributed index (if you need it, of course).

The index is split vertically so that each shard contains a disjoint set of the entire index. Solr will query and merge results across those shards. The following diagram illustrates a Solr deployment with 3 nodes; this deployment consists of two cores (**C1** and **C2**) divided into three shards (**S1**, **S2**, and **S3**):



When using shards, only query requests are distributed. This means that it's up to the indexer to add and distribute the data across nodes, and to subsequently forward a change request (that is, delete, replace, and commit) for a given document to the appropriate shard (the shard that owns the document).

The Solr Wiki recommends a simple, hash-based algorithm to determine the shard where a given document should be indexed:

```
documentId.hashCode() % numServers
```

Using this approach is also useful in order to know in advance where to send delete or update requests for a given document.

On the opposite side, a searcher client will send a query request to any node, but it has to specify an additional `shards` parameter that declares the target shards that will be queried. In the following example, assuming that two shards are hosted in two servers listening to ports 8080 and 8081, the same request when sent to both nodes will produce the same result:

```
http://localhost:8080/solr/c1/query?q=*:*&shards=localhost:8080/
solr/c1,localhost:8081/solr/c2

http://localhost:8081/solr/c2/query?q=*:*&shards=localhost:8080/
solr/c1,localhost:8081/solr/c2
```

When sending a query request, a client can optionally include a pseudofield associated with the `[shard]` transformer. In this case, as a part of each returned document, there will be additional information indicating the owning shard. This is an example of such a request:

```
http://localhost:8080/solr/c1/query?q=*:*&shards=localhost:8080/
solr/c1,localhost:8081/solr/c2&src_shard:[shard]
```

Here is the corresponding response (note the pseudofield aliased as `src_shard`):

```
<result name="response" numFound="192" start="0">
  <doc>
    <str name="id">9920</str>
    <str name="brand">Fender</str>
    <str name="model">Jazz Bass</str>
    <arr name="artist">
    <str>Marcus Miller</str>
    </arr><str name="series">Marcus Miller signature</str>
    <str name="src_shard">localhost:8080/solr/shard1</str>
  </doc>
  …
  <doc>
    <str name="id">4392</str>
    <str name="brand">Music Man</str>
    <str name="model">Sting Ray</str>
    <arr name="artist"><str>Tony Levin</str></arr>
    <str name="series">5 strings DeLuxe</str>
    <str name="src_shard">localhost:8081/solr/shard2</str>
  </doc>
</result>
```

The following are a few things to keep in mind when using this deployment scenario:

- The schema must have a `uniqueKey` field. This field must be declared as stored and indexed; in addition, it is supposed to be unique across all shards.
- **Inverse Document Frequency** (**IDF**) calculations cannot be distributed.
- IDF is computed per shard.
- Joins between documents belonging to different shards are not supported.
- If a shard receives both index and query requests, the index may change during a query execution, thus compromising the outgoing results (for example, a matching document that has been deleted).

Master/slaves scenario

In a master/slaves scenario, there are two types of Solr servers: an indexer (the master) and one or more searchers (the slaves).

The master is the server that manages the index. It receives update requests and applies those changes. A searcher, on the other hand, is a Solr server that exposes search services to external clients.

The index, in terms of data files, is replicated from the indexer to the searcher through HTTP by means of a built-in `RequestHandler` that must be configured on both the indexer side and searcher side (within the `solrconfig.xml` configuration file).

On the indexer (master), a replication configuration looks like this:

```
<requestHandler
  name="/replication"
  class="solr.ReplicationHandler">
  <lst name="master">
    <str name="replicateAfter">startup</str>
    <str name="replicateAfter">optimize</str>
    <str name="confFiles">schema.xml,stopwords.txt</str>
  </lst>
</requestHandler>
```

The replication mechanism can be configured to be triggered after one of the following events:

- **Commit**: A commit has been applied
- **Optimize**: The index has been optimized
- **Startup**: The Solr instance has started

In the preceding example, we want the index to be replicated after startup and optimize commands. Using the `confFiles` parameter, we can also indicate a set of configuration files (`schema.xml` and `stopwords.txt`, in the example) that must be replicated together with the index.

Remember that changes on those files don't trigger any replication. Only a change in the index, in conjunction with one of the events we defined in the `replicateAfter` parameter, will mark the index (and the configuration files) as replicable.

On the searcher side, the configuration looks like the following:

```
<requestHandler
  name="/replication"
  class="solr.ReplicationHandler">
  <lst name="slave">
    <str name="masterUrl">http://<localhost>:<port>/solrmaster</str>
    <str name="pollInterval">00:00:10</str>
  </lst>
</requestHandler>
```

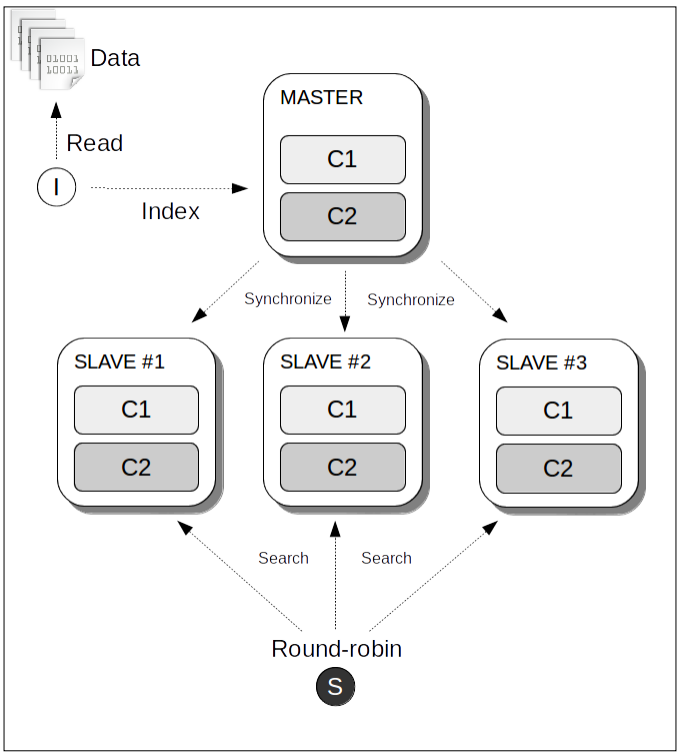
You can see that a searcher periodically keeps polling the master (the `pollInterval` parameter) to check whether a newer version of the index is available. If it is, the searcher will start the replication mechanism by issuing a request to the master, which is completely unaware of the searchers.

The replicability status of the index is actually indicated by a version number. If the searcher has the same version as the master, it means the index is the same. If the versions are different, it means that a newer version of the index is available on the master, and replication can start.

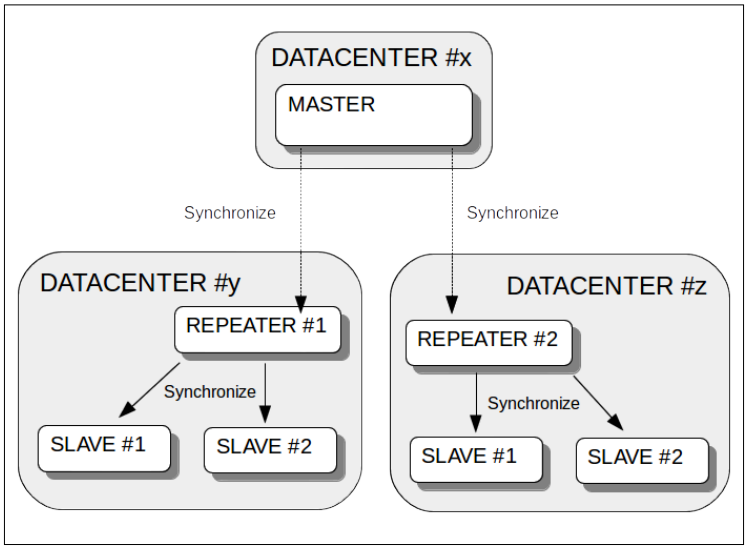
Other than separating responsibilities, this deployment configuration allows us to have a so-called **diamond** architecture, consisting of one indexer and several searchers. When the replication is triggered, each searcher in the ring will receive a whole copy of the index. This allows the following:

- Load balancing of the incoming (query) requests.
- An increment to the availability of the whole system. In the event of a server crash, the other searchers will continue to serve the incoming requests.

The following diagram illustrates a master/slave deployment scenario with one indexer, three searchers, and two cores:

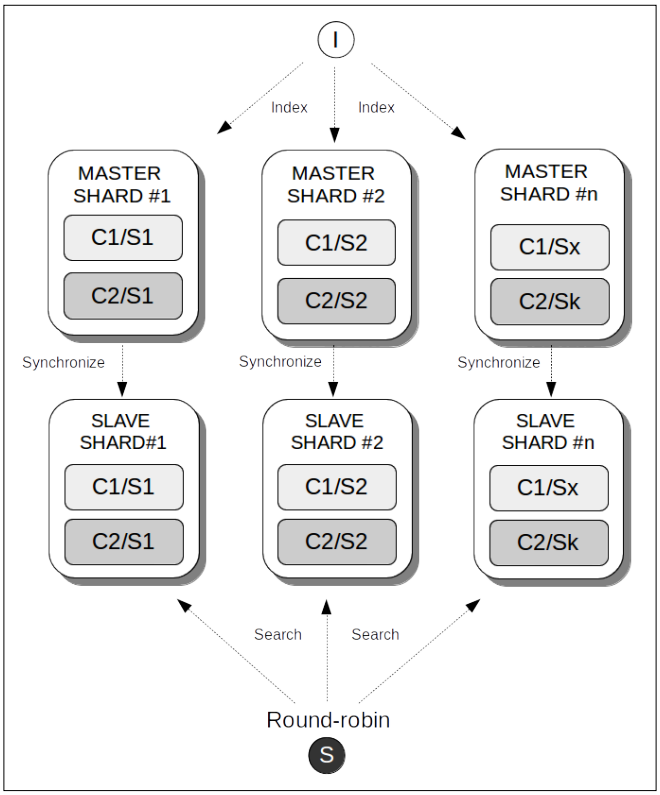


If the searchers are in several geographically dislocated data centers, an additional role called **repeater** can be configured in each data center in order to rationalize the replication data traffic flow between nodes. A repeater is simply a node that acts as both a master and a slave. It is a slave of the main master, and at the same time, it acts as master of the searchers within the same data center, as shown in this diagram:



Shards with replication

This scenario combines shards and replication in order to have a scalable system with high throughput and availability. There is one indexer and one or more searchers for each shard, allowing load balancing between (query) shard requests. The following diagram illustrates a scenario with two cores, three shards, one indexer, and (due to problems with available space), only one searcher for each shard:

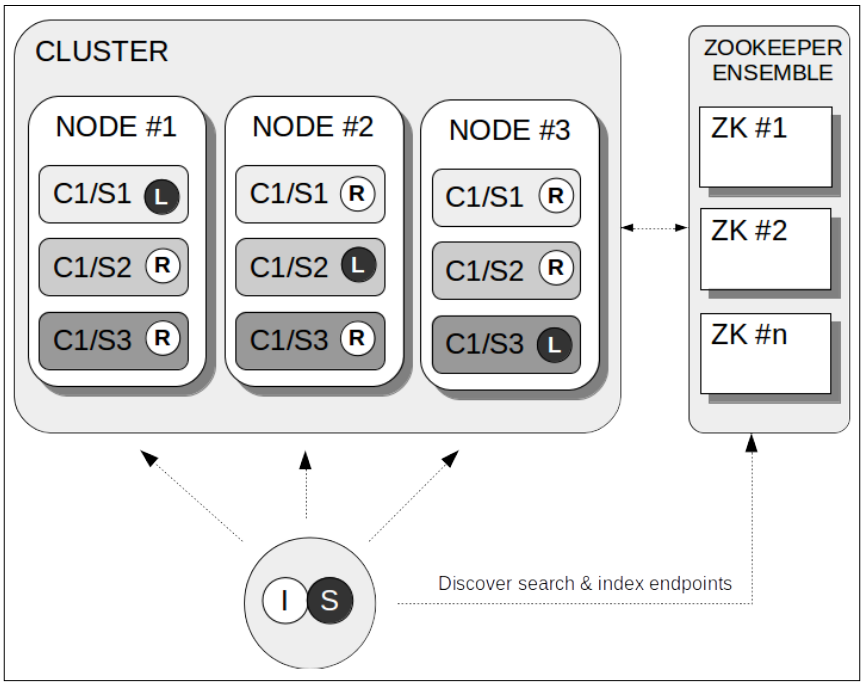


The drawback of this approach is undoubtedly the overall growing complexity of the system that requires more effort in terms of maintainability, manageability, and system administration. In addition to this, each searcher is an independent node, and we don't have a central administration console where a system administrator can get a quick overview of system health.

These disadvantages have been either mitigated or overcome in **SolrCloud**, which is described in the next section.

SolrCloud

SolrCloud is a highly available, fault-tolerant cluster of Solr servers that provides distributed indexing and search capabilities. The following diagram illustrates a simple SolrCloud scenario:



Although SolrCloud introduced a new terminology to define things in a distributed domain, the preceding diagram has been drawn with the same concepts that we saw in the previous scenarios, for better understanding.

Starting from Solr 4.10.0, the download bundle contains an interactive, wizard-like command-line setup for a sample SolrCloud installation. A step-by-step guide for this is available at `https://cwiki.apache.org/confluence/display/solr/Getting+Started+with+SolrCloud`.

The following sections will describe the relevant aspects of SolrCloud.

Cluster management

Apache Zookeeper was introduced in SolrCloud for cluster coordination and configuration. This means it is a central actor in this scenario, providing discovery, configuration, and lookup services for other components (including clients) to gather information about the Solr cluster.

Apache Zookeeper, being a central component, can be organized in a cluster itself (as depicted in the previous diagram) in order to avoid a single point of failure. A cluster of Zookeeper nodes is called **ensemble**.

For more information about Apache Zookeeper, visit `http://zookeeper.apache.org`, the project homepage.

Replication factor, leaders, and replicas

In the preceding diagram, we have only one core (C1) with three shards (S1, S2, and S3). Now, the main difference between the previous distributed scenario (where we met shards) and this scenario is that here, there's a copy of each shard in every node. That copy is called a **replica**. In this example, we have three copies for each shard, but this is just for simplicity; you can have as many copies as you want.

More specifically, SolrCloud has a property called **replication factor,** that determines the total number of copies in the cluster for each shard. Among the copies, one is elected as the **leader** (the letter "L" on C1/S1 on the first node) while the remaining are replicas (the letter "R").

In the preceding diagram, the replication factor is 3 and it is equal to the number of nodes. Keep in mind that this is a coincidence; those measures could be different, and they actually depend on your cluster configuration and needs.

This replication feature satisfies three important nonfunctional requirements: **load balancing**, **high availability**, and **backup**. We have already described how the classic replication mechanism provides load balancing. Having the same data within more than one node allows a searcher to issue query requests to those nodes in a round-robin fashion, thus expanding the overall capacity of the system in terms of queries per second. Here, the context is the same; each shard, regardless of whether it is a leader or a replica, can be found on *n* nodes (where *n* is the replication factor); therefore, a client can use those nodes for load balancing requests.

High availability is a direct consequence of the redundancy introduced with shard replication. The presence of the same data (and the same search services) on several nodes means that, even if one of those node crashes, a client can continue to send requests to the remaining nodes.

The redundancy introduced with the replication also works as a backup mechanism. Having the same things in several places provides a better guarantee against data loss. After all, this is the underlying principle of the popular cloud data services (for example, Dropbox, ICloud, and Copy).

Durability and recovery

Each node maintains a write-ahead transaction log, where any change is recorded before being applied to the index. Therefore, the transaction log is available for leaders and replicas, and it will be used to determine which content needs to be part of a chosen replica during synchronization. For instance, when a new replica is created, it refers to its leader and its transaction log to know which content to get.

The transaction log will also be used when restarting a server that didn't shut down gracefully. Its content will be "replayed" in order to synchronize local leaders and replicas.

Write-ahead logging is widely used in distributed systems. For more information about it, see `https://cwiki.apache.org/confluence/display/solr/NRT%2C+Replication%2C+and+Disaster+Recovery+with+SolrCloud`.

The transaction log path can be configured in an appropriate section of the `solconfig.xml` file.

The new terminology

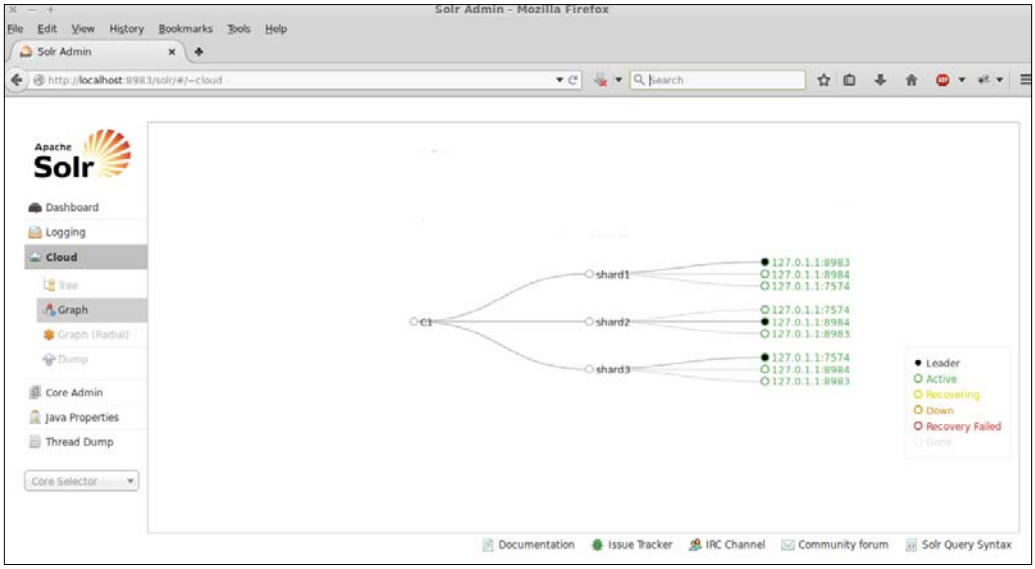
Now that the main features of SolrCloud have been explained, we can stop thinking about it as an evolution of the shard scenario and cover its own terminology:

Parameter	Description
Node	This is a Java Virtual Machine running Solr.
Cluster	A set of Solr nodes that form a single unit of service.
Shard	We previously defined a shard as a vertical subset of the index, that is, a subset of all documents in the index. A shard is a single copy of that subset. In SolrCloud, it can be a leader or a replica.

Parameter	Description
Partition/slice	A subset of the whole index replicated on one or more nodes. A slice is basically composed of all shards (leader and replicas) belonging to the same subset.
Leader	Each shard has one node identified as its leader. This role is crucial for the update workflow. All the updates belonging to a partition route through the leader.
Replica	The replication factor determines the total number of copies each shard has. Among all of those copies, one is elected as the leader, while the others are called replicas. While querying can be done across all shards, updates are always directed (or forwarded by replicas) to leaders.
Replication factor	The number of copies of a shard (and hence, of a document) maintained by the cluster.
Collection	A core that is logically and physically distributed across the cluster. In our example, we have only one collection (C1).

Administration console

In a SolrCloud deployment, the administration console of each node will report an additional menu item called **Cloud**, where it's possible to get an overall view of the cluster. You can choose between several graphic representations of the cluster (tree, graph, and radial), but all of them have a common aim—giving an immediate overview of the cluster in terms of nodes, shards, and collections. This is a screenshot from the administration console of the SolrCloud used in this section:



Collections API

The Collections API is used to manage the cluster, including collections, shards, and metadata about the cluster. This interface is composed of a single HTTP service endpoint located at `http://<hostname>:<port>/<context root>/admin/collections`.

The Collections API accepts an action parameter, which is a mnemonic code associated with the command that we want to execute. Each command has its own set of parameters that depend on the goal of the command. The following table lists the allowed values for the action parameter (that is, the available commands):

Action	Description
CREATE	Creates a new collection.
RELOAD	Reloads a collection. This is used when a configuration has been changed in ZooKeeper.
DELETE	Deletes a collection.
LIST	Returns the names of the collections in the cluster.
CREATESHARD	Creates a new shard.
SPLITSHARD	Splits an existing shard into two new shards.
DELETESHARD	Deletes an inactive shard.
CREATEALIAS	Creates or replaces an alias for an existing collection.
DELETEALIAS	Deletes an alias.
ADDREPLICA	Adds a new replica for a given shard.
DELETEREPLICA	Deletes a replica of a shard.
CLUSTERPROP	Adds, edits, or deletes a cluster property.
MIGRATE	Moves documents between collections.
ADDROLE	Adds a role to a node. At the time of writing this book, the only supported role is an **overseer**. This is the cluster leader responsible for shard assignments and node management operations.
REMOVEROLE	Removes a role from a node.
OVERSEERSTATUS	Returns the current status of the overseer, including some stats about services calls (for example, create collection and create shard).
CLUSTERSTATUS	Returns the cluster status, including shards, collections, replicas, aliases, and cluster properties.
REQUESTSTATUS	Returns the status of those requests that have been executed asynchronously (for example, `MIGRATE`, `SPLITSHARD`, and `CREATE COLLECTION`).

Action	Description
ADDREPLICAPROP	Adds or replaces a replica property.
DELETEREPLICAPROP	Deletes a replica property.
BALANCESHARDUNIQUE	Distributes a given property evenly among the physical nodes that make up a collection.

The complete list of parameters for each command is available at `https://cwiki.apache.org/confluence/display/solr/Collections+API`.

Distributed search

Queries can be sent to any node performing a full distributed search across the cluster with load balancing and failover. SolrCloud also allows partial queries, that is, queries executed against a group of shards, a list of servers, or a list of collections.

If you are using Java on client the side, `CloudSolrServer` in Solrj completely simplifies communication between the client, Zookeeper, and the cluster. As a developer, you will work with the usual `SolrServer` interface.

Cluster-aware index

A drawback of the first distributed scenario we met (that is, shards) was that a client that wants to issue an update request needs to explicitly point to the target shard. This is no longer valid in a SolrCloud context because, for a given shard, there could be more than one copy (that is, a leader and zero or more copies). So the update path becomes the following:

- Updates can be sent to any node in the cluster
- If the target node is the leader of the shard owning the document, the update is executed there, and then it is forwarded to all replicas
- If the target node is a replica, then the update request is forwarded to its leader, and the flow described in the previous point applies

The `CloudSolrServer` in Solrj asks Zookeeper about the leader's location before sending updates. Thus, requests are always targeted at leaders, avoiding additional network round-trips.

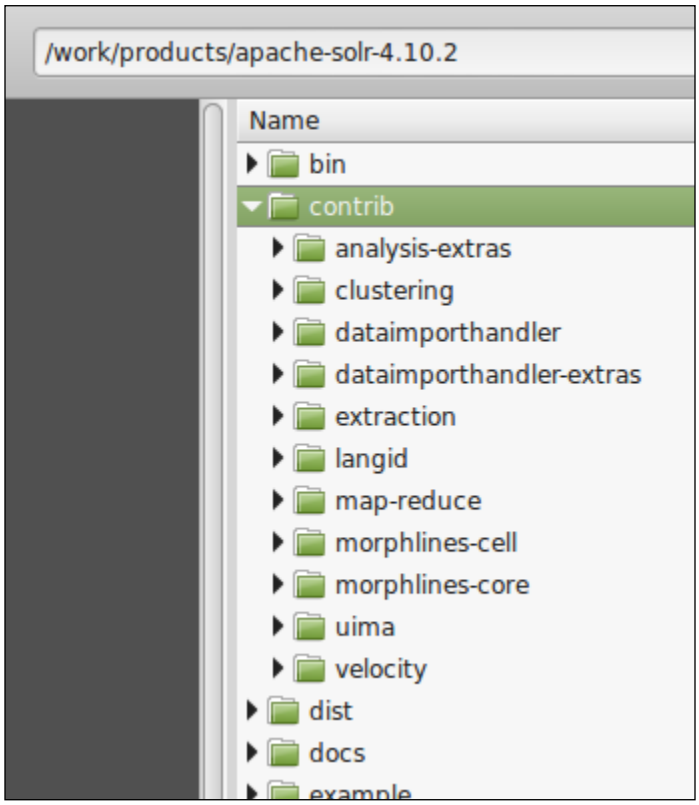
Summary

In this chapter, we described various ways in which you can deploy Solr. Each deployment scenario has specific features, advantages, and drawbacks that make a choice ideal for one context and bad for another. A good thing is that the different scenarios are not strictly exclusive; they follow an incremental approach. In an ideal context, things should start immediately with the perfect scenario that fits your needs. However, unless your requirements are clear right from the start, you can begin with a simple configuration and then change it, depending on how your application evolves.

In the next chapter, we will walk through some useful add-ons that are not part of the core distribution but are included in the Solr download bundle.

7
Solr Extensions

Every popular open source project usually includes a `contrib` folder containing several extra modules to solve common use case implementation problems. In Solr, you can find such modules within the download bundle, as depicted in the following screenshot:



Suppose your data is in a relational database, an XML file with a custom format, or a mail server; you need to index data coming from a Content Management System (such as Drupal, Joomla!, or WordPress); or you have rich documents (such as PDFs or Microsoft Office documents) and you want to do some kind of automatic keyword extraction. In general, these requirements are not covered by the core part of Solr. You will have to plug in and configure those contribution modules.

The aim of this chapter is to describe such modules. In order to do that, we will make use of a preloaded sample Solr instance, with those extensions. To start this instance, you have to check out the source project associated with the chapter, change the directory to the `ch7` folder, and type this from the command line:

```
# mvn clean package cargo:run
```

If you checked out the project using Eclipse, you might have noticed that, under the `src/dev/eclipse` folder, there is preconfigured launcher. Right-click on it and choose the **Debug as…** menu item.

Regardless of the way you choose, you will see something like this at the end:

```
[INFO] Jetty 8.1.15.v20140411 Embedded started on port [8983]
[INFO] Press Ctrl-C to stop the container...
```

This means that the sample instance is up and running. This chapter will cover the following points:

- Importing data from several data sources
- Text and metadata extraction from digital documents
- Language identification
- Solritas (that is, Solr and Velocity)
- Other contrib modules

DataImportHandler

The `DataImportHandler` is a module that enables Solr to load data from several types of data sources. The most frequent type of storage where applications put their data is undoubtedly a relational database, but in general, we could have a lot of scenarios here: filesystems, websites, emails, FTP servers, LDAP, NoSQL databases, and so on.

The `DataImportHandler` module, other than providing a lot of ready-to-use connectors, is an extensible framework where developers are free to inject their storage-specific connector logic. The configuration happens in two different places: the first is the `solrconfig.xml` file (as usual), where the handler is declared as follows:

```
<requestHandler name="/import"
  class="org.apache.solr.handler.dataimport.DataImportHandler">
  <lst name="defaults">
    <str name="config">dih-config.xml</str>
  </lst>
</requestHandler>
```

The second is the handler configuration file (in the preceding example, we called it `dih-config.xml`). Although the specific content of that file could vary, mainly depending on the kind of data source we are using, the building blocks of a `DataImportHandler` domain are data sources, documents, entities, fields, transformers, and processors.

Data sources

A **data source** is a collection of records that store data. Although you are probably thinking of relational databases, data sources can also be associated with other kinds of sources and protocols, such as websites (HTTP), FTP servers, LDAP, mail servers, and so on.

A data source declaration is probably the first thing you will meet in a `DataImportHandler` configuration file. First of all, you must declare where your data is:

```
<dataSource
  type="JdbcDataSource"
  driver="com.mysql.jdbc.Driver"  url="jdbc:mysql://host/
  database-  name"
  user="database_username"
  password="database_password"/>

<dataSource
  type="FileDataSource" encoding="UTF-8"/>
```

Note that it's possible to declare more than one data source (for example, a database and a filesystem or two different databases). Each data source has its own specific properties that depend on its nature. The following table describes the available data sources:

Name	Description
JdbcDataSource	This connects to a database (a direct connection or JNDI data source) using a JDBC driver. Note that Solr doesn't come with any JDBC driver shipped. You must obtain it separately and put that library under the server class path or under the core `lib` folder.
URLDataSource	Reads character files using HTTP.
BinURLDataSource	Reads binary files using HTTP.
FileDataSource	Reads from local character files.
BinFileDataSource	Reads from local binary files.
ContentStreamDataSource	Reads from the `ContentStream` of a POST request using `java.io.Reader`.
BinContentStreamDataSource	Reads from the `ContentStream` of a POST request using `java.io.InputStream`.
FieldReaderDataSource	Used in conjunction with other data sources, when a given field contains text that needs further processing (for example, when it contains an XML document).
FieldStreamDataSource	Used in conjunction with other data sources when a given field contains binary content that needs further processing (for example, when it contains the value of a BLOB database column).

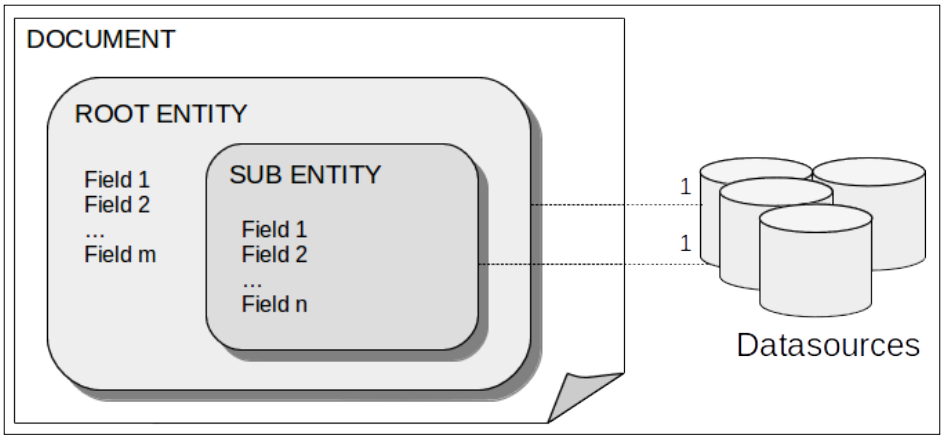
Documents, entities, and fields

Mapping between external data and Solr is done using **documents**, **entities**, and **fields**.

A document represents a logical type (such as products, books, and associations). It contains one or more entities.

Entities are called **root** or **sub** entities depending on their nesting level. Root-entities are direct children of a document. Sub-entities are children of another entity. They have a relationship with their parents; within their configuration, it's possible to use an expression language to refer to their parents.

Fields are concrete places where the mapping between the external data source and Solr document occurs. The following figure schematizes these relationships:



A single document can have one or more root entities. Each entity defines the logic to gather its data and populate its fields.

In the following example, a Solr schema contains books. Each book consists of an identifier (`id`), a title (`title`), and one or more authors. There are two database tables, `BOOKS` and `AUTHORS`, with a **1:n** relationship (this means that a book can have more than one author).

First, let's see how the root entity (the book) is defined:

```
<document name="books">
  <entity name="book" dataSource="my-ds"
  query="SELECT BOOK_ID,TITLE FROM BOOKS" onError="skip">
  <field column="BOOK_ID" name="id"/>
  <field column="TITLE" name="title"/>
```

As you can see, the entity is associated with a data source called my-ds. It is configured with a query, and for each record of the outcoming `ResultSet`, we are interested in two fields: `BOOK_ID` and `TITLE`. They are mapped with the `id` and `title` fields in the Solr schema.

If the name of the column (or the alias) in `ResultSet` coincides with the name of the Solr field (case insensitive), the `<field>` declaration can be omitted. Solr will perform the mapping automatically. So, in the preceding example, the `TITLE` mapping can be removed.

Now, since the cardinality of the relationship between books and authors is **1:n**, we need to define a sub-entity. For each book, we must query the data source again to find the corresponding authors:

```
<entity name="book" dataSource="my-ds"
  query="SELECT BOOK_ID,TITLE FROM BOOKS" onError="skip">
  <field column="BOOK_ID" name="id"/>
  <field column="TITLE" name="title"/>

<entity name="author" dataSource="my-ds"
  query="SELECT NAME FROM AUTHORS WHERE BOOK_ID=${book.BOOK_ID}">
  <field column="NAME" name="author"/>
```

The `author` sub-entity declares a query on the `AUTHORS` table. It uses a simple expression language to refer to the identifier of the current (parent) book:

```
${<parent entity name>.<database alias or column name>}
```

Obviously, this is a really simplified example. In a real production scenario, you will probably meet complicated relational schemas, but the `DataImportHandler` logic will be always the same—detect and configure entities or fields in order to denormalize your data model.

Transformers

A **transformer** is a function associated with an entity (root or nested) that can manipulate the fields fetched by the entity itself. The transformer must be declared as an attribute of the target entity:

```
<entity name="author" transformer="script:createAuthorFullName">
```

The corresponding function will be called for each set of fields (record) fetched by the query associated with the entity. The function has complete control over the fetched record. It can remove, add, or replace fields.

In the previous example, the Solr schema includes an `author` field that is supposed to hold the complete name of the author (for example, Dante Alighieri). Now let's imagine that the `AUTHORS` table contains two separate columns instead—`FIRST_NAME` and `LAST_NAME`. With the help of a built-in script transformer, we can write a simple JavaScript function to combine the two fields:

```
<script><![CDATA[
  function createAuthorFullName(record) {
    var first = record.remove('FIRST_NAME');
    var last = record.remove('LAST_NAME');
```

```
    record.put('author', first + ' ' + last);
    return record;
  }
]]></script>
```

Note how we manipulated the current record by adding a new field (`author`) and removing the `LAST_NAME` and `FIRST_NAME` fields.

The following table lists the available built-in transformers:

Name	Description
ScriptTransformer	Executes a function written in JavaScript or another scripting language supported by Java.
DateFormatTransformer	Creates `java.util.Date` instances from string literals.
HTMLStripTransformer	Strips off HTML tags from field values.
LogTransformer	Logs messages using a given template.
NumberFormatTransformer	Creates number instances from string literals.
RegexTransformer	Uses regular expressions to manipulate data in fields.
TemplateTransformer	Puts values in a column by resolving an expression containing other columns. For example, the concatenation we got with the `ScriptTransformer` can also be done using this transformer: `<field name="author" template="${author.FIRST NAME}  ${author.LAST_NAME}"`

A transformer is simply a class that extends `org.apache.solr.handler.dataimport.Transformer` so, if the built-in portfolio doesn't meet your needs, it is always possible to create a custom implementation.

Entity processors

Each entity is handled by a so-called **EntityProcessor** that defaults to **SQLEntityProcessor**. This is because the relational database is the most popular type of data source.

However, when using a different data source such as HTTP, files or streams, the entity management logic should have its own specific requirements that most probably fall outside the area covered by `SQLEntityProcessor`. In these cases, you can override the default settings by explicitly declaring an `EntityProcessor` for a given entity.

As usual, there are a lot of built-in `EntityProcessor` instances but it is always possible to create a custom implementation by extending the `org.apache.solr.handler.dataimport.Entityprocessor` class.

The following table lists and describes available entity processors:

Name	Description
SqlEntityProcessor	This is the default entity processor assigned to each entity. It provides support to read and cache data from databases. It is used in conjunction with `JdbcDataSource`.
FileListEntityProcessor	Enumerates the list of files from a filesystem based on criteria specified in the associated entity (for example, base path, recursive, and filename pattern).
LineEntityProcessor	Reads from a datasource on a line-by-line basis and produces a field called `rawLine` for each line read.
MailEntityProcessor	Handles emails and attachments from POP3 or IMAP sources.
PlainTextEntityProcessor	Reads from a datasource and returns a field called plainText. This field contains a string representing the source content.
SolrEntityProcessor	Reads values from another Solr instance using Solrj. Each returned record is a `SolrDocument` instance.
TikaEntityProcessor	Extracts metadata and text from rich documents by means of Apache Tika. Later, we will see the Content Extraction Library, which also uses Tika as the extraction engine.
XPathEntityProcessor	Uses a streaming XPATH parser to extract values from XML documents.

Event listeners

The document element in the `DataImportHandler` configuration allows us to declare two event listeners to intercept the most relevant events of a data import life cycle—`onImportStart` and `onImportEnd`:

```
<document
  onImportStart="com.foo.MyImportStartEventListener"
  onImportEnd="com.foo.MyImportEndEventListener">
```

The event listeners must implement the `org.apache.solr.handler.dataimport.EventListener` interface, which gives them access (by means of an `org.apache.solr.handler.dataimport.Context` instance) to most `DataImportHandler` objects and event statistics such as documents skipped, indexed, failed, and so on.

Content Extraction Library

The **Content Extraction Library** (also known as **SolrCell**) integrates the popular **Apache Tika** framework to detect and extract metadata and text from a large variety of file types such as PDF, Microsoft Office, Libre Office, and Open Office documents.

Apache Tika provides a façade parser interface on top of several low-level frameworks that are able to manage and manipulate specific file types (for example, **PDFBox** for PDFs and **Apache POI** for Microsoft documents). Its simple interface also provides automatic mime-type detection, so the framework itself is able to understand the correct parser that needs to be applied for a given file.

On the Solr side, a dedicated `ExtractingRequestHandler` will be in charge of getting the input data (files) sent by clients and extracting metadata and text by means of Tika.

The configuration of `ExtractingRequestHandler` follows the same procedure that we saw for the other handlers. Specifically, it has to be declared in `solrconfig.xml`, as follows:

```
<requestHandler name="/update/extract"
  class="solr.extraction.ExtractingRequestHandler">
  <lst name="defaults">
    …
  </lst>
</requestHandler>
```

SolrCell has several options that can be configured to fine-tune its behavior. Most of them are related to metadata handling, field name mapping, and custom Tika configuration.

For a complete list of all configuration parameters, go to `https://cwiki.apache.org/confluence/display/solr/Uploading+Data+with+Solr+Cell+using+Apache+Tika`.

The `src/solr/solr-home/example-data` folder in the example project contains a document that can be sent to SolrCell. Open a shell and type the following (replace the `PROJECT_HOME` placeholder with your `ch7` project local path):

```
# curl
  "http://localhost:8983/solr/example/update/extract?commit=true"
  -F data=@PROJECT_HOME/ch7/src/solr/solr-home/example-
  data/libreoffice-writer.odt
```

Wait for a moment, and then you should see a response like this:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">572</int>
  </lst>
</response>
```

The document (the LibreOffice document in this case, but you can also try other files) has been indexed. You can see that, when you open the browser and type `http://127.0.0.1:8983/solr/example/select?q=stream_name:libreoffice-writer.odt&indent=true`, the XML response shows the extracted text (under the text attribute) and all the metadata fields that have been detected for that document.

Language Identifier

The language Identifier extension detects the language (or languages) of fields belonging to a given document. This is a very useful add-on to use in conjunction with the previously described extraction library, to get additional information about data that has been indexed.

The component is implemented as an `UpdateRequestProcessor` subclass that intercepts and analyzes the incoming data:

```
<processor class="org.apache.solr.update.processor.
  TikaLanguageIdentifierUpdateProcessorFactory">
  <str name="langid.fl">text</str>
  <str name="langid.langField">language</str>
  <str name="langid.fallback">en</str>
</processor>
```

As you can see, this processor can be configured with several options. We can declare the fields of the incoming documents that must be analyzed, the name of the field that will hold the results of language detection, or a default fallback language in case no detection is possible.

In the example project associated with this chapter, you will find a `solrconfig.xml` file where the chain is already defined but the `UpdateRequestProcessor` is commented out. Just remove the comment markers, reload the core using the Administration Console, and reindex the documents under the `example-data` folder, following the same procedure as we described in the previous section. At the end, you will see an additional "language" field in each document; that is the result of the language detection component.

You should know that declaring the processor within the `solrconfig.xml` file is not enough. We need to insert that into an update request processor chain, and finally associate that chain with an `UpdateRequestHandler`. Only those update requests that will be received by that handler will pass through the language detection analysis chain.

Rapid prototyping with Solaritas

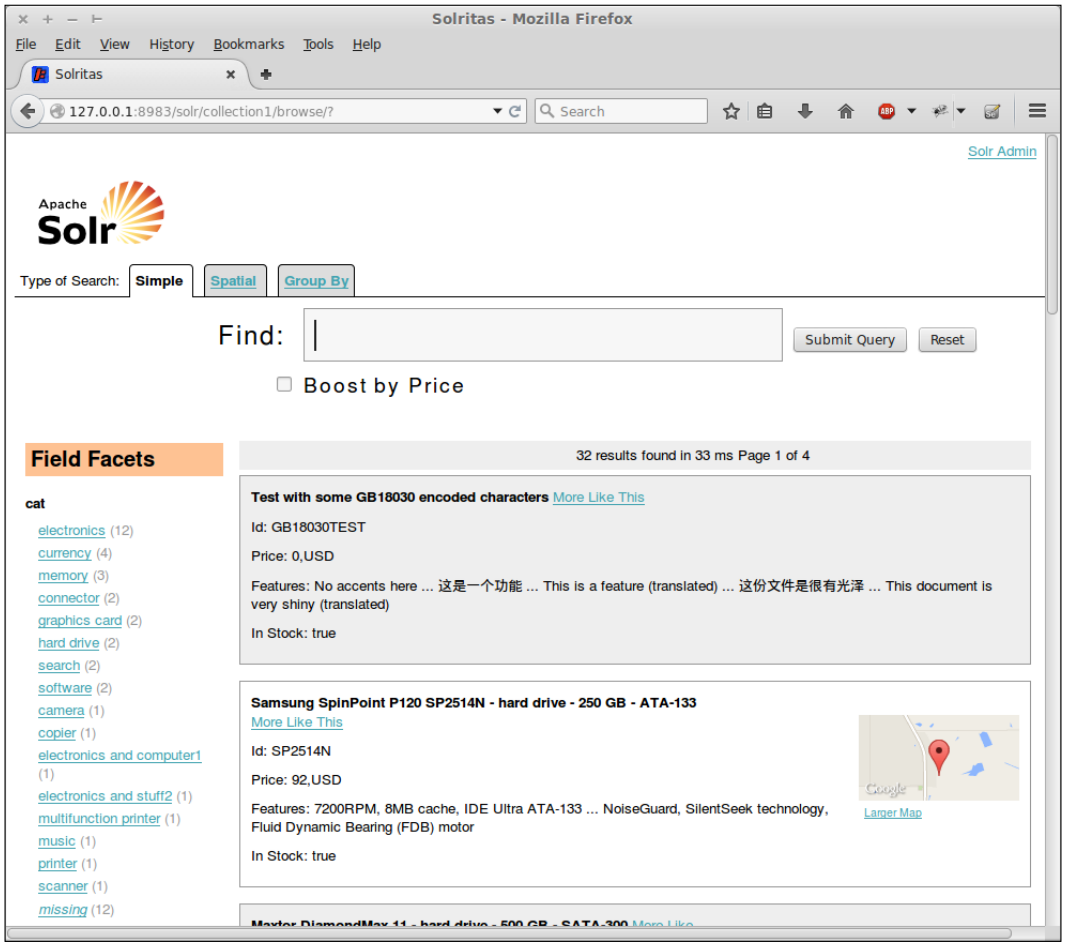
Solritas is the name of a contribution module that integrates Solr with **Apache Velocity**. It is basically a response writer that uses the Apache Velocity template engine to render Solr responses with a graphical user interface.

A set of ready-to-use Velocity templates is combined with Solr responses in order to provide a search GUI with a lot of features (for example, faceting, highlighting, and autocompletion).

You can find the Velocity templates under the `src/solr/solr-home/example/conf/velocity` folder of the `ch7` project, or under the `example/solr/collection1/conf/velocity` folder of the Solr download bundle.

As this GUI is directly provided by transforming the emerging Solr responses, there's no need for an external web application to execute searches and graphically see the corresponding results.

Okay, one could now say, "This is already possible with the Solr REST services", but that is definitely more technically complex and the search results are displayed in XML or JSON or whatever format. Here, a more user-friendly interface is provided, as shown in the following screenshot:



That makes Solritas an ideal choice to build rapid prototypes. The sample instance you started at the beginning of this chapter has Solritas configured in `solrconfig.xml`. It responds to the `/solritas` endpoint, so after indexing some data from the previous paragraph, open your browser and type `http://127.0.0.1:8983/solr/example/solritas`.

The Velocity templates have been copied from the Solr download bundle, so some areas (such as Google Maps widgets, spatial queries, and range queries) might not be visible or might not make sense with the chapter's sample data. If you want to see all of them in action, just start the Solr example in the download bundle and navigate to `http://127.0.0.1:8983/solr/browse` address.

You should see Solritas' results page, which is preloaded with a *:* query by default.

Other extensions

The `contrib` folder contains other modules or plugins that are briefly described in the following sections.

Clustering

The clustering module is a framework used to plug in third-party (clustering) implementations. At the time of writing this book, it provides support for clustering search results using the **Carrot2** project.

The Solr example that comes with the download bundle already contains a `ClusteringComponent` within the `solrconfig.xml` configuration file. The declaration happens in two phases. First, the component has to be configured:

```
<searchComponent
  name="clustering"
  enable="${solr.clustering.enabled:false}"
  class="solr.clustering.ClusteringComponent" >
  <lst name="engine">
    <str name="name">lingo</str>
    <str name="carrot.algorithm">org.carrot2.
    clustering.lingo.LingoClusteringAlgorithm</str>
    <str name="carrot.resourcesDir">clustering/carrot2</str>
  </lst>
  …
</searchComponent>
```

After this, as with any other `SearchComponent`, you should enable it by including its name in the `RequestHandler` instance where it is supposed to play:

```
<requestHandler name="/myRequestHandler"
  class="solr.SearchHandler">
  …
  <arr name="last-components">
    <str>clustering</str>
  </arr>
</requestHandler>
```

In this way, it can contribute to search results by adding a "clusters" section, like this:

```
<response>
  <result>
    …
  </result>
  <arr name="clusters">
  <arr name="labels">
    <str>iPod</str>
  </arr>
  <double name="score">1.3174612693376382</double>
  <arr name="docs">
    <str>F8V7067-APL-KIT</str>
    <str>IW-02</str>
    …
  </arr>
  <arr name="labels">
    <str>Hard Drive</str>
  </arr>
  …
</response>
```

If you want to try this yourself, open a shell and type the following commands:

```
# cd $INSTALL_DIR/example
# java -Dsolr.clustering.enabled=true -jar start.jar
```

These will start Solr with the `ClusteringComponent` enabled. Now, on another shell type this:

```
# cd $INSTALL_DIR/example/exampledocs
# ./post.sh *.xml
```

Finally, open a browser and execute this query:

```
http://localhost:8983/solr/clustering?q=*:*&rows=10
```

You should get a response similar to the preceding example, with the "clusters" section at the bottom.

UIMA Metadata Extraction Library

This module integrates **Apache UIMA** in Solr by providing a powerful Metadata Extraction Library that can be used for tasks such as automatic keyword extraction and Named Entity Recognition (for example, places, names, concepts, and dates).

The plugin can be provided both as an `UpdateRequestProcessor` subclass, to decorate the index process chain, or as a set of `Tokenizers`/`Filters`, to add such behavior in the (index or query) text analysis phase.

Using this module, you can enrich your Solr documents with additional metadata information extracted from the input data. UIMA provides an analysis engine that involves several components arranged in a pipeline. The default pipeline supports the use of existing analysis engines such as **Alchemy** or **OpenCalais**. Keep in mind that these engines are not free-of-charge, but they provide a free trial period. You can register and obtain an API key that must be configured in the `solrconfig.xml` file. Other components are used for language and sentence detection.

Under the `contrib/uima` folder, you will find a `README` file with detailed information about the Solr UIMA module usage.

The UIMA `UpdateRequestProcessor` intercepts the documents that are being indexed and sends them to its analysis pipeline. Those documents will be automatically enriched with extracted information such as sentences, languages, or named entities (for example, places or names).

MapReduce

The **MapReduce** contrib module provides integration with **Apache Hadoop**. MapReduce is the name of a paradigm (programming model) that is implemented in Apache Hadoop to process large datasets with a parallel and distributed algorithm.

The contribution contains a MapReduce job to build Solr indexes and merge them into a Solr cluster.

Summary

In this chapter, we illustrated a set of contribution modules that are not part of the Solr core but definitely useful in a lot of real scenarios. The Solr download bundle contains all of them, and their installation is very easy. Each module folder has a `README` file that guides you through installation and setup steps (basically, it's just a matter of copying, pasting, and configuring).

In the next chapter, we will conclude our Solr path with an overview about the Solr code base. You will learn how to work with it and eventually how to contribute to the open source community process.

8
Contributing to Solr

A friend of mine used to say, "Is there a better way to start a new year than contributing to an open source project?" I strongly agree; a great way to get involved in the open source world is to contribute to the projects you're using.

Being a user of an open source software, you are already part of that world—an important part that makes that software useful. But there's more; you can delve more deeply into what actually happens behind the scenes.

By the end of this chapter, you will have a good understanding of the following topics:

- The constituent pieces of the open source world
- The Apache contribution process
- How to work with Solr source code in your IDE

Identifying your needs

Why are you interested in the open source contribution process? Why do you want to have the Solr source code in your IDE? These are crucial questions you should answer before doing all that is described in this chapter. In my opinion, you could fall under one of these scenarios:

- **Curiosity**: You want to inspect and see with your eyes how things are working behind the scenes.
- **Bug fixing**: You want to fix a bug that you met in your Solr installation. In this way you, will satisfy your customer and the community will benefit from your work.

- **Improvement**: You've got an idea about an interesting feature not yet implemented. Probably, a customer requirement led to that idea, and you believe that it could be useful for other users if (once implemented) it would be integrated in Solr.
- **Wanting to contribute**: You simply want to contribute by fixing an existing issue and participating in the development/contribution process.

While curiosity could be a good reason to start investigating source code, sooner or later (and I would add most probably), you will fall into one of the other categories. At that time, you will necessarily start communicating with other people and the communities associated with the project.

You can find a general introduction about the Apache contribution process at `http://www.apache.org/foundation/getinvolved.html`.

That interaction will involve some general aspects such as issue tracking, mailing lists, software development, and so on. Once you have identified your needs and goals, you can look at upcoming sections to get a description about those cross-cutting concepts.

An example – SOLR-3191

In 2013, I was working on an **Online Public Access Catalogue** (**OPAC**) project for a big library. The schema definition became huge very soon, because the MARC, the standard representation for bibliographic records, is an old and proven standard that classifies each minimal piece of information about a catalog item.

Obviously, our customer required all that richness in the search application, so we started with a small schema and quickly ended up with a lot of fields.

Another requirement was the capability to download each item in MARCXML format (MARCXML is the XML representation of a MARC record) in the end user application. So, in order to satisfy that requirement, we put the whole MARC representation in a dedicated stored field called, not surprisingly, `marc_xml`.

What was the problem? On the Solr side, we defined a lot of `SearchHandler` instances, one for each kind of search (for example, any keyword, author, title, or subject). As you know, for each handler we have to declare all (stored) fields that must be in the search results using the `fl` parameter.

In the first approach, we simply put a wildcard (`*`) as a value for the `fl` parameter, as most parts of those fields were needed in the user interface. But after it had been running for a while in production, the IT department, in charge of monitoring the system, raised an issue about the network traffic between the frontend application and the Solr server. After doing some analysis, we discovered a lot of records with a huge `marc_xml` field returned to the client. "Ok," said one of the IT guys to us, "just exclude the marc_xml field from the `fl` parameter".

The `fl` parameter accepts a list of fields that *must* be returned, but there's no way to tell it what *must not* be in the search results. Eight handlers were defined in the `solrconfig.xml` file, and for each of them (later, we discovered the XInclude feature, but that's another story), we had to declare all stored fields, excluding the `marc_xml` field. This was terrible and unmaintainable!

After googling a bit, I found several guys facing the same problem, so I decided to take a look at an existing JIRA issue. Thus, I met the (unsolved) **SOLR-3191** issue at `https://issues.apache.org/jira/browse/SOLR-3191`, which describes the problem:

> *SOLR-3191 field exclusion from fl*
>
> *I think it would be useful to add a way to exclude field from the Solr response. If I have for example 100 stored fields and I want to return all of them but one, it would be handy to list just the field I want to exclude instead of the 99 fields for inclusion through fl*

So I thought to myself: why don't you try to implement that feature? And I did what I'm going to describe in this chapter. If you take a look at that issue, you will see I submitted two patches and had some exchange with a couple of Solr guys.

Subscribing to mailing lists

If you haven't subscribed to a Solr mailing list (or lists) yet, you should do that before going ahead. User and developer lists are the primary place where things such as doubts, questions, features, and bugs are discussed.

It's mainly there that you should look to solve your problem and meet people with similar requirements. Like any other Apache project, Solr has the following mailing lists:

- A user list – `solr-user@lucene.apache.org`
- A dev list – `dev@lucene.apache.org`
- A commits list – `commits@lucene.apache.org`

Every Solr user should be subscribed to the user list. This usually avoids the need to reinvent the wheel by getting ideas and solutions from users and developers.

The dev list is meant for listening or participation in discussions on Lucene and Solr internals, developments, upcoming features, and so on. The focus here is more technical.

Finally, the commits list is used to receive notifications about every Solr or Lucene commit.

Subscribing to a list is very easy; just send an empty email to `solr-user-subscribe@lucene.apache.org`, `dev-subscribe@lucene.apache.org`, or `commits-subscribe@lucene.apache.org`, and then follow the procedure written in the answering mail.

Signing up on JIRA

The issue tracker is another important building block of the open source contribution process. Whenever an idea, question, bug, or feature becomes something that could affect the code, a new JIRA issue is filled, and all things related to that (for example, tasks, discussions, patches, code, and commit logs) will be put there.

Issues in JIRA are public, so if you want to only see or read them there's no need to have an account (you should have already read the SOLR-3191 issue on JIRA, without having an account).

However, if you want to participate in a discussion, post a patch, or create or update issues, you must sign up at `https://issues.apache.org/jira/secure/Signup!default.jspa`.

Ultimately, you can sign in using the login form at `https://issues.apache.org/jira/login.jsp`.

That's all! Welcome to the Apache Issue Tracker! Note that, before opening a new issue, it is always better to ping the dev list and discuss it. Maybe, a similar issue already exists and someone is working on it.

Setting up the development environment

Following the same logic that was used in the previous chapters, I will assume you have Eclipse installed. If that is not the case, that is, if you followed the examples using some other IDE (for example, IntelliJ), a few steps could be a bit different.

In order to be able to modify, build, and run Solr from the source code, you need the following:

- An IDE such as Eclipse or IntelliJ
- A Subversion client, which can be a standalone client (such as the svn command-line tool or TortoiseSVN) or a plugin in your IDE (for example, Subclipse or Subversive)
- Apache ANT (`http://ant.apache.org/bindownload.cgi`)

Version control

Subversion is an open source version control system that is used to maintain the source code of the Apache projects, including Solr.

As a first step, you need to check out the Solr source code from the SVN repository. Depending on your role, you should point to one of the following addresses:

- `http://svn.apache.org/repos/asf/lucene/dev/<branch>`
- `https://svn.apache.org/repos/asf/lucene/dev/<branch>`

As you can see, the only difference in the preceding links is in the protocol. The first link, which uses **http**, is for anonymous checkout, and the other, which uses **https**, is for committers. Committers are those people who have commit rights, that is, active members of the development community with write permissions on the repository. I assume you don't fall within this last category, so the correct link is the first.

The link also contains a `<branch>` placeholder. This must be replaced with the correct target version you will work on. That strictly depends on the task you would like to do. If you want to fix a bug in a past version (for example, 4.7.2), you should point to the corresponding branch. If you want to pick up an existing enhancement or bug that has been scheduled for the next major release, you should point to the "trunk" leg. The following table describes how the repository tree is organized (`http://svn.apache.org/repos/asf/lucene/dev/`):

Folder	Description
`branches`	Development branches.
`branches/branch_5x`	The development branch for the next version, 5.x.
... `branches/Lucene_solr_3_6` ... `branches/Lucene_solr_4_10`	The development branches for versions that have been released. Apart from some tasks that have been scheduled for a given release, most of the development activities done in these branches are bug fixes.
`tags`	When a new version is released, the corresponding source code is copied here, in a dedicated folder (for example, `tags/lucene_solr_3_6_1` and `tags/lucene_solr_4_10_3`).
`trunk`	This is the main center of development.

The target branch depends on what you would like to do. If you pick up an existing JIRA among its attributes, you will also find the affected version. Besides, you may want to fix an issue in an older version (for example, 3.6.1) because your customer is using that specific version.

Keep in mind that most development tasks are done in the trunk and then reported to the corresponding active development branch (under the `branches` folder). Anyway, before starting, it is always recommended to ping the dev list explaining what you want to do.

Code style

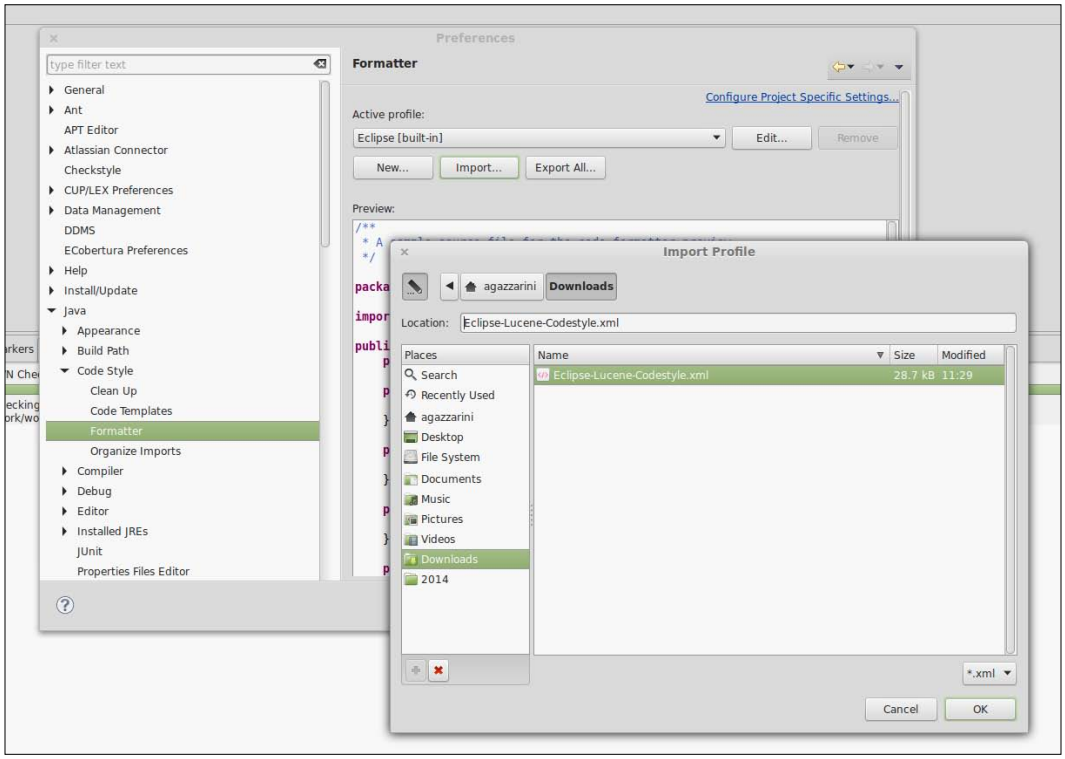
One of the common problems in a distributed development is the agreement about source code formalisms: comments, naming conventions, and so on.

That's the reason the Solr development team provided two useful configuration files—one for Eclipse and another for IntelliJ. These files can be imported to those IDEs to automate a lot of things such as indentation, braces positions, line wrapping, comments, and so on.

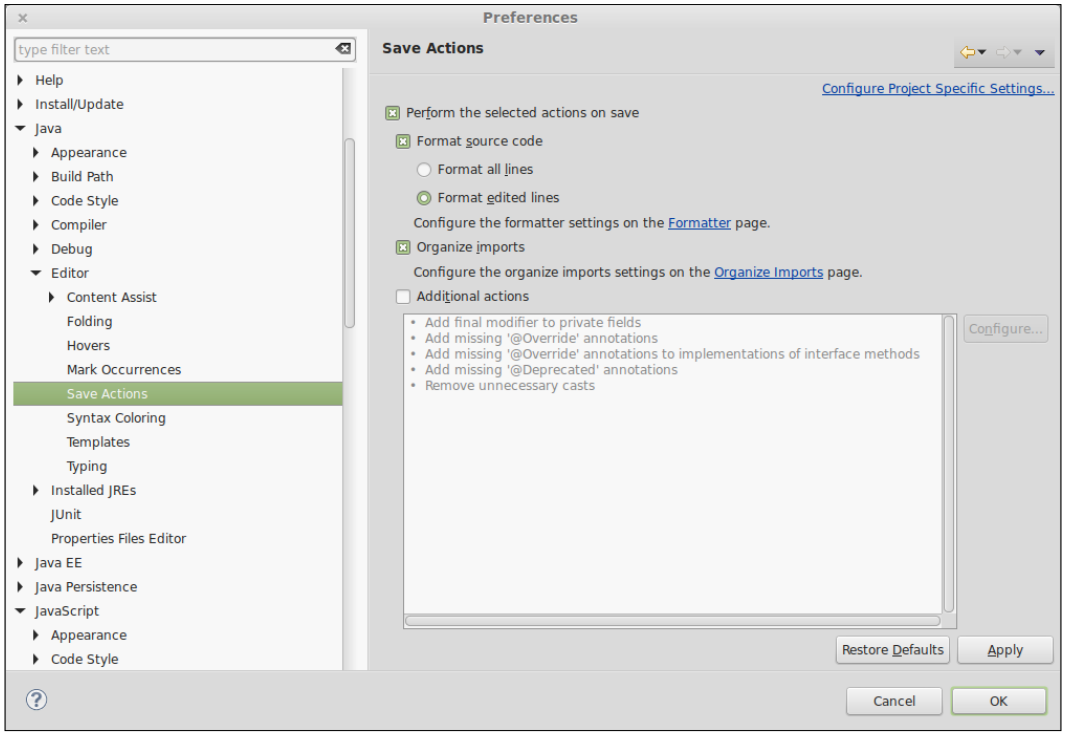
Pick up that file from one of the following addresses, depending on your favorite IDE:

- **Eclipse**: `http://people.apache.org/~rmuir/Eclipse-Lucene-Codestyle.xml`
- **IntelliJ**: `http://people.apache.org/~erick/Intellij-Lucene-Codestyle.xml`

In Eclipse, the configuration file can be imported by going to **Window** | **Preferences** | **Java** | **Code Style** | **Formatter** and then clicking on the **Import** button, as shown in the following screenshot:



After that, navigate to **Java** | **Editor** | **Save Actions**. Select the **Perform the selected actions on save** checkbox and the **Format edited lines** radio button, as shown in this screenshot:



Checking out the code

Once you have identified the target branch to work on, check out the source code using the svn command-line tool or your favorite tool (for example, TortoiseSVN).

SOLR-3191 was considered a new feature at that time, so I checked out the trunk. The current trunk requires Java 8 in order to build so, to execute the steps needed in this chapter, let's point to a different branch (`5_x`). Open a shell and type the following command:

```
# cd /work/solrdev
# svn checkout
http://svn.apache.org/repos/asf/lucene/dev/branches/branch_5x solr_5
```

Bear in mind the following:

- I'm not a committer, so I pointed to the read-only (`http`) address.
- The name of the local folder that will contain the downloaded source is `solr_5`. If it doesn't exist, it will be automatically created.
- The `/work/solrdev/solr_5` folder is a local working folder on my machine. You can choose whatever name you like.

When you execute that command, a lot of files will be downloaded. In the end, you should see something like this:

```
…
A    solr_5/solr/test-framework/src/java/overview.html
A    solr_5/.hgignore
U   solr_5
Checked out revision 1651057.
```

Now the source code of Solr 5_x is in your machine.

Creating the project in your IDE

Getting the source code is not enough, unless you want to develop your patch using Vim. You will have to create a project in your IDE. Assuming you are in the `/work/solrdev/solr_5` folder you created in the previous step, type the following:

```
# ant clean test
```

The `ant` command will immediately fail because the build requires **Ivy** (a dependency management tool), and you don't have that on your machine. No problem! There's a dedicated task that can install Ivy for you. Type this command:

```
# ant ivy-bootstrap
```

You should see something like this:

```
…
ivy-bootstrap2:
ivy-checksum:
ivy-bootstrap:

BUILD SUCCESSFUL
Total time: 3 seconds
```

Now we can retry the first command:

```
# ant clean test
```

This will execute the whole test suite, which is very huge, so take a long coffee break!

Although this step is not mandatory, it is strongly recommended to check the state of your build before making any change. In this way, you can see whether there's something failing, something that doesn't have to do with your changes.

Once the test suite has been executed, type this command if you are using Eclipse:

```
# ant eclipse
```

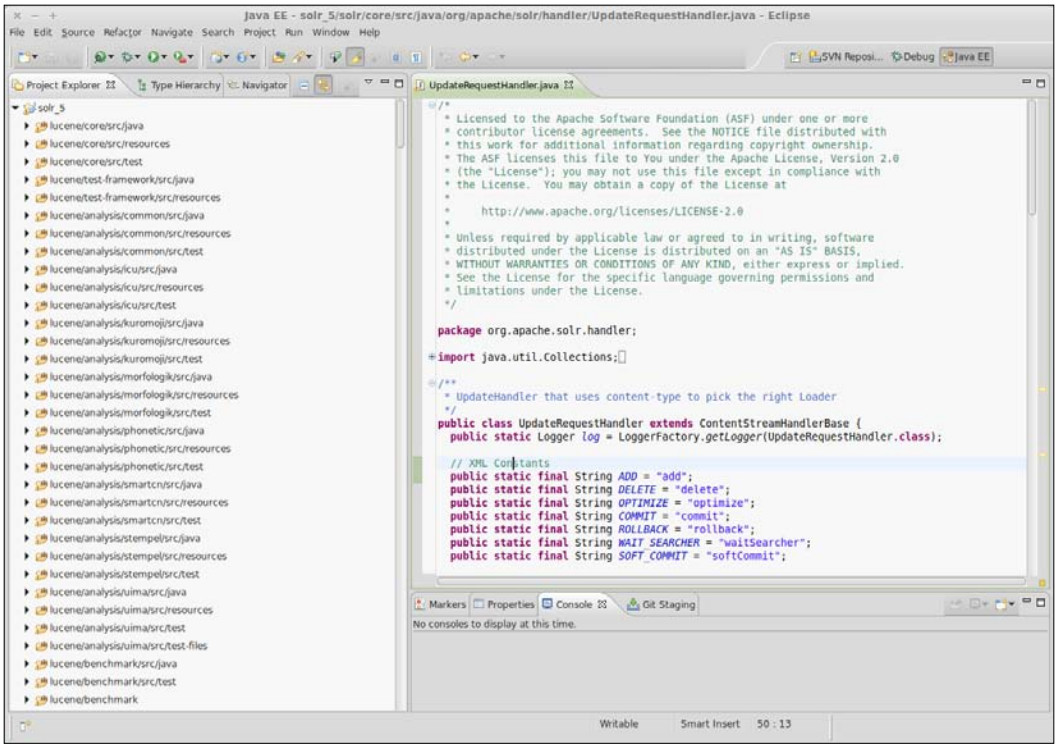
If you are using IntelliJ, type the following command:

```
# ant idea
```

This will generate the IDE project files within the current directory (`solr_5`). From here on, I will assume you're using Eclipse, but the steps are basically the same for IntelliJ.

Open Eclipse and create a new workspace (you can also use the workspace where you loaded the sample projects of this book).

Open the **File** menu and choose **Import**. From the dialog that appears, go to **General | Existing Projects into Workspace**. Using the **Browse** button, select the `/work/solrdev/solr_5` folder. Press **Ok** and then **Confirm**. The dialog will close and the project will be imported, as shown in this screenshot:



Once the project has been built, you shouldn't have any errors. Everything is ready, and you can proceed with your change.

Making your changes

We won't dig very deep in this step because it basically depends on the nature of the task you picked up. For instance, my SOLR-3191 patch contains four existing classes that I changed to implement that specific behavior.

Since nobody knows you and your changes will be hopefully integrated in a very popular framework, the most important things to keep in mind are as follows:

- **Correctness**: The implementation must do what it is supposed to do, according to the requirements expressed in the JIRA issue
- **Documentation**: Javadoc at class and method levels (don't include the `@author` tag)
- **Unit tests**: These describe and validate your changes

Returning to the SOLR-3191 example, I changed two classes:

- `org.apache.solr.search.ReturnFields`
- `org.apache.solr.search.SolrReturnFields`

These classes contain the logic required by the issue. At the same time, I updated two `TestCase` classes with several unit tests demonstrating and validating my changes:

- `org.apache.solr.search.ReturnFieldsTest`
- `org.apache.solr.search.TestPseudoReturnFields`

During development, it's better to periodically execute the test suite, in order to ensure that your changes didn't introduce any side-effect.

When working in a distributed development environment, it is strongly recommended you run an `svn update` command frequently. In this way, you will always be working with the latest version of the branch you checked out.

Okay, take your time and make your changes. Remember to post a message in the issue page in JIRA for every relevant doubt. In this way, all of the history of your work will be in one place.

Creating and submitting a patch

Once the implementation has been completed, everything is working, and the tests are green, it's time to submit the patch.

Before doing that, open a shell on the `/work/solrdev/solr_5` working folder and type this:

```
# ant precommit
```

This task will look for problems related to tab indentation, author tags, and broken or wrong links in javadoc. At the end, type the following command:

```
# svn stat
```

You will see a list of source files that have been changed. If all of them are associated with your changes, just type this command in order to include them in the patch:

```
# svn stat | grep "^?" | awk '{print $2}' | xargs svn add
```

Alternatively, you can add those files one by one, using the following command:

```
# svn add <file>
```

Finally, type this command to generate a patch:

```
# svn diff > /work/patches/SOLR-XXXX.patch
```

That will create a new file (`SOLR-XXXX.patch`) under the `/work/patches` local folder. Here are a couple of things to note:

- `/work/patches` is a sample local directory that I've created on my machine. You can put the patch in a different folder.
- `XXXX` is supposed to be replaced with the number of the corresponding JIRA issue. If you are updating an existing patch, the name should always follow this convention because JIRA will take care of highlighting the newest version.

If you've installed an SVN plugin on your IDE (such as Subclipse or Subversive in Eclipse), you can do everything without using the command-line. In Subclipse, for example, there's a **Create Patch** under **Team** that will guide you through the necessary steps with an easy wizard.

Once you've got the patch file, open a browser, log in to JIRA, go to the issue page, and upload the patch. It is recommended you post a comment with information (including a description) about your submission. That's all! Now you should follow your issue because several things can happen:

- The patch is perfect, so it's just a matter of time and it will be applied.
- Some questions come from JIRA users. In that case, you may want to participate in a discussion that might eventually request a new version of the patch.

Anyway, the big part is done! You've actively participated in the contribution process, and hopefully your artifact will be integrated with Solr. Congrats!

Other ways to contribute

Besides writing code, there are other ways to participate in an open source project. After all, the software is just a component of a final product. We can find support and documentation, which in most cases make the real difference between a good and a bad product from the user's perspective.

Documentation

Software quality is described by a combination of several factors: functional and non-functional features, internal and external qualities, and last but not least, documentation.

By "documentation", I personally mean a complex and huge world made up of different types of information for different types of target audience:

- **Technical internal documentation**: Strictly needed by active developers to inform about the structure or the implementation of the system.
- **Technical external documentation**: Crucial for open source projects representing frameworks, things that can be extended. This is sometimes called the **developer guide**. This kind of information documents the public API and the extension points that let developers integrate the product with their applications.
- **User documentation**: This enables end users to understand the usage and power of a given system. It is sometimes called a **user guide** and is the primary source of information for an end user.

Solr has two main places where documentation can be found:

- The reference guide, available online at `https://cwiki.apache.org/confluence/display/solr/About+This+Guide`, or in PDF format
- The Solr community Wiki, at `https://wiki.apache.org/solr`

The first is a guide constituting the official reference documentation. It is created and maintained by Solr committers. On the other hand, the Wiki is a public and collaborative tool. Anyone can potentially edit its content by creating an account and then requesting write grants from the Solr team. For detailed instructions refer to `http://wiki.apache.org/solr/#How_to_edit_this_Wiki`.

Mailing list moderator

A list moderator is a kind of supervisor for a given mailing list and a user with elevated privileges. He can get a list of all subscribers and manually subscribe or unsubscribe a given user.

He checks emails sent to the list from addresses that are not subscribed in order to improve spam filter rules. He also helps users who face issues related with lists (for example, subscription and un-subscription).

Summary

In this final chapter, we illustrated the overall contribution process. Being an open source project, the Solr team warmly welcomes any kind of contribution: source code, bug fixing, documentation, and active participation in the mailing lists. There's no need to be a committer, which would be surely an ambitious goal for a developer. It's always possible to download the source code, change it, and eventually (if you think the changes could also be useful for other people) create a patch and submit it to the community.

Index

Symbols

A

B

C

D

E

F

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

Thank you for buying
Apache Solr Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

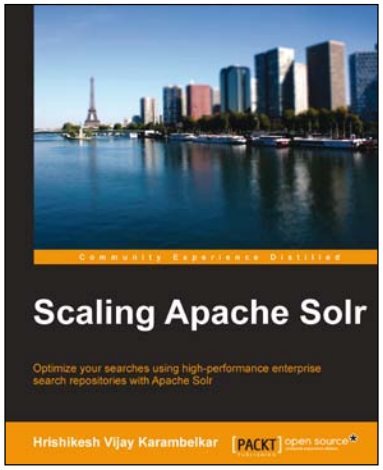
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

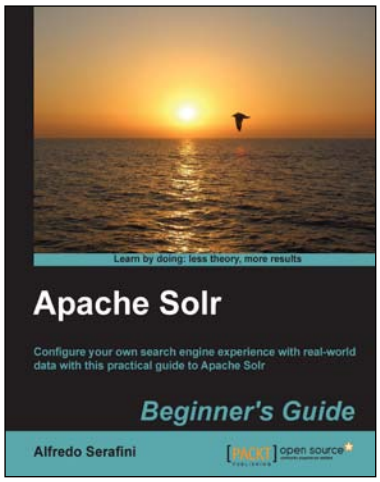


Scaling Apache Solr

ISBN: 978-1-78398-174-8 Paperback: 298 pages

Optimize your searches using high-performance enterprise search repositories with Apache Solr

1. Get an introduction to the basics of Apache Solr in a step-by-step manner with lots of examples.
2. Develop and understand the workings of enterprise search solution using various techniques and real-life use cases.
3. Gain a practical insight into the advanced ways of optimizing and making an enterprise search solution cloud ready.

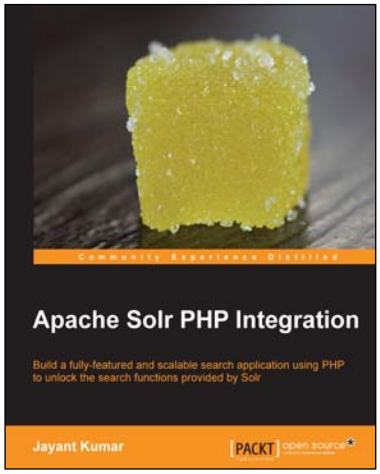


Apache Solr Beginner's Guide

ISBN: 978-1-78216-252-0 Paperback: 324 pages

Configure your own search engine experience with real-world data with this practical guide to Apache Solr

1. Learn to use Solr in real-world contexts, even if you are not a programmer, using simple configuration examples.
2. Define simple configurations for searching data in several ways in your specific context, from suggestions to advanced faceted navigation.
3. Teaches you in an easy-to-follow style, full of examples, illustrations, and tips to suit the demands of beginners.



Apache Solr PHP Integration

ISBN: 978-1-78216-492-0 Paperback: 118 pages

Build a fully-featured and scalable search application using PHP to unlock the search functions provided by Solr

1. Understand the tools that can be used to communicate between PHP and Solr, and how they work internally.
2. Explore the essential search functions of Solr such as sorting, boosting, faceting, and highlighting using your PHP code.
3. Take a look at some advanced features of Solr such as spell checking, grouping, and auto complete with implementations using PHP code.

Getting started with Apache Solr Search Server [Video]

ISBN: 978-1-78216-084-7 Duration: 2:30 hrs

Integrate Solr as a blazing-fast open-source search solution into your enterprise web application and take your application to the next level

1. Teaches you everything you need to know to get started with Apache Solr such as indexing, querying, configuration, and implementation.
2. Learn how to define a search architecture specific to your business needs.
3. Includes walk-throughs on the Solr admin interface.