



Quick answers to common problems

# Apache Solr 3.1 Cookbook

Over 100 recipes to discover new ways to work with Apache's  
Enterprise Search Server

Rafał Kuć

[**PACKT**] open source\*  
PUBLISHING community experience distilled

# Apache Solr 3.1 Cookbook

Over 100 recipes to discover new ways to work with  
Apache's Enterprise Search Server

**Rafał Kuć**

**[PACKT]** open source   
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

# **Apache Solr 3.1 Cookbook**

Copyright © 2011 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: July 2011

Production Reference: 1140711

Published by Packt Publishing Ltd.  
32 Lincoln Road  
Olton  
Birmingham, B27 6PA, UK.

ISBN 978-1-849512-18-3

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Rakesh Shejwal ([shejwal.rakesh@gmail.com](mailto:shejwal.rakesh@gmail.com))

# Credits

**Author**

Rafał Kuć

**Reviewers**

Ravindra Bharathi

Juan Grande

**Acquisition Editor**

Sarah Cullington

**Development Editor**

Meeta Rajani

**Technical Editors**

Manasi Poonthottam

Sakina Kaydawala

**Copy Editors**

Kriti Sharma

Leonard D'Silva

**Project Coordinator**

Srimoyee Ghoshal

**Proofreader**

Samantha Lyon

**Indexer**

Hemangini Bari

**Production Coordinators**

Adline Swetha Jesuthas

Kruthika Bangera

**Cover Work**

Kruthika Bangera

# About the Author

**Rafał Kuć** is a born team leader and software developer. Right now, he holds the position of Software Architect, and Solr and Lucene specialist. He has eight years of experience in various software branches, from banking software to the e-commerce products. He is mainly focused on Java, but is open to every tool and programming language that will make the achievement of his goal easier and faster. Rafał is also one of the founders of the `solr.pl` site where he tries to share his knowledge and help people with their problems with Solr and Lucene.

Rafał began his journey with Lucene in 2002 and it wasn't love at first sight. When he came back to Lucene in late 2003, he revised his thoughts about the framework and saw the potential in search technologies. Then Solr came and that was it. From then, Rafał has concentrated on search technologies and data analysis. Right now, Lucene and Solr are his main points of interest. Many Polish e-commerce sites use Solr deployments that were made with the guidance of Rafał and with the use of tools developed by him.

---

The writing of this book was a big thing for me. It took almost all of my free time for the last six months. Of course, this was my sacrifice and if I were to make that decision today, it would be the same. However, the ones that suffered from this the most was my family—my wife Agnes and our son Philip. Without their patience for the always busy husband and father, the writing of this book wouldn't have been possible. Furthermore, the last drafts and corrections were made, when we were waiting for our daughter to be born, so it was even harder for them.

I would like to thank all the people involved in creating, developing, and maintaining Lucene and Solr projects for their work and passion. Without them, this book wouldn't be written.

Last, but not least, I would like to thank all the people who made the writing of this book possible, and who listened to my endless doubts and thoughts, especially Marek Rogoziński. Your point-of-view helped me a lot. I'm not able to mention you all, but you all know who you are.

Once again, thank you.

---

# About the Reviewers

**Ravindra Bharathi** holds a master's degree in Structural Engineering and started his career in the construction industry. He took up software development in 2001 and has since worked in domains such as education, digital media marketing/advertising, enterprise search, and energy management systems. He has a keen interest in search-based applications that involve data visualization, mashups, and dashboards. He blogs at <http://ravindrabharathi.blogspot.com>.

**Juan Grande** is an Informatics Engineering student at Universidad de Buenos Aires, Argentina. He has worked with Java-related technologies since 2005, when he started as web applications developer, gaining experience in technologies such as Struts, Hibernate, and Spring. Since 2010, he has worked as search consultant for Plug Tree. He also works as an undergraduate teaching assistant in programming-related subjects since 2006. His other topic of interest is experimental physics, where he has a paper published.

---

I want to acknowledge Plug Tree for letting me get involved in this project, and specially Tomás Fernández Löbbe for introducing me to the fascinating world of Solr.

---

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

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](http://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](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://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.



<http://PacktLib.PacktPub.com>

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

## Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Apache Solr Configuration</b>	<b>5</b>
Introduction	5
Running Solr on Jetty	6
Running Solr on Apache Tomcat	9
Using the Suggester component	11
Handling multiple languages in a single index	15
Indexing fields in a dynamic way	17
Making multilingual data searchable with multicore deployment	19
Solr cache configuration	23
How to fetch and index web pages	28
Getting the most relevant results with early query termination	31
How to set up Extracting Request Handler	33
<b>Chapter 2: Indexing your Data</b>	<b>35</b>
Introduction	35
Indexing data in CSV format	36
Indexing data in XML format	38
Indexing data in JSON format	40
Indexing PDF files	43
Indexing Microsoft Office files	45
Extracting metadata from binary files	47
How to properly configure Data Import Handler with JDBC	49
Indexing data from a database using Data Import Handler	52
How to import data using Data Import Handler and delta query	55
How to use Data Import Handler with URL Data Source	57
How to modify data while importing with Data Import Handler	60



<b>Chapter 3: Analyzing your Text Data</b>	<b>63</b>
Introduction	64
Storing additional information using payloads	64
Eliminating XML and HTML tags from the text	66
Copying the contents of one field to another	68
Changing words to other words	70
Splitting text by camel case	72
Splitting text by whitespace only	74
Making plural words singular, but without stemming	75
Lowercasing the whole string	77
Storing geographical points in the index	78
Stemming your data	81
Preparing text to do efficient trailing wildcard search	83
Splitting text by numbers and non-white space characters	86
<b>Chapter 4: Solr Administration</b>	<b>89</b>
Introduction	89
Monitoring Solr via JMX	90
How to check the cache status	93
How to check how the data type or field behave	95
How to check Solr query handler usage	98
How to check Solr update handler usage	100
How to change Solr instance logging configuration	102
How to check the Java based replication status	104
How to check the script based replication status	106
Setting up a Java based index replication	108
Setting up script based replication	110
How to manage Java based replication status using HTTP commands	113
How to analyze your index structure	116
<b>Chapter 5: Querying Solr</b>	<b>121</b>
Introduction	121
Asking for a particular field value	122
Sorting results by a field value	123
Choosing a different query parser	125
How to search for a phrase, not a single word	126
Boosting phrases over words	128
Positioning some documents over others on a query	131
Positioning documents with words closer to each other first	136
Sorting results by a distance from a point	138
Getting documents with only a partial match	141
Affecting scoring with function	143

Nesting queries	147
<b>Chapter 6: Using Faceting Mechanism</b>	<b>151</b>
Introduction	151
Getting the number of documents with the same field value	152
Getting the number of documents with the same date range	155
Getting the number of documents with the same value range	158
Getting the number of documents matching the query and sub query	161
How to remove filters from faceting results	164
How to name different faceting results	167
How to sort faceting results in an alphabetical order	170
How to implement the autosuggest feature using faceting	173
How to get the number of documents that don't have a value in the field	176
How to get all the faceting results, not just the first hundred ones	179
How to have two different facet limits for two different fields in the same query	181
<b>Chapter 7: Improving Solr Performance</b>	<b>187</b>
Introduction	187
Paging your results quickly	188
Configuring the document cache	189
Configuring the query result cache	190
Configuring the filter cache	191
Improving Solr performance right after the startup or commit operation	193
Setting up a sharded deployment	195
Caching whole result pages	198
Improving faceting performance	199
What to do when Solr slows down during indexing when using Data Import Handler	201
Getting the first top documents fast when having millions of them	202
<b>Chapter 8: Creating Applications that use Solr and Developing your Own Solr Modules</b>	<b>205</b>
Introduction	205
Choosing a different response format than the default one	206
Using Solr with PHP	208
Using Solr with Ruby	210
Using SolrJ to query Solr	212
Developing your own request handler	215
Developing your own filter	217
Developing your own search component	221
Developing your own field type	224

<b>Chapter 9: Using Additional Solr Functionalities</b>	<b>229</b>
Introduction	229
Getting more documents similar to those returned in the results list	230
Presenting search results in a fast and easy way	232
Highlighting matched words	235
How to highlight long text fields and get good performance	238
Sorting results by a function value	240
Searching words by how they sound	243
Ignoring defined words	245
Computing statistics for the search results	247
Checking user's spelling mistakes	250
Using "group by" like functionalities in Solr	254
<b>Chapter 10: Dealing with Problems</b>	<b>259</b>
Introduction	259
How to deal with a corrupted index	259
How to reduce the number of files the index is made of	262
How to deal with a locked index	263
How to deal with too many opened files	264
How to deal with out of memory problems	266
How to sort non-English languages properly	267
How to deal with the infinite loop exception when using shards	271
How to deal with garbage collection running too long	272
How to update only one field in all documents without the need of full indexation	274
How to make your index smaller	276
<b>Index</b>	<b>279</b>

# Preface

Welcome to the Solr cookbook. You will be taken on a tour through the most common problems when dealing with Apache Solr. You will learn how to deal with the problems with Solr configuration and setup, how to handle common querying problems, how to fine-tune Solr instances, how to write Solr extensions, and many more things. Every recipe is based on real-life problems, and each recipe includes solutions along with detailed descriptions of the configuration and code that was used.

## What this book covers

*Chapter 1, Apache Solr Configuration:* In this chapter, you will see how to handle Solr configuration, such as setting up Solr on different servlet containers, how to set up single and multicore deployments, and how to use Solr with Apache Nutch.

*Chapter 2, Indexing your Data:* In this chapter, you will learn how to send your data in different formats, how to handle binary file format, and what can be done when using the Data Import Handler.

*Chapter 3, Analyzing your Text Data:* This chapter will tell you how to overcome common problems you may encounter while analyzing your text data.

*Chapter 4, Solr Administration:* This chapter will guide you through the Solr administration panel, index replication setup, and its maintenance.

*Chapter 5, Querying Solr:* This chapter will help you with common problems when sending queries to Solr, such as searching for a word, phrase, boosting, sorting, and so on.

*Chapter 6, Using Faceting Mechanism:* This chapter will show you the beautiful world of the Apache Solr faceting mechanism, including tasks like getting the number of documents with the same field value, matching the same query, matching given range values, and so on.

*Chapter 7, Improving Solr Performance:* This chapter will help you troubleshoot performance problems with your Apache Solr instance. Here you will find tips about setting up your Solr instance caches, improving faceting performance, setting up a sharded deployment, or getting the first top documents out of millions fast.

*Chapter 8, Creating Applications that Use Solr and Developing your Own Solr Modules:* In this chapter, you will learn how to use Solr with different programming languages and how to develop your own modules for Apache Solr server.

*Chapter 9, Using Additional Solr Functionalities:* In this chapter, you will find recipes that will show you how to use additional Solr components, like the highlighting mechanism, spellchecker, statistics, or grouping mechanism.

*Chapter 10, Dealing with Problems:* This chapter concentrates on helping you with common problems when running the Apache Solr search server. In this part of the book, you will find recipes that will help you in dealing with a corrupted index, with out-of-memory errors, garbage collection, and so on.

## What you need for this book

In order to be able to run most of the examples in this book, you will need an Oracle Java Runtime Environment version 1.5 or newer, and of course the 3.1 version of Solr search server.

A few chapters require additional software, such as Apache Nutch 1.2 or your favorite IDE and a Java compiler to be able to edit and compile examples provided in the book.

## Who this book is for

Developers who are working with Apache Solr and would like to know how to combat common problems will find this book of great use. Knowledge of Apache Lucene would be a bonus, but is not required.

## Conventions

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

Code words in text are shown as follows: "It's based on the `solr.PointType` class and is defined by two attributes."

A block of code is set as follows:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="name">Solr cookbook</field>
  <field name="description">This is a book that I'll show</field>
</doc>
</add>
```


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


```
<fieldType name="text_stem" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.SnowballPorterFilterFactory" />
  </analyzer>
</fieldType>
```

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

```
java -jar post.jar ch3_html.xml
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "If you want to analyze the field type, just type in the appropriate name type and change the **name** selection to **field**."

 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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on [www.packtpub.com](http://www.packtpub.com) or e-mail [suggest@packtpub.com](mailto:suggest@packtpub.com).

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 on [www.packtpub.com/authors](http://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 for all Packt books you have purchased from your account at <http://www.PacktPub.com>. 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.

## 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 would 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/support>, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

## Piracy

Piracy of copyright 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](mailto: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

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1

# Apache Solr Configuration

In this chapter, we will cover:

- ▶ Running Solr on Jetty
- ▶ Running Solr on Apache Tomcat
- ▶ Using the Suggester component
- ▶ Handling multiple languages in a single index
- ▶ Indexing fields in a dynamic way
- ▶ Making multilingual data searchable with multicore deployment
- ▶ Solr cache configuration
- ▶ How to fetch and index web pages
- ▶ Getting the most relevant results with early query termination
- ▶ How to set up the Extracting Request Handler

## Introduction

Setting up an example Solr instance is not a hard task, at least when setting up the simplest configuration. The simplest way is to run the example provided with the Solr distribution, which shows how to use the embedded Jetty servlet container.

So far so good. We have a simple configuration, simple index structure described by the `schema.xml` file, and we can run indexing.



In this chapter, we will go a little further. You'll see how to configure and use the more advanced Solr modules. You'll see how to run Solr in different containers and how to prepare your configuration to meet different requirements. Finally, you will learn how to configure Solr cache to meet your needs and how to pre-sort your Solr indexes to be able to use early query termination techniques efficiently.

If you don't have any experience with Apache Solr, please refer to the Apache Solr tutorial that can be found at <http://lucene.apache.org/solr/tutorial.html> before reading this book.



During the writing of this chapter, I used Solr version 3.1 and Jetty version 6.1.26, and those versions are covered in the tips of the following chapter. If another version of Solr is mandatory for a feature to run, then it will be mentioned.

## Running Solr on Jetty

The simplest way to run Apache Solr on a Jetty servlet container is to run the provided example configuration based on the embedded Jetty, but it's not the case here. In this recipe, I would like to show you how to configure and run Solr on a standalone Jetty container.

### Getting ready

First of all, you need to download the Jetty servlet container for your platform. You can get your download package from an automatic installer (like `apt -get`) or you can download it yourself from <http://jetty.codehaus.org/jetty/>. Of course, you also need `solr.war` and other configuration files that come with Solr (you can get them from the example distribution that comes with Solr).

### How to do it...

There are a few common mistakes that people do when setting up Jetty with Solr, but if you follow the following instructions, the configuration process will be simple, fast, and will work flawlessly.

The first thing is to install the Jetty servlet container. For now, let's assume that you have Jetty installed.

Now we need to copy the `jetty.xml` and `webdefault.xml` files from the `example/etc` directory of the Solr distribution to the configuration directory of Jetty. In my Debian Linux distribution, it's `/etc/jetty`. After that, we have our Jetty installation configured.

The third step is to deploy the Solr web application by simply copying the `solr.war` file to the `webapps` directory of Jetty.

The next step is to copy the Solr configuration files to the appropriate directory. I'm talking about files like `schema.xml`, `solrconfig.xml`, and so on. Those files should be in the directory specified by the `jetty.home` system variable (in my case, this was the `/usr/share/jetty` directory). Please remember to preserve the directory structure you see in the example.

We can now run Jetty to see if everything is ok. To start Jetty that was installed, for example, using the `apt -get` command, use the following command:

```
/etc/init.d/jetty start
```

If there were no exceptions during start up, we have a running Jetty with Solr deployed and configured. To check if Solr is running, try going to the following address with your web browser: `http://localhost:8983/solr/`.

You should see the Solr front page with cores, or a single core, mentioned. Congratulations, you have just successfully installed, configured, and ran the Jetty servlet container with Solr deployed.

## How it works...

For the purpose of this recipe, I assumed that we needed a single core installation with only `schema.xml` and `solrconfig.xml` configuration files. Multicore installation is very similar—it differs only in terms of the Solr configuration files.

Sometimes there is a need for some additional libraries for Solr to see. If you need those, just create a directory called `lib` in the same directory that you have the `conf` folder and put the additional libraries there. It is handy when you are working not only with the standard Solr package, but you want to include your own code as a standalone Java library.

The third step is to provide configuration files for the Solr web application. Those files should be in the directory specified by the system variable `jetty.home` or `solr.solr.home`. I decided to use the `jetty.home` directory, but whenever you need to put Solr configuration files in a different directory than Jetty, just ensure that you set the `solr.solr.home` property properly. When copying Solr configuration files, you should remember to include all the files and the exact directory structure that Solr needs. For the record, you need to ensure that all the configuration files are stored in the `conf` directory for Solr to recognize them.

After all those steps, we are ready to launch Jetty. The example command has been run from the Jetty installation directory.

After running the example query in your web browser, you should see the Solr front page as a single core. Congratulations, you have just successfully configured and ran the Jetty servlet container with Solr deployed.

## There's more...

There are a few tasks you can do to counter some problems when running Solr within the Jetty servlet container. Here are the most common ones that I encountered during my work.

### I want Jetty to run on a different port

Sometimes it's necessary to run Jetty on a different port other than the default one. We have two ways to achieve that:

1. Adding an additional start up parameter, `jetty.port`. The start up command would look like this:

```
java -Djetty.port=9999 -jar start.jar
```

2. Changing the `jetty.xml` file—to do that, you need to change the following line:

```
<Set name="port"><SystemProperty name="jetty.port"
default="8983"/></Set>
```

to:

```
<Set name="port"><SystemProperty name="jetty.port"
default="9999"/></Set>
```

### Buffer size is too small

Buffer overflow is a common problem when our queries are getting too long and too complex—for example, when using many logical operators or long phrases. When the standard HEAD buffer is not enough, you can resize it to meet your needs. To do that, you add the following line to the Jetty connector in the `jetty.xml` file. Of course, the value shown in the example can be changed to the one that you need:

```
<Set name="headerBufferSize">32768</Set>
```

After adding the value, the connector definition should look more or less like this:

```
<Call name="addConnector">
  <Arg>
    <New class="org.mortbay.jetty.bio.SocketConnector">
      <Set name="port"><SystemProperty name="jetty.port" default="8080"/></Set>
      <Set name="maxIdleTime">50000</Set>
      <Set name="lowResourceMaxIdleTime">1500</Set>
      <Set name="headerBufferSize">32768</Set>
    </New>
  </Arg>
</Call>
```



### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. 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.

## Running Solr on Apache Tomcat

Sometimes you need to choose a different servlet container other than Jetty. Maybe it's because your client has other applications running on another servlet container, maybe it's because you just don't like Jetty. Whatever your requirements are that put Jetty out of the scope of your interest, the first thing that comes to mind is a popular and powerful servlet container—Apache Tomcat. This recipe will give you an idea of how to properly set up and run Solr in the Apache Tomcat environment.

### Getting ready

First of all, we need an Apache Tomcat servlet container. It can be found at the Apache Tomcat website—<http://tomcat.apache.org>. I concentrated on Tomcat version 6.x because of two things—version 5 is pretty old right now, and version 7 is the opposite—it's too young in my opinion. That is why I decided to show you how to deploy Solr on Tomcat version 6.0.29, which was the newest one while writing this book.

### How to do it...

To run Solr on Apache Tomcat, we need to perform the following six simple steps:

1. Firstly, you need to install Apache Tomcat. The Tomcat installation is beyond the scope of this book, so we will assume that you have already installed this servlet container in the directory specified by the `$TOMCAT_HOME` system variable.
2. The next step is preparing the Apache Tomcat configuration files. To do that, we need to add the following inscription to the connector definition in the `server.xml` configuration file:

```
URIEncoding="UTF-8"
```

The portion of the modified `server.xml` file should look like this:

```
<Connector port="8080" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443"
    URIEncoding="UTF-8" />
```

3. Create a proper context file. To do that, create a `solr.xml` file in the `$TOMCAT_HOME/conf/Catalina/localhost` directory. The contents of the file should look like this:  

```
<Context path="/solr">
  <Environment name="solr/home" type="java.lang.String" value="/home/solr/configuration/" override="true"/>
</Context>
```
4. The next thing is the Solr deployment. To do that, we need a `solr.war` file that contains the necessary files and libraries to run Solr to be copied to the Tomcat `webapps` directory. If you need some additional libraries for Solr to see, you should add them to the `$TOMCAT_HOME/lib` directory.
5. The last thing we need to do is add the Solr configuration files. The files that you need to copy are files like `schema.xml`, `solrconfig.xml`, and so on. Those files should be placed in the directory specified by the `solr/home` variable (in our case, `/home/solr/configuration/`). Please don't forget that you need to ensure the proper directory structure. If you are not familiar with the Solr directory structure, please take a look at the example deployment that is provided with standard Solr package.
6. Now we can start the servlet container by running the following command:

```
bin/catalina.sh start
```

In the log file, you should see a message like this:

```
Info: Server startup in 3097 ms
```

To ensure that Solr is running properly, you can run a browser, and point it to an address where Solr should be visible, like the following: `http://localhost:8080/solr/`.

If you see the page with links to administration pages of each of the cores defined, that means that your Solr is up and running.

## How it works...

Let's start from the second step, as the installation part is beyond the scope of this book. As you probably know, Solr uses UTF-8 file encoding. That means that we need to ensure that Apache Tomcat will be informed that all requests and responses made should use that encoding. To do that, we modify the `server.xml` in the way shown in the example.

The Catalina context file (called `solr.xml` in our example) says that our Solr application will be available under the `/solr` context (the `path` attribute), the war file will be placed in the `/home/tomcat/webapps/` directory. `solr/home` is also defined, and that is where we need to put our Solr configuration files. The shell command that is shown starts Apache Tomcat. I think that I should mention some other options of `catalina.sh` (or `catalina.bat`) script:

- ▶ `stop`—stops Apache Tomcat
- ▶ `restart`—restarts Apache Tomcat

- ▶ debug—start Apache Tomcat in debug mode

After running the example address in a web browser, you should see a Solr front page with a core (or cores if you have a multicore deployment). Congratulations, you have just successfully configured and ran the Apache Tomcat servlet container with Solr deployed.

## There's more...

Here are some other tasks that are common problems when running Solr on Apache Tomcat.

### Changing the port on which we see Solr running on Tomcat

Sometimes it is necessary to run Apache Tomcat on a port other than the 8080 default one. To do that, you need to modify the `port` variable of the connector definition in the `server.xml` file located in the `$TOMCAT_HOME/conf` directory. If you would like your Tomcat to run on port 9999, this definition should look like this:

```
<Connector port="9999" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443"
    URIEncoding="UTF-8" />
```

While the original definition looks like this:

```
<Connector port="8080" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443"
    URIEncoding="UTF-8" />
```

## Using the Suggester component

Nowadays, it's common for web pages to give a search suggestion (or autocomplete as I tend to call it), just like many "big" search engines do—just like Google, Microsoft, and others. Lately, Solr developers came up with a new component called **Suggester**. It provides Solr with flexible ways to add suggestions to your application using Solr. This recipe will guide you through the process of configuring and using this new component.

## How to do it...

First we need to add the search component definition in the `solrconfig.xml` file. That definition should look like this:

```
<searchComponent class="solr.SpellCheckComponent" name="suggester">
  <lst name="spellchecker">
    <str name="name">suggester</str>
```

```
<str name="classname"> org.apache.solr.spelling.suggest.Suggester</str>
<str name="lookupImpl"> org.apache.solr.spelling.suggest.tst.TSTLookup</str>
<str name="field">name</str>
<str name="threshold">2</str>
</lst>
</searchComponent>
```

Now we can define an appropriate request handler. To do that, we modify the `solrconfig.xml` file and add the following lines:

```
<requestHandler class="org.apache.solr.handler.component.SearchHandler" name="/suggester">
  <lst name="defaults">
    <str name="spellcheck">true</str>
    <str name="spellcheck.dictionary">suggester</str>
    <str name="spellcheck.count">10</str>
  </lst>
  <arr name="components">
    <str>suggester</str>
  </arr>
</requestHandler>
```

Now if all went well (Solr server started without an error), we can make a Solr query like this:

```
http://localhost:8983/solr/suggester/?q=a
```

You should see the response as follows:

```
<?xml version="1.0" encoding="UTF-8">
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">16</int>
  </lst>
  <lst name="spellcheck">
    <lst name="suggestions">
      <lst name="a">
        <int name="numFound">8</int>
        <int name="startOffset">0</int>
        <int name="endOffset">1</int>
        <arr name="suggestions">
          <str>a</str>
          <str>asus</str>
          <str>ati</str>
          <str>ata</str>
```

```
<str>adata</str>
<str>all</str>
<str>allinone</str>
<str>apple</str>
</arr>
<lst>
<lst>
</lst>
</response>
```

## How it works...

After reading the aforementioned search component configuration, you may wonder why we use `solr.SpellCheckComponent` as our search component implementation class. Actually, the Suggester component relies on spellchecker and reuses most of its classes and logic. That is why Solr developers decided to reuse spellchecker code.

Anyway, back to our configuration. We have some interesting configuration options. We have the component name (`name` variable), we have the logic implementation class (`classname` variable), the word lookup implementation (`lookupImpl` variable), the field on which suggestions will be based (`field` variable), and a `threshold` parameter which defines the minimum fraction of documents where a term should appear to be visible in the response.

Following the definition, there is a request handler definition present. Of course, it is not mandatory, but useful. You don't need to pass all the required parameters with your query all the time. Instead, you just write those parameters to the `solrconfig.xml` file along with request handler definition. We called our example request handler `/suggester` and with that name, it'll be exposed in the servlet container. We have three parameters saying that, in Solr, when using the defined request handler it should always include suggestions (`suggest` parameter set to `true`), it should use the dictionary named `suggester` that is actually our component (`spellcheck.dictionary` parameter), and that the maximum numbers of suggestions should be 10 (`spellcheck.count` parameter).

As mentioned before, the Suggester component reuses most of the spellchecker logic, so most of all spellchecker parameters can be used as the configuration of the behavior of the Suggester component. Remember that, because you never know when you'll need some of the non-standard parameters.

## There's more...

There are a few things that are good to know about when using the Suggester component. The following are the three most common things that people tend to ask about:



## Suggestions from a static dictionary

There are some situations when you'll need to get suggestions not from a defined field in an index, but from a file. To do that, you need to have a text dictionary (encoded in UTF-8) that looks as follows:

```
Suggestion 1
Suggestion 2
Suggestion 3
```

Second, you need to add another parameter to the Suggest component definition named `sourceLocation` that specifies the dictionary location (the file that will be used for the suggestions base):

```
<str name="sourceLocation">suggest_dictionary.txt</str>
```

So the whole Suggester component definition would look like this:

```
<searchComponent class="solr.SpellCheckComponent" name="suggester">
  <lst name="spellchecker">
    <str name="name">suggester</str>
    <str name="classname"> org.apache.solr.spelling.suggest.Suggester</str>
    <str name="lookupImpl"> org.apache.solr.spelling.suggest.tst.TSTLookup</str>
    <str name=" sourceLocation">suggest_dictionary.txt</str>
  </lst>
</searchComponent>
```

## Rebuilding the suggestion word base after commit

If you have field-based suggestions, you'll probably want to rebuild your suggestion dictionary after every change in your index. Of course, you can do that manually by invoking the appropriate command (the `spellcheckbuild` parameter with the `true` value), but we don't want to do it this way—we want Solr to do it for us. To do it, just add another parameter to your Suggester component configuration. The new parameter should look like this:

```
<str name="buildOnCommit">true</str>
```

## Removing uncommon words from suggestions

How to remove language mistakes and uncommon words from suggestions is a common concern. It's not only your data—most data has some kind of mistakes or errors—and the process of data cleaning is time consuming and difficult. Solr has an answer for you. We can add another parameter to our Suggester component configuration and we won't see any of the mistakes and uncommon words in the suggestions. Add the following parameter:

```
<float name="threshold">0.05</float>
```

This parameter takes values from 0 to 1. It tells the Suggester component the minimum fraction of documents of the total where a word should appear to be included as the suggestion.

## See also

If you don't need a sophisticated Suggester component like the one described, you should take a look at the *How to implement an autosuggest feature using faceting recipe in Chapter 6, Using Faceting Mechanism*.

## Handling multiple languages in a single index

There are many examples where multilingual applications and multilingual searches are mandatory—for example, libraries having books in multiple languages. This recipe will cover how to set up Solr, so we can make our multilingual data searchable using a single query and a single response. This task doesn't cover how to analyze your data and automatically detect the language used. That is beyond the scope of this book.

## How to do it...

First of all, you need to identify what languages your applications will use. For example, my latest application uses two languages—English and German.

After that, we need to know which fields need to be separate. For example—a field with an ISBN number or an identifier field can be shared, because they don't need to be indexed in a language-specific manner, but titles and descriptions should be separate. Let's assume that our example documents consist of four fields:

- ▶ ID
- ▶ ISBN
- ▶ Title
- ▶ Description

We have two languages—English and German—and we want all documents to be searchable with one query within one index.

First of all, we need to define some language-specific field types. To add the field type for English, we need to add the following lines to the `schema.xml` file:

```
<fieldType name="text_en" class="solr.TextField"
positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
```

```
generateNumberParts="1" catenateWords="1" catenateNumbers="1"
catenateAll="0" splitOnCaseChange="1"/>
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.PorterStemFilterFactory"/>
</analyzer>
</fieldType>
```

Next, we need to add the field type for fields containing a German title and description:

```
<fieldType name="text_de" class="solr.TextField"
positionIncrementGap="100">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
generateNumberParts="1" catenateWords="1" catenateNumbers="1"
catenateAll="0" splitOnCaseChange="1"/>
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.SnowballPorterFilterFactory" language="German2"/>
</analyzer>
</fieldType>
```

Now we need to define the document's fields using the previously defined field types. To do that, we add the following lines to the `schema.xml` file in the field section.

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="isbn" type="string" indexed="true" stored="true"
required="true" />
<field name="title_en" type="text_en" indexed="true" stored="true" />
<field name="description_en" type="text_en" indexed="true"
stored="true" />
<field name="title_de" type="text_de" indexed="true" stored="true" />
<field name="description_de" type="text_de" indexed="true"
stored="true" />
```

And that's all we need to do with configuration. After the indexation, we can start to query our Solr server. Next thing is to query Solr for the documents that are in English or German. To do that, we send the following query to Solr:

```
q=title_en:harry+OR+description_en:harry+OR+title_
de:harry+OR+description_de:harry
```

## How it works...

First of all, we added a `text_en` field type to analyze the English title and description. We tell Solr to split the data on whitespaces, split on case change, make text parts lowercased, and finally to stem text with the appropriate algorithm—in this case, it is one of the English stemming algorithms available in Solr and Lucene. Let's stop for a minute to explain what stemming is. All you need to know, for now, is that stemming is the process for reducing inflected or derived words into their stem, root, or base forms. It's good to use a stemming algorithm in full text search engines because we can show the same set of documents on words like 'cat' and 'cats'.

Of course, we don't want to use English stemming algorithms with German words. That's why we've added another field type—`text_ger`. It differs from the `text_en` field in one matter—it uses `solr.SnowballPorterFilterFactory` with the attribute specifying German language instead of `solr.PorterStemFilterFactory`.

The last thing we do is to add field definitions. We have six fields. Two of those six are `id` (unique identifier) and `isbn`. The next two are English title and description, and the last two are German title and description.

As you can see in the example query, we can get documents regardless of where the data is. To achieve that, I used the `OR` logical operator to get the documents that the searched word, in any of the specified fields. If we have a match in English, we will get the desired documents; if we get a match in German, we will also get the desired documents.

## See also

If you don't need to get all multilingual results with a single query, you can look at the *Making multilingual data searchable with multicore deployment* recipe description provided in this chapter. If you want to sort the data we talked about, please refer to *Chapter 10, Dealing With Problems* and the recipe *How to sort non-English languages properly*.

## Indexing fields in a dynamic way

Sometimes you don't know the names and number of fields you have to store in your index. You can only determine the types that you'll use and nothing else, either because it is a requirement or maybe the data is coming from different sources or the data is very complex, and no one has enough knowledge to specify how many and what fields are needed to be present in the index. There is a cure for that problem—dynamic fields. This recipe will guide you through the definition and usage of dynamic fields.

## How to do it...

Looking from the scope of this book, we just need to do one thing—prepare the `schema.xml` file, so that Solr will be able to determine where to put our data. The example `schema.xml` file which comes with Solr provides definitions of some dynamic fields. Here are the example dynamic fields:

```
<dynamicField name="*_i" type="int" indexed="true" stored="true"/>
<dynamicField name="*_s" type="string" indexed="true" stored="true"/>
<dynamicField name="*_l" type="long" indexed="true" stored="true"/>
<dynamicField name="*_t" type="text" indexed="true" stored="true"/>
<dynamicField name="*_b" type="boolean" indexed="true" stored="true"/>
<dynamicField name="*_f" type="float" indexed="true" stored="true"/>
<dynamicField name="*_d" type="double" indexed="true" stored="true"/>
```

Having those definitions in the `schema.xml` file, we can update data without the need for a static field definition. Here is an example document that can be sent to Solr:

```
<add>
<doc>
<field name="name_s">Solr Cookbook</field>
<field name="name_t">Solr Cookbook</field>
<field name="price_d">19.99</field>
<field name="quantity_i">12</field>
<field name="available_b">true</field>
</doc>
</add>
```

## How it works...

When defining dynamic fields, you tell Solr to observe (at indexing time) every expansion that the pattern you wrote matches. For example, the pattern `*_i` will expand to field names like `quantity_i` and `grade_i`, but not for `gradei`.

However, you must know one thing—Solr will always try to choose the field that matches the pattern and has the shortest field pattern. What's more, Solr will always choose static field over dynamic field.

Let's get back to our definitions. We have a few dynamic fields defined in the `schema.xml` file. As you can see, dynamic field definition is just a little bit different from a static field definition. Definitions begin with an appropriate XML tag `dynamicField` and have the same attributes as static field definition. One thing that's uncommon is the name—it's a pattern that will be used to match the field name. There is one more thing to remember—you can only use one wildcard character—`*` as the start or end of a dynamic field name. Any other usage will result in a Solr start up error.

As you see in the example document, we have five fields filled. The field named `name_s` will be matched to the pattern `*_s`, the field named `name_t` will be matched to the pattern `*_t`, and the rest will behave similarly.

Of course, this is just a simple example of a document. The documents may consist of hundreds of fields both static and dynamic. Also the dynamic field definitions may be much more sophisticated than the one in the preceding example. You should just remember that using dynamic fields is perfectly eligible and you should use them whenever your deployment needs them.

### See also

If you need your data not only to be indexed dynamically, but also to be copied from one field to another, please refer to the recipe entitled *Copying contents of one field to another* in *Chapter 3, Analyzing your Text Data*.

## Making multilingual data searchable with multicore deployment

You don't always need to handle multiple languages in a single index, either, it's because you have your application in multiple languages showing only one of them at the same time or some other requirement. Whatever your cause is, this recipe will guide you on how to handle separable data in a single instance of Solr server through the use of multicore deployment.

### How to do it...

First of all, you need to create the `solr.xml` file and place it in your `$SOLR_HOME` directory. Let's assume that our application will handle two languages—English and German. The sample `solr.xml` file might look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<solr>
  <cores adminPath="/admin/cores/">
    <core name="en" instanceDir="cores/en">
      <property name="dataDir" value="cores/en/data" />
    </core>
    <core name="de" instanceDir="cores/de">
      <property name="dataDir" value="cores/de/data" />
    </core>
  </cores>
</solr>
```

Let's create directories mentioned in the `solr.xml` file. For the purpose of the example, I assumed that the `$SOLR_HOME` points to the `/usr/share/solr/directory`. We need to create the following directories:

- ▶ `$$SOLR_HOME/cores`
- ▶ `$SOLR_HOME/cores/en`
- ▶ `$SOLR_HOME/cores/de`
- ▶ `$SOLR_HOME/cores/en/conf`
- ▶ `$SOLR_HOME/cores/en/data`
- ▶ `$SOLR_HOME/cores/de/conf`
- ▶ `$SOLR_HOME/cores/de/data`

We will use the sample `solrconfig.xml` file provided with the example deployment of multicore Solr version 3.1. Just copy this file to the `conf` directory of both cores. For the record, the file should contain:

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
<updateHandler class="solr.DirectUpdateHandler2" />
<requestDispatcher handleSelect="true" >
<requestParsers enableRemoteStreaming="false"
multipartUploadLimitInKB="2048" />
</requestDispatcher>
<requestHandler name="standard" class="solr.StandardRequestHandler"
default="true" />
<requestHandler name="/update" class="solr.XmlUpdateRequestHandler" />
<requestHandler name="/admin/" class="org.apache.solr.handler.admin.
AdminHandlers" />
<admin>
<defaultQuery>solr</defaultQuery>
</admin>
</config>
```

Now we should prepare a simple `schema.xml` file. To make it simple, let's just add two field types to the example Solr `schema.xml`.

To the `schema.xml` file that will describe the index containing English documents, let's add the following field type (just add it in the types section of the `schema.xml` file):

```
<fieldType name="text_en" class="solr.TextField"
positionIncrementGap="100">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
```

```

<filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
generateNumberParts="1" catenateWords="1" catenateNumbers="1"
catenateAll="0" splitOnCaseChange="1"/>
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.PorterStemFilterFactory"/>
</analyzer>
</fieldType>

```

To the `schema.xml` file, describing the index containing the document in German, let's add the following field type:

```

<fieldType name="text_de" class="solr.TextField"
positionIncrementGap="100">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
generateNumberParts="1" catenateWords="1" catenateNumbers="1"
catenateAll="0" splitOnCaseChange="1"/>
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.SnowballPorterFilterFactory" language="German2"/>
</analyzer>
</fieldType>

```

The field definition for the English `schema.xml` should look like this:

```

<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="isbn" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text_en" indexed="true" stored="true" />
<field name="description" type="text_en" indexed="true" stored="true"
/>

```

The field definition for the German `schema.xml` should look like this:

```

<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="isbn" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text_de" indexed="true" stored="true" />
<field name="description" type="text_de" indexed="true" stored="true"
/>

```

Now you should copy the files you've just created to the appropriate directories:

- ▶ German `schema.xml` file to `$SOLR_HOME/cores/de/conf`
- ▶ English `schema.xml` file to `$SOLR_HOME/cores/en/conf`



That's all in terms of the configuration. You can now start your Solr instance as you always do.

Now all the index update requests should be made to the following addresses:

- ▶ English documents should go to `http://localhost:8983/solr/en/update`
- ▶ German documents should go to `http://localhost:8983/solr/de/update`

It is similar when querying Solr, for example, I've made two queries, first for the documents in English and second for the documents in German:

```
http://localhost:8983/solr/en/select?q=harry
http://localhost:8983/solr/de/select?q=harry
```

## How it works...

First of all, we create the `solr.xml` file to tell Solr that the deployment will consist of one or more cores. What is a core? Multiple cores let you have multiple separate indexes inside a single Solr server instance. Of course you can run multiple Solr servers, but every one of them would have its own process (actually a servlet container process), its own memory space assigned, and so on. The multicore deployment lets you use multiple indexes inside a single Solr instance, a single servlet container process, and with the same memory space.

Following that, we have two cores defined. Every core is defined in its own `core` tag and has some attributes defining its properties. Like the core home directory (`instanceDir` attribute) or where the data will be stored (`dataDir` property). You can have multiple cores in one instance of Solr, almost an infinite number of cores in theory, but in practice, don't use too many.

There are some things about the `solr.xml` file that need to be discussed further. First of all, the `adminPath` attribute of the tag `cores`—it defines where the core admin interface attribute will be available. With the value shown in the example, the core admin will be available under the following address: `http://localhost:8983/solr/admin/cores`.

The field type definition for each of the cores is pretty straightforward. The file that describes the index for English documents uses an English stemmer for text data, and the file that describes the index for German documents uses a German stemmer for text data.

The only difference in field definition is the type that the description and title fields use—for the German `schema.xml` they use the `text_de` field type, and for the English `schema.xml` they use the `text_en` field type.

As for the queries, you must know one thing. When using multicore with more than one core, the address under which Solr offers its handlers is different from the one when not using cores. Solr adds a core name before the handler name. So if you have a handler named `/simple` in the core named `example`, it will be available under the context `/solr/example/simple`, not `/solr/simple`. When you know that, you'll be able to know where to point your applications that use Solr and a multicore deployment.

There is one more thing—you need to remember that every core has a separate index. That means that you can't combine results from different cores, at least not automatically. For example, you can't automatically get results with a combination of documents in English and German; you must do it by yourself or choose a different architecture for your Solr deployment.

### There's more...

If you need more information about cores, maybe some of the following information will be helpful.

#### More information about core admin interface

If you seek for more information about the core admin interface commands, please refer to the Solr wiki pages found at <http://wiki.apache.org/solr/CoreAdmin>.

### See also

If you need to handle multiple languages in a single index, please refer to the *Handling multiple languages in a single index* recipe in this chapter.

## Solr cache configuration

As you may already know, cache plays a major role in a Solr deployment. And I'm not talking about some exterior cache—I'm talking about the three Solr caches:

- ▶ Filter cache—used for storing filter (query parameter `fq`) results and enum type facets mainly
- ▶ Document cache—used for storing Lucene documents that hold stored fields
- ▶ Query result cache—used for storing results of queries

There is a fourth cache—Lucene's internal cache—the field cache, but you can't control its behavior—it is managed by Lucene and created when it is first used by the Searcher object.

With the help of these caches, we can tune the behavior of the Solr searcher instance. In this recipe, we will focus on how to configure your Solr caches to suit most needs. There is one thing to remember—Solr cache sizes should be tuned to the number of documents in the index, the queries, and the number of results you usually get from Solr.

## Getting ready

Before you start tuning Solr caches, you should get some information about your Solr instance. That information is:

- ▶ Number of documents in your index
- ▶ Number of queries per second made to that index
- ▶ Number of unique filter (`fq` parameter) values in your queries
- ▶ Maximum number of documents returned in a single query
- ▶ Number of different queries and different sorts

All those numbers can be derived from the Solr logs and by using the Solr admin interface.

## How to do it...

For the purpose of this task, I assumed the following numbers:

- ▶ Number of documents in the index: 1,000,000
- ▶ Number of queries per second: 100
- ▶ Number of unique filters: 200
- ▶ Maximum number of documents returned in a single query: 100
- ▶ Number of different queries and different sorts: 500

Let's open the `solrconfig.xml` file and tune our caches. All the changes should be made in the query section of the file (the section between the `<query>` and `</query>` XML tags).

First goes the filter cache:

```
<filterCache
  class="solr.FastLRUCache"
  size="200"
  initialSize="200"
  autowarmCount="100"/>
```

Second goes the query result cache:

```
<queryResultCache
  class="solr.FastLRUCache"
  size="500"
  initialSize="500"
  autowarmCount="250"/>
```

Third, we have the document cache:

```
<documentCache
  class="solr.FastLRUCache"
  size="11000"
  initialSize="11000" />
```

Of course, the preceding configuration is based on the example values.

Furthermore, let's set our result window to match our needs—we sometimes need to get 20–30 more results than we need during query execution. So, we change the appropriate value in the `solrconfig.xml` to something like this:

```
<queryResultWindowSize>200</queryResultWindowSize>
```

And that's all.

## How it works...

Let's start with a small explanation. First of all, we use the `solr.FastLRUCache` implementation instead of the `solr.LRUCache`. This is a new type of cache implementation found in the 1.4 version of Solr. So called `FastLRUCache` tends to be faster when Solr puts less into caches and gets more. This is the opposite to `LRUCache` that tends to be more efficient when there is more "puts" than "gets" operations. That's why we use it.

This may be the first time you have seen cache configuration, so I'll explain what cache configuration parameters mean:

- ▶ `class`—you probably figured that out by now. Yes, this is the class implementing the cache
- ▶ `size`—this is the maximum size that the cache can have.
- ▶ `initialSize`—this is the initial size that the cache will have.
- ▶ `autowarmCount`—this is the number of cache entries that will be copied to the new instance of the same cache when Solr invalidates the `Searcher` object—for example, during commit operation.

As you can see, I tend to use the same number of entries for `size` and `initialSize`, and half of those values for the `autowarmCount`.

There is one thing you should be aware of. Some of the Solr caches (the document cache, actually) operate on internal identifiers called `docid`. Those caches cannot be automatically warmed. That's because the `docid` is changing after every commit operation and thus copying the `docid` is useless.

Now let's take a look at the cache types and what they are used for.

## Filter cache

So first we have the filter cache. This cache is responsible for holding information about the filters and the documents that match the filter. Actually, this cache holds an unordered set of document `ids` that match the filter. If you don't use the faceting mechanism with filter cache, you should set its size to the number of unique filters that are present in your queries at least. This way, it will be possible for Solr to store all the unique filters with their matching documents `ids` and this will speed up queries that use filters.

## Query result cache

The next cache is the query result cache. It holds the ordered set of internal `ids` of documents that match the given query and the sort specified. That's why, if you use caches, you should add as much filters as you can and keep your query (`q` parameter) as clean as possible (for example, pass only the search box content of your search application to the query parameter). If the same query will be run more than once and the cache has enough size to hold the entry, information available in the cache will be used. This will allow Solr to save precious I/O operation for the queries that are not in the cache—resulting in performance boost.



The maximum size of this cache I tend to set is the number of unique queries and their sorts that Solr handles in the time between Searchers object invalidation. This tends to be enough in the most cases.

## Document cache

The last type of cache is the document cache. It holds the Lucene documents that were fetched from the index. Basically, this cache holds the stored fields of all the documents that are gathered from the Solr index. The size of this cache should always be greater than the number of concurrent queries multiplied by the maximum results you get from Solr. This cache can't be automatically warmed—because every commit is changing the internal IDs of the documents. Remember that the cache can be memory-consuming in case you have many stored fields.

## Query result window

The last is the query result window. This parameter tells Solr how many documents to fetch from the index in a single Lucene query. This is a kind of super set of documents fetched. In our example, we tell Solr that we want maximum of one hundred documents as a result of a single query. Our query result window tells Solr to always gather two hundred documents. Then when we will need some more documents that follow the first hundred they will be fetched from cache and therefore we will be saving our resources. The size of the query result window is mostly dependent on the application and how it is using Solr. If you tend to do multiple paging, you should consider using a higher query result window value.



You should remember that the size of caches shown in this task is not final and you should adapt them to your application needs. The values and the method of their calculation should be only taken as a starting point to further observation and optimizing process. Also, please remember to monitor your Solr instance memory usage as using caches will affect the memory that is used by the JVM.

### There's more...

There are a few things that you should know when configuring your caches.

#### Using filter cache with faceting

If you use the term enumeration faceting method (parameter `facet.method=enum`), Solr will use the filter cache to check each term. Remember that if you use this method, your filter cache size should have at least the size of the number of unique facet values in all your faceted fields. This is crucial and you may experience performance loss if this cache is not configured the right way.

#### When we have no cache hits

When your Solr instance has a low cache hit ratio, you should consider not using caches at all (to see the hit ratio, you can use the administration pages of Solr). Cache insertion is not free—it costs CPU time and resources. So if you see that you have very low cache hit ratio, you should consider turning your caches off—it may speed up your Solr instance. Before you turn off the caches, please ensure that you had the right cache set up—a small hit ratio can be a result of a bad cache configuration.

#### When we have more "puts" than "gets"

When your Solr instance uses put operations more than get operations, you should consider using the `solr.LRUCache` implementation. It's confirmed that this implementation behaves better when there are more insertions into the cache than lookups.

### See also

There is another way to warm your caches, if you know the most common queries that are sent to your Solr instance—auto warming queries.

- ▶ To see how to configure them, you should refer to *Chapter 7, Improving Solr Performance* and the recipe *Improving Solr performance right after start up or commit operation*
- ▶ To see how to use the administration pages of the Solr server, you should refer to *Chapter 4, Solr Administration*

- For information on how to cache whole pages of results, please refer to *Chapter 7*, the recipe *Caching whole result pages*.

## How to fetch and index web pages

There are many ways to index web pages. We could download them, parse them, and index with the use of Lucene and Solr. The indexing part is not a problem, at least in most cases. But there is another problem—how do you fetch them? We could possibly create our own software to do that, but that takes time and resources. That's why this recipe will cover how to fetch and index web pages using Apache Nutch.

### Getting ready

For the purpose of this recipe we will be using version 1.2 of Apache Nutch. To download the binary package of Apache Nutch, please go to the download section of <http://nutch.apache.org>.

### How to do it...

First of all, we need to install Apache Nutch. To do that, we just need to extract the downloaded archive to the directory of our choice, for example, I installed it in the directory: `/nutch`. This directory will be referred to as `$NUTCH_HOME`.

Open the file `$NUTCH_HOME/conf/nutch-default.xml` and set the value `http.agent.name` to the desired name of your crawler. It should look like this:

```
<property>
<name>http.agent.name</name>
<value>SolrCookbookCrawler</value>
<description>HTTP 'User-Agent' request header.</description>
</property>
```

Now let's create an empty directory called `crawl` in the `$NUTCH_HOME` directory. Then create the `nutch` directory in the `$NUTCH_HOME/crawl` directory.

The next step is to create a directory `urls` in the `$NUTCH_HOME/crawl/nutch` directory.

Now add the file named `nutch` to the directory `$$NUTCH_HOME/crawl/nutch/site`. For the purpose of this book, we will be crawling Solr and Lucene pages, so this file should contain the following: `http://lucene.apache.org`.

Now we need to edit the `$NUTCH_HOME/conf/crawl-urlfilter.txt` file. Replace the `MY.DOMAIN.NAME` with the `http://lucene.apache.org`. So the appropriate entry should look like:

```
+^http://lucene.apache.org/
```

One last thing before fetching the data is Solr configuration. The only thing that we need to do is to copy the file `$NUTCH_HOME/conf/schema.xml` to the directory `$$SOLR_HOME/conf`.

Now we can start fetching web pages.

Run the following command from the `$NUTCH_HOME` directory:

```
bin/nutch crawl crawl/nutch/site -dir crawl -depth 3 -topN 50
```

Depending on your Internet connection and your machine configuration, you should finally see the following message:

```
crawl finished: crawl
```

This means that the crawl is completed and the data is fetched. Now we should invert the fetched data to be able to index anchor text to the indexed pages. To do that, we invoke the following command:

```
bin/nutch invertlinks crawl/linkdb -dir crawl/segments
```

Sometime later, you'll see a message that will inform you that the invert process is completed:

```
LinkDb: finished at 2010-10-18 21:35:44, elapsed: 00:00:15
```

We can now send our data to Solr (you can find the appropriate `schema.xml` file in the Nutch distribution in the `conf/schema.xml` directory). To do that, you should run the following command:

```
bin/nutch solrindex http://127.0.0.1:8983/solr/ crawl/crawldb crawl/
linkdb crawl/segments/*
```

After a period of time, depending on the size of your crawl database, you should see a message informing you that the indexing process was finished:

```
SolrIndexer: finished at 2010-10-18 21:39:28, elapsed: 00:00:26
```

## How it works...

After installing Nutch and Solr, the first thing we do is set our crawler name. Nutch does not allow empty names, so we must choose one. The file `nutch-default.xml` defines more properties than the mentioned one, but at this time, we only need to know about that one.

The next step is the creation of directories where the crawl database will be stored. It doesn't have to be exactly the same directory as the example `crawl`. You can place it on a different partition or another hard disk drive.

The file `site` we created in the directory `$$NUTCH_HOME/crawl/nutch` should contain information about the sites from which we want information to be fetched. In the example, we have only one site—`http://lucene.apache.org`.



The `crawl-urlfilter.txt` file contains information about the filters that will be used to check the URLs that Nutch will crawl. In the example, we told Nutch to accept every URL that begins with `http://lucene.apache.org`.

Next, we start with some "Nutch magic". First of all we run the crawling command. The `crawl` command of Nutch command line utility needs some parameters, they are as follows:

- ▶ File with addresses to fetch defined.
- ▶ Directory where the fetch database will be stored.
- ▶ How deep to go after the links are defined—in our example, we told Nutch to go for a maximum of three links from the main page.
- ▶ How many documents to get from each level of the depth. In our example, we told Nutch to get a maximum of 50 documents per level of depth.

The next big thing is the link inversion process. This process is performed to generate link database so that Nutch can index the anchor with the associated pages. The `invertlinks` command of Nutch command line utility was run with two parameters:

- ▶ Output directory where the newly created link database should be created
- ▶ Directory where the data segments were written during the crawl process

The last command that was run was the one that pushed the data into Solr. This process uses the `javabin` format and uses the `/update` handler, so remember to have both of these functionalities configured in your Solr instance. The `solrindex` command of the Nutch command line utility was run with the following parameters:

- ▶ Address of the Solr server instance
- ▶ Directory containing the crawl database created by the `crawl` command
- ▶ Directory containing the link database created by the `invertlinks` command
- ▶ List of segments that contain crawl data

## There's more...

There is one more thing worth knowing when you start a journey in the land of Apache Nutch.

### Multiple thread crawling

The `crawl` command of the Nutch command-line utility has another option—it can be configured to run crawling with multiple threads. To achieve that, you add the parameter:

`-threads N`

So if you would like to crawl with 10 threads, you should run the `crawl` command as follows:

```
bin/nutch crawl crawl/nutch/site -dir crawl -depth 3 -topN 50 -threads 10
```

## See also

If you seek more information about Apache Nutch, please refer to <http://nutch.apache.org> and go to the wiki section.

## Getting the most relevant results with early query termination

When we have millions of documents, large indexes, and many shards, there are situations where you don't need to show all the results on a given query. It is very probable that you only want to show your user the top N results. It's time when you can use the early termination techniques to terminate long queries after a set amount of time. But using early termination techniques is a bit tricky. There are a few things that need to be addressed before you can use early termination techniques. One of those things is getting the most relevant results. The tool for sorting the Lucene index was only available in Apache Nutch, but that is history now because the Lucene version of this tool was committed to the SVN repository. This recipe will guide you through the process of index pre-sorting and explain why to use this new feature of Lucene and Solr and how to get the most relevant results with the help of this tool.

## Getting ready

During the writing of this book, the `IndexSorter` tool was only available in `branch_3x` of the Lucene and Solr SVN repository. After downloading the appropriate version, compiling and installing it, we can begin using this tool.

## How to do it...

`IndexSorter` is an index post processing tool. This means that it should be used after data is indexed. Let's assume that we have our data indexed. For the purpose of showing how to use the tool, I modified my `schema.xml` file to consist only of these fields (add the following to the fields section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
multiValued="false" required="true"/>
<field name="isbn" type="string" indexed="true" stored="true"
multiValued="false" />
<field name="title" type="text" indexed="true" stored="true"
multiValued="false" />
<field name="description" type="text" indexed="true" stored="true"
multiValued="false" />
<field name="author" type="string" indexed="true" stored="true"
multiValued="false" />
<field name="value" type="string" indexed="true" stored="true"
multiValued="false" />
```

Let's assume that we have a requirement that we need to show our data sorted by the `value` field (the field contains float values, the higher the value the more important the document is), but our index is so big, that a single query is taking more time than a client will be willing to wait for the results. That's why we need to pre-sort the index by the required field. To do that, we will use the tool named `IndexSorter`. There is one more thing before you can run the `IndexSorter` tool

So let's run the following command:

```
java -cp lucene-lubs/* org.apache.lucene.index.IndexSorter solr/data/
index solr/data/new_index author
```

After some time, we should see a message like the following one:

```
IndexSorter: done, 9112 total milliseconds
```

This message means that everything went well and our index is sorted by a `value` field. The sorted index is written in the `solr/data/new_index` directory and the old index is not altered in any way. To use the new index, you should replace the contents of the old index directory (that is, `solr/data/index`) with the contents of the `solr/data/new_index` directory.

## How it works...

I think that the field definitions do not need to be explained. The only thing worth looking at is the `value` field which is the field on which we will be sorting.

But how does this tool work? Basically, it sorts your Lucene index in a static way. What does that mean? Let's start with some explanation. When indexing documents with Solr (and Lucene of course), they are automatically given an internal identification number—a document ID. Documents with low internal ID will be chosen by Lucene first. During the indexing process, we don't have the possibility to set the internal document IDs. So what happens when we use `TimeLimitingCollector` (and therefore ending a query after a set amount of time) in combination with sorting by the `author` field on millions of data? We get some amount of data, but not all data, because we end a query after a set amount of time. Then Solr sorts that data and return it to the application or a user. You can imagine that because the data set is not complete, the end user can get random results. This is because Solr, and therefore Lucene, will choose the documents with low ID first.

To avoid it, and get the most relevant result, we can use the `IndexSorter` tool, to change the IDs of documents we are interested in and store them with low internal IDs. And that is what the `IndexSorter` tool is for—sorting our index on the basis of a defined field. Why do we only want to return the first amount of documents? When we have millions of documents, the user usually want to see the most relevant ones, not all.

One thing to remember is that the sorting is static. You cannot change it during query execution. So if you need sorting on multiple fields, you should consider multicore deployment where one core holds unsorted data, and other cores hold indexes sorted using the `IndexSorter` tool. Therefore, you'll be able to use the early termination techniques and get the most relevant data sorted on the basis of different fields.

## See also

To see how to use the early termination technique with Solr, refer to *Chapter 7*, the recipe *How to get the first top documents fast when having millions of them*.

## How to set up Extracting Request Handler

Sometimes indexing prepared text files (XML, CSV, JSON, and so on) is not enough. There are numerous situations where you need to extract data from binary files. For example, one of my clients wanted to index PDF files—actually their contents. To do that, we either need to parse the data in some external application or setup Solr to use Apache Tika. This recipe will guide you through the process of setting up Apache Tika with Solr.

## How to do it...

First, let's edit our Solr instance `solrconfig.xml` and add the following configuration:

```
<requestHandler name="/update/extract" class="org.apache.solr.handler.extraction.ExtractingRequestHandler">
  <lst name="defaults">
    <str name="fmap.content">text</str>
    <str name="lowernames">true</str>
    <str name="uprefix">attr_</str>
    <str name="captureAttr">true</str>
  </lst>
</requestHandler>
```

Next, create the `lib` folder next to the `conf` directory (the directory where you place your Solr configuration files) and place the `apache-solr-cell-3.1-SNAPSHOT.jar` file from the `dist` directory (looking from the official Solr distribution package) there. After that, you have to copy all the libraries from the `contrib/extraction/lib/` directory to the `lib` directory you created before.

And that's actually all that you need to do in terms of configuration.

To simplify the example, I decided to choose the standard `schema.xml` file distributed with Solr.

To test the indexing process, I've created a PDF file `book.pdf` using PDFCreator, which contained only the following text: This is a Solr cookbook. To index that file, I've used the following command:

```
curl "http://localhost:8983/solr/update/extract?literal.id=1&commit=true"
-F "myfile=@example.pdf"
```

You should see the following response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">578</int>
  </lst>
</response>
```

## How it works...

Binary file parsing is implemented using the Apache Tika framework. Tika is a toolkit for detecting and extracting metadata and structured text from various types of documents, not only binary files, but also HTML and XML files. To add a handler which uses Apache Tika, we need to add a handler based on the `org.apache.solr.handler.extraction.ExtractingRequestHandler` class to our `solrconfig.xml` file, as shown in the example.

So we added a new request handler with some default parameters. Those parameters tell Solr how to handle the data that Tika returns. The `fmap.content` parameter tells Solr to what field content of the parsed document should be put. In our case, the parsed content will go to the field named `text`. The next parameter `lowernames` set to `true` tells Solr to lower all names that comes from Tika and make them lowercased. The next parameter, `uprefix`, is very important. It tells Solr how to handle fields that are not defined in the `schema.xml` file. The name of the field returned from Tika will be added to the value of the parameter and send to Solr. For example, if Tika returned a field named `creator` and we don't have such a field in our index, than Solr would try to index it under a field named `attr_creator`, which is a dynamic field. The last parameter tells Solr to index Tika XHTML elements into separate fields named after those elements.

Next we have a command that sends a PDF file to Solr. We are sending a file to the `/update/extract` handler with two parameters. First we define a unique identifier. It's useful to be able to do that during document sending because most of the binary documents won't have an identifier in its contents. To pass the identifier, we use the `literal.id` parameter. The second parameter we send to Solr is information to perform commit right after document processing.

## See also

To see how to index binary files, please take a look at *Chapter 2, Indexing Your Data*, the recipes: *Indexing PDF files*, *Indexing Microsoft Office files*, and *Extracting metadata from binary files*.

# 2

## Indexing your Data

In this chapter, we will cover:

- ▶ Indexing data in CSV format
- ▶ Indexing data in XML format
- ▶ Indexing data in JSON format
- ▶ Indexing PDF files
- ▶ Indexing Microsoft Office files
- ▶ Extracting metadata from binary files
- ▶ How to properly configure Data Import Handler with JDBC
- ▶ Indexing data from a database using Data Import Handler
- ▶ How to import data using Data Import Handler and delta query
- ▶ How to use Data Import Handler with URL Data Source
- ▶ How to modify data while importing with Data Import Handler

### Introduction

Indexing data is one of the most crucial things in every Lucene and Solr deployment. When your data is not indexed properly, your search results will be poor. When the search results are poor, it's almost certain that the users will not be satisfied with the application that uses Solr. That's why we need our data to be prepared and indexed well.

On the other hand, preparing data is not an easy task. Nowadays we have more and more data floating around. We need to index multiple formats of data from multiple sources. Do we need to parse the data manually and prepare the data in XML format? The answer is no—we can let Solr do that for us. This chapter will concentrate on the indexing process and data preparation, from indexing XML format, through PDF and Microsoft Office files, to using the Data Import Handler functionalities.

## Indexing data in CSV format

Let's imagine you have a simple SQL data dump that is provided by your client in CSV format. You don't have the time and resources to read it in an external application, change it to XML, and send it to Solr; you need fast results. Well, Solr can help you with that by providing a handler that allows you to index data in a CSV or comma-separated values (or character-separated values).

### How to do it...

Let's assume that we have an index with four fields: `id`, `name`, `isbn`, `description`. So the field definition part of `schema.xml` would look like this:

```
<field name="id" type="string" stored="true" indexed="true"/>
<field name="name" type="string" stored="true" indexed="true"/>
<field name="isbn" type="string" stored="true" indexed="true"/>
<field name="description" type="text" stored="true" indexed="true"/>
```

The data file called `data.csv` would look like this:

```
id;name;isbn;description;publish_date
1;Solr cookbook;ABC123456;"Solr Cookbook";2011-06
2;Some book 1;JHDS871281;"Some book";2010-01
3;Some book 2;9182KSC98;"Another book";2010-02
```

So now, let's modify the `solrconfig.xml` file and add the following:

```
<requestHandler name="/update/csv" class="solr.CSVRequestHandler"
startup="lazy">
  <lst name="defaults">
    <str name="separator">;</str>
    <str name="header">true</str>
    <str name="skip">publish_date</str>
    <str name="encapsulator">"</str>
  </lst>
</requestHandler>
```

Now, let's index the example data. To index the data, I've used the following command:

```
curl http://localhost:8983/solr/update/csv?commit=true --data-binary @
data.csv -H 'Content-type:text/plain; charset=utf-8'
```

After seeing a Solr response like the following, we can be sure that the indexing process went well and our data is indexed:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
```

```
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">250</int>
</lst>
</response>
```

## How it works...

I'll skip the comments about field definition as it's simple and obvious.

As you can see, the data file begins with a header in the first row. If configured properly, the header will tell Solr how to treat the data. The description is surrounded by quotation marks. You'll see how to handle that in a moment. One more thing about the data is that there is one additional field (`publish_date`), which we don't want to index for several reasons.

Now let's take a look at the update handler configuration. We have a Java class defined, `solr.CSVRequestHandler`, which is the actual handler implementation. We also set it so that the handler will be visible under `/update/csv` context; so, if your Solr is running on a local machine, the full URL would be like this: `http://localhost:8983/solr/update/csv`.

We also told Solr to instantiate this update handler when it's used for the first time by adding the `startup="lazy"` parameter.

Looking deeper into configuration, we have some additional parameters which define the behavior of the update handler. We tell Solr that the separator for the data will be the `;` character (`separator` parameter). We tell Solr that the data will be preceded with a header (parameter `header` set to `true`). We also tell Solr which part of the data to skip during indexing (`skip` parameter). And at the end, we say that the data may be encapsulated with the quotation mark (`encapsulator` parameter).

To send the example data to Solr, I used Ubuntu Linux and the `curl` command. First of all, I've added a new parameter to inform the update handler to perform commit after the data is read by adding the `commit` parameter set to `true`. You could ask why I didn't add it in the configuration. I could, but I want to control when my data will be committed. That's because I may have multiple CSV files which I don't want to be committed until the last of them is read; that's why I would add this parameter only with the last file I'm sending to Solr.

At the end, there is a Solr response which tells us that all went well and we should be able to see our data.

There is one thing you need to know: with the CSV format, you can't use index time boosting of fields and documents. If you need to boost your data during indexing, you should consider another file format.



## There's more...

There are a few things worth mentioning.

### Splitting encapsulated data

There are certain situations where the data is seen without the header. To inform Solr of how to parse such data, we need to pass another parameter `split`, either as a request parameter or a request handler configuration option.

If you set the `split` parameter to `true`, Solr will split the data even if it's encapsulated. For example, if we had a field named `title`, which was multivalued, we could set this parameter to split the data like this:

```
id;isbn;title
1;ABCD;"Solr Cookbook;Another title"
```

This would result in a document with `id=1`, `isbn=ABCD`, and two titles, the first one being "Solr Cookbook" and the second one being "Another title".

## Indexing data in XML format

XML is the most common of the indexing formats. It enables you to not only index data, but also to boost documents and fields, thus changing their importance. In this recipe, I'll concentrate on indexing XML files and what we can do with them.

## How to do it...

Let's assume that we have an index with three fields: `id`, `name`, and `isbn`. The field definition part of `schema.xml` would look like this:

```
<field name="id" type="string" stored="true" indexed="true"/>
<field name="name" type="string" stored="true" indexed="true"
omitNorms="false"/>
<field name="isbn" type="string" stored="true" indexed="true"/>
```

We have a file called `books.xml` which contains our data. We want some of the documents to be more important than others. We also want the `name` field to be more important than the rest of the fields. This is how the contents of the file could look:

```
<add overwrite="true" commitWithin="10000">
<doc>
<field name="id">1</field>
<field name="isbn">ABC1234</field>
<field name="name" boost="2">Some Book</field>
</doc>
```

```

<doc boost="2.5">
  <field name="id">2</field>
  <field name="isbn">ZYVW9821</field>
  <field name="name" boost="2">Important Book</field>
</doc>
<doc>
  <field name="id">3</field>
  <field name="isbn">NXJS1234</field>
  <field name="name" boost="2">Some other book</field>
</doc>
</add>

```

To index the data, I use the Linux `curl` command which is run as follows:

```

curl http://localhost:8983/solr/update --data-binary @books.xml -H
'Content-type:text/xml; charset=utf-8'

```

After running the preceding command, you should see a message like the following saying that everything went well:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">140</int>
  </lst>
</response>

```

## How it works...

There is one thing in the field definition that requires a comment—the `omitNorms="false"` attribute. If you ever plan to use index time field boosting like we did, you must tell Solr to use the field length normalization process on that field (for more information, please refer to <http://wiki.apache.org/solr/SchemaXml> wiki page). Otherwise, Solr won't be able to boost that field during indexing. Remember that if you have some fields with `omitNorms` set to `true`, those fields won't be boosted during indexing.

Our `books.xml` file holds our data. First of all, we tell Solr that we want to add data to our index by adding the `<add>` root tag in the document. We also tell Solr that we want documents that already exist in our index to be overwritten by the ones from the XML file. This is done by adding the `overwrite="true"` attribute. To overwrite a document in the index, the new one has to have the same unique identifier. The last thing is informing Solr to run the `commit` command automatically after 10,000 milliseconds. This time is measured after the documents are processed.

Next, we have document definitions. Every document is defined between two tags: `<doc>` and its closing tag `</doc>`. The second document has an additional attribute saying that the document is more important than the others. That attribute is `boost`, which takes a float value. The default value for that attribute is 1.0, everything above this value means that the document will be more important than the others, and everything below that value means that the document will be less important than the others.

Every document is defined by a set of fields. Every field is defined by a `<field>` XML tag and a mandatory attribute `name`, which is the name of the field that the data will be indexed into. The `name` field has an additional `boost` attribute that defines the importance of that field in the scope of the document. It follows the same guidelines for the values as the `boost` attribute of the `<doc>` XML tag.

The Linux `curl` command I've used is pretty simple—it just packs the data in the binary form, sets the appropriate header, and sends the data to Solr. If you don't use Linux or UNIX, you can write a simple Java code using, for example, SolrJ.

After running the preceding command, you should see a message like the one in the example saying that everything went well.

## Indexing data in JSON format

Assume that we want to use Solr with a JavaScript-based application. XML file format is not an option, it's usually big and it demands parsing. We want our data format to be as light as possible. JSON (JavaScript Object Notation) comes to the rescue. Although it is completely language independent, it is a good choice when dealing with JavaScript. This recipe will show you how to use this newest update file format available in Solr.

### How to do it...

Let's assume that we have an index with three fields: `id`, `name`, and `author`, which is multivalued. So the field definition part of `schema.xml` would look like this:

```
<field name="id" type="string" stored="true" indexed="true"/>
<field name="name" type="string" stored="true" indexed="true"
omitNorms="false"/>
<field name="author" type="string" stored="true" indexed="true" multiValued="true"/>
```

We have a file called `books.json` which contains our data. We want some of the documents to be more important than others. We also want the `name` field to be more important than the rest of the fields. This is how the contents of the file should look:

```
{
  "add": {
    "overwrite": true,
```

```

    "doc": {
      "id": 1,
      "name": "Some book",
      "author": ["John", "Marry"]
    }
  },
  "add": {
    "overwrite": true,
    "boost": 2.5,
    "doc": {
      "id": 2,
      "name": "Important Book",
      "author": ["Harry", "Jane"]
    }
  },
  "add": {
    "overwrite": true,
    "doc": {
      "id": 3,
      "name": "Some other book",
      "author": "Marry"
    }
  }
}

```

We must also add the appropriate request handler. To do that, we modify the `solrconfig.xml` file and add the following line:

```

<requestHandler name="/update/json" class="solr.
  JsonUpdateRequestHandler" />

```

To index the data, I use the Linux `curl` command which is run as follows:

```

curl http://localhost:8983/solr/update/json --data-binary @books.json -H
'Content-type:text/json; charset=utf-8'

```

After running the preceding command, you should see a message saying that everything went well, as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
<int name="status">0</int>
<int name="QTime">111</int>
</lst>
</response>

```

## How it works...

The field definition is pretty simple. So, I'll skip the part about the field types. However, there is one thing worth mentioning—the `omitNorms="false"` attribute. If you ever plan to use index time field boosting like we did, you must tell Solr to use the field length normalization process on that field (for more information please refer to the <http://wiki.apache.org/solr/SchemaXml> wiki page). Otherwise, Solr won't be able to boost that field during indexing. Remember that if you have some fields with `omitNorms` set to `true`, those fields won't be boosted during indexing.

What is more interesting comes with the second example. As you can see, the JSON file format differs from XML and CSV. Let's start from the beginning. Every property name and property value is surrounded by quotation marks. You can skip the quotation marks for simple data types like numbers or Boolean values, but I suggest surrounding data like strings with them. This will avoid confusion and will be easier to read by a human.

The data definition starts with an opening curly bracket and ends with a closing one. This also applies to every `add` command and to some values.

To add a document, we need to add an `add` part followed by a `:` character and curly brackets, between which the document will be defined. The `add` commands are separated from each other by a semicolon character. As you can see, unlike the XML format, a single document requires a separate `add` command. We also told Solr to overwrite documents in the index by the ones that come from the JSON file by adding an `overwrite` parameter set to `true`. This parameter can be defined for every `add` command separately.

Next, we have document definitions. Every document is defined by a `doc` part of the file. Similar to the `add` command, the `doc` command is followed by a `:` character and curly brackets, between which fields and document attributes will be defined. The document definition can be boosted during the indexing process. To do that, you need to add a `boost` parameter, as shown in the example with the second document. The default value for the `boost` attribute is 1.0, everything above that value means that the document will be more important than the others, everything below that value means that the document will be less important than the others.

A document consists of fields. As you may guess, the field definition is similar to the document and add definition. It starts with a field name surrounded by quotation marks. Following that is a `:` character and a value, or another pair of curly brackets or a pair of square brackets. If you plan to add just a value, you don't need to include any of the brackets. If you want to include a list of values, you should use square brackets (values are separated from each other by semicolons).

The Linux `curl` command I've used is pretty simple—it just packs the data in the binary form, sets the appropriate header, and sends the data to Solr. If you don't use Linux or UNIX, you can write a simple Java code using, for example, SolrJ.

After running the preceding command, you should see a message like the one in the example, saying that everything went well.

## Indexing PDF files

The library on the corner that we used to go to wants to expand its collection and make it available for the wider public through the World Wide Web. It asked its book suppliers to provide sample chapters of all the books in the PDF format so they can share it with the online users. With all the samples provided by the suppliers came a problem—how to extract data for the search box from more than 900,000 PDF files? Solr can do it with the use of Apache Tika. This recipe will show you how to handle such a recipe.

### Getting ready

Before you start getting deeper into the task, please have a look at the *How to set up Extracting Request Handler* recipe discussed in *Chapter 1, Apache Solr Configuration*, which will guide you through the process of configuring Solr to use Apache Tika.

### How to do it...

To test the indexing process, I've created a PDF file `book.pdf` using the PDF Creator which contains the text `This is a Solr cookbook`. To index that file, I've used the following command:

```
curl "http://localhost:8983/solr/update/extract?literal.id=1&commit=true"
-F "myfile=@cookbook.pdf"
```

You should then see the following response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">578</int>
  </lst>
</response>
```

To see what was indexed, I've run the following within a web browser:

```
http://localhost:8983/solr/select/?q=text:solr.
```

In return, I get:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  ...
```

```
<result name="response" numFound="1" start="0">
<doc>
<arr name="attr_created"><str>Thu Oct 21 16:11:51 CEST 2010</str></arr>
<arr name="attr_creator"><str>PDFCreator Version 1.0.1</str></arr>
<arr name="attr_producer"><str>GPL Ghostscript 8.71</str></arr>
<arr name="attr_stream_content_type"><str>application/octet-stream</str></arr>
<arr name="attr_stream_name"><str>cookbook.pdf</str></arr>
<arr name="attr_stream_size"><str>3209</str></arr>
<arr name="attr_stream_source_info"><str>myfile</str></arr>
<str name="author">Gr0</str>
<arr name="content_type"><str>application/pdf</str></arr>
<str name="id">1</str>
<str name="keywords"/>
<date name="last_modified">2010-10-21T14:11:51Z</date>
<str name="subject"/>
<arr name="title"><str>cookbook</str></arr>
</doc>
</result>
</response>
```

## How it works...

The `curl` command we used sends a PDF file to Solr. We are sending a file to the `/update/extract` handler along with two parameters. We define a unique identifier. It's useful to be able to do that during sending document because most of the binary documents won't have an identifier in their contents. To pass the identifier we use the `literal.id` parameter. The second parameter we send tells Solr to perform the commit operation right after document processing.

The test file I've created for the purpose of the recipe contains a simple sentence: `This is a Solr cookbook`.

Remember the contents of the PDF file I've created? It contained the word "solr". That's why I asked Solr to give me documents which contain the word "solr" in a field named `text`.

In response, I got one document which matched the given query. To simplify the example, I removed the response header part. As you can see in the response, there were a few fields that were indexed dynamically, and their names start with `attr_`. Those fields contained information about the file, such as the size, the application that created it, and so on. As we see, we have our ID indexed as we wished and some other fields that were present in the `schema.xml` file, and that Apache Tika could parse and return to Solr.

## Indexing Microsoft Office files

Imagine a law office with tons of documents in the Microsoft Word format. Now imagine that your task is to make them all searchable with Solr; a nightmare right? With the use of Apache Tika it can turn into quite a nice task with an easy execution. This recipe will show you how to achieve it.

### Getting ready

Before you start getting deeper into the task, please have a look at the *How to set up Extracting Request Handler* recipe discussed in *Chapter 1, Apache Solr Configuration*, which will guide you through the process of configuring Solr to use Apache Tika.

### How to do it...

First of all, we need to create a proper `schema.xml` file. So here it is:

```
<field name="id" type="string" indexed="true" stored="true"
required="true"/>
<field name="author" type="string" indexed="true" stored="true"
multiValued="true"/>
<field name="comments" type="text" indexed="true" stored="true"
multiValued="false"/>
<field name="keywords" type="text" indexed="true" stored="true"
multiValued="false"/>
<field name="contents" type="text" indexed="true" stored="true"
multiValued="false"/>
<field name="title" type="text" indexed="true" stored="true"
multiValued="false"/>
<field name="revision_number" type="string" indexed="true"
stored="true" multiValued="false"/>
<dynamicField name="ignored_*" type="ignored" indexed="false" stored="
false" multiValued="true"/>
```

Now let's get the `solrconfig.xml` file ready:

```
<requestHandler name="/update/extract" class="org.apache.solr.handler.
extraction.ExtractingRequestHandler">
<lst name="defaults">
<str name="fmap.content">contents</str>
<str name="lowernames">true</str>
<str name="uprefix">ignored_</str>
<str name="captureAttr">true</str>
</lst>
</requestHandler>
```



Now we can start sending the documents to Solr. To do that, let's run the following command:

```
curl "http://localhost:8983/solr/update/extract?literal.id=1&commit=true"
-F "myfile=@word.doc"
```

Let's check how the document was indexed. To do that, type a query similar to the following one in your web browser:

```
http://localhost:8983/solr/select/?q=contents:solr
```

As a result, we get the following document:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  ...
  <result name="response" numFound="1" start="0">
    <doc>
      <arr name="author"><str>Rafał Ku</str></arr>
      <str name="comments">Here is an comment</str>
      <str name="contents">Solr cookbook example This is a Solr cookbook
example word document</str>
      <str name="id">1</str>
      <str name="keywords">solr, cookbook, example</str>
      <str name="revision_number">2</str>
      <str name="title">Solr cookbook example</str>
    </doc>
  </result>
</response>
```

## How it works...

At first we define an appropriate index structure. I decided that besides a unique ID, I want to store some additional data like title, author, contents, comments, keywords, and the revision date. We also defined a dynamic field called `ignored`, to handle the data we don't want to index.

The next step is to define a new request handler to handle our updates, as you already know. We also added a few default parameters to define our handler behavior. In our case, `fmap.content` parameter tells Solr to index contents of the file to a field named `contents`. The next parameter, `uprefix` tells Solr to index all unknown fields (those that are not defined in `schema.xml` file) to the dynamic field whose name begins with `ignored_`. The last parameter (`captureAttr`) tells Solr to index Tika XHTML elements into separate fields named after those elements.

Next, we have a command that sends a doc file to Solr. We are sending a file to the `/update/extract` handler with two parameters. First, we define a unique identifier and pass that identifier to Solr using the `literal.id` parameter. The second parameter we send to Solr is the information to perform a commit right after document processing.

The last listing is an XML with Solr response. As you can see, there are only fields that are explicitly defined in `schema.xml`; no dynamic fields. Solr and Tika managed to extract the contents of the file and some of the metadata like author, comments, and keywords.

## See also

If you want to index other types of binary files, please take a look at the *Indexing PDF files* and *Extracting metadata from binary files* recipe in this chapter.

## Extracting metadata from binary files

Suppose that our current client has a video and music store. Not the e-commerce one, just the regular one; just around the corner. And now he wants to expand his business to e-commerce. He wants to sell the products online. But his IT department said that this will be tricky because they need to hire someone to fill up the database with the product names and their metadata. And that is where you come in and say that you can extract titles and authors from the MP3 files that are available, as samples. Now let's see how that can be achieved.

## Getting ready

Before you start getting deeper into the task, please have a look at the *How to set up Extracting Request Handler* recipe discussed in *Chapter 1, Apache Solr Configuration*, which will guide you through the process of configuring Solr to use Apache Tika.

## How to do it...

First we define an index structure in the `schema.xml` file. The field definition section should look like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true"/>
<field name="author" type="string" indexed="true" stored="true"
multiValued="true"/>
<field name="title" type="text" indexed="true" stored="true"/>
<dynamicField name="ignored_*" type="ignored" indexed="false" stored="
false"multiValued="true"/>
```

Now let's get the `solrconfig.xml` file ready:

```
<requestHandler name="/update/extract" class="org.apache.solr.handler.extraction.ExtractingRequestHandler">
  <lst name="defaults">
    <str name="lowernames">true</str>
    <str name="uprefix">ignored_</str>
    <str name="captureAttr">true</str>
  </lst>
</requestHandler>
```

Now we can start sending the documents to Solr. To do that, let's run the following command:

```
curl "http://localhost:8983/solr/update/extract?literal.id=1&commit=true"
-F "myfile=@sample.mp3"
```

Let's check how the document was indexed. To do that, type a query like the following to your web browser:

```
http://localhost:8983/solr/select/?q=title:207
```

As a result, we get the following document:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
    <lst name="params">
      <str name="q">title:207</str>
    </lst>
  </lst>
  <result name="response" numFound="1" start="0">
    <doc>
      <str name="author">Armin Van Buuren</str>
      <str name="id">1</str>
      <str name="title">Desiderium 207 (Feat Susana)</str>
    </doc>
  </result>
</response>
```

So it seems that everything went well.

## How it works...

First we define an index structure that will suit our needs. I decided that besides unique ID, I need to store title and author. We also defined a dynamic field called `ignored` to handle the data we don't want to index.

The next step is to define a new request handler to handle our updates, as you already know. We also added a few default parameters to define our handler behavior. In our case, the parameter `uprefix` tells Solr to index all unknown fields to the dynamic field whose name begins with `ignored_`, thus the additional data will not be visible in the index. The last parameter tells Solr to index Tika XHTML elements into separate fields named after those elements.

Next, we have a command that sends an MP3 file to Solr. We are sending a file to the `/update/extract` handler with two parameters. First we define a unique identifier and pass that identifier to Solr using the `literal.id` parameter. The second parameter we send to Solr is the information to perform commit right after document processing.

The query is a simple one, so I'll skip commenting this part.

The last listing is an XML with Solr response. As you can see, there are only fields that are explicitly defined in `schema.xml`; no dynamic fields. Solr and Tika managed to extract the name and author of the file.

## See also

If you want to index other types of binary files, please take a look at the *Indexing PDF files* and *Indexing Microsoft Office Files* recipes in this chapter.

## How to properly configure Data Import Handler with JDBC

One of our clients is having a problem. His database of users has grown to such size that even simple SQL selection is taking too much time and he seeks how to improve the search times. Of course he has heard about Solr but he doesn't want to generate XML or any other data format and push it to Solr. He would like the data to be fetched. What can we do about it? Well there is one thing: we can use one of the contribute modules of Solr – Data Import Handler. This recipe will show you how to configure the basic setup of the Data Import Handler and how to use it.

## How to do it...

First of all, copy the appropriate libraries to the `WEB-INF` directory of your Solr application. You have to copy all the libraries from the `contrib/dataimporthandler/libs` directory.

Next, we need to modify the `solrconfig.xml` file. You should add an entry like this:

```
<requestHandler name="/dataimport" class="org.apache.solr.handler.
dataimport.DataImportHandler">
  <lst name="defaults">
```

```
<str name="config">db-data-config.xml</str>
</lst>
</requestHandler>
```

Next, we create the `db-data-config.xml` file. It should have contents like the following example:

```
<dataConfig>
<dataSource driver="org.postgresql.Driver" url="jdbc:postgresql://
localhost:5432/users" user="users" password="secret" />
<document>
<entity name="user" query="SELECT user_id, user_name from users">
<field column="user_id" name="id" />
<field column="user_name" name="name" />
<entity name="user_desc" query="select desc from users_description
where user_id=${user.user_id}">
<field column="desc" name="description" />
</entity>
</entity>
</document>
</dataConfig>
```

If you want to use any other database engine, please change the `driver`, `url`, `user`, and `password`.

The index structure in the `schema.xml` file looks like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true"/>
<field name="name" type="text" indexed="true" stored="true" />
<field name="user_desc" type="text" indexed="true" stored="true"/>
<field name="description" type="text" indexed="true" stored="true"/>
```

One more thing before the indexation: you should copy an appropriate JDBC driver to the `lib` directory. You can get the library for PostgreSQL here <http://jdbc.postgresql.org/download.html>.

Now we can start indexing. We run the following query to Solr:

```
http://localhost:8983/solr/dataimport?command=full-import
```

As you may know, the HTTP protocol is asynchronous, and thus you won't be updated on how the process of indexing is going. To check the status of the indexing process, you can run the command once again.

And that's how we configure the Data Import Handler.

## How it works...

First we have a `solrconfig.xml` part which actually defines a new request handler—the Data Import Handler to be used by Solr. The `<str name="config">` XML tag specifies the name of the Data Import Handler configuration file.

The second listing is the actual configuration of the Data Import Handler. I used the JDBC source connection sample to illustrate how to configure the Data Import Handler. The contents of this configuration file start with the root tag named `dataConfig`, which is followed by a second tag defining a data source, named `dataSource`. In the example, I used the PostgreSQL database, and thus the JDBC driver is `org.postgresql.Driver`. We also define the database connection URL (attribute named `url`) and the database credentials (attributes `user` and `password`).

Next we have a document definition tag named `document`. This is the section containing information about the document that will be sent to Solr. Document definition is made of database queries—the entities.

The entity is defined by a name (`name` attribute) and an SQL query (`query` attribute). The entity name can be used to reference values in sub-queries; you can see an example of such behavior in the second entity named `user_desc`. As you may have already noticed, entities can be nested to handle sub-queries. The SQL query is there to fetch the data from the database and use it to fill the entity variables which will be indexed.

After the entity comes the mapping definition. There is a single `field` tag for every column returned by a query, but that is not a must – Data Import Handler can guess what the mapping is (for example, where entity field name matches column name), but I tend to use mappings because I find it easier to maintain. However, let's get back to the field. A field is defined by two attributes: `column`, which is the column name returned by a query and `name`, which is the field to which the data will be written.

Next we have a Solr query to start the indexing process. There are actually five commands that can be run:

- ▶ `/dataimport`: This command will return the actual status.
- ▶ `/dataimport?command=full-import`: This command will start the full import process. Remember that the default behavior is to delete the index contents at the beginning.
- ▶ `/dataimport?command=delta-import`: This command will start the incremental indexing process.
- ▶ `/dataimport?command=reload-config`: This command will force configuration reload.
- ▶ `/dataimport?command=abort`: This command will stop the indexing process.

## There's more...

There is one more thing that I think you should know.

### How to change the default behavior of deleting index contents at the beginning of a full import

If you don't want to delete the index contents at the start of full indexing using the Data Import Handler, add the `clean=false` parameter to your query. An example query should look like this:

```
http://localhost:8983/solr/data?command=full-import&clean=false
```

## Indexing data from a database using Data Import Handler

Let's assume that we want to index Wikipedia data and we don't want to parse the whole Wikipedia data and make another XML file. Instead, we asked our DB expert to import the data dump info, the PostgreSQL database, so we could fetch that data. Did I say fetch? Yes, it is possible to do so with the use of the Data Import Handler and JDBC data source. This recipe will guide you how to do it.

## Getting ready

Please take a look at the recipe named *How to properly configure Data Import Handler with JDBC* in this chapter, to know the basics about configuring the Data Import Handler.

## How to do it...

The Wikipedia data I used in this example is available under the Wikipedia downloads page <http://download.wikimedia.org/>.

The index structure in the `schema.xml` file looks like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true"/>
<field name="name" type="string" indexed="true" stored="true"/>
<field name="revision_id" type="string" indexed="true" stored="true"/>
<field name="contents" type="text" indexed="true" stored="true"/>
```

The next step is to add the request handler definition to the `solrconfig.xml` file. This is done as follows:

```
<requestHandler name="/dataimport" class="org.apache.solr.handler.
dataimport.DataImportHandler">
  <lst name="defaults">
    <str name="config">db-data-config.xml</str>
  </lst>
</requestHandler>
```

Now we have to add a `db-data-config.xml` file to the `conf` directory of the Solr instance (or core):

```
<dataConfig>
<dataSource driver="org.postgresql.Driver" url="jdbc:postgresql://
localhost:5432/wikipedia" user="wikipedia" password="secret" />
<document>
<entity name="page" query="SELECT page_id, page_title from page">
  <field column="page_id" name="id" />
  <field column="page_title" name="name" />
<entity name="revision" query="select rev_id from revision where rev_
page=${page.page_id}">
  <field column="rev_id" name="revision_id" />
<entity name="pagecontent" query="select old_text from pagecontent
where old_id=${revision.rev_id}">
  <field column="old_text" name="contents" />
</entity>
</entity>
</entity>
</document>
</dataConfig>
```

Now let's start indexing. To do that, type the following URL into your browser:

```
http://localhost:8983/solr/data?command=full-import
```

Let's check the indexing status during import. To do that, we run the following query:

```
http://localhost:8983/solr/data
```

Solr will show us a response similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
```



```
</lst>
<lst name="initArgs">
<lst name="defaults">
<str name="config">db-data-config.xml</str>
</lst>
</lst>
<str name="status">busy</str>
<str name="importResponse">A command is still running...</str>
<lst name="statusMessages">
<str name="Time Elapsed">0:1:15.460</str>
<str name="Total Requests made to DataSource">39547</str>
<str name="Total Rows Fetched">59319</str>
<str name="Total Documents Processed">19772</str>
<str name="Total Documents Skipped">0</str>
<str name="Full Dump Started">2010-10-25 14:28:00</str>
</lst>
<str name="WARNING">This response format is experimental. It is
likely to change in the future.</str>
</response>
```

Running the same query after the importing process is done should result in a response similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
<int name="status">0</int>
<int name="QTime">0</int>
</lst>
<lst name="initArgs">
<lst name="defaults">
<str name="config">db-data-config.xml</str>
</lst>
</lst>
<str name="status">idle</str>
<str name="importResponse"/>
<lst name="statusMessages">
<str name="Total Requests made to DataSource">2118645</str>
<str name="Total Rows Fetched">3177966</str>
<str name="Total Documents Skipped">0</str>
<str name="Full Dump Started">2010-10-25 14:28:00</str>
<str name="">Indexing completed. Added/Updated: 1059322 documents.
Deleted 0 documents.</str>
<str name="Committed">2010-10-25 14:55:20</str>
<str name="Optimized">2010-10-25 14:55:20</str>
```

```

<str name="Total Documents Processed">1059322</str>
<str name="Time taken ">0:27:20.325</str>
</lst>
<str name="WARNING">This response format is experimental. It is
likely to change in the future.</str>
</response>

```

## How it works...

To illustrate how the Data Import Handler works, I decided to index the Polish Wikipedia data. I decided to store four fields: page identifier, page name, page revision number, and its contents. The field definition part is fairly simple, so I decided to skip commenting on this.

The request handler definition, Data Import Handler configuration, and command queries were discussed in the *How to properly configure Data Import Handler with JDBC* recipe. The portions of interest in this recipe are in `db-data-config.xml`.

As you can see, we have three entities defined. The first entity gathers data from the `page` table and maps two of the columns to the index fields. The next entity is nested inside the first one and gathers the revision identifier from the table `revision` with the appropriate condition. The revision identifier is then mapped to the index field. The last entity is nested inside the second and gathers data from the `pagecontent` table, again with the appropriate condition. And again, the returned column is mapped to the index field.

We have the response which shows us that the import is still running (the listing with `<str name="importResponse">A command is still running...</str>`). As you can see, there is information about how many data rows were fetched, how many requests to the database were made, how many Solr documents were processed, and how many were deleted. There is also information about the start of the indexing process. One thing you should be aware of is that this response can change in the next versions of Solr and Data Import Handler.

The last listing shows us the summary of the indexation process.

## How to import data using Data Import Handler and delta query

Do you remember the task with the users import from the recipe named *How to properly configure Data Import Handler*? We imported all the users from our client database but it took ages (about two weeks). Our client is very happy with the results; his database is now not used for searching but only updating. And yes, that is the problem for us—how do we update the data in the index? We can't fetch the whole data every time. What we can do is an incremental import, which will modify only the data that has changed since the last import. This task will show you how to do that.

## Getting ready

Please take a look at the recipe named *How to properly configure Data Import Handler with JDBC* in this chapter, to get to know the basics of the Data Import Handler configuration.

## How to do it...

The first thing you should do is add an additional column to the tables you use. So, in our case, let's assume that we added a column named `last_modified`. Now our `db-data-config.xml` would look like this:

```
<dataConfig>
<dataSource driver="org.postgresql.Driver" url="jdbc:postgresql://
localhost:5432/users" user="users" password="secret" />
<document>
<entity name="user" query="SELECT user_id, user_name FROM users"
deltaImportQuery="select user_id, user_name FROM users WHERE user_id =
'${dataimporter.delta.user_id}'" deltaQuery="select user_id FROM users
WHERE last_modified> '${dataimporter.last_index_time}'">
<field column="user_id" name="id" />
<field column="user_name" name="name" />
<entity name="user_desc" query="select desc from users_description
where user_id=${user.user_id}">
<field column="desc" name="description" />
</entity>
</entity>
</document>
</dataConfig>
```

After that, we run a new kind of query to start delta import:

```
http://localhost:8983/solr/data?command=delta-import
```

## How it works...

We modified our database table to include a column named `last_modified`. We need to ensure that the column will be modified at the same time as the table is. Solr will not modify the database, so you have to ensure that your application does.

When running a delta import, Data Import Handler will create a file named `dataimport.properties` inside a Solr configuration directory. In that file, the last index time will be stored as a timestamp. This timestamp will be later used to distinguish whether the data was changed or not. It can be used in a query by using a special variable `${dataimporter.last_index_time}`.

You may have already noticed the two differences: two additional attributes defining an entity named `user-deltaQuery` and `deltaImportQuery`. The first one is responsible for getting the information about which users were modified since the last index. Actually, it only gets the user's unique identifiers. It uses the `last_modified` field to determine which users were modified since the last import. Then the second query, `deltaImportQuery`, is executed. This query gets users with the appropriate unique identifier, to get all the data which we want to index. One thing worth noticing is the way that I used the user identifier in the `deltaImportQuery`. I used the `delta` variable with its `user_id` (the same name as the table column name) variable to get it: `${dataimporter.delta.user_id}`.

You might have noticed that I left the `query` attribute in the entity definition. It's left on purpose. You may need to index the full data once again, so that configuration will be useful for full imports as well as for the partial ones.

Next, we have a query that shows how to run the delta import.

The statuses that are returned by Solr are the same as with the full import, so please refer to the appropriate chapters to see what information they carry.

The delta queries are only supported for the default `SqlEntityProcessor`. This means that you can only use the queries with JDBC data sources.

## How to use Data Import Handler with URL Data Source

Do you remember the first example with the Wikipedia data? We asked our fellow DB expert to import the data dump into PostgreSQL and we fetched the data from there. But what if our colleague is sick and can't help us, and we need to import that data? We can parse the data and send it to Solr, but that's not an option. We don't have much time to do that. So what to do? Yes, you guessed it right: We can use the Data Import Handler and one of its data sources—file data source. This recipe will show you how to do that.

### Getting ready

Please take a look at the recipe named *How to properly configure Data Import Handler with JDBC* in this chapter to get to know the basics of the Data Import Handler configuration.

## How to do it...

Let's take a look at our data source. To be consistent, I chose to index the Wikipedia data. First of all, we work on the index structure. So, our field definition part of `schema.xml` should look like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true"/>
<field name="name" type="string" indexed="true" stored="true"/>
<field name="revision_id" type="string" indexed="true" stored="true"/>
<field name="contents" type="text" indexed="true" stored="true"/>
```

Let's define a Data Import Handler request handler (put that definition in the `solrconfig.xml` file):

```
<requestHandler name="/data" class="org.apache.solr.handler.
dataimport.DataImportHandler">
  <lst name="defaults">
    <str name="config">data-config.xml</str>
  </lst>
</requestHandler>
```

And now, the `data-config.xml` file:

```
<dataConfig>
  <dataSource type="FileDataSource" encoding="UTF-8" />
  <document>
    <entity name="page" processor="XPathEntityProcessor" stream="true"
forEach="/mediawiki/page/" url="/solrcookbook/data/enwiki-20100904-
pages-articles.xml" transformer="RegexTransformer">
      <field column="id" xpath="/mediawiki/page/id" />
      <field column="name" xpath="/mediawiki/page/title" />
      <field column="revision_id" xpath="/mediawiki/page/revision/id" />
      <field column="contents" xpath="/mediawiki/page/revision/text" />
      <field column="$skipDoc" regex="^#REDIRECT .*" replaceWith="true"
sourceColName="contents"/>
    </entity>
  </document>
</dataConfig>
```

Now let's start indexing by sending the following query to Solr:

```
http://localhost:8983/solr/data?command=full-import
```

After the import is done, we will have the data indexed.

## How it works...

The Wikipedia data I used in this example is available under the Wikipedia downloads page <http://download.wikimedia.org/enwiki/>. I've chosen the `pages-articles.xml.bz2` file (actually it was named `enwiki-20100904-pages-articles.xml.bz2`), which was about 6GB. We only want to index some of the data from the file: page identifier, name, revision, and page contents. I also wanted to skip the articles that are only linking to other articles in Wikipedia.

The field definition part of the `theschema.xml` file is fairly simple and contains only four fields.

`solrconfig.xml` contains the handler definition with the information about the Data Import Handler configuration filename.

Next we have the `data-config.xml` file, where the actual configuration is written. We have a new data source type here, named `FileDataSource`. This data source will read the data from a local directory. You can use `HttpDataSource` if you want to read the data from an outer location. The XML tag defining the data source also specifies file encoding (`encoding` attribute) – in our example it's `UTF-8`. Next we have an entity definition, which has a name under which it will be visible: a processor which will process our data. The `processor` attribute is only mandatory when not using a database source. This value must be set to `XPathEntityProcessor` in our case. The `stream` attribute, which is set to `true`, informs the Data Import Handler to stream the data from the file, which is a must in our case when the data is large. Following that, we have a `forEach` attribute, which specifies the `xPath` expression. This path will be iterated over. There is a location of the data file defined in the `url` attribute and a transformer defined in the `transformer` attribute. Transformer is a mechanism that will transform every row of the data and process it before sending it to Solr.

Under the entity definition, we have field mapping definitions. We have columns which are the same as the index field names, thus I skipped the `name` field. There is one additional attribute named `xpath` in the mapping definitions. It specifies the `xPath` expression that defines where the data is located in the XML file. If you are not familiar with `xPath`, please refer to the <http://www.w3schools.com/xpath/default.asp> tutorial.

We also have a special column named `$skipDoc`. It tells Solr which documents to skip (if the value of the column is `true`, then Solr will skip the document). The column is defined by a regular expression (attribute `regex`), a column to which the regular expression applies (attribute `sourceColName`), and the value that will replace all the occurrences of the given regular expression (`replaceWith` attribute). If the regular expression matches (in this case, if the data in the column specified by the `sourceColName` attribute starts with `#REDIRECT`), then the `$skipDoc` column will be set to `true`, and thus the document will be skipped.

The actual indexing time was more than four hours on my machine. So, if you try to index the sample Wikipedia data, please take that into consideration.

## How to modify data while importing with Data Import Handler

After we indexed the users and made the indexing incremental (*How to properly configure Data Import Handler with JDBC* and *How to import data using Data Import Handler and delta query tasks*), we were asked if we can modify the data a bit. Actually, it would be perfect if we could split the name and surname into two fields in the index, while those two reside in a single column in the database. And of course, updating the database is not an option (trust me, it almost never is). Can we do that? Of course we can. We just need to add some more configuration details in the Data Import Handler and use transformer. This recipe will show you how to do that.

### Getting ready

Please take a look at the recipe named *How to properly configure Data Import Handler with JDBC* in this chapter to get to know the basics about the Data Import Handler configuration.

Also, to be able to run the examples given in this chapter, you need to run Solr in the servlet container that is run on Java 6 or later.

### How to do it...

Let's assume that we have a database table. To select users from our table, we use the following SQL query:

```
SELECT id, name, description FROM users
```

The response may look like this:

id	name	description
1	John Kowalski	superuser
2	Amanda Looks	user

Our task is to split the name from the surname, and place it in two fields: name and surname.

First of all, change the index structure. Our field definition part of `schema.xml` should look like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true"/>
<field name="firstname" type="string" indexed="true" stored="true"/>
<field name="surname" type="string" indexed="true" stored="true"/>
<field name="description" type="text" indexed="true" stored="true"/>
```

Now we have to add a `db-data-config.xml` file:

```
<dataConfig>
<dataSource driver="org.postgresql.Driver" url="jdbc:postgresql://
localhost:5432/users" user="users" password="secret" />
<script><![CDATA[
    function splitName(row) {
varnameTable = row.get('name').split(' ');
row.put('firstname', nameTable[0]);
row.put('surname', nameTable[1]);
row.remove('name');
    return row;
    }
  ]]></script>
<document>
<entity name="user" transformer="script:splitName" query="SELECT id,
name, description from users">
<field column="id" />
<field column="firstname" />
<field column="surname" />
<field column="description" />
</entity>
</entity>
</document>
</dataConfig>
```

And now you can follow the normal indexing procedure which was discussed in the *How to properly configure Data Import Handler with JDBC* recipe.

## How it works...

The first two listings are the sample SQL query and the result given by a database. Next we have a field definition part of the `schema.xml` file, which defines four fields. Please look at the example database rows once again. Can you see the difference? We have four fields in our index, while our database rows have only three columns. We must split the contents of the `name` column into two index fields: `firstname` and `surname`. To do that, we will use the JavaScript language and the script transformer functionality of the Data Import Handler.

The `solrconfig.xml` file is the same as the one discussed in the *How to properly configure Data Import Handler with JDBC* recipe in this chapter, so I'll skip that as well.



Next, we have the updated contents of the `db-data-config.xml` file, which we use to define the behavior of the Data Import Handler. The first and the biggest difference is the `script` tag that will be holding our scripts that parse the data. The scripts should be held in the CDATA section. I defined a simple function called `splitName` that takes one parameter, the database row (remember that the functions that operate on entity data should always take one parameter). The first thing in the function is getting the content of the `name` column, split it with the space character, and assign it to the JavaScript table. Then we create two additional columns in the processed row—`firstname` and `surname`. The contents of those rows come from the JavaScript table we created. Then we remove the `name` column because we don't want it to be indexed. The last operation is the returning of the processed row.

To enable script processing, you must add one additional attribute to the entity definition—the `transformer` attribute with the contents like `script:functionName`. In our example, it looks like this: `transformer:"script:splitName"`. It tells the Data Import Handler to use the defined function name for every row returned by the query.

And that's how it works. The rest is the usual indexing process described in the *How to properly configure Data Import Handler with JDBC* recipe in this chapter.

### There's more...

There are a few things worth noticing.

#### Using scripts other than JavaScript

If you want to use a different language than JavaScript, then you have to specify it in the `language` attribute of the `<script>` tag. Just remember that the scripting language that you want to use must be supported by Java 6. The example definition would look like this:

```
<script language="ECMAScript">...</script>
```

# 3

## Analyzing your Text Data

In this chapter, we will cover:

- ▶ Storing additional information using payloads
- ▶ Eliminating XML and HTML tags from the text
- ▶ Copying the contents of one field to another
- ▶ Changing words to other words
- ▶ Splitting text by camel case
- ▶ Splitting text by whitespace only
- ▶ Making plural words singular, but without stemming
- ▶ Lowercasing the whole string
- ▶ Storing geographical points in the index
- ▶ Stemming your data
- ▶ Preparing text to do efficient trailing wildcard search
- ▶ Splitting text by numbers and non-white space characters

## Introduction

The process of data indexing can be divided into different parts. One of the parts, actually one of the last parts, of this process is data analysis. It's one of the crucial parts of data preparation. It defines how your data will be written into index, its structure, and so on. In Solr, data behavior is defined by types. Type's behavior can be defined in the context of the indexing process or the context of the query process, or both. Furthermore, type definition is composed of tokenizers and filters (both token filters and character filters). Tokenizer specifies how your data will be preprocessed after it is sent to the appropriate field. Analyzer operates on the whole data that is sent to the field. Types can only have one tokenizer. The result of the tokenizer work is a stream of objects called tokens. Next in the analysis chain are the filters. They operate on the tokens in the token stream. And they can do anything with the tokens—changing them, removing them, or for example, making them lowercase. Types can have multiple filters.

One additional type of filter is the character filter. The character filters do not operate on tokens from the token stream. They operate on the data that is sent to the field and they are invoked before the data is sent to the analyzer.

This chapter will focus on the data analysis and how to handle the common day-to-day analysis questions and problems.

## Storing additional information using payloads

Imagine that you have a powerful preprocessing tool that can extract information about all the words in the text. Your boss would like you to use it with Solr or at least store the information it returns in Solr. So what can you do? We can use something that is called payload and use it to store that data. This recipe will show you how to do it.

### How to do it...

I assumed that we already have an application that takes care of recognizing the part of speech in our text data. Now we need to add it to the Solr index. To do that we will use payloads, a metadata that can be stored with each occurrence of a term.

First of all, you need to modify the index structure. For this, we will add the new field type to the `schema.xml` file:

```
<fieldtype name="partofspeech" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.DelimitedPayloadTokenFilterFactory"
      encoder="integer" delimiter="|"/>
  </analyzer>
</fieldtype>
```

```
</analyzer>
</fieldtype>
```

Now add the field definition part to the `schema.xml` file:

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="text" type="text" indexed="true" stored="true" />
<field name="speech" type="partofspeech" indexed="true" stored="true"
multivalued="true" />
```

Now let's look at what the example data looks like (I named it `ch3_payload.xml`):

```
<add>
<doc>
<field name="id">1</field>
<field name="text">ugly human</field>
<field name="speech">ugly|3 human|6</field>
</doc>
<doc>
<field name="id">2</field>
<field name="text">big book example</field>
<field name="speech">big|3 book|6 example|1</field>
</doc>
</add>
```

Let's index our data. To do that, we run the following command from the `exampledocs` directory (put the `ch3_payload.xml` file there):

```
java -jarpost.jar ch3_payload.xml
```

## How it works...

What information can payload hold? It may hold information that is compatible with the encoder type you define for the `solr.DelimitedPayloadTokenFilterFactory` filter. In our case, we don't need to write our own encoder—we will use the supplied one to store integers. We will use it to store the boost of the term. For example, nouns will be given a token boost value of 6, while the adjectives will be given a boost value of 3.

First we have the type definition. We defined a new type in the `schema.xml` file, named `partofspeech` based on the Solr text field (attribute `class="solr.TextField"`). Our tokenizer splits the given text on whitespace characters. Then we have a new filter which handles our payloads. The filter defines an encoder, which in our case is an integer (attribute `encoder="integer"`). Furthermore, it defines a delimiter which separates the term from the payload. In our case, the separator is the pipe character `|`.

Next we have the field definitions. In our example, we only define three fields:

- ▶ Identifier
- ▶ Text
- ▶ Recognized speech part with payload

Now let's take a look at the example data. We have two simple fields: `id` and `text`. The one that we are interested in is the `speech` field. Look how it is defined. It contains pairs which are made of a term, delimiter, and boost value. For example, `book|6`. In the example, I decided to boost the nouns with a boost value of 6 and adjectives with the boost value of 3. I also decided that words that cannot be identified by my application, which is used to identify parts of speech, will be given a boost of 1. Pairs are separated with a space character, which in our case will be used to split those pairs. This is the task of the tokenizer which we defined earlier.

To index the documents, we use simple post tools provided with the example deployment of Solr. To use it, we invoke the command shown in the example. The post tools will send the data to the default update handler found under the address `http://localhost:8983/solr/update`. The following parameter is the file that is going to be sent to Solr. You can also post a list of files, not just a single one.

That is how you index payloads in Solr. In the 1.4.1 version of Solr, there is no further support for payloads. Hopefully this will change. But for now, you need to write your own query parser and similarity class (or extend the ones present in Solr) to use them.

## Eliminating XML and HTML tags from the text

There are many real-life situations when you have to clean your data. Let's assume that you want to index web pages that your client sends you. You don't know anything about the structure of that page—one thing you know is that you must provide a search mechanism that will enable searching through the content of the pages. Of course, you could index the whole page by splitting it by whitespaces, but then you would probably hear the clients complain about the HTML tags being searchable and so on. So, before we enable searching on the contents of the page, we need to clean the data. In this example, we need to remove the HTML tags. This recipe will show you how to do it with Solr.

### How to do it...

Let's suppose our data looks like this (the `ch3_html.xml` file):

```
<add>
<doc>
<field name="id">1</field>
<field name="html"><![CDATA[<html><head><title>My page</title></
```

```

    head><body><p>This is a <b>my</b><i>sample</i> page</body></html>]]></
    field>
  </doc>
</add>

```

Now let's take care of the `schema.xml` file. First add the type definition to the `schema.xml` file:

```

<fieldType name="html_strip" class="solr.TextField">
  <analyzer>
    <charFilter class="solr.HTMLStripCharFilterFactory"/>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>

```

And now, add the following to the field definition part of the `schema.xml` file:

```

<field name="id" type="string" indexed="true" stored="true"
  required="true" />
<field name="html" type="html_strip" indexed="true" stored="false" />

```

Let's index our data. To do that, we run the following command from the `exampledocs` directory (put the `ch3_html.xml` file there):

```
java -jar post.jar ch3_html.xml
```

If there were no errors, you should see a response like this:

```

SimplePostTool: version 1.2
SimplePostTool: WARNING: Make sure your XML documents are encoded in UTF-
8, other encodings are not currently supported
SimplePostTool: POSTing files to http://localhost:8983/solr/update..
SimplePostTool: POSTingfile ch3_html.xml
SimplePostTool: COMMITting Solr index changes..

```

## How it works...

First of all, we have the data example. In the example, we see one file with two fields; the identifier and some HTML data nested in the CDATA section. You must remember to surround the HTML data in CDATA tags if they are full pages, and start from HTML tags like our example, otherwise Solr will have problems with parsing the data. However, if you only have some tags present in the data, you shouldn't worry.

Next, we have the `html_strip` type definition. It is based on `solr.TextField` to enable full-text searching. Following that, we have a character filter which handles the HTML and the XML tags stripping. This is something new in Solr 1.4. The character filters are invoked before the data is sent to the tokenizer. This way they operate on untokenized data. In our case, the character filter strips the HTML and XML tags, attributes, and so on. Then it sends the data to the tokenizer, which splits the data by whitespace characters. The one and only filter defined in our type makes the tokens lowercase to simplify the search.

To index the documents, we use simple post tools provided with the example deployment of Solr. To use it we invoke the command shown in the example. The post tools will send the data to the default update handler found under the address `http://localhost:8983/solr/update`. The parameter of the command execution is the file that is going to be sent to Solr. You can also post a list of files, not just a single one.

As you can see, the sample response from the post tools is rather informative. It provides information about the update handler address, files that were sent, and information about commits being performed.

If you want to check how your data was indexed, remember not to be mistaken when you choose to store the field contents (attribute `stored="true"`). The stored value is the original one sent to Solr, so you won't be able to see the filters in action. If you wish to check the actual data structures, please take a look at the Luke utility (a utility that lets you see the index structure, field values, and operate on the index). Luke can be found at the following address: `http://code.google.com/p/luke`.

Solr provides a tool that lets you see how your data is analyzed. That tool is a part of Solr administration pages. More information about it can be found in *Chapter 4, Solr Administration*.

## Copying the contents of one field to another

Imagine that you have many big XML files that hold information about the books that are stored on library shelves. There is not much data, just the unique identifier, name of the book, and the name of the author. One day your boss comes to you and says: "Hey, we want to facet and sort on the basis of the book author". You can change your XML and add two fields, but why do that when you can use Solr to do that for you? Well, Solr won't modify your data, but it can copy the data from one field to another. This recipe will show you how to do that.

### How to do it...

Let's assume that our data looks like this:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="name">Solr Cookbook</field>
  <field name="author">John Kowalsky</field>
```

```

</doc>
<doc>
<field name="id">2</field>
<field name="name">Some other book</field>
<field name="author">Jane Kowalsky</field>
</doc>
</add>

```

We want the contents of the `author` field to be present in the fields named `author`, `author_facet`, and `author_sort`. So let's define the copy fields in the `schema.xml` file (place the following right after the fields section):

```

<copyField source="author"dest="author_facet"/>
<copyField source="author"dest="author_sort"/>

```

And that's all. Solr will take care of the rest.

The field definition part of the `schema.xml` file could look like this:

```

<field name="id" type="string" indexed="true" stored="true"
required="true"/>
<field name="author" type="text" indexed="true" stored="true"
multiValued="true"/>
<field name="name" type="text" indexed="true" stored="true"/>
<field name="author_facet" type="string" indexed="true"
stored="false"/>
<field name="author_sort" type="alphaOnlySort" indexed="true"
stored="false"/>

```

Let's index our data. To do that, we run the following command from the `exampledocs` directory (put the `ch3_html.xml` file there):

```
java -jar post.jar data.xml
```

## How it works...

As you can see in the example, we have only three fields defined in our sample data XML file. There are two fields which we are not particularly interested in: `id` and `name`. The field that interests us the most is the `author` field. As I have mentioned earlier, we want to place the contents of that field in three fields:

- ▶ `Author` (the actual field that will be holding the data)
- ▶ `author_sort`
- ▶ `author_facet`

To do that we use the copy fields. Those instructions are defined in the `schema.xml` file, right after the field definitions, that is, after the `</fields>` tag. To define a copy field, we need to specify a source field (attribute `source`) and a destination field (attribute `dest`).



After the definitions, like those in the example, Solr will copy the contents of the source fields to the destination fields during the indexing process. There is one thing that you have to be aware of—the content is copied before the analysis process takes place. This means that the data is copied as it is stored in the source.

### There's more...

There are a few things worth noting when talking about copying contents of the field to another field.

#### Copying the contents of dynamic fields to one field

You can also copy multiple field content to one field. To do that, you should define a copy field like this:

```
<copyField source="*_author" dest="authors"/>
```

The definition like the one above would copy all of the fields that end with `_author` to one field named `authors`. Remember that if you copy multiple fields to one field, the destination field should be defined as multivalued.

#### Limiting the number of characters copied

There may be situations where you only need to copy a defined number of characters from one field to another. To do that we add the `maxChars` attribute to the copy field definition. It can look like this:

```
<copyField source="author" dest="author_facet" maxChars="200"/>
```

The above definition tells Solr to copy up to 200 characters from the `author` field to the `author_facet` field. This attribute can be very useful when copying the content of multiple fields to one field.

### Changing words to other words

Let's assume we have an e-commerce client and we are providing a search system based on Solr. Our index has hundreds of thousands of documents which mainly consist of books, and everything works fine. Then one day, someone from the marketing department comes into your office and says that he wants to be able to find all the books that contain the word "machine" when he types "electronics" into the search box. The first thing that comes to mind is: 'hey, do it in the source and I'll index that!'. But that is not an option this time, because there can be many documents in the database that have those words. We don't want to change the whole database. That's when synonyms come into play and this recipe will show you how to use it.

## How to do it...

To make the example as simple as possible, I assumed that we only have two fields in our index. This is how the field definition section in the `schema.xml` file looks like (just add it to your `schema.xml` file into the field section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="description" type="text_syn" indexed="true" stored="true"
/>
```

Now let's add the `text_syn` type definition to the `schema.xml` file, as shown in the code snippet:

```
<fieldType name="text_syn" class="solr.TextField">
<analyzer type="query">
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
<analyzer type="index">
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="false" />
<filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
</fieldType>
```

As you notice, there is a file mentioned `synonyms.txt`. Let's take a look at its contents:

```
machine => electronics
```

The `synonyms.txt` file should be placed in the same directory as the other configuration files.

## How it works...

First we have our field definition. There are two fields: identifier and description. The second one should be of our interest right now. It's based on the new type `text_syn`, which is shown in the second listing.

Now about the new type, `text_syn`—it's based on the `solr.TextField` class. Its definition is divided. It behaves one way while indexing and in a different way while querying. So the first thing we see is the query time analyzer definition. It consists of the tokenizer that splits the data on the basis of whitespace characters and then the lowercase filter makes all the tokens lowercase. The interesting part is the index time behavior. It starts with the same tokenizer, but then the synonyms filter comes into play. Its definition starts like all the other filters with the factory definition. Next we have a `synonyms` attribute which defines which file contains the synonyms definition. Following that we have the `ignoreCase` attribute which tells Solr to ignore the case of the tokens and the contents of the synonyms file.

The last attribute named `expand` is set to `false`. This means that Solr won't be expanding the synonyms and all equivalent synonyms will be reduced to the first synonym in the line. If the attribute is set to `true`, all synonyms will be expanded to all equivalent forms.

The example `synonyms.txt` file tells Solr that when the word `machine` appears in the field based on the `text_syn` type, it should be replaced by `electronics`, but not the other way round. Each synonym rule should be placed in a separate line in the `synonyms.txt` file. Also, remember that the file should be written in the UTF-8 file encoding. This is crucial and you should always remember it, because Solr will expect the file to be encoded in UTF-8.

### There's more...

There is one more thing I would like to add when talking about synonyms.

#### Equivalent synonyms setup

Let's get back to our example for a second. But what if the person from the marketing department says that he wants not only to be able to find books that have the word `machine` to be found when entering the word `electronics`, but also all the books that have the word `electronics` to be found when entering the word `machine`. The answer is simple. First, we would set the `expand` attribute (of the filter) to `true`. Then we would change our `synonyms.txt` file to something like this:

```
machine, electronics
```

And as I said earlier, Solr would expand synonyms to equivalent forms.

### Splitting text by camel case

Let's suppose that you run an e-commerce site with an electronic assortment. The marketing department can be a source of many great ideas. Imagine that one time your colleague from this department comes to you and says that they would like your search application to be able to find documents containing the word "PowerShot" by entering the words "power" and "shot" into the search box. So can we do that? Of course, and this recipe will show you how.

## How to do it...

Let's assume that we have the following index structure (add this to your `schema.xml` file, to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="description" type="text_split" indexed="true"
stored="true" />
```

To split text in the `description` field, we should add the following type definition to the `schema.xml` file:

```
<fieldType name="text_split" class="solr.TextField">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
splitOnCaseChange="1"/>
<filter class="solr.LowerCaseFilterFactory" />
</analyzer>
</fieldType>
```

To test our type, I've indexed the following XML file:

```
<add>
<doc>
<field name="id">1</field>
<field name="description">TextTest</field>
</doc>
</add>
```

Then I run the following query in the web browser:

```
http://localhost:8983/solr/select?q=description:test
```

You should get the indexed document in response.

## How it works...

Let's see how things work. First of all, we have the field definition part of the `schema.xml` file. This is pretty straightforward. We have two fields defined: one that is responsible for holding the information about the identifier (`id` field) and the second is responsible for the product description (`description` field).

Next, we see the interesting part. We name our type `text_split` and base it on a text type, `solr.TextField`. We also told Solr that we want our text to be tokenized by whitespaces by adding the whitespace tokenizer (`tokenizer` tag). To do what we want to do (split by case change) we need more than this. Actually we need a filter named `WordDelimiterFilter`, which is created by the `solr.WordDelimiterFilterFactory` class and a `filter` tag. We also need to define the appropriate behavior of the filter, so we add two attributes: `generateWordParts` and `splitOnCaseChange`. The values of these two parameters are set to 1, which means that they are turned on. The first attribute tells Solr to generate word parts, which means that the filter will split the data on non-letter characters. We also add the second attribute which tells Solr to split the tokens by case change.

What will that configuration do with our sample data? As you can see, we have one document sent to Solr. The data in the `description` field will be split into two words: `text` and `test`. You can check it yourself by running the example query in your web browser.

## Splitting text by whitespace only

One of the most common problems that you have probably come across is having to split the text with whitespaces in order to segregate words from each other, to be able to process it further. This recipe will show you how to do it.

### How to do it...

Let's assume that we have the following index structure (add this to your `schema.xml` file in the field definition section):

```
<field name="description_string" type="string" indexed="true"
stored="true" />
<field name="description_split" type="text_split" indexed="true"
stored="true" />
```

To split the text in the `description` field, we should add the following type definition:

```
<fieldType name="text_split" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  </analyzer>
</fieldType>
```

To test our type, I've indexed the following XML file:

```
<add>
  <doc>
    <field name="description_string">test text</field>
    <field name="description_text">test text</field>
  </doc>
</add>
```

Then I run the following query in the web browser:

```
http://localhost:8983/solr/select?q=description_split:text
```

You should get the indexed document in response.

On the other hand, you won't get the indexed document in response after running the following query:

```
http://localhost:8983/solr/select?q=description_string:text
```

### How it works...

Let's see how things work. First of all we have the field definition part of the `schema.xml` file. This is pretty straightforward. We have two fields defined: one named `description_string`, which is based on a string field and thus not analyzed, and the second is the `description_split` field, which is based on our `text_split` type and will be tokenized on the basis of whitespace characters.

Next, we see the interesting part. We named our type `text_split` and based it on a text type, `solr.TextField`. We told Solr that we want our text to be tokenized by whitespaces by adding a whitespace tokenizer (`tokenizer` tag). Because there are no filters defined, the text will be tokenized only by whitespace characters and nothing more.

That's why our sample data in the `description_text` field will be split into two words: `test` and `text`. On the other hand, the text in the `description_string` field won't be split. That's why the first example query will result in one document in response, while the second example won't find the example document.

## Making plural words singular, but without stemming

Nowadays it's nice to have stemming algorithms in your application. But let's imagine that you have a search engine that searches through the contents of the books in the library. One of the requirements is changing the plural forms of the words from plural to singular – nothing less, nothing more. Can Solr do that? Yes, the newest one can, and this recipe will show you how.

### How to do it...

Let's assume that our index consists of two fields (add this to your `schema.xml` file, to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
```

```
<field name="description" type="text_light_stem" indexed="true"
stored="true" />
```

Our `text_light_stem` type should look like this (add this to your `schema.xml` file):

```
<fieldType name="text_light_stem" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.EnglishMinimalStemFilterFactory" />
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

If you would like to check the analysis tool of Solr administration pages, you should see that words like `ways` and `keys` are changed to their singular forms.

## How it works...

First of all, we need to define the fields in the `schema.xml` file. To do that, we add the contents from the first example into that file. It tells Solr that our index will consist of two fields: the `id` field, which will be responsible for holding the information about the unique identifier of the document and the `description` field, which will be responsible for holding the document description.

The `description` field is actually where the magic is being done. We defined a new field type for that field and we called it `text_light_stem`. The field definition consists of a tokenizer and two filters. If you want to know how this tokenizer behaves, please refer to the *Splitting text by whitespace only* recipe in this chapter. The first filter is a new one. This is the light stemming filter that we will use to perform minimal stemming. The class that enables Solr to use that filter is `solr.EnglishMinimalStemFilterFactory`. This filter takes care of the process of light stemming. You can see that by using the analysis tools of the Solr administration panel. The second filter defined is the lowercase filter; you can see how it works by referring to the *Lowercasing the whole string* recipe in this chapter.

After adding this to your `schema.xml` file, you should be able to use the light stemming algorithm.

## There's more...

Light stemming supports a number of different languages. To use the light stemmers for your respective language, add the following filters to your type:

Language	Filter
Russian	<code>solr.RussianLightStemFilterFactory</code>
Portuguese	<code>solr.PortugueseLightStemFilterFactory</code>
French	<code>solr.FrenchLightStemFilterFactory</code>
German	<code>solr.GermanLightStemFilterFactory</code>
Italian	<code>solr.ItalianLightStemFilterFactory</code>
Spanish	<code>solr.SpanishLightStemFilterFactory</code>
Hungarian	<code>solr.HungarianLightStemFilterFactory</code>
Swedish	<code>solr.SwedishLightStemFilterFactory</code>

## Lowercasing the whole string

Let's get back to our books search example. This time your boss comes to you and says that all book names should be searchable when the user types lower or uppercase characters. Of course, Solr can do that, and this recipe will describe how to do it.

### How to do it...

Let's assume that we have the following index structure (add this to your `schema.xml` file in the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="string_lowercase" indexed="true"
stored="true" />
<field name="description" type="text" indexed="true" stored="true" />
```

To make our strings lowercase, we should add the following type definition to the `schema.xml` file:

```
<fieldType name="string_lowercase" class="solr.TextField">
<analyzer>
<tokenizer class="solr.KeywordTokenizerFactory"/>
<filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
</fieldType>
```

To test our type, I've indexed the following XML file:

```
<add>
<doc>
<field name="id">1</field>
<field name="name">Solr Cookbook</field>
```



```
<field name="description">Simple description</field>
</doc>
</add>
```

Then I run the following query in the web browser:

```
http://localhost:8983/solr/select?q=name:"solr cookbook"
```

You should get the indexed document in the response.

On the other hand, you should be able to get the indexed document in the response after running the following query:

```
http://localhost:8983/solr/select?q=name:"solr Cookbook"
```

## How it works...

Let's see how things work. First of all, we have the field definition part of the `schema.xml` file. This is pretty straightforward. We have three fields defined: first the field named `id`, which is responsible for holding our unique identifier; the second is the `name` field, which is actually our lowercase string field; and the third field will be holding the description of our documents, and is based on the standard `text` type defined in the example Solr deployment.

Now let's get back to our `name` field. It's based on the `string_lowercase` type. Let's look at that type. It consists of an analyzer, which is defined as a tokenizer, and one filter. `solr.KeywordTokenizerFactory` tells Solr that the data in that field should not be tokenized in any way. It should just be passed as a single token to the token stream. Next we have our filter, which is changing all the characters to their lowercase equivalents. And that's how this field analysis is performed.

The example queries show how the field behaves. It doesn't matter if you type lower or uppercase characters, the document will be found anyway. What matters is that you must type the whole string as it is, because we used the keyword tokenizer which, as I already said, is not tokenizing but just passing the whole data through the token stream as a single token.

## Storing geographical points in the index

Imagine that till now your application stores information about companies. Not much information, only the unique identification and the company name. But now, your client wants to store the location of the companies. Not a problem—just two additional fields. But a company can have multiple addresses and thus can have multiple geographical points assigned. So now, how to do that in Solr? Of course, we can add multiple dynamic fields and remember the field names in our application, but that isn't comfortable. In this recipe, I'll show how to store pairs of fields; in our case, the geographical point.

## How to do it...

Let's assume that the companies that we store are defined by three fields (add this to your `schema.xml` file, to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="location" type="point" indexed="true"
stored="true" multiValued="true" />
```

We should also have one dynamic field defined (add this to your `schema.xml` file in the field definition section):

```
<dynamicField name="*_d" type="double" indexed="true" stored="true"/>
```

Our point type should look like this:

```
<fieldType name="point" class="solr.PointType" dimension="2"
subFieldSuffix="_d"/>
```

Now let's see how our example data will look (I named the data file `task9.xml`):

```
<add>
<doc>
<field name="id">1</field>
<field name="name">Solr.pl company</field>
<field name="location">10,10</field>
<field name="location">20,20</field>
</doc>
</add>
```

Let's index our data. To do that, we run the following command from the `exampledocs` directory (put the `task9.xml` file there):

```
java -jar post.jar task9.xml
```

After indexing we should be able to use the query, like the following one, to get our data:

```
http://localhost:8983/solr/select?q=location:10,10
```

The response should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
<int name="status">0</int>
<int name="QTime">2</int>
<lst name="params">
```

```
<str name="q">location:10,10</str>
</lst>
</lst>
<result name="response" numFound="1" start="0">
<doc>
<str name="id">1</str>
<arr name="location">
<str>10,10</str>
<str>20,20</str>
</arr>
<arr name="location_0_d">
<double>10.0</double>
<double>20.0</double>
</arr>
<arr name="location_1_d">
<double>10.0</double>
<double>20.0</double>
</arr>
<str name="name">Solr.pl company</str>
</doc>
</result>
</response>
```

## How it works...

First of all, we have three fields and one dynamic field defined in our `schema.xml` file. The first field is the one responsible for holding the unique identifier. The second one is responsible for holding the name of the company. The third one, named `location`, is responsible for holding the geographical points and can have multiple values. The dynamic field will be used as a helper for the `point` type.

Next we have our `point` type definition. It's based on the `solr.PointType` class and is defined by two attributes:

- ▶ `dimension`: The number of dimensions that the field will be storing. In our case, we will need to store pairs of values, so we need to set this attribute to 2.
- ▶ `subFieldSuffix`: The field that will be used to store the actual values of the field. This is where we need our dynamic field. We tell Solr that our helper field will be the dynamic field ending with the suffix of `_d`.

So how does this type of field actually work? When defining a two dimensional field, like we did, there are actually three fields created in the index. The first field is named like the field we added in the `schema.xml` file, so in our case it is `location`. This field will be responsible for holding the stored value of the field. And one more thing, this field will only be created when we set the field attribute `store` to `true`.

The next two fields are based on the defined dynamic field. Their names will be `field_0_d` and `field_1_d`. First we have the field name, then the `_` character, then the index of the value, then another `_` character, and finally the suffix defined by the `subFieldSuffix` attribute of the type.

We can now look at the way the data is indexed. Please take a look at the example data file. You can see that the values in each pair are separated by the comma character. And that's how you can add the data to the index.

Querying is just the same as indexing in the way the pairs should be represented. The example query shows how you should be making your queries. It differs from the standard, one-valued fields with only one thing—each value in the pair is separated by a comma character and passed to the query.

The response is shown just to illustrate how the fields are stored. You can see, beside the `location` field, that there were two dynamic fields `location_0_d` and `location_1_d` created.

### There's more...

If you wish to store more than two dimensions in a field, you should change the `dimensions` attribute of the field. For example, if you want to store four dimensions in a field, your definition should look like this:

```
<fieldType name="tetragon" class="solr.PointType" dimension="4" subFieldSuffix="_i"/>
```

## Stemming your data

One of the most common requirements I meet is stemming. Let's imagine the book e-commerce store, where you store the books' names and descriptions. We want to be able to find words like `shown` and `showed` when we type the word `show`, and vice versa. To achieve that, we can use stemming algorithms. This recipe will show you how to add stemming to your data analysis.

### How to do it...

Let's assume that our index consists of three fields (add this to your `schema.xml` file, to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="description" type="text_stem" indexed="true"
stored="true" />
```

Our `text_stem` type should look like this:

```
<fieldType name="text_stem" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.SnowballPorterFilterFactory" />
  </analyzer>
</fieldType>
```

Now you can index your data. Let's create an example data file:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="name">Solr cookbook</field>
  <field name="description">This is a book that I'll show</field>
</doc>
<doc>
  <field name="id">2</field>
  <field name="name">Solr cookbook 2</field>
  <field name="description">This is a book I showed</field>
</doc>
</add>
```

After indexing, we can test how our data was analyzed. To do that, let's run the following query:

```
http://localhost:8983/solr/select?q=description:show
```

That's right, Solr found two documents matching the query, which means that our field is working as intended.

## How it works...

Our index consists of three fields: one holding the unique identifier of the document, the second one holding the name of the document, and the third one holding the document description. The last field is the field that will be stemmed.

The stemmed field is based on a Solr text field and has an analyzer that is used at the query and indexing time. It is tokenized on the basis of the whitespace characters. Then the stemming filter is used. What does the filter do? It tries to bring the words to its root form, meaning that words like `shows`, `showing`, and `show` will all be changed to `show`, or at least they should be changed to that form.

Please note that in order to properly use stemming algorithms, they should be used on the query and indexing times. It is a must because of the stemming results.

As you can see, our test data consists of two documents. Please take a look at the description. One of the documents has the word `showed` and the other has the word `show` in their description fields. After indexing and running the sample query, Solr would return two documents in the result, which means that the stemming did its job.

### There's more...

There are two other things I would like to mention when talking about stemming.

#### Alternative English stemmer

If you find that the snowball porter stemmer is not sufficient for your needs (for example, if the first one is too invasive or too slow), you can try the other stemmer for English available in Solr. To do that, you change your stemming filter to the following one:

```
<filter class="solr.PorterStemFilterFactory" />
```

#### Stemming other languages

There are too many languages that have stemming support integrated into Solr to mention them all. If you are using a language other than English, please refer to the <http://wiki.apache.org/solr/LanguageAnalysis> page of the Solr wiki to find the appropriate filter.

## Preparing text to do efficient trailing wildcard search

Many users coming from traditional RDBMS systems are used to wildcard searches. The most common of them are the ones using the `*` characters, which means zero or more characters. You have probably seen searches like:

```
AND name LIKE 'ABC12%'
```

So, how to do that with Solr and not kill our Solr server? This recipe will show you how to prepare your data and make efficient searches.

### How to do it...

Let's assume we have the following index structure (add this to your `schema.xml` file, to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="string_wildcard" indexed="true" stored="true"
/>
```

Now, let's define our `string_wildcard` type (add this to the `schema.xml` file).

```
<fieldType name="string_wildcard" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.EdgeNGramFilterFactory" minGramSize="1"
      maxGramSize="25" side="front"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  </analyzer>
</fieldType>
```

The example data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">XYZ1234ABC12POI</field>
  </doc>
</add>
```

Now, send the following query to Solr:

```
http://localhost:8983/solr/select?q=name:XYZ1
```

As you see that document has been found, it means our setup is working as intended.

## How it works...

First of all, let's look at our index structure defined in the `schema.xml` file. We have two fields: one holding the unique identifier of the document (`id` field) and the second one holding the name of the document (`name` field), which is actually the field we are interested in.

The `name` field is based on the new type we defined, `string_wildcard`. This type is responsible for enabling trailing wildcards, the ones that will enable SQL queries like `LIKE 'WORD%'`. As you can see, the field type is divided into two analyzers: one for the data analysis during indexing and the other for queries processing. The querying one is straightforward—it just tokenizes the data on the basis of whitespace characters. Nothing more and nothing less.

Now, the indexing time analysis (of course we are talking about the `name` field). Similar to the query time, the data is tokenized on the basis of whitespace characters during indexing, but there is also an additional filter defined. `solr.EdgeNGramFilterFactory` is responsible for generating the so called n-grams. In our setup, we tell Solr that the minimum length of an n-gram is 1 (`minGramSize` attribute) and the maximum length is 25 (`maxGramSize` attribute). We also defined that the analysis should start from the beginning of the text (`side` attribute set to `front`). So what would Solr do with our example data? It will create the following tokens from the example text: `x`, `xy`, `xyz`, `xyz1`, `xyz12`, and so on. It will create tokens by adding the next character from the string to the previous token, up to the maximum length of n-gram that is given in the filter configuration.

So by typing the example query, we can be sure that the example document will be found because of the n-gram filter defined in the configuration of the field. We also didn't define the n-gram in the querying stage of analysis because we don't want our query to be analyzed in such a way, because that could lead to false positive hits and we don't want that to happen.

By the way, this functionality, as described, can be successfully used to provide autocomplete (if you are not familiar with the autocomplete feature, please take a look at <http://en.wikipedia.org/wiki/Autocomplete>) features to your application.

### There's more...

If you would like your field to be able to simulate SQL `LIKE '%ABC'` queries, you should change the `side` attribute of `solr.EdgeNGramFilterFactory` to the `back` value. The configuration should look like this:

```
<filter class="solr.EdgeNGramFilterFactory"minGramSize="1"maxGramSize="25" side="back"/>
```

It would change the end from which Solr starts to analyze the data. In our case, it would start from the end and thus would produce n-grams like: `I`, `OI`, `POI`, `2POI`, `12POI`, and so on.

### See also

If you want to propose another solution for that kind of search, please refer to the recipe *Splitting text by numbers and non-white space characters*.



## Splitting text by numbers and non-white space characters

Analyzing the text data is not only about stemming, removing diacritics (if you are not familiar with the word, please take a look at <http://en.wikipedia.org/wiki/Diacritic>), and choosing the right format for the data. Let's assume that our client wants to be able to search by words and numbers that construct product identifiers. For example, he would like to be able to find the product identifier ABC1234XYZ with the use of ABC, 1234, or XYZ.

### How to do it...

Let's assume that our index consists of three fields (add this to your `schema.xml` file, to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true"/>
<field name="description" type="text_split" indexed="true"
stored="true" />
```

Our `text_split` type should look like this (add this to your `schema.xml` file):

```
<fieldType name="text_split" class="solr.TextField">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
generateNumberParts="1" splitOnNumerics="1" />
<filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
</fieldType>
```

Now you can index your data. Let's create an example data file:

```
<add>
<doc>
<field name="id">1</field>
<field name="name">Test document</field>
<field name="description">ABC1234DEF BL-123_456 adding-documents</
field>
</doc>
</add>
```

After indexing we can test how our data was analyzed. To do that, let's run the following query:

```
http://localhost:8983/solr/select?q=description:1234
```

Solr found our document, which means that our field is working as intended.

## How it works...

We have our index defined as three fields in the `schema.xml` file. We have a unique identifier (an `id` field) indexed as a string. We have a document name (a `name` field) indexed as text (type which is provided with the example deployment of Solr) and a document description (a `description` field), which is based on the `text_split` field which we defined ourselves.

Our type is defined to make the same text analysis both on the query time and the index time. It consists of the whitespace tokenizer and two filters. The first filter is where the magic is done. The `solr.WordDelimiterFilterFactory` behavior, in our case, is defined by these parameters:

- ▶ `generateWordParts`: If set to 1, it tells the filter to generate parts of the word that are connected by non-alphanumeric characters like the dash character. For example, token `ABC-EFG` would be split to `ABC` and `EFG`.
- ▶ `generateNumberParts`: If set to 1, it tells the filter to generate words from numbers connected by non-numeric characters like the dash character. For example, token `123-456` would be split to `123` and `456`.
- ▶ `splitOnNumerics`: If set to 1, it tells the filter to split letters and numbers from each other. This means that token `ABC123` will be split to `ABC` and `123`.

The second filter is responsible for changing the words that lowercase the equivalents and is discussed in the recipe *Lowercasing the whole string* in this chapter.

So after sending our test data to Solr, we can run the example query to see if we defined our filter properly. And you probably know the result; yes the result will contain one document, the one that we send to Solr. That's because the word `ABC1234DEF` is split to `ABC`, `1234`, and `DEF` tokens, and can thus be found by the example query.

## There's more...

In case you would like to preserve the original token that is passed to `solr.WordDelimiterFilterFactory`, add the following attribute to the filter definition:

```
preserveOriginal="1"
```

## See also

If you would like to know more about `solr.WordDelimiterFilterFactory`, please refer to the recipe *Splitting text by camel case* in this chapter.



# 4

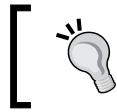
## Solr Administration

In this chapter, we will cover:

- ▶ Monitoring Solr via JMX
- ▶ How to check the cache status
- ▶ How to check how the data type or field behave
- ▶ How to check Solr query handler usage
- ▶ How to check Solr update handler usage
- ▶ How to change Solr instance logging configuration
- ▶ How to check the Java based replication status
- ▶ How to check the script based replication status
- ▶ Setting up Java based index replication
- ▶ Setting up script based replication
- ▶ How to manage a Java based replication status using HTTP commands
- ▶ How to analyze your index structure

### Introduction

Solr administration pages can be very useful in everyday work whenever you need to check your cache status, see how indexes replicate, or check if the type you just created does what it is supposed to do. This chapter will show you to use Solr server instances. It will not only show you the administration pages and what information you can find there, but also guide you through common mistakes made while setting up both Java based and shell script based replications. So please take a seat and see if you can find the information that will interest you. Hope it'll help with your work.



Most of the recipes presented in this chapter are based on a single core deployment of Solr unless stated otherwise.



## Monitoring Solr via JMX

One day your client comes to you and says: "We want to be able to monitor the Solr instance you have set up for us."

"Yes, that's possible" you say, "here is the administration interface and you have access to it."

"No, no, no – we want it to be automatic and secure", says your client.

Can we do it? Of course we can, with the use of Java Management Extension – JMX. This recipe will show you how to set up Solr to use JMX.

### How to do it...

To configure JMX support in Solr, we need to add the following line to the `solrconfig.xml` file:

```
<jmx />
```

Now you can start your example Solr server with the following command:

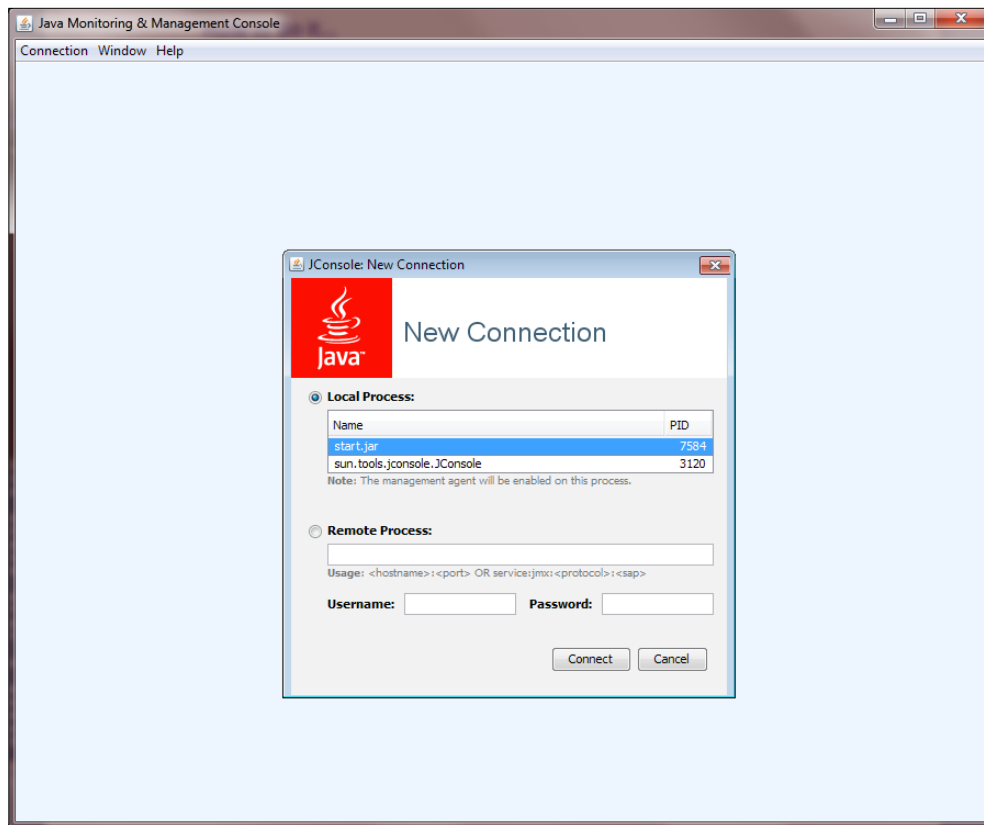
```
java -Dcom.sun.management.jmxremote -jar start.jar
```

And that's all. You should be able to connect to your Solr instance with the use of JMX, for example, with the `jconsole` program which is distributed with Java.

Run the `jconsole` application with the following command:

```
jconsole
```

You should see a list of Java processes you can connect to. Choose `start.jar`.



If no error occurred, you are now connected to Solr MBeans and you can monitor Solr by using it.

## How it works...

First there is a little change in the `solrconfig.xml` file. It enables JMX support in Solr, but only if there is an MBean server already.

After altering the `solrconfig.xml` file, you can run the example Solr deployment with the shown command. This tells JVM that it should enable JMX support for local access only. If Solr started without an error, you can connect to it by using JMX.

One of the simplest ways to do that is to use the `jconsole` application distributed with the Java Development Kit. After running `jconsole` (path `PATH_TO_JDK_INSTALL/bin`), you should see a window which allows you to choose which application to connect to. Choose `start.jar` to connect to the Jetty servlet container running Solr if it is the only Jetty you are running. If not, you must check the process ID (pid) of the Jetty running Solr and choose the right one.

After connecting to Solr, you have control over MBeans. I encourage you to play with it a bit and see what information you can check. There are a plenty of them so I won't mention them all.

### There's more...

There are some other things that are nice to know about dealing with JMX and Solr.

#### Connecting to an existing JMX agent

If you would like Solr to connect to a defined MBean server whose agent identifier is known to you, you can modify the `solrconfig.xml` file to try to connect to that agent. To do that, add the following line to your `solrconfig.xml` file instead of the line shown in the recipe:

```
<jmx agentId="myMBeanServer" />
```

The preceding configuration will tell Solr to connect to an MBean server named `myMBeanServer`.

#### Connecting to an existing MBean server

If you would like Solr to create a new MBean server, replace the line shown in the recipe with the following one:

```
<jmx serviceUrl="service:jmx:rmi:///jndi/rmi://localhost:9999/solr"/>
```

The `serviceUrl` attribute tells Solr to create a new MBean server under the specified JNDI address. If you would like to know more about MBean server creation, please refer to the Java JMX documentation.

#### Running a remote JXM server

If you need your Solr JMX Mbeans to be available at remote machines, you can add the following properties to the start command:

```
-Dcom.sun.management.jmxremote.port=9999  
-Dcom.sun.management.jmxremote.ssl=false  
-Dcom.sun.management.jmxremote.authenticate=false
```

Adding these parameters will enable remote JMX on port 9999 without SSL and without authentication. Remember to be careful with it—everyone can connect if you won't filter connections to that machine.

## How to check the cache status

Cache plays a major role in the Solr instance configuration. However, you should know one thing—cache tuning is not a one-time job. It requires a small amount of time every now and then to check how it behaves and if there is something that we can tune in order to have a better working cache. This recipe will show you how to check the Solr cache status using the Solr administration panel.

### How to do it...

To access the cache pages of Solr admin, you should type the following address in to your web browser:

```
http://localhost:8983/solr/admin/stats.jsp#cache.
```

On that page, you can see five types of Solr caches (queryResultCache, fieldCache, fieldValueCache, documentCache, and filterCache), as shown in the following screenshot:

<b>name:</b>	filterCache
<b>class:</b>	org.apache.solr.search.FastLRUCache
<b>version:</b>	1.0
<b>description:</b>	Concurrent LRU Cache(maxSize=512, initialSize=512, minSize=460, acceptableSize=486, cleanupThread=false)
<b>stats:</b>	lookups: 7 hits: 0 hitratio: 0.00 inserts: 7 evictions: 0 size: 7 warmupTime: 0 cumulative_lookups: 7 cumulative_hits: 0 cumulative_hitratio: 0.00 cumulative_inserts: 7 cumulative_evictions: 0

Now let's see what they mean.

### How it works...

The address of the Solr admin cache pages can be different if you are using multiple cores. If that is the case, you should go to the page defined by your `solr.xml` configuration file. For example, if you have a core named `shop`, your Solr cache pages will probably be available at `http://localhost:8983/solr/shop/admin/stats.jsp#cache`. However, if you are using the single core deployment, then the address shown in the example should guide you to the Solr cache section of the statistics page of Solr admin.

The cache section consists of entries about every type of Solr cache that is defined in the `solrconfig.xml` file. For every cache type there are a few options that show its usage. Those main options are:

- ▶ **name:** Name of the cache type
- ▶ **class:** Implementation class that is used for that type of cache



- ▶ **version:** Cache version
- ▶ **description:** Information about cache configuration
- ▶ **stats:** Usage statistics

The **stats** option is the one that should attract us the most. It tells us how our caches are used, what the hit rates are, and how many evictions there were. We can divide those variables into two sections. The first one describes the cache behavior since the last cache invalidation (for example, the last commit operation), the second describes the cache behavior since the startup. Let's start with the first group and what the parameters mean:

- ▶ **lookups:** This variable tells us how many "questions" were sent to the cache to check if it contains the value that Solr was interested in
- ▶ **hits:** This variable tells us how many of the questions that were sent to the cache resulted in a positive match in the cache
- ▶ **hitratio:** This variable tells us the percentage of hits in all questions sent to the cache
- ▶ **inserts:** This tells us how many values were inserted into the cache
- ▶ **evictions:** This tells us how many deletions were made in the cache
- ▶ **size:** This tells us the actual size of the cache (number of entries)
- ▶ **warmupTime:** This tells us how many milliseconds the last warm up operation took

Now let's take a look at the second group:

- ▶ **cumulative\_lookups:** This gives us the same information as the `lookups` variable, but only since the start of the Solr server
- ▶ **cumulative\_hits:** This variable tells us the same as the `hits` variable, but since the start of Solr server
- ▶ **cumulative\_hitratio:** This variable tells us the same as `hitratio` variable, but since the start of Solr server
- ▶ **cumulative\_inserts:** This variable tells us the same as `inserts` variable, but since the start of Solr server
- ▶ **cumulative\_evictions:** This variable tells us the same as `evictions` variable, but since the start of Solr server

## See also

If you want to know more about cache configuration, please have a look at the *Solr cache configuration* recipe discussed in *Chapter 1, Apache Solr Configuration*.

## How to check how the data type or field behave

Every time we add or modify a field or type, we should check how that field behaves. Of course, we can check the field behavior with a simple query, but this will only show us if we have the field configuration the way we want it or not. Solr allows us to check the behavior of the tokenizer and each of the filters in the Solr administration pages. This recipe will show you how to use that part of the administration pages of Solr.

### How to do it...

We will be using the example `schema.xml` file that is distributed with the example Solr deployment. After starting Solr, go to the following page:

```
http://localhost:8983/solr/admin/analysis.jsp?highlight=on.
```

The address of the Solr admin cache pages can be different if you are using multiple cores. If that is the case, you should go to the page defined by your `solr.xml` configuration file. For example, if you have a core named `shop`, your Solr analysis page will probably be available at `http://localhost:8983/solr/shop/admin/analysis.jsp`.

It will show you the analysis page, which looks similar to the following screenshot:

**Solr Admin (example)**  
192.168.10.211:8983  
cwd=E:\Solr SolrHome=solr/

Apache Solr

### Field Analysis

Field name ▾	
<b>Field value (Index)</b> verbose output <input type="checkbox"/> highlight matches <input checked="" type="checkbox"/>	
<b>Field value (Query)</b> verbose output <input type="checkbox"/>	
<input type="button" value="Analyze"/>	

Now, in the **Field name**, type **description** and check all the checkboxes. In the **Field value (Index)**, type the text: **This is a test index content** and type **test indexes** in the **Field value (Query)** field. It should look similar to the following screenshot:

**Solr Admin (example)**  
192.168.10.211:8983  
cwd=E:\Solr SolrHome=solr/

**Field Analysis**

<b>Field</b> name ▾	description
<b>Field value (Index)</b> verbose output <input checked="" type="checkbox"/> highlight matches <input checked="" type="checkbox"/>	This is a test index content
<b>Field value (Query)</b> verbose output <input checked="" type="checkbox"/>	test indexes
<input type="button" value="Analyze"/>	

Now click on the **Analyze** button. Solr will show us the analysis results, which should look like this:

<b>Index Analyzer</b>						
<b>org.apache.solr.analysis.WhitespaceTokenizerFactory {}</b>						
term position	1	2	3	4	5	6
term text	This	is	a	test	index	content
term type	word	word	word	word	word	word
source start,end	0,4	5,7	8,9	10,14	15,20	21,28
payload						
<b>org.apache.solr.analysis.StopFilterFactory {words=stopwords.txt, ignoreCase=true, enablePositionIncrements=true}</b>						
term position	4	5	6			
term text	test	index	content			
term type	word	word	word			
source start,end	10,14	15,20	21,28			
payload						
<b>org.apache.solr.analysis.WordDelimiterFilterFactory {splitOnCaseChange=1, generateNumberParts=1, catenateWords=1, generateWordParts=1, catenateAll=0, catenateNumbers=1}</b>						
term position	4	5	6			
term text	test	index	content			
term type	word	word	word			
source start,end	10,14	15,20	21,28			
payload						
<b>org.apache.solr.analysis.LowerCaseFilterFactory {}</b>						
term position	4	5	6			
term text	test	index	content			
term type	word	word	word			
source start,end	10,14	15,20	21,28			
payload						
<b>org.apache.solr.analysis.SnowballPorterFilterFactory {protected=protowords.txt, language=English}</b>						
term position	4	5	6			
term text	test	index	content			
term type	word	word	word			
source start,end	10,14	15,20	21,28			
payload						

Now let's look at what Solr told us.

## How it works...

I didn't show the type that I analyzed because it's the standard type that is available in the `schema.xml` file of example that is provided with the standard Solr distribution.

The analysis page of the Solr administration section gives us a few parameters we can type in. First of all, we type the name of the field or the name of the field type that we want to analyze. If you want to check the field behavior, type in the field name you want to check (it is useful because fields can have their own attributes and can differ from types). If you want to analyze the field type, just type in the appropriate name type and change the **name** selection to **field**.

The next section of the analysis page is the index section. We can choose three things here. First, we can choose whether to show the detailed index time analysis (the **verbose output** checkbox) and whether to highlight the matching tokens (the `highlight matches` attribute). The big text field is a place where you can type in the document contents that you want to check—just type in what your test document contents are. The next section of the analysis page is the query section. We can choose two things here. First we can choose to show the detailed query time analysis (the **verbose output** checkbox). The second is the text field, where you should type the query string.

Let's take a look at the analysis output. The analysis of the field is divided into two phases, just like an ordinary search—index time and query time. In the example screenshot, we have the index time analysis. As you can see, it starts with **org.apache.solr.analysis.WhitespaceTokenizerFactory**, which divides the example text into tokens. As you can see there are a few details available with each token. They are:

- ▶ **term position:** The position in the token stream
- ▶ **term text:** The text of the token
- ▶ **term type:** The type of token
- ▶ **source start, end:** The index of the string and ending character of the term in the scope of the token stream (for example, the `test` word starts at position 10 and ends at position 14 of the index contents)
- ▶ **payload:** The payload associated with the token

Next you can see how each filter behaves. For example, you can see that **org.apache.solr.analysis.StopFilterFactory** removed some of the tokens from the token stream, leaving the others untouched.

One thing to notice: you can see the terms with the blue background. Those terms are the ones that got matched with the terms from the query. It is useful when you check how your type will behave with some data in the index and other data in the query.

This is how Solr administration analysis page looks like and works. Use it to test your types and fields—you can save yourself much time when you know how your fields and types will behave.

## How to check Solr query handler usage

Sometimes it is good to know how your query handlers are used: how many requests were made to that handler, how many errors were encountered, and information like that. Of course, you can always implement such behavior in your own application that uses Solr and count those statistics. However, why do that when Solr can give us that information in a simple way? Just go to the appropriate admin page and look at them. This recipe will show you what you can learn about your handlers when they are working.

## How to do it...

You need to open an administration interface of Solr that can be found at the following web page:

```
http://localhost:8983/solr/admin/stats.jsp#query
```

You should see a page with a list of handlers that are defined, which looks similar to the following screenshot:

QUERY HANDLERS	
<b>name:</b>	/admin/plugins
<b>class:</b>	org.apache.solr.handler.admin.PluginInfoHandler
<b>version:</b>	\$Revision: 790580 \$
<b>description:</b>	Registry
<b>stats:</b>	handlerStart : 1289937446041 requests : 0 errors : 0 timeouts : 0 totalTime : 0 avgTimePerRequest : NaN avgRequestsPerSecond : 0.0
<b>name:</b>	/admin/system
<b>class:</b>	org.apache.solr.handler.admin.SystemInfoHandler

Each handler is described by a number of statistics. We can see them in the next screenshot:

<b>name:</b>	dismax
<b>class:</b>	org.apache.solr.handler.component.SearchHandler
<b>version:</b>	\$Revision: 766412 \$
<b>description:</b>	Search using components: org.apache.solr.handler.component.QueryComponent, org.apache.
<b>stats:</b>	handlerStart : 1289937445759 requests : 9 errors : 0 timeouts : 0 totalTime : 146 avgTimePerRequest : 16.222221 avgRequestsPerSecond : 0.0018375451

Now let's see what they mean.

## How it works...

First of all, the address I've shown in the example is only right when we are talking about single core deployment. In multicore deployment, you should look at your `solr.xml` file and see what path you have defined for your core's admin. For example, if you have a core named `shop`, your Solr query handler statistics page will probably be available at `http://localhost:8983/solr/shop/admin/stats.jsp#query`.

In the first screenshot, you see a list of query handlers defined in the `solrconfig.xml` file. Each handler is listed here. Both handlers that start during the startup of Solr are defined as the lazy ones.

The second screenshot shows the actual description of a single handler. I chose the `dismax` handler (that's its name defined by the `name` attribute). Next we have the `class` attribute which tells us which class is responsible for the implementation of that handler. Then we have the `version` attribute, which tells us about the revision number of the SVN repository from which the class originates. Another thing is the `description` attribute, which tells us a bit more about the origin of the handler implementation.

Next is the statistics we are interested in, grouped under the `stats` attribute. So first we have `handlerStats`, which tells us how many milliseconds since the handler initialization elapsed. Next, we have the information about how many requests were made to that handler (the `requests` attribute), how many errors were encountered (the `errors` attribute), and how many requests didn't finish their execution (the `timeouts` attribute). Following that, we have information about the total execution time of request handled by that handler (the `totalTime` attribute), the average time per single request (the `avgTimePerRequest` attribute), and the average request per second made to that handler (the `avgRequestsPerSecond` attribute).

You should remember that all of those statistics are gathered from the startup of Solr. Solr doesn't store that information, so they are discarded when Solr is stopped.

## How to check Solr update handler usage

Knowing how your handlers work is good knowledge. It is always good to know how your update handler, or handlers if you have many of them, behaves. This can help you optimize the indexing process. Solr provides such information using the administration interface. This recipe will show you how to get and interpret this information.

## How to do it...

You need to open an administration interface of Solr that can be found at the following web page:

```
http://localhost:8983/solr/admin/stats.jsp#update
```

You should see a page with a list of handlers that are defined, which looks similar to the following screenshot:

UPDATE HANDLERS	
<b>name:</b>	updateHandler
<b>class:</b>	org.apache.solr.update.DirectUpdateHandler2
<b>version:</b>	1.0
<b>description:</b>	Update handler that efficiently directly updates the on-disk main lucene index
<b>stats:</b>	commits : 1 autocommits : 0 optimizes : 0 rollbacks : 0 expungeDeletes : 0 docsPending : 0 adds : 0 deletesById : 0 deletesByQuery : 0 errors : 0 cumulative_adds : 19 cumulative_deletesById : 0 cumulative_deletesByQuery : 0 cumulative_errors : 0

This page would contain more than one update handler if there were more defined. In the example Solr deployment, we have only one update handler.

Now let's look at what those attributes mean.

## How it works...

First of all the address I've shown in the example are only right when we are talking about single core deployment. In multicore deployment, you should look at your `solr.xml` file and see what path you have defined for your core's admin. For example, if you have a core named `shop`, your Solr update handler statistics page will probably be available at `http://localhost:8983/solr/shop/admin/stats.jsp#update`.

Next, we have some standard attributes defining all the handlers. Those attributes were discussed in the *How to check Solr query handler usage* recipe in this chapter. Please take a look.



Next, we have some information grouped under the `stats` attribute. First we have a `commit` statistic, which tells us how many commit operations were performed since the Solr startup. The same tells us about the `autocommits` statistic, but it tells us about the automatic commits. Following that, we have the `rollbacks` and `optimizes` statistics, which tell us about the number of rollback and optimize operations since the Solr startup. `expungeDeletes` tells us how many merge operations were made after deletions, also since the start of Solr. Next thing we see is the `docsPending` statistic, which says how many documents were added since the last commit, and are still waiting to be committed. Following that, you can see information about how many add operations were performed since the last commit (you must remember that a single add operation can add multiple documents). Next we have the attributes telling us about the delete operations—`deletesById` and `deletesByQuery`, telling how many deletions were done by the unique identifier and how many with the use of query since the last commit. Following that, we have information about how many errors happened since the last commit operation and that tells us about the `errors` statistic.

Finally, we can go to the cumulative statistics section. There are four statistics of that kind: `cumulative_adds` telling about add operations, `cumulative_deletesById` telling about deletions by the unique identifier, `cumulative_deletesByQuery` telling us about deletions by query, and `cumulative_errors` telling us about the happened errors. Those four statistics carry information since the initialization of the handler.

## How to change Solr instance logging configuration

As you may know, Solr is a Java web application that uses SLF4J API for logging. Logs can be useful in many ways, showing things that we are not seeing when Solr is running in the production environment. That's why it is useful to know how you can change the behavior of Solr logging and thus have access to information other than the ones that are logged in/out of the box. This recipe will show you how to do it.


### How to do it...

You need to open an administration interface of Solr that can be found at the following web page:

`http://localhost:8983/solr/admin/logging.jsp`

You should see a page with a list of handlers that are defined, which looks similar to the following screenshot:

## JDK Log Level Selector



Below is the complete JDK Log hierarchy with intermediate logger/categories synthesized. The effective logging level is shown to the far right. If a logger has unset level, then the effective level is that of the nearest ancestor with a level setting. Note that this only shows JDK Log levels.

Logger/Category name	Level										
(Dark rows don't yet exist.)	unset	FINEST	FINE	CONFIG	INFO	WARNING	SEVERE	OFF	Effective		
root	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
global	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>			
org.apache	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>			
org.apache.solr	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>			
org.apache.solr.analysis	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>			
org.apache.solr.analysis.BaseTokenFilterFactory	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.analysis.BaseTokenizerFactory	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.common	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>			
org.apache.solr.common.util	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>			
org.apache.solr.common.util.ConcurrentLRUCache	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.core	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>			
org.apache.solr.core.Config	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.core.CoreContainer	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.core.JmxMonitoredMap	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.core.RequestHandlers	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.core.SolrConfig	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.core.SolrCore	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.core.SolrResourceLoader	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.handler	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>			
org.apache.solr.handler.DocumentAnalysisRequestHandler	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		
org.apache.solr.handler.XmlUpdateRequestHandler	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	INFO		

Now, let's see what we have here.

## How it works...

First of all, the address I've shown in the example is only right when we are talking about single core deployment. In multicore deployment, you should look at your `solr.xml` file and see what path you have defined for your core's admin. For example, if you have a core named `shop`, your Solr logging page will probably be available at `http://localhost:8983/solr/shop/admin/logging.jsp`.

On the logging configuration page of Solr administration, you can see a simple table. The first column of the table informs us about the component—package or a class. The next columns inform us about the available levels of logging—from the most informative to the least informative levels (reading from left to right). The last column tells us what the actual level for the component is (because it can be different from the chosen one).

Every row of the table is responsible for the configuration of a single element. For example, you can see `root` logging set to the `INFO` level. One thing worth noticing is that if the element logging level is set to the `UNSET` value, then it will take the logging level of the nearest configured ancestor. In our example, we have only configured the `root` element, which means that all other elements which are defined (not colored gray), will have the `INFO` logging level.

On the other hand, the elements which are colored gray won't be logged at all, as they don't have any level defined.

To change the logging configuration, just select the desired level for the chosen components and click the `Set` button.

## How to check the Java based replication status

Java based replication is one of those features that was introduced in Solr 1.4. Basically, it allows you to fetch indexes from the master Solr through an HTTP protocol and has one big feature—it's platform independent, so you can use it on any platform that can run Solr. In this recipe, I'll show you how to check the Java based replication status using Solr administration pages.

### Getting ready

This recipe assumes that you have a Java based replication setup.

### How to do it...

You need to open an administration interface of Solr that can be found at the following web page:

```
http://localhost:8983/solr/admin/replication/index.jsp
```

If you are on a master server, you should see a page similar to the following screenshot:

### Solr replication (example) Master

192.168.2.200:8983  
cwd=E:\Solr SolrHome=solr/  
(WHAT IS THIS PAGE?)

Local Index	Index Version: 1289389571397, Generation: 5
	Location: E:\Solr\solr\data\index
	Size: 20,53 KB


Current Time: Wed Nov 17 23:36:48 CET 2010
Server Start At: Wed Nov 17 23:35:06 CET 2010

[RETURN TO ADMIN PAGE](#)

Apache  
**Solr**

On the other hand, if you are on the slave server, you will see more information:

## Solr replication (example) Slave



192.168.2.200:9999  
 cwd=E:\SolrSlave SolrHome=solr/

[\(WHAT IS THIS PAGE?\)](#)

<b>Master</b>	http://localhost:8983/solr/replication
<b>Poll Interval</b>	00:00:60
<b>Local Index</b>	Index Version: 1289389571398, Generation: 6
	Location: E:\SolrSlave\solr\data\index
	Size: 39,38 KB
	Times Replicated Since Startup: 3
	Previous Replication Done At: Wed Nov 17 23:39:00 CET 2010
	Config Files Replicated At: null
	Config Files Replicated: null
	Times Config Files Replicated Since Startup: null
	Next Replication Cycle At: Wed Nov 17 23:40:00 CET 2010
<b>Controls</b>	<div style="border: 1px solid gray; padding: 2px; display: inline-block;">Disable Poll</div>
	<div style="border: 1px solid gray; padding: 2px; display: inline-block;">Replicate Now</div>

Current Time: Wed Nov 17 23:39:20 CET 2010

Server Start At: Wed Nov 17 23:36:00 CET 2010

[RETURN TO ADMIN PAGE](#)

Now let's see what that information means.

## How it works...

First of all, the address I've shown in the example is only right when we are talking about single core deployment. In multicore deployment, you should look at your `solr.xml` file and see what path you have defined for your cores admin. For example, if you have a core named `shop`, your Solr replication status page will probably be available at `http://localhost:8983/solr/shop/admin/replication/index.jsp`.

As you can see, the master server informs us about just a few things. These are: the index version (the timestamp indicating when the index was created) and generation (the number of commit operations made to the index from its creation), the location and the size. The version and generation of the index are increased after every commit and optimize commands.

Things get more interesting when we come to the slave server. Replication pages of the Solr admin running on the slave provide much more information. We can see what the master server is bound to (the `master` row), and what the polling interval is (the `poll interval` row). Next, we have information about the local index that is used by the slave server. We can see the index version and generation, its location, and size. We can see how many times the index was replicated since the server startup and when the last replication took place. We also have information about how many configuration files were replicated, how many times they were replicated, and when the configuration files replication took place. The last information tells us about the time the slave will start the next replication procedure.

You can also see that you can disable the polling process or start replication immediately with the use of the provided buttons.

If there is replication going on, you will also be able to see the speed of it and what file is actually being fetched. You will also be able to cancel the replication process.

Additionally, both master and slave servers provide information about current time in the server JVM locale time zone and the information about the server start time.

## How to check the script based replication status

Script based replication is the replication type that was the first one available in Solr. Basically, it allows you to have one server that is responsible for data indexing and multiple servers responsible for serving that data to the clients. The shell based replication uses the `rsync` command and `ssh` to fetch data from the master server. In this recipe, I'll show you how to check if everything is going as it should.

### Getting ready

This recipe assumes that you have a script based replication setup.

### How to do it...

With script based replication, we are mainly interested in slave servers. However, let's see what information we can find on the master server.

There are two files that can show us how the master is handling the `rsyncd`-enabled file and the `rsyncd.log` file. The second one should contain information like this:

```
2010/11/18 04:10:02 [18832] rsync on solr/snapshot.20101118040701/
from www (127.0.0.1)
2010/11/18 04:10:17 [18832] sent 885331541 bytes received 1860 bytes
total size 9861304921
```

And that is all when talking about the master. So let's look at the slave server. Here we have a few files. First, the empty file `snappuller-enabled`. Next, the `snappuller.status` file with contents like this:

- ▶ `rsync of snapshot.20101118050701 started:20101118-051001 ended:20101118-051006 rsync-elapsed:5`

The next file is the `snapshot.current` file, the contents of which should look similar to this:

```
/var/data/solr/snapshot.20101118041906
```

Now let's look at the files that are holding information about the work of the scripts. The first one, the `snappuller.log` file, the contents of which should look similar to this:

```
2010/11/18 04:10:01 started by root
2010/11/18 04:10:01 command: ./snappuller
2010/11/18 04:10:01 pulling snapshot snapshot.20101118040701
2010/11/18 04:10:23 ended (elapsed time: 22 sec)
2010/11/18 04:25:01 started by root
2010/11/18 04:25:01 command: ./snappuller
2010/11/18 04:25:01 pulling snapshot snapshot.20101118041906
2010/11/18 04:25:13 ended (elapsed time: 12 sec)
```

The last one is the `snapinstaller.log` file, the contents of which are similar to the following:

```
2010/11/18 04:10:23 started by root
2010/11/18 04:10:23 command: ./snapinstaller
2010/11/18 04:10:23 installing snapshot /var/data/solr/
snapshot.20101118040701
2010/11/18 04:10:23 notifying Solr to open a new Searcher
2010/11/18 04:10:35 ended (elapsed time: 12 sec)
2010/11/18 04:25:13 started by root
2010/11/18 04:25:13 command: ./snapinstaller
2010/11/18 04:25:13 installing snapshot /var/data/solr/
snapshot.20101118041906
2010/11/18 04:25:13 notifying Solr to open a new Searcher
2010/11/18 04:25:27 ended (elapsed time: 14 sec)
```

Now let's see what that data means.

## How it works...

So what can the master Solr server tell us? First of all, the empty `rsyncd-enabled` file tells us that the `rsyncd` daemon is running. Without that file, you can be sure that the master won't respond to the slave. The `rsyncd.log` file tells us about the flow of actual data—when it happened, what server (name and IP address) fetched the data, and how much data was sent.

Now let's look at the slave server. First the empty file `snappuller-enabled` tells us if the `snappuller` script is enabled. Without it, the replication won't work.

Next we have a file (`snapshot.current`), the content of which tells us about the latest snapshot that was pulled from the master. As you can see, we see not only the name of the snapshot, but also the start and end time, and how many seconds the `rsync` operation lasted.

The `snapshot.current` file tells us about the latest snapshot that is installed on slave.

Next, we have two log files that inform us about the operations that the following two scripts are performing, `snappuller` and `snapinstaller`. The first log tells us about the user who started the script, what command was run, what snapshot was pulled from the master, and how the pulling process ended. The second log file is similar, but tells us about the process of installing the pulled snapshot. These two can also hold information about the failures with a bit of information about the error cause.

## Setting up a Java based index replication

One day your client comes to your office and says, "We want to get rid of the `ssh` daemon from all of our machines and of course we don't want the `rsync` daemon either – they are evil!". When leaving your office, your client mentions that synonyms and stop words should be replicated too. But hey, you have a script based replication running. In a panic, you think what the other options are. Are there any? Yes, there is a Java based replication. With Solr 1.4, you have an option either to use scripts to replicate your indexes or to use the replication handler. This recipe will show you how to use the replication handler, and thus how to set up Java based replication.

## How to do it...

I assumed that we have two example Solr instances, master and slave, both running on the same machine. Master Solr is running at the 8983 port, while the Solr slave server is running at the 9999 port.

First let's take care of the master server. Add the following handler definition to the master server `solrconfig.xml` file:

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="replicateAfter">commit</str>
    <str name="replicateAfter">startup</str>
    <str name="replicateAfter">optimize</str>
    <str name="confFiles">synonyms.txt, stopwords.txt</str>
  </lst>
</requestHandler>
```

Now add the following to the slave server `solrconfig.xml` file:

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="slave">
    <str name="masterUrl">http://localhost:8983/solr/replication</str>
    <str name="pollInterval">00:00:60</str>
  </lst>
</requestHandler>
```

And that's all that needs to be done. Simple right? Now let's look at what the added configuration means.

## How it works...

Maybe it is time for some insights on how the replication works. Master Solr is the server where the data is indexed. After the commit (or optimize) operation takes place, the so called index generation is changed and the index is ready to be replicated. Slave servers query to the master about the index generation. If slave is out of sync from the master, then it fetches the new index and updates itself. That's how it works in a telegraphic shortcut. Now let's see what the configuration entries mean.

Both the master and slave servers must have a new handler defined—`solr.ReplicationHandler`. The master configuration options are grouped by the list with the `name="master"` attribute. On the other hand, the slave configuration options are grouped by the list with the attribute `name="slave"`.

There are two configuration values in the master Solr configuration. The first one is a multivalued one—can be specified multiple times—`replicateAfter`. It tells the Solr master when to replicate the data. The values shown in the example tell Solr to enable replication of the index after operations such as commit, optimization, and after server startup. These are the only options. The `confFiles` configuration variable tells Solr master which configuration files should be replicated with the index. The value should be a list separated by commas, of the files that can be visible to Solr.



Next, we have the slave server configuration. First, we have the `masterUrl` variable, which tells Solr where the master server is. Remember, slave can be connected to only one master. The next configuration variable is `pollInterval`, which tells us when to poll the master server for a new index version. Its format is `HH:mm:ss`. If you won't define this, then automatic polling will be disabled, but you'll always be able to start replication from the administration pages.

### There's more...

There are two things I would like to mention when talking about replication.

#### Slave and HTTP Basic authorization

If your master is configured to validate users with HTTP Basic, then you must ensure that your slave servers will have the proper information. Add the following to the slave replication handler configuration:

```
<str name="httpBasicAuthUser">username</str>
<str name="httpBasicAuthPassword">password</str>
```

This will tell the slave server to authenticate a `username` user with a `password` password.

#### Changing the configuration file names when replicating

You can change the configuration filename when replicating indexes. For example, you want to replicate the `solrconfig_slave.xml` file from the master and place it on your slave under the `solrconfig.xml` name. To do this, change the line specifying the configuration file to the one like this:

```
<str name="confFiles">solrconfig_slave.xml:solrconfig.xml</str>
```

The preceding line of code will tell Solr to fetch the `solrconfig_slave.xml` file from the master server and place it under the `solrconfig.xml` name on the slave server.

### See also

If you want to know how to check the Java based replication status, please refer to the *How to check a Java based replication status* recipe in this chapter.

## Setting up script based replication

Imagine a situation like this: you have a large index which is only getting incremental updates with once a week optimization. The data updates are not frequent but the index is getting too large to fetch every time. You have to think about something like incremental updates or something like that. Can we do that with the standard Solr tool? Sure we can, we need to use script based replication. This recipe will show you how to do that.

## Getting ready

You need to ensure that the slave can connect to the master through the `ssh` and `rsync`. Personally, I prefer key based authorization between the slaves and master. However, as that's not a subject of this book, I'll pass on this part of the topic.

## How to do it...

You must be aware of one thing—script based replication will only work on Unix and Linux based operating systems because of the bash shell usage and command that are not standard Microsoft Windows operating systems commands (unless you use `Cygwin`, for example).

Let's start by enabling the `rsync` daemon. To do that, run the following commands from the `bin` directory of your Solr installation:

```
rsyncd-enable
```

and

```
rsyncd-start
```

Now add the following listener to the master server update handler definition in the `solrconfig.xml` file:

```
<listener event="postCommit" class="solr.RunExecutableListener">
  <str name="exe">/solr/bin/snapshooter</str>
  <bool name="wait">true</bool>
  <arr name="args"></arr>
  <arr name="env"></arr>
</listener>
```

Now let's do the slave side configuration. First of all, we need to enable the `snappuller` script. To do that, run the following command:

```
snappuller-enable
```

Now, in your `conf` directory, create a file named `scripts.conf` with the following content:

```
solr_port=8983
rsyncd_port=18280
data_dir=/solrslave/data/
master_host=192.168.2.2
master_data_dir=/solr/data/
master_status_dir=/solr/logs/slaves-clients
webapp_name=solr
```

We are almost done. We need to add the pulling and installing scripts to the system's `cron` table to run them periodically. For example, add the following configuration to the slave `cron` table:

```
0,30 * * * * /solr/bin/snappuller;/solr/bin/snapinstaller
```

And that's all. If you want to know how to check if the replication is running, please refer to the *How to check script based replication status* recipe in this chapter.

## How it works...

I assumed that the master and slave are running on separate machines and that the slave can connect to the master through `ssh` with the use of a key and without the need of a password.

So let's start with the master. First, we enable the `rsync` daemon and then we start it. On the other hand, we add that listener definition to the `solrconfig.xml` file into the update handler section. It tells Solr to run the executable defined under the `exe` configuration attribute. The `wait` attribute tells Solr to wait for the end of the executable execution (actually not for Solr to wait, but for the thread that run the executable). The `args` variable specifies the argument that is needed to be passed to the executable, we don't want to pass anything here. And the last, `env` variable, tells us about the environment variables that are needed to be set, also empty in our case.

Basically, when the `commit` command is called on the master Solr server, a new snapshot will be generated—a directory containing the difference between the old index and the new index (the one before and after the `commit` operation). That's the directory that will be pulled by the slaves.

So now let's look at slave's configuration. The first thing we do is enable the `snappuller` script, the one that is responsible for pulling snapshots from the master. The command shown in the example will create a single, empty file named `snappuller-enabled`, which will enable the `snappuller` script to run.

The `scripts.conf` file is responsible for holding the configuration information for the scripts. As you see in the example, there are a few things defined there. The port that we can find the slave server on which the scripts run is defined by the `solr_port` variable. Following that, we have the master `rsync` daemon port, defined by the `rsyncd_port` variable. Next, `data_dir` tells the script where the slave data directory is located. `master_host` holds information about the IP address (or the name) of the master server. The `master_data_dir` variable tells us where the index is located on the master's machine. Next, the `master_status_dir` variable tells us about the location of the file with the actual status of replication. The last variable, `webapp_name`, specifies the context under which Solr slave can be found.

The `cron` command is simple, it tells the operating system to run `snappuller` and `snapinstaller` after every one and a half hours.

One more thing to remember: the time on the machines should match because the snapshots are generated on the basis of the following format: `snapshot.yyyymmddHHMMSS`. If your time does not match, then the scripts can be fooled and won't be able to fetch and install the latest snapshots.

That is how the master and each of the slaves should be configured.

### See also

If you want to know how to check the script based replication status, please refer to the *How to check a script based replication status* recipe in this chapter.

Full information about parameter that can be passed to distribution scripts can be found on Solr wiki: <http://wiki.apache.org/solr/CollectionDistribution>.

## How to manage Java based replication status using HTTP commands

You were drinking coffee while watching the ASCII mode YouTube videos and setting up Solr instances. You thought that it will be another quiet day at work. And then when you ran your Solr instances for the first time, something happened—replication is not working. Where to begin? How to check it? I know, administration pages, but you only have curl on your operating system and nothing more. What to do? It's simple, don't panic, this recipe will show you how to manage Solr replication using HTTP commands.

### Getting ready

I assume that you already have replication configured. If you don't, please take a look at the *Setting up Java based replication* recipe in this chapter.

### How to do it...

Let's see what replication handler can say about itself. First, let's enable index polling on the slave. To do this, we run the following command:

```
curl http://slave:8983/solr/replication?command=enablepoll
```

The response should be like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">4</int></lst>
```

```
<str name="status">OK</str>
</response>
```

So we enabled polling, but still slave doesn't want to fetch the master index. Let's check the slave configuration details. To do that, run the following command:

```
curl http://slave:8983/solr/replication?command=details
```

The response should be like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">170</int>
  </lst>
  <lst name="details">
    <str name="indexSize">58,23 KB</str>
    <str name="indexPath">E:\SolrSlave\solr\data\index</str>
    <arr name="commits"/>
    <str name="isMaster">false</str>
    <str name="isSlave">true</str>
    <long name="indexVersion">1289389571399</long>
    <long name="generation">7</long>
    <lst name="slave">
      <lst name="masterDetails">
        <str name="indexSize">58,23 KB</str>
        <str name="indexPath">E:\Solr\solr\data\index</str>
        <arr name="commits">
          <lst>
            <long name="indexVersion">1289389571399</long>
            <long name="generation">7</long>
            <arr name="filelist">
              <str>_0.tis</str>
              (...)
              <str>_3.tis</str>
            </arr>
          </lst>
        </arr>
        <str name="isMaster">true</str>
        <str name="isSlave">false</str>
        <long name="indexVersion">1289389571399</long>
        <long name="generation">7</long>
      </lst>
      <str name="masterUrl">http://localhost:8983/solr/replication</
str>
    <str name="isPollingDisabled">false</str>
    <str name="isReplicating">false</str>
```

```

    </lst>
  </lst>
  <str name="WARNING">This response format is experimental. It is
likely to change in the future.</str>
</response>

```

Hmmm, too much, too much, but hey, I see an index version. Let's check the index version on the master. The following command will be good:

```
curl http://master:8983/solr/replication?command=indexversion
```

And the response is:

```

<response>
<lst name="responseHeader"><int name="status">0</int><int
name="QTime">0</int></lst><long name="indexversion">1289389571399</
long><long name="generation">7</long>
</response>

```

Hmmm, they are the same. However, for the sake of my sanity, let's force replication.

```
curl http://slave:8983/solr/replication?command=fetchindex
```

And again, the response:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader"><int name="status">0</int><int
name="QTime">1</int></lst><str name="status">OK</str>
</response>

```

Seems like there was nothing to worry about.

## How it works...

So let's see what those commands mean and do. As you may already notice, all commands sent to the replication handler are passed as values of the `command` parameter.

So, first we have the `enablepoll` command, which enables index polling by the slave server. This command can be useful when upgrading Solr, for example, or changing the data structure, and so on. You can disable the poll with the `disablepoll` command, do your work, and then enable the replication again. The response generated by Solr tells us that everything went well by showing the `OK` status.

The next command that we run is the `details` command telling Solr to show us details of the configuration and index on the machine that we run it on (yes, we can run it on the master and slave). As you can see in the response, there is much information about the deployment. There is information about the index (like size, version, location, generation, or files that are present in the index directory), master, about the actual replication status, and about polling (enabled/disabled).

Although we can get the index version with the `details` command, it also carries much other, not always useful, information. That's why the `indexversion` command was introduced. It enables us to get information about the index version from the master server. As the response shows, it returns the index version and the index generation. In our case, they are the same on the master as on the slave.

As we see no mistakes in our configuration, we run the `fetchindex` command (on the Solr slave server) to force index fetching. If everything went well, the `OK` status will be returned by Solr as shown in the example. Remember one thing—the `OK` status is not telling us that the index was fetched; it just tells us that the command was processed correctly. In our case, the index version and generation were the same on slave and master and thus the `fetchindex` command didn't force replication. That right, Solr won't replicate the index. If the master and slave index generation are identical, Solr won't fetch the index, no matter what.

### There's more...

There are a few commands that may be useful when controlling Java based replication with HTTP commands. They are:

#### Aborting index fetching

To abort index, copy operation just sends the following command to the Solr slave server:

```
http://slave:8983/solr/replication?command=abortfetch
```

#### Disabling replication

To disable replication for all slave servers, just run the following command on the master Solr server:

```
http://master:8983/solr/replication?command=disablereplication
```

#### Enabling replication

To enable replication for all slave servers, just run the following command on the master Solr server:

```
http://slave:8983/solr/replication?command=enablereplication
```

## How to analyze your index structure

Imagine a situation where you need some information about the index. Not only the ones that can be found in the `schema.xml` file. You would like to know the top terms for a given field or the number of terms in the whole index. Can we do it with the Solr administration panel? The answer is yes, we can and this recipe will show you how to do it.

## How to do it...

You need to open an administration interface of Solr that can be found at the following web page:

`http://localhost:8983/solr/admin/schema.jsp`

You should see a page like this:



The screenshot shows the Solr Admin interface. At the top left, it says "Solr Admin (example)" followed by the IP address "192.168.2.200:8983" and the path "cwd=E:\solr SolrHome=solr/". On the top right is the Apache Solr logo. Below the header, there is a navigation bar with "Schema Browser" and a link "See RAW SCHEMA.XML". A sidebar on the left contains links: "HOME", "FIELDS", "DYNAMIC FIELDS", and "FIELD TYPES". The main content area is titled "Schema Information" and lists various configuration details:

- Unique Key: [ID](#)
- Default Search Field: [TEXT](#)
- numDocs: 20
- maxDoc: 60
- numTerms: 943
- version: 1289389571399
- optimized: false
- current: true
- hasDeletions: true
- directory: org.apache.lucene.store.SimpleFSDirectory:org.apache.lucene.store.SimpleFSDirectory@E:\Solr\solr\data\index
- lastModified: 2010-11-20T11:21:17.103Z



Some nice information we have there. Let's see what can Solr tell us about the **name** field. To do that click **FIELDS** from the left menu and then choose the **name** field. You should see a page similar to the one in the following screenshot:

**Solr Admin (example)**  
 192.168.2.200:8983  
 cwd=E:\solr SolrHome=solr/

**Schema Browser** | See [RAW SCHEMA.XML](#)

**Field: name**

Field Type: **TEXTGEN**

Properties: Indexed, Tokenized, Stored

Schema: Indexed, Tokenized, Stored

Index: Indexed, Tokenized, Stored

Copied Into: **TEXT**

Position Increment Gap: 100

Index Analyzer: org.apache.solr.analysis.TokenizerChain [DETAILS](#)

Query Analyzer: org.apache.solr.analysis.TokenizerChain [DETAILS](#)

Docs: 20

Distinct: 126

**Top 10 Terms**

term	frequency
gb	27
memory	18
ddr	18
pc	18
184	18
pin	18
1	18
3200	18
400	18
sdram	18

**Histogram**

Bin	Count
1	0
2	13
4	77
8	18
16	6
32	12

This is the information that we can see, now let's see what that information means.

## How it works...

First of all, the address I've shown in the example is only right when we are talking about single core deployment. In multicore deployment, you should look at your `solr.xml` file and see what path you have defined for your core's admin. For example, if you have a core named `shop`, your Solr index structure analysis page will probably be available at `http://localhost:8983/solr/shop/admin/schema.jsp`.

Once you open the schema analysis page, you can see some useful information there. We start with the unique key, default search field, number of documents, maximum number of documents that were in the index, and the number of terms in the index. Following that, we have the version, attribute telling about optimization of the index (optimized/not optimized) and if the index consists of files to delete, and of course the last modification date.

Then we've chosen the **name** field. In the screenshot presenting the page, you can see that there is some information that can't be found in the `schema.xml` file. The page starts with some basic information such as the field type and its properties: the ones defined in the type are shown under **Properties**, the ones that are defined in the field are marked **Schema**, and the ones actually indexed are shown under **Index**. One thing comes into mind—can schema be different from the index? Yes, it can. For example, when we upgrade Solr and other operations like that.

Next, Solr tells us where the data from that field is copied to and about the position increment gap of the field. There is also information about the analyzers, both index and query. To see the details, click the **Details** button. I'll skip commenting on that.

Now comes the real interesting part. First, we have the number of documents that have that field defined (not empty), the **Docs** attribute. Next, is the number of unique terms in the field (the **Distinct** attribute).

Next is the N **Top Terms** (default N is **10**, but you can specify the value that suits you the most). On the right side of the top terms is the **Histogram** field, which shows the number of documents with the highest number of terms. For example, you can see that there are 77 documents with the four distinct terms.

Of course, there are many things more than you can check with this administration page—if you would like to see and discover them, you should check this page.



# 5

## Querying Solr

In this chapter, we will cover:

- ▶ Asking for a particular field value
- ▶ Sorting results by a field value
- ▶ Choosing a different query parser
- ▶ How to search for a phrase, not a single word
- ▶ Boosting phrases over words
- ▶ Positioning some documents over others on a query
- ▶ Positioning documents with words closer to each other first
- ▶ Sorting results by a distance from a point
- ▶ Getting documents with only a partial match
- ▶ Affecting scoring with functions
- ▶ Nesting queries

### Introduction

Making a simple query is not a hard task, but making a complex one with faceting, local params, parameters dereferencing, and phrase queries can be a challenging task. On top of all that, you must remember to write your query with some performance factors in mind. That's why something that seems simple at first sight can turn into something more challenging like writing a good, complex query. This chapter will try to guide you through some of the tasks you may encounter during your everyday work with Solr.

## Asking for a particular field value

There are many cases when you might want to ask for a particular field value, for example, when searching for the author of a book on the Internet library or e-commerce shop. Of course, Solr can do that and this recipe will show you how to do it.

### How to do it...

Let's assume we have the following index structure (just add the following to the field definition section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
<field name="author" type="string" indexed="true" stored="true"/>
```

To ask for a value in the author field, send the following query to Solr:

```
http://localhost:8983/solr/select?q=author:rafal
```

And that's all! The documents you'll get from Solr will be the ones that have the given value in the `author` field. Remember that the query shown in the example is using standard query parser, not `dismax`.

### How it works...

We defined three fields in the index, but it's only for the purpose of the example. As you can see in the preceding query to ask for a particular field value, you need to send a `q` parameter syntax such as `FIELD_NAME:VALUE` and that's it. Of course, you can add the logical operator to the query to make it more complex. Remember that if you omit the field name from the query, your values will be checked again in the default search field that is defined in the `schema.xml` file.

### There's more...

When asking for a particular field value, there are things that are useful to know.

#### Querying for a particular value using `dismax` query parser

You may sometimes need to ask for a particular field value when using the `dismax` query parser. Unfortunately the `dismax` query parser doesn't support full Lucene query syntax and thus you can't send a query like that; however, there is a solution. You can use the extended `dismax` query parser, which is an evolved `dismax` query parser. It has the same list of functionalities as the `dismax` query parser and it also supports full Lucene query syntax. The query shown in this recipe, but using `edismax`, would look like this:

```
http://localhost:8983/solr/select?q=author:rafal&defType=edismax
```

### Querying for multiple values in the same field

You may sometimes need to ask for multiple values in a single field. For example, let's suppose that you want to find the `solr` and `cookbook` values in the `title` field. To do that, you should run the following query (notice the brackets surrounding the values):

```
http://localhost:8983/solr/select?q=author:(solr cookbook)
```

## Sorting results by a field value

Imagine an e-commerce site where you can't choose the sorting order of the results, you can only browse the search results page-by-page and nothing more. That's terrible, right? Yes and that's why with Solr you can specify the sort fields and the order in which your search results should be sorted. This recipe will show you how to do it.

### How to do it...

Let's assume that you want to sort your data by additional field—the field that contains the author of the book. So first, add the following to your `schema.xml` file field section:

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
<field name="author" type="string" indexed="true" stored="true"/>
```

Now, the data file can look like this:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="title">Solr cookbook</field>
  <field name="author">Rafal Kuc</field>
</doc>
<doc>
  <field name="id">2</field>
  <field name="title">Solr results</field>
  <field name="author">John Doe</field>
</doc>
<doc>
  <field name="id">3</field>
  <field name="title">Solr perfect search</field>
  <field name="author">John Doe</field>
</doc>
</add>
```

As I wrote earlier, we want to sort the result list by the author in ascending order. Additionally, we want the books that have the same author to be sorted by relevance in descending order. To do that, we must send the following query to Solr:

```
http://localhost:8983/solr/select?q=solr&sort=author+asc,score+desc
```

The results returned by Solr should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
    <lst name="params">
      <str name="q">solr</str>
      <str name="sort">author asc,score desc</str>
    </lst>
  </lst>
  <result name="response" numFound="3" start="0">
    <doc>
      <str name="author">John Doe</str>
      <str name="id">2</str>
      <str name="title">Solr results</str>
    </doc>
    <doc>
      <str name="author">John Doe</str>
      <str name="id">3</str>
      <str name="title">Solr perfect search</str>
    </doc>
    <doc>
      <str name="author">Rafal Kuc</str>
      <str name="id">1</str>
      <str name="title">Solr cookbook</str>
    </doc>
  </result>
</response>
```

As you can see, our data is sorted exactly as we wanted.

## How it works...

As you can see, I defined three fields in our index. The most interesting one to us is the `author` field on the basis of which we will perform the sorting operation. Notice one thing, the type on which the field is based is the `string` type. If you want to sort on a type other than the default `score` fields, please prepare your data well—use the appropriate number types (the ones based on the `Trie` types). To sort on the text field, use the `string` field type (or text type using the `KeywordTokenizer` and lowercase filter).

Following that, you see the data which is very simple—it only adds three documents to the index.

I've added one additional parameter to the query that was sent to Solr, the `sort` parameter. This parameter defines the sort field with order. Each field must be followed by the order in which the data should be sorted—`asc`, which tells Solr to sort in the ascending order and `desc`, which tells Solr to sort in the descending order. Pairs of field and order should be delimited with the comma character, as shown in the example.

The result list that Solr returned is telling us that we did a perfect job on defining the sort order.

## Choosing a different query parser

There are situations where you need to choose a different query parser than the one that you are using. For example, imagine a situation when some of your queries are run in a way where different words go to different fields, while the others require `mm` parameter to work properly. The first query would require the standard Lucene query parser, while the second one would require the use of the `dismax` query parser. Of course, this is just an example. You should be aware that there is not only one query parser and you can choose which to use during query time. This recipe will show you how to do it.

## How to do it...

Choosing a query parser is very simple, you just need to pass the appropriate parameter to the query you send to Solr. For example, let's choose the `dismax` query parser instead of the standard one. To do that, add the following parameter to the query:

```
defType=dismax
```

The actual query would look like this:

```
http://localhost:8983/solr/select?q=book&qf=title&defType=dismax
```



## How it works...

The `defType` parameter specifies the name of the query parsers that Solr should use, either the standard ones or the ones created by you. The standard query parsers that are available to Solr are:

- ▶ `dismax`: The `dismax` query parser
- ▶ `edismax`: The extended `dismax` query parser
- ▶ The standard query parser

Yes it's that simple, nothing more, nothing less.

## How to search for a phrase, not a single word

Imagine that you have an application that searches within millions of documents that are generated by a law company. One of the requirements is to search the titles of the documents as a phrase, but with stemming and lowercasing, so string-based field is not an option. So is it possible to achieve using Solr? Yes, and this recipe will show you how.

## How to do it...

Assume that we have the following type defined (add this part to your `schema.xml` file):

```
<fieldType name="text" class="solr.TextField"
positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.SnowballPorterFilterFactory"
language="English"/>
  </analyzer>
</fieldType>
```

And the following fields to your `schema.xml` file:

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
```

Our data looks like this:

```
<add>
<doc>
```

```

    <field name="id">1</field>
    <field name="title">2010 report</field>
  </doc>
</doc>
  <field name="id">2</field>
  <field name="title">2009 report</field>
</doc>
</doc>
  <field name="id">3</field>
  <field name="title">2010 draft report</field>
</doc>
</add>

```

Now let's try to find the documents that have a 2010 report phrase. To do that, make the following query to Solr:

```
http://localhost:8983/solr/select?q=title:"2010 report"
```

The result should be like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="q">title:"2010 report"</str>
    </lst>
  </lst>
  <result name="response" numFound="1" start="0">
    <doc>
      <str name="id">1</str>
      <str name="title">2010 report</str>
    </doc>
  </result>
</response>

```

The debug query (debugQuery=on parameter) shows us what Lucene query was made:

```
<str name="parsedquery">PhraseQuery(title:"2010 report")</str>
```

As you can see, we only got one document which is perfectly good. Now let's see why that happened.

## How it works...

As I said in the introduction, our requirement was to phrase searching over field that is stemmed and lowercased. If you want to know more about stemming and lowercasing, please have a look at *Stemming your data* and *Lowercasing the whole string* recipes discussed in *Chapter 3, Analyzing your Text Data*.

We only need two fields because we will only search the title, and return the title and unique identifier of the field. Thus the configuration is as shown in the example.

The example data is quite simple, so I'll skip commenting on it.

The query is something that we should be more interested in. The query is made to the standard Solr query parser, thus we can specify the field name and the value we are looking for. The query differs from the standard word-searching query by the use of the " character at the start and end of the query. It tells Solr to use the phrase query instead of the term query. Using the phrase query means that Solr will search for the whole phrase, not a single word. That's why only the document with identifier 1 was found, because the third document did not match the phrase.

The debug query only ensured that the phrase query was made instead of the usual term query and Solr showed us that we made the query right.

## There's more...

When using queries, there is one thing that is very useful to know.

### Defining the distance between words in a phrase

You may sometimes need to find documents that match the phrase, but are separated by some other words. Let's assume that you would like to find the first and third documents in our example. This means that you want documents that can have an additional word between the words `2010` and `report`. To do that, we add so-called phrase slop to the phrase. In our case, the distance (slop) between words can be a maximum of one word, so we add the `~1` part after the phrase definition.

```
http://localhost:8983/solr/select?q=title:"2010 report"~1
```

## Boosting phrases over words

Imagine that you are a search expert at a leading e-commerce shop in your region. One day, disaster strikes—your marketing department says that the search results are not good enough. They would like you to favor documents that have the exact phrase typed by the user over the documents that have matches in separate words. Can you do it? Of course you can, and this recipe will show you how to achieve that.

## Getting ready

Before you start reading this recipe, I suggest reading the *How to search for a phrases not a single word* recipe in this chapter. It will allow you to understand the recipe better.

## How to do it...

I assume that we will be using the `dismax` query parser, not the standard one. We will also use the same `schema.xml` that was shown in the *How to search for a phrases not a single word* recipe in this chapter.

Our sample data file looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Annual 2010 report last draft</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">2009 report</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">2010 draft report</field>
  </doc>
</add>
```

As I already mentioned, we would like to boost those documents that have phrase matches over others matching the query. To do that, run the following query to your Solr instance:

```
http://localhost:8983/solr/select? defType=dismax&pf=title^100&q=2010
+report&qf=title
```

You should get the following response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="qf">title</str>
      <str name="pf">title^100</str>
      <str name="q">2010 report</str>
      <str name="defType">dismax</str>
```

```
</lst>
</lst>
<result name="response" numFound="2" start="0">
  <doc>
    <str name="id">1</str>
    <str name="title">Annual 2010 report last draft</str>
  </doc>
  <doc>
    <str name="id">3</str>
    <str name="title">2010 draft report</str>
  </doc>
</result>
</response>
```

To visualize the results better, I decided to include the results returned by Solr for the same query, but without adding the pf parameter:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="qf">title</str>
      <str name="q">2010 AND report</str>
      <str name="defType">dismax</str>
    </lst>
  </lst>
  <result name="response" numFound="2" start="0">
    <doc>
      <str name="id">3</str>
      <str name="title">2010 draft report</str>
    </doc>
    <doc>
      <str name="id">1</str>
      <str name="title">Annual 2010 report last draft</str>
    </doc>
  </result>
</response>
```

As you can see, we fulfilled our requirement.

## How it works...

Some of the parameters that are present in the example query can be new to you. The first parameter is `defType` which tells Solr which query parser we will be using. In this example, we will be using the `dismax` query parser (if you are not familiar with the `dismax` query parser, please take a look at the following address <http://wiki.apache.org/solr/DisMax>). One of the features of this query parser is the ability to tell which field should be used to search for phrases—we do that by adding the `pf` parameter. The `pf` parameter takes a list of fields with the boost that corresponds to them, for example, `pf=title^100` which means that the phrase found in the `title` field will be boosted with a value of 100. The `q` parameter is the standard query parameter that you are familiar with. This time, we passed the words we are searching for and the logical operator `AND`. That query means that we are looking for documents which contain the words `2010` and `report`. You should remember that you can't pass a query such as `fieldname:value` to the `q` parameter and use `dismax` query parser. The fields you are searching against should be specified using the `qf` parameter. In our case, we told Solr that we will be searching against the `title` field.

The results show us that we found two documents. The one that matches the exact query is returned first and that is what we were looking to achieve.

## There's more...

There is one more thing I would like you to know.

### Boosting phrases with standard query parser

You can, of course, boost phrases with standard query parsers, but that's not as elegant as the `dismax` query parser method. To achieve similar results, you should run the following query to your Solr instance:

```
http://localhost:8983/solr/select?q=title:(2010+AND+report)+OR+title:"
2010+report"^100
```

The preceding query tells Solr to search for the words `2010` and `report` in a `title` field and search for a `2010 report` phrase, and if found, then boost that phrase with the value of 100.

## Positioning some documents over others on a query

Imagine a situation when your client tells you that he wants to promote some of his products by placing them at the top of the search result list. Additionally, he would like the products list to be flexible, he would like to be able to define the list for some queries and not for others. Many thoughts come to your mind, such as boosting, index time boosting, or maybe some special field to achieve that. However, don't bother—Solr can help you with the recipe with a component that is known as `QueryElevationComponent`.

## How to do it...

First of all, let's modify the `solrconfig.xml` document. We need to add the component definition. To do that, add the following section to your `solrconfig.xml` file:

```
<searchComponent name="elevator" class="solr.QueryElevationComponent"
>
  <str name="queryFieldType">string</str>
  <str name="config-file">elevate.xml</str>
</searchComponent>
```

Now let's add the proper request handler that will include the elevation component. We will name it `/promotion`. Add this to your `solrconfig.xml` file:

```
<requestHandler name="/promotion" class="solr.SearchHandler">
  <arr name="last-components">
    <str>elevator</str>
  </arr>
</requestHandler>
```

You may notice that the query elevation component contained information about a mysterious `elevate.xml` file. Let's assume that we want the documents with identifiers 3 and 1 to be on the first two places in the results list for the `solr` query. For now, you need to create that file in the configuration directory of your Solr instance and paste the following contents:

```
<?xml version="1.0" encoding="UTF-8" ?>
<elevate>
  <query text="solr">
    <doc id="3" />
    <doc id="1" />
  </query>
</elevate>
```

Now the `schema.xml` file. Our field definition part of it is as follows:

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
```

I have indexed the following data file:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Solr cookbook</field>
  </doc>
</add>
```

```

    <field name="id">2</field>
    <field name="name">Solr master pieces</field>
  </doc>
</doc>
  <field name="id">3</field>
  <field name="name">Solr annual report</field>
</doc>
</add>

```

Now we can run Solr and test our configuration. To do that, let's run the following query:

```
http://localhost:8983/solr/promotion?q=solr
```

The preceding query should return the following result:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="q">solr</str>
    </lst>
  </lst>
  <result name="response" numFound="3" start="0">
    <doc>
      <str name="id">3</str>
      <str name="name">Solr annual report</str>
    </doc>
    <doc>
      <str name="id">1</str>
      <str name="name">Solr cookbook</str>
    </doc>
    <doc>
      <str name="id">2</str>
      <str name="name">Solr master pieces</str>
    </doc>
  </result>
</response>

```

The query without using the elevation component returned the following result:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>

```



```
<int name="QTime">1</int>
<lst name="params">
  <str name="q">solr</str>
</lst>
</lst>
<result name="response" numFound="3" start="0">
  <doc>
    <str name="id">1</str>
    <str name="name">Solr cookbook</str>
  </doc>
  <doc>
    <str name="id">2</str>
    <str name="name">Solr master pieces</str>
  </doc>
  <doc>
    <str name="id">3</str>
    <str name="name">Solr annual report</str>
  </doc>
</result>
</response>
```

As you can see, the component worked. Now let's see how it works.

### How it works...

The first part of the configuration defines a new search component with a name under which the component will be visible to other components and the search handler (name attribute). In our case, the component name is `elevator` and it's based on the `solr.QueryElevationComponent` class (the `class` attribute). Following that, we have two additional attributes that define the elevation component behavior:

- ▶ `queryFieldType`: This attribute tells Solr which type of field should be used to parse the query text that is given to the component (for example, if you want the component to ignore letter case, you should set this parameter to the field type that makes its contents lowercase)
- ▶ `config-file`: The configuration file which will be used by the component

The next part of the `solrconfig.xml` configuration procedure is search handler definition. It simply tells Solr to create a new search handler with the name of `/promotion` (the `name` attribute) and using the `solr.SearchHandler` class (the `class` attribute). In addition to that, the handler definition also tells Solr to include a component named `elevator` in this search handler. This means that this search handler will use our defined component. Just so you know—you can use more than one search component in a single search handler.

What we see next is the actual configuration of the `elevate` component. You can see that there is a query defined (the `query` XML tag) with an attribute `text="solr"`. This defines the behavior of the component when a user passes `solr` to the `q` parameter. Under this tag, you see a list of unique identifiers of documents that will be placed on top of the results list for the defined query. Each document is defined by a `doc` tag and an `id` attribute (which have to be defined on the basis of `solr.StrField`) which holds the unique identifier. You can have multiple `query` tags in a single configuration file, which means that the elevation component can be used for a variety of queries.

The index configuration and example data file are fairly simple. The index contains two fields that are responsible for holding information about the document. In the example data file, we can see three documents present. As it is not crucial to the explanation, I'll skip discussing it further.

The query you see in the example is returning all the documents. The query is made to our new handler with just a simple one word `q` parameter (the default search field is set to `name` in the `schema.xml`). Recall the `elevate.xml` file and the documents we defined for the query we just passed to Solr. Yes, we told Solr that we want a document with `id=3` in the first place of the results list and we want a document with the `id=1` in the second place of the results list. As you can see, the documents were positioned exactly as we wanted them, so it seems that the component did its job.

### There's more...

There is one more thing I would like to mention.

### Excluding documents with QueryElevationComponent

The `elevate` component cannot only place documents on top of the results list, it can also exclude documents from the results list. To do that, you should add the `exclude="true"` attribute to the document definition in your `elevate.xml` file. This is how the example file would look like:

```
<?xml version="1.0" encoding="UTF-8" ?>
<elevate>
  <query text="solr">
    <doc id="3" />
    <doc id="1" exclude="true" />
    <doc id="2" exclude="true" />
  </query>
</elevate>
```

## Positioning documents with words closer to each other first

Imagine an e-commerce bookshop where the users have only one way to find books—by searching. Imagine an angry user calls the call centre and says that by typing "solr cookbook", the first few pages of results are not relevant to the query he typed in, so in other words, this is not what he searched for. And that's the problem, what to do? One of the answers is phrase boosting in addition to the standard search. This recipe will show you how to boost documents that have words closer to each other.

### How to do it...

For the purpose of this recipe, I assumed that we will be using the dismax query parser.

Let's assume we have the following index structure (just add the following to your schema.xml file to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
<field name="author" type="string" indexed="true" stored="true"/>
```

And our data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Solr perfect search cookbook</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">Solr example cookbook</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">Solr cookbook</field>
  </doc>
</add>
```

As I wrote earlier, we want to get the documents with the words typed by the user close to each other first in the result list. To do that, we modify our query and send the following one to Solr:

```
http://localhost:8983/solr/select?defType=dismax&pf=title^100&q=solr+cookbook&qf=title&fl=score,id,title
```

The result list returned by Solr is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">1</int>
  <lst name="params">
    <str name="pf">title^100</str>
    <str name="fl">score,id,title</str>
    <str name="q">solr cookbook</str>
    <str name="qf">title</str>
    <str name="defType">dismax</str>
  </lst>
</lst>
<result name="response" numFound="3" start="0" maxScore="0.894827">
  <doc>
    <float name="score">0.894827</float>
    <str name="id">3</str>
    <str name="title">Solr cookbook</str>
  </doc>
  <doc>
    <float name="score">0.0035615005</float>
    <str name="id">1</str>
    <str name="title">Solr perfect search cookbook</str>
  </doc>
  <doc>
    <float name="score">0.0035615005</float>
    <str name="id">2</str>
    <str name="title">Solr example cookbook</str>
  </doc>
</result>
</response>
```

We received the documents in the way we wanted. Now let's look at how that happened.

## How it works...

First of all, we have the index structure. For the purpose of the example, I assumed that our book description will consist of three fields and we will be searching for the use of the `title` field, which is based on the standard text field defined in the standard `schema.xml` file provided with the Solr distribution.

As you can see in the provided data file example, there are three books. The first book has two other words between the word `solr` and `cookbook`. The second book has one other word between the given words. And the third book has the words next to each other. Ideally, we would like to have the third book from the example file as the first one on the result list, the second book from the file in second place, and the first book from the example data file as the last result in the results list.

With the query given in the example, it's possible. First, the query is telling Solr to use `dismax` query parser. More about changing the type of the query parser is discussed in the *Choosing a different query parser* recipe in this chapter. Then we have the `pf` parameter which tells `dismax` query parser which field to use for phrase boosting. In our case, it's the `title` field and we gave that field a boost of 100 for phrase matching.

The rest of the parameters are the ones that you should be aware of. The `q` parameter tells Solr what the query is, the `qf` parameter tells the `dismax` query parser which fields are taken into consideration when searching, and the `fl` parameter tells Solr about the fields we want to see in the results list.

The last thing is the results list. As you can see, the documents are sorted in the way we wanted them to be. You should take a look at one thing—the `score` field. This field shows how relevant the document is to the query we sent to Solr.

### There's more...

There is one more thing I think is worth noting.

#### Phrase boosting using standard query parser

It is also possible to boost phrases over words. To achieve results similar to the ones presented in the recipe, but using standard query parser, send the following query to Solr:

```
http://localhost:8983/solr/select?q=solr+AND+cookbook+AND+"solr+cookbook"^100
```

### Sorting results by a distance from a point

Suppose we have a search application that is storing information about the companies. Every company is described by a name and two floating-point numbers that represent the geographical location of the company. One day your boss comes to your room and says that he wants the search results to be sorted by distance from the user's location. This recipe will show you how to do it.

## How to do it...

Let's assume that we have the following index (add the following to the field section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="string" indexed="true" stored="true" />
<field name="x" type="float" indexed="true" stored="true" />
<field name="y" type="float" indexed="true" stored="true" />
```

I assumed that the user location will be provided from the application that is making a query.

The data file looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Company 1</field>
    <field name="x">56.4</field>
    <field name="y">40.2</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Company 2</field>
    <field name="x">50.1</field>
    <field name="y">48.9</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Company 3</field>
    <field name="x">23.18</field>
    <field name="y">39.1</field>
  </doc>
</add>
```

So our user is standing at the North Pole and is using our search application. The query to find the companies and sort them on the basis of the distance from the North Pole would look like this:

```
http://localhost:8983/solr/select?q=company&sort=dist(2,x,y,0,0)+asc
```

The result of that query would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
```

```
<int name="status">0</int>
<int name="QTime">2</int>
<lst name="params">
  <str name="q">company</str>
  <str name="sort">dist(2,x,y,0,0) asc</str>
</lst>
</lst>
<result name="response" numFound="3" start="0">
  <doc>
    <str name="id">3</str>
    <str name="name">Company 3</str>
    <float name="x">23.18</float>
    <float name="y">39.1</float>
  </doc>
  <doc>
    <str name="id">1</str>
    <str name="name">Company 1</str>
    <float name="x">56.4</float>
    <float name="y">40.2</float>
  </doc>
  <doc>
    <str name="id">2</str>
    <str name="name">Company 2</str>
    <float name="x">50.1</float>
    <float name="y">48.9</float>
  </doc>
</result>
</response>
```

If you would like to calculate the distance by hand, you would see that the results are sorted as they should be.

## How it works...

As you can see in the index structure and in the data, every company is described by four fields:

- ▶ id: The unique identifier
- ▶ name: The company name
- ▶ x: The latitude of the company's location
- ▶ y: The longitude of the company's location

We wanted to get the companies that match the given query and are sorted in the ascending order from the North Pole. To do that, we run a standard query with a non-standard sort. The sort parameter consists of a function name, `dist`, which calculates the distance between points. In our example, the function takes five parameters:

- ▶ The first parameter mentions the algorithm used to calculate the distance. In our case, the value `2` tells Solr to calculate the Euclidean distance.
- ▶ The second parameter is the name of the field which contains the latitude.
- ▶ The third parameter is the name of the field which contains the longitude.
- ▶ The fourth parameter is the latitude value of the point from which the distance will be calculated.
- ▶ The fifth parameter is the longitude value of the point from which the distance will be calculated.

After the function, there is the order of the sort, which in our case is `asc`, meaning ascending order.

### See also

If you would like to know more about the functions available in Solr, please go to the Solr wiki page at the following address: <http://wiki.apache.org/solr/FunctionQuery>.

## Getting documents with only a partial match

Imagine a situation where you have an e-commerce library and you want to make a search algorithm that tries to bring the best search results to your customers. However, you noticed that many of your customers tend to make queries with too many words, which results in an empty results list. So you decided to make a query that will require a maximum of two of the words that a user entered to be matched. This recipe will show you how to do it.

### Getting ready

This method can only be used with the `dismax` query parser. The standard query parser doesn't support the `mm` parameter.

### How to do it...

Let's assume that we have the following index structure (add this to the field definition section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
```



We also defined the default logical operator as AND (add this to the field definition section of your `schema.xml` file):

```
<solrQueryParser defaultOperator="AND"/>
```

As you can see, our books are described by two fields. Now let's look at the example data:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Solrcook book revised</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">Some book that was revised</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">Another revised book</field>
  </doc>
</add>
```

The query that will satisfy the requirements would look like this:

```
http://localhost:8983/solr/select?q=book+revised+another+ different+wo
rd+that+doesnt+count&defType=dismax&mm=2
```

The preceding query will return the following results:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="qf">title</str>
      <str name="mm">2</str>
      <str name="q">book revised another different word that doesnt
count</str>
      <str name="defType">dismax</str>
    </lst>
  </lst>
  <result name="response" numFound="3" start="0">
    <doc>
      <str name="id">3</str>
      <str name="title">Another revised book</str>
    </doc>
```

```
<doc>
  <str name="id">2</str>
  <str name="title">Some book that was revised</str>
</doc>
<doc>
  <str name="id">1</str>
  <str name="title">Solrcook book revised</str>
</doc>
</result>
</response>
```

As you can see, even though the query is made of too many words, the result list contains all the documents from the example file. Now let's see why that happened.

### How it works...

The index structure and the data are fairly simple. Every book is described by two fields, namely a unique identifier and a title. We also changed the standard logical operator from OR to AND.

The query is the thing that we are interested in. We have passed about eight words to Solr (`q` parameter), we defined that we want to use the dismax query parser (`defType` parameter), and we sent the mysterious `mm` parameter set to the value of 2. Yes, you are right, the `mm` parameter, also called minimum, tells the dismax query parser how many of the words passed into the query must be matched with the document to state that the document is a match. In our case, we told the dismax query parser that there should be two or more words matched to identify the document as a match.

You should also note one thing—the document that has three words matched is at the top of the list. The relevance algorithm is still there, which means that the documents that have more words that matched the query will be higher on the result list than those that have less words that match the query. The documentation about the `mm` parameter can be found at <http://wiki.apache.org/solr/DisMaxQParserPlugin>.

## Affecting scoring with function

There are many situations where you would like to have an influence on how the score of the documents is calculated. For example, you would like to book the documents on the basis of its purchases. In an e-commerce book store, you would like to show relevant results, but you would like to influence them by adding yet another factor to their score. Is it possible? Yes, and this recipe will show you how to do it.

## How to do it...

Let's assume that we have the following data (just add the following to the field section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
<field name="sold" type="int" indexed="true" stored="true" />
```

The example data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Solrcook book revised</field>
    <field name="sold">5</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">Some book revised</field>
    <field name="sold">200</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">Another revised book</field>
    <field name="sold">60</field>
  </doc>
</add>
```

So we want to boost our documents on the basis of the `sold` field while retaining the relevance of sorting. Our user typed `revised` in the search box. So the query would look like this:

```
http://localhost:8983/solr/select?defType=dismax&qf=title&q=revised&fl=*,score
```

And the results would be like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="qf">title</str>
      <str name="fl">*,score</str>
      <str name="q">revised</str>
```

```

    <str name="defType">dismax</str>
  </lst>
</lst>
<result name="response" numFound="3" start="0" maxScore="0.35615897">
  <doc>
    <float name="score">0.35615897</float>
    <str name="id">1</str>
    <int name="sold">5</int>
    <str name="title">Solrcook book revised</str>
  </doc>
  <doc>
    <float name="score">0.35615897</float>
    <str name="id">2</str>
    <int name="sold">200</int>
    <str name="title">Some book revised</str>
  </doc>
  <doc>
    <float name="score">0.35615897</float>
    <str name="id">3</str>
    <int name="sold">60</int>
    <str name="title">Another revised book</str>
  </doc>
</result>
</response>

```

Now let's add the `sold` factor by adding the following to the query:

```
bf=product(sold)
```

So our modified query would look like this:

```
http://localhost:8983/solr/select?defType=dismax&qf=title&q=revised&fl=*,score&bf=product(sold)
```

And the results for the preceding query would be:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="fl">*,score</str>
      <str name="q">revised</str>
      <str name="qf">title</str>
      <str name="bf">product(sold)</str>
    </lst>
  </lst>

```

```
<str name="defType">dismax</str>
</lst>
</lst>
<result name="response" numFound="3" start="0" maxScore="163.1048">
  <doc>
    <float name="score">163.1048</float>
    <str name="id">2</str>
    <int name="sold">200</int>
    <str name="title">Some book revised</str>
  </doc>
  <doc>
    <float name="score">49.07608</float>
    <str name="id">3</str>
    <int name="sold">60</int>
    <str name="title">Another revised book</str>
  </doc>
  <doc>
    <float name="score">4.279089</float>
    <str name="id">1</str>
    <int name="sold">5</int>
    <str name="title">Solrcook book revised</str>
  </doc>
</result>
</response>
```

As you see, adding the parameter changed the whole results list. Now let's see why that happened.

## How it works...

The schema.xml file is simple. It contains three fields:

- ▶ `id`: Responsible for holding the unique identifier of the book
- ▶ `title`: The book title
- ▶ `sold`: The number of pieces that have been sold during the last month

In the data, we have three books. Each of the books has the same number of words in the title. That's why when typing the first query, all documents got the same score. As you see, the first book is the one with the least pieces sold and that's not what we want to achieve.

That's why we added the `bf` parameter. It tells Solr which function to use to affect the scoring computation (in this case, the result of the function will be added to the score of the document). In our case, it is the `product` function which returns the product of the values we provide as its arguments—in our case, the one and only argument of the function will be the value of the book's `sold` field.

The result list of the modified query clearly shows how the scoring was affected by the function. On the first place of the results list, we have the book that was the most popular last week. The next book is the one which was less popular than the first book, but more popular than the last book. The last book in the results is the least popular.

## See also

If you would like to know more about the functions available in Solr, please go to the Solr wiki page at the following address: <http://wiki.apache.org/solr/FunctionQuery>.

## Nesting queries

Imagine a situation where you need a query nested inside another query. Let's imagine that you want to run a query using the standard request handler, but you need to embed a query that is parsed by the dismax query parser inside it. It is possible with Solr 1.4, and this recipe will show you how to do it.

## How to do it...

Let's assume that we have the following data (just add the following to the field section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
```

The example data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Revised solrcook book</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">Some book revised</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">Another revised little book</field>
  </doc>
</add>
```

Imagine that you are using the standard query parser to support Lucene query syntax, but you would like to boost phrases using the dismax query parser. At first it seems that it is impossible, but let's suppose that we want to find books that have the words `revised` and `book` in their `title` field, and we want to boost the `revised book` phrase by 10. Let's send a query like this:

```
http://localhost:8983/solr/select?q=revised+AND+book+AND+_
query_:"{!dismax qf=title pf=title^10 v=$qq}"&qq=revised+book
```

The results of the preceding query should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">2</int>
    <lst name="params">
      <str name="fl">*,score</str>
      <str name="qq">book revised</str>
      <str name="q">book AND revised AND _query_:"{!dismax qf=title
pf=title^10 v=$qq}"</str>
    </lst>
  </lst>
  <result name="response" numFound="3" start="0" maxScore="0.77966106">
    <doc>
      <float name="score">0.77966106</float>
      <str name="id">2</str>
      <str name="title">Some book revised</str>
    </doc>
    <doc>
      <float name="score">0.07087828</float>
      <str name="id">1</str>
      <str name="title">Revised solrcook book</str>
    </doc>
    <doc>
      <float name="score">0.07087828</float>
      <str name="id">3</str>
      <str name="title">Another revised little book</str>
    </doc>
  </result>
</response>
```

As you can see, the results list was sorted exactly the way we wanted. Now let's see how it works.

## How it works...

As you can see, our index is very simple. It consists of two fields—one to hold the unique identifier (the `id` field) and the other to hold the title of the book (the `title` field).

Let's look at the query. The `q` parameter is built from two parts connected with the `AND` operator. The first one `revised+AND+book` is just a usual query with a logical operator `AND` defined. The second part that is making the query is starting with a strange looking expression, `_query_`. This expression tells Solr that another query should be made that will affect the results list. Notice that the expression is surrounded with `"` characters. Then we see the expression stating that Solr should use the `dismax` query parser (the `!dismax` part) and the parameters that will be passed to the parser (`qf` and `pf`). The `v` parameter is an abbreviation for value and it is used to pass the value of the `q` parameter. The value passed to the `dismax` query parser will, in our case, be `revised+book`. Why? You can see something that is called parameter dereferencing. Using the `$qq` expression, we tell Solr to use the value of the `qq` parameter. Of course, we could pass the value to the `v` parameter, but I wanted to show you how to use the dereferencing mechanism.

The `qq` parameter is set to `revised+book` and it is used by Solr as a parameter for the query that was passed to the `dismax` query parser.

The results show that we achieved exactly what we wanted.





# 6

## Using Faceting Mechanism

In this chapter, we will cover:

- ▶ Getting the number of documents with the same field value
- ▶ Getting the number of documents with the same date range
- ▶ Getting the number of documents with the same value range
- ▶ Getting the number of documents matching the query and the sub query
- ▶ How to remove filters from faceting results
- ▶ How to name different faceting results
- ▶ How to sort faceting results in an alphabetical order
- ▶ How to implement the autosuggest feature using faceting
- ▶ How to get the number of documents that don't have a value in the field
- ▶ How to get all the faceting results, not just the first hundred ones
- ▶ How to have two different facet limits for two different fields in the same query

### Introduction

One of the advantages of Solr is the ability to group results on the basis of the field's contents. Solr classification mechanism, called faceting, provides the functionalities which can help us in several tasks that we need to do in everyday work. For example, getting the number of documents with the same values in a field (such as the companies from the same city) through the ability of date and ranges grouping, to the autocomplete features based on the faceting mechanism. This chapter will show you how to handle some of the common tasks when using the faceting mechanism.

## Getting the number of documents with the same field value

Imagine a situation where besides the search results, you have to return the number of documents with the same field value. For example, imagine you have an application that allows the user to search for companies in Europe, and your client wants to have the number of companies in the cities where the companies that were found by the query are located. To do this, you could of course run several queries, but Solr provides a mechanism called faceting that can do that for you. This recipe will show you how to do it.

### How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `city` field to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="city" type="string" indexed="true" stored="true" />
```

And the example data can look like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Company 1</field>
    <field name="city">New York</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Company 2</field>
    <field name="city">New Orleans</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Company 3</field>
    <field name="city">New York</field>
  </doc>
</add>
```

Let's suppose that our hypothetical user searches for the word `company`. The query that will get us what we want should look like this:

```
http://localhost:8983/solr/select?q=name:company&facet=true&facet.
field=city
```

The result of this query should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="facet">true</str>
      <str name="facet.field">city</str>
      <str name="q">name:company</str>
    </lst>
  </lst>
  <result name="response" numFound="3" start="0">
    <doc>
      <str name="city">New York</str>
      <str name="id">1</str>
      <str name="name">Company 1</str>
    </doc>
    <doc>
      <str name="city">New Orleans</str>
      <str name="id">2</str>
      <str name="name">Company 2</str>
    </doc>
    <doc>
      <str name="city">New York</str>
      <str name="id">3</str>
      <str name="name">Company 3</str>
    </doc>
  </result>
  <lst name="facet_counts">
    <lst name="facet_queries"/>
    <lst name="facet_fields">
      <lst name="city">
        <int name="New York">2</int>
        <int name="New Orleans">1</int>
      </lst>
    </lst>
    <lst name="facet_dates"/>
  </facet_counts>
</response>
```

```
</lst>  
</response>
```

As you can see, besides the normal results list, we got the faceting results with the numbers that we wanted. Now let's see how that happened.

### How it works...

The index structure and data are pretty simple and they make the example easier to understand. The company is described by three fields. We are particularly interested in the `city` field. This is the field that we want to use to get the number of companies that have the same value in this field—which basically means that they are in the same city.

To do that, we run a query to Solr and inform the query parser that we want the documents that have the word `company` in the `title` field. Additionally, we say that we also want to enable the faceting mechanism—we say that by using the `facet=true` parameter. The `facet.field` parameter tells Solr which field to use to calculate the faceting numbers. You can specify the `facet.field` parameter multiple times to get the faceting numbers for different fields in the same query.

As you can see in the results list, the results of all types of faceting are grouped in the list with the `name="facet_counts"` attribute. The field based faceting is grouped under the list with the `name="facet_fields"` attribute. Every field that you specified using the `facet.field` parameter has its own list which has the `name` attribute same as the value of the parameter in the query—in our case it is `city`. Then we can finally see the results that we are interested in—the pairs of values (the `name` attribute) and how many documents have that value in the specified field.

### There's more...

There is some more information that can be useful when using faceting.

#### How to show facets with counts greater than zero

The default behavior of Solr is to show all the faceting results irrespective of the counts. If you want to show only the facets with counts greater than zero, then you should add the `facet.mincount=1` parameter to the query (you can set this parameter to other values if you are interested in any arbitrary value).

#### Lexicographical sorting of the faceting results

If you want to sort the faceting results lexicographically, not by the highest count (which is the default behavior), then you need to add the `facet.sort=index` parameter.

## Getting the number of documents with the same date range

Imagine a situation where you have an application that monitors the books that were added to the e-commerce library. One of the requirements of your application is to show how many books were added during certain periods—actually defined periods of one week. This recipe will show you how to do it in an efficient and easy way.

### How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `added` field to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="added" type="tdate" indexed="true" stored="true" />
```

Our data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Lucene or Solr ?</field>
    <field name="added">2010-12-06T12:12:12Z</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">My Solr and the rest of the world</field>
    <field name="added">2010-12-07T11:11:11Z</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">Solr recipes</field>
    <field name="added">2010-11-30T12:12:12Z</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="title">Solr cookbook</field>
    <field name="added">2010-11-29T12:12:12Z</field>
  </doc>
</add>
```

Let's recall what we want to achieve. We want to get the list of periods (we need every period to cover one week) that will show us how many books were added in each period. We also want the data for the last 30 days. To do that, we will send the following query to Solr:

```
http://localhost:8983/solr/select?q=*:*&rows=0&facet=true&facet.  
date=added&facet.date.start=NOW/DAY-30DAYS&facet.date.end=NOW/  
DAY&facet.date.gap=%2B7DAY
```

This query will result in the following response from Solr:

```
<?xml version="1.0" encoding="UTF-8"?>  
<response>  
  <lst name="responseHeader">  
    <int name="status">0</int>  
    <int name="QTime">2</int>  
    <lst name="params">  
      <str name="facet.date.start">NOW/DAY-30DAYS</str>  
      <str name="facet">true</str>  
      <str name="q">*:*</str>  
      <str name="facet.date">added</str>  
      <str name="facet.date.gap">+7DAY</str>  
      <str name="facet.date.end">NOW/DAY</str>  
      <str name="rows">0</str>  
    </lst>  
  </lst>  
  <result name="response" numFound="4" start="0"/>  
  <lst name="facet_counts">  
    <lst name="facet_queries"/>  
    <lst name="facet_fields"/>  
    <lst name="facet_dates">  
      <lst name="added">  
        <int name="2010-11-08T00:00:00Z">0</int>  
        <int name="2010-11-15T00:00:00Z">0</int>  
        <int name="2010-11-22T00:00:00Z">0</int>  
        <int name="2010-11-29T00:00:00Z">2</int>  
        <int name="2010-12-06T00:00:00Z">2</int>  
        <str name="gap">+7DAY</str>  
        <date name="end">2010-12-13T00:00:00Z</date>  
      </lst>  
    </lst>  
  </lst>  
</response>
```

Now let's see how it works.

## How it works...

As you can see, the index structure is simple. There is just one thing that should catch our attention—the `added` field. This field is responsible for holding the information about the date when the book was added to the index. To allow faster date faceting, I decided not to use the `date` type, but the `tdate` type. This type is one of the standard field types that are present in the default Solr `schema.xml` file. It has a higher precision step than the `date` field type and thus will be faster for our needs.

You may notice that dates are written in their canonical representation. This is because Lucene and Solr only support dates in such a format.

Now the interesting part—the query. As we are not interested in the search results, we ask for all the documents in the index (`q=*:*` parameter) and we tell Solr not to return the search results (`rows=0` parameter). Then we tell Solr that we want the faceting mechanism to be enabled for the query (`facet=true` parameter). Now comes the fun part.

We will not be using the standard faceting mechanism—the field-based faceting. Instead, we will use date faceting which is optimized to work with dates. So, we tell Solr which field will be used for date faceting by adding the parameter `facet.date` with the `added` value. This means that the `added` field will be used for the date faceting calculation. Then we specify the date from which the calculation will begin, by adding the `facet.date.start` parameter. We used the two constants and the functionality of their arithmetic's by specifying the following value: `NOW/DAY-30DAY` (which mean the following day minus thirty days). It says that we want to start from the date that was 30 days from now. We can also specify the date in the canonical form, but I think that using the constants is a more convenient way to do that—for example, we don't need to worry about passing the correct date every time. Next we have the `facet.date.end` parameter, which tells Solr about the maximum date the calculation should be made for. In our case we used `NOW/DAY`, which means that we want the calculation to stop at today's date—we don't have any books that will be added tomorrow in the index, so why bother? The last parameter (`facet.date.gap`) informs Solr of the length of the periods that will be calculated. Our requirement was that we wanted to calculate the books that were added in a period of 7 days, and that is what we told Solr by setting the last parameter to the value of `+7DAY`. You may notice that the `+` character was encoded; that's because the `+` character is a special character for Lucene and in our case it needs to be encoded to be properly recognized.

Remember that when using the date faceting mechanism, you must specify the three parameters:

- ▶ `facet.date.start`
- ▶ `facet.date.end`
- ▶ `facet.date.gap`

Or the date faceting mechanism won't work.



## See also

To see more constants that can be used with date faceting, please refer to the Lucene date math parser documentation. It can be found at the following address: <http://lucene.apache.org/solr/api/org/apache/solr/util/DateMathParser.html>.

## Getting the number of documents with the same value range

Imagine that you have an application where users can search the index to find a car for rent. One of the requirements of the application is to show a navigation panel, where users can choose the price range for the cars they are interested in. To do it in an efficient way, we will use range faceting and this recipe will show you how to do it.

## How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `price` field to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="price" type="float" indexed="true" stored="true" />
```

The example data looks like this:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="name">Super Mazda</field>
  <field name="price">50</field>
</doc>
<doc>
  <field name="id">2</field>
  <field name="name">Mercedes Benz</field>
  <field name="price">210</field>
</doc>
<doc>
  <field name="id">3</field>
  <field name="name">Bentley</field>
  <field name="price">290</field>
</doc>
</add>
```

```

    <field name="id">2</field>
    <field name="name">Super Honda</field>
    <field name="price">99.90</field>
  </doc>
</add>

```

Now, as you may recall, our requirement was to show the navigation panel with the price ranges. To do that, we need to get the data from Solr. We also know that the minimum price for car rent is 1 dollar and the maximum is 400 dollars. To get the price ranges from Solr, we send the following query:

```

http://localhost:8983/solr/select?q=*:*&rows=0&facet=true&facet.
range=price&facet.range.start=0&facet.range.end=400&facet.range.
gap=100

```

This query would result in the following result list:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">3</int>
    <lst name="params">
      <str name="facet">true</str>
      <str name="q">*:*</str>
      <str name="facet.range.start">0</str>
      <str name="facet.range">price</str>
      <str name="facet.range.end">400</str>
      <str name="facet.range.gap">100</str>
      <str name="rows">0</str>
    </lst>
  </lst>
  <result name="response" numFound="4" start="0"/>
  <lst name="facet_counts">
    <lst name="facet_queries"/>
    <lst name="facet_fields"/>
    <lst name="facet_dates"/>
    <lst name="facet_ranges">
      <lst name="price">
        <lst name="counts">
          <int name="0.0">2</int>
          <int name="100.0">0</int>
          <int name="200.0">2</int>
          <int name="300.0">0</int>
        </lst>
        <float name="gap">100.0</float>
      </lst>
    </lst>
  </lst>

```

```
<float name="start">0.0</float>
<float name="end">400.0</float>
</lst>
</lst>
</lst>
</response>
```

So, we got exactly what we wanted. Now let's see how it works.

### How it works...

As you can see, the index structure is simple. There are three fields, one responsible for the unique identifier, one for the car name, and the last one for the price of rent.

The query is where all the magic is done. As we are not interested in the search results, we ask for all the documents in the index (`q=*:*` parameter) and we tell Solr not to return the search results (`rows=0` parameter). Then we tell Solr that we want the faceting mechanism to be enabled for the query (`facet=true` parameter). We will not be using the standard faceting mechanism—the field based faceting. Instead, we will use range faceting, which is optimized to work with ranges. So, we tell Solr which field will be used for range faceting by adding the parameter `facet.range` with the `price` value. This means that the `price` field will be used for the range faceting calculation. Then we specify the lower boundary from which the range faceting calculation will begin. We do this by adding the `facet.range.start` parameter—in our example, we set it to 0. Next we have the `facet.range.end` parameter which tells Solr when to stop the calculation of the range faceting. The last parameter (`facet.range.gap`) informs Solr about the length of the periods that will be calculated.

Remember that when using the range faceting mechanism, you must specify the three parameters:

- ▶ `facet.range.start`
- ▶ `facet.range.end`
- ▶ `facet.range.gap`

Or the range faceting mechanism won't work.

In the faceting results, you can see the periods and the number of documents that were found in each of them. First period can be found under the `<int name="0.0">` tag. This period consists of prices from 0 to 100 (in mathematical notation it would be `<0; 100>`). It contains two cars. The next period can be found under the `<int name="100.0">` tag and consists of prices from 100 to 200 (in mathematical notation it would be `<100; 200>`), and so on.

## Getting the number of documents matching the query and sub query

Imagine a situation where you have an application that has a search feature for cars. One of the requirements is not only to show search results, but also to show the number of cars with the price period chosen by the user. There is also another thing—those queries must be fast because of the number of queries that will be run. Can Solr handle that? The answer is yes. This recipe will show you how to do it.

### How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `price` field to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="price" type="float" indexed="true" stored="true" />
```

The example data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Car 1</field>
    <field name="price">70</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Car 2</field>
    <field name="price">101</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Car 3</field>
    <field name="price">201</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="name">Car 4</field>
    <field name="price">99.90</field>
  </doc>
</add>
```

Now, as you may recall, our requirements are: cars that match the query (let's suppose that our user typed car) and to show the counts in the chosen price periods. For the purpose of the recipe, let's assume that the user has chosen two periods of prices:

- ▶ 10 – 80
- ▶ 90 – 300

The query to achieve such a requirement should look like this:

```
http://localhost:8983/solr/select?q=name:car&facet=true&facet.  
query=price:[10 TO 80]&facet.query=price:[90 TO 300]
```

The result list of this query should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>  
<response>  
  <lst name="responseHeader">  
    <int name="status">0</int>  
    <int name="QTime">1</int>  
    <lst name="params">  
      <str name="facet">true</str>  
      <arr name="facet.query">  
        <str>price:[10 TO 80]</str>  
        <str>price:[90 TO 300]</str>  
      </arr>  
      <str name="q">name:car</str>  
    </lst>  
  </lst>  
  <result name="response" numFound="4" start="0">  
    <doc>  
      <str name="id">1</str>  
      <str name="name">Car 1</str>  
      <float name="price">70.0</float>  
    </doc>  
    <doc>  
      <str name="id">2</str>  
      <str name="name">Car 2</str>  
      <float name="price">101.0</float>  
    </doc>  
    <doc>  
      <str name="id">3</str>  
      <str name="name">Car 3</str>  
      <float name="price">201.0</float>  
    </doc>  
    <doc>  
      <str name="id">4</str>
```

```

    <str name="name">Car 4</str>
    <float name="price">99.9</float>
  </doc>
</result>
<lst name="facet_counts">
  <lst name="facet_queries">
    <int name="price: [10 TO 80]">1</int>
    <int name="price: [90 TO 300]">3</int>
  </lst>
  <lst name="facet_fields"/>
  <lst name="facet_dates"/>
</lst>
</response>

```

### How it works...

As you can see, the index structure is simple. There are three fields, one responsible for the unique identifier, one for the car name, and the last one for the price.

Next, we have the query. First you can see a standard query where we tell Solr that we want to get all the documents that have the `car` word in the `name` field (`q=name:car` parameter). Next, we say that we want to use the faceting mechanism by adding the `facet=true` parameter to the query. This time we will use the faceting type query. This means that we can pass the query to the faceting mechanism and as a result, we get the number of documents that match the given query. In our example case, we wanted two periods:

- ▶ One from the price of 10 to 80
- ▶ Another from the price of 90 to 300

This is achieved by adding the `facet.query` parameter with the appropriate value. The first period is defined as a standard range query to the `price` field—`price: [10 TO 80]`. The second query is very similar, just different values. The value passed to the `facet.query` parameter should be a Lucene query written using the default query syntax.

As you can see in the result, the query faceting results are grouped under the `<lst name="facet_queries">` XML tag with the names exactly as in the passed queries. You can see that Solr correctly calculated the number of cars in each of the periods, which means that this is a perfect solution for us, when we can't use the range faceting mechanism.

## How to remove filters from faceting results

Let's assume for the purpose of this recipe that you have an application that can search for companies within a city and state. But the requirements say that not only should you show the search results, but also the number of companies in each city and the number of companies in each state (to say in the Solr way—you want to exclude the filter query from the faceting results). Can Solr do that in an efficient way? Sure it can. And this recipe will show you how to do it.

### Getting ready

Before you start reading this recipe, please take a look at the *Getting the number of documents with the same field value* recipe in this chapter.

### How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `city` and `state` fields to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="city" type="string" indexed="true" stored="true" />
<field name="state" type="string" indexed="true" stored="true" />
```

And the example data can look like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Company 1</field>
    <field name="city">New York</field>
    <field name="state">New York</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Company 2</field>
    <field name="city">New Orleans</field>
    <field name="state">Luiziana</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Company 3</field>
    <field name="city">New York</field>
```

```

    <field name="state">New York</field>
  </doc>
</doc>
  <field name="id">4</field>
  <field name="name">Company 4</field>
  <field name="city">New York</field>
  <field name="state">New York</field>
</doc>
</add>

```

Let's suppose that our hypothetical user searches for the word `company`, and tells our application that he needs the companies matching the word in the state of New York. In that case, the query that will fulfill our requirement should look like this:

```

http://localhost:8983/solr/select?q=name:company&facet=true
&fq={!tag=stateTag}state:"New York"&facet.field={!ex=stateTag}
city&facet.field={!ex=stateTag}state

```

The result for this query will look as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="facet">true</str>
      <arr name="facet.field">
        <str>{!ex=stateTag}city</str>
        <str>{!ex=stateTag}state</str>
      </arr>
      <str name="fq">{!tag=stateTag}state:"New York"</str>
      <str name="q">name:company</str>
    </lst>
  </lst>
  <result name="response" numFound="3" start="0">
    <doc>
      <str name="city">New York</str>
      <str name="id">1</str>
      <str name="name">Company 1</str>
      <str name="state">New York</str>
    </doc>
    <doc>
      <str name="city">New York</str>
      <str name="id">3</str>
      <str name="name">Company 3</str>
    </doc>
  </result>
</response>

```



```
<str name="state">New York</str>
</doc>
<doc>
  <str name="city">New York</str>
  <str name="id">4</str>
  <str name="name">Company 4</str>
  <str name="state">New York</str>
</doc>
</result>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="city">
      <int name="New York">3</int>
      <int name="New Orleans">1</int>
    </lst>
    <lst name="state">
      <int name="New York">3</int>
      <int name="Luiziana">1</int>
    </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
</response>
```

Now let's see how it works.

## How it works...

The index structure is pretty simple—it contains four fields that describe the company. The search will be performed against the `name` field, while the filtering and the faceting is done with the use of the `state` and the `city` fields.

So let's get on with the query. As you can see, we have some typical elements there. First the `q` parameter that just tells Solr where and what to search for. Then the `facet=true` parameter that enables the faceting mechanism. So far, so good. Following that, you have a strange looking filter query (the `fq` parameter) with the `fq={!tag=stateTag}state:"New York"` value. It tells Solr to only show those results that have `New York` in the `state` field. By adding the `{!tag=stateTag}` part, we basically gave that filter query a name (`stateTag`) which we will use further.

Now look at the two `facet.field` parameters. Our requirement was to show the number of companies in all the states and cities. The only thing that was preventing us from getting those numbers was the filter query that we added. So let's exclude it from the faceting results. How to do it? It's simple—just add `{!ex=stateTag}` to the beginning of each of the `facet.field` parameters like this: `facet.field={!ex=stateTag}city`. It tells Solr to exclude the filter with the passed name.

As you can see in the results list, we got the numbers correctly, which means that the exclude works as intended.

## How to name different faceting results

There are some situations where it would be nice to be able to name the faceting results lists. Of course it's just a thing of comfort, but I think that it is a worthy piece of knowledge. You can make your life a bit easier when you know what to look for in the results list returned by Solr. This recipe will show you how to name your faceting results.

### How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `city` and `state` fields to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="city" type="string" indexed="true" stored="true" />
<field name="state" type="string" indexed="true" stored="true" />
```

This index structure is responsible for holding information about companies and their location. The example data could look like this:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="name">Company 1</field>
  <field name="city">New York</field>
  <field name="state">New York</field>
</doc>
<doc>
  <field name="id">2</field>
  <field name="name">Company 2</field>
  <field name="city">New Orleans</field>
  <field name="state">Luiziana</field>
</doc>
</add>
```

```
<field name="id">3</field>
<field name="name">Company 3</field>
<field name="city">New York</field>
<field name="state">New York</field>
</doc>
<doc>
  <field name="id">4</field>
  <field name="name">Company 4</field>
  <field name="city">New York</field>
  <field name="state">New York</field>
</doc>
</add>
```

For the purpose of the example, let's assume that our application is querying Solr for the results of two field faceting operations and the standard query results. The original query could look like this:

```
http://localhost:8983/solr/select?q=name:company&facet=true
&fq={!tag=stateTag}state:Luiziana&facet.field=city&facet.
field={!ex=stateTag}state
```

Now let's suppose that we want to name our faceting results. We want the first faceting shown under the name `cityFiltered` and the second one under the name `stateUnfiltered`. To do that, we should change the previous query to the following:

```
http://localhost:8983/solr/select?q=name:company&facet=true&fq={!tag
=stateTag}state:Luiziana&facet.field={!key=stateFiltered}city&facet.
field={!ex=stateTag key=stateUnfiltered}state
```

The result of the query looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="facet">true </str>
      <arr name="facet.field">
        <str>{!key=stateFiltered}city</str>
        <str>{!ex=stateTag key=stateUnfiltered}state</str>
      </arr>
      <str name="fq">{!tag=stateTag}state:Luiziana</str>
      <str name="q">name:company</str>
    </lst>
  </lst>
  <result name="response" numFound="1" start="0">
```

```

<doc>
  <str name="city">New Orleans</str>
  <str name="id">2</str>
  <str name="name">Company 2</str>
  <str name="state">Luiziana</str>
</doc>
</result>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="stateFiltered">
      <int name="New Orleans">1</int>
      <int name="New York">0</int>
    </lst>
    <lst name="stateUnfiltered">
      <int name="New York">3</int>
      <int name="Luiziana">1</int>
    </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
</response>

```

Now let's see how it works.

## How it works...

The index structure and the example data are only here to help us make a query, so I'll skip discussing them.

The first query is just an example of how faceting can be used. The first query contains some field faceting and one filter query exclusion. Then comes the second query which is the one that we are interested in. As you can see, the second query differs from the first one in only two respects—`key=cityFiltered` and `key=stateUnfiltered`. This is another example of the so-called local params, but this time we use it to give a name for the faceting lists.

The first field faceting calculation is defined as follows: `facet.`

`field={!key=stateFiltered}city`. This means that we want to get the calculation for the field `city` and name it `stateFiltered` (we do it by passing the `key` argument). The second, `facet.field={!ex=stateTag key=stateUnfiltered}state`, is telling Solr that we want to get the field facets for the `state` field and name it `stateUnfiltered`. We also want to exclude a filter query named `stateTag`.

As you can see in the results list, our query worked perfectly. We don't have lists named on the basis of fields; instead we got two lists with the names exactly the same as we passed with the use of the `key` parameter.

## How to sort faceting results in an alphabetical order

Imagine a situation where you have a website and you present some kind of advertisements—for example, house rental advertisements. One of the requirements is to show a list of cities where the offers that the query typed by the user matched are located. So the first thing you think is to use the faceting mechanism—and that's a good idea. But then, your boss tells you that he is not interested in the counts and you have to sort the results in an alphabetical order. So, is Solr able to do it? Of course it is, and this recipe will show you how to do it.

### Getting ready

Before you start reading this recipe, please take a look at the *Getting the number of documents with the same field value* recipe in this chapter.

### How to do it...

For the purpose of the recipe, let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `city` field to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="city" type="string" indexed="true" stored="true" />
```

This index structure is responsible for holding the information about companies and their location. The example data could look as follows:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">House 1</field>
    <field name="city">New York</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">House 2</field>
    <field name="city">Washington</field>
  </doc>
```

```

<doc>
  <field name="id">3</field>
  <field name="name">House 3</field>
  <field name="city">Washington</field>
</doc>
<doc>
  <field name="id">4</field>
  <field name="name">House 4</field>
  <field name="city">San Francisco</field>
</doc>
</add>

```

Let's assume that our hypothetical user typed `house` in the search box. The query to return the search results with the faceting results, sorted alphabetically, should be like this:

```

http://localhost:8983/solr/select?q=name:house&facet=true&facet.
field=city&facet.sort=index

```

The results returned by Solr for the above query should look like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">1</int>
  <lst name="params">
    <str name="facet">true</str>
    <str name="facet.field">city</str>
    <str name="facet.sort">index</str>
    <str name="q">name:house</str>
  </lst>
</lst>
<result name="response" numFound="4" start="0">
  <doc>
    <str name="city">New York</str>
    <str name="id">1</str>
    <str name="name">House 1</str>
  </doc>
  <doc>
    <str name="city">Washington</str>
    <str name="id">2</str>
    <str name="name">House 2</str>
  </doc>
  <doc>
    <str name="city">Washington</str>

```

```
<str name="id">3</str>
<str name="name">House 3</str>
</doc>
<doc>
  <str name="city">San Francisco</str>
  <str name="id">4</str>
  <str name="name">House 4</str>
</doc>
</result>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="city">
      <int name="New York">1</int>
      <int name="San Francisco">1</int>
      <int name="Washington">2</int>
    </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
</response>
```

As you can see, the faceting results returned by Solr are not sorted by counts but in alphabetical order. Now let's see how it works.

## How it works...

The index structure and the example data are only here to help us make a query, so I'll skip discussing them.

The query shown in the recipe differs from the standard faceting query by only one parameter—`facet.sort`. It tells Solr how to sort the faceting results. The parameter can be assigned one of the two values:

- ▶ `count`—Tells Solr to sort the faceting results placing the highest counts first
- ▶ `index`—Tells Solr to sort the faceting results by index order, which means that the results will be sorted lexicographically

For the purpose of the recipe, we choose the second option and as you can see in the returned results, we got what we wanted.

## There's more...

There is one more thing.

### Choosing the sort order in Solr earlier than 1.4

If you want to set the faceting results sorting in Solr earlier than 1.4, you can't use the values of the `facet.sort` parameter shown in the recipe. Instead of the `count` value, you should use the `true` value, and instead of the `index` value, you should use the `false` value.

## How to implement the autosuggest feature using faceting

There are plenty of web-based applications that help the users choose what they want to search for. One of the features that helps users is the autocomplete (or autosuggest) feature—like the one that most of the commonly used world search engines have. Let's assume that we have an e-commerce library and we want to help a user to choose a book title—we want to enable autosuggest on the basis of the title. This recipe will show you how to do that.

## Getting ready

Before you start reading this recipe, please take a look at the *Getting the number of documents with the same field value* recipe in this chapter.

## How to do it...

For the purpose of this recipe, let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
<field name="title_autocomplete" type="lowercase" indexed="true"
stored="true">
```

We also want to add some field copying to make some operations automatically. Just add the following after the fields section in your `schema.xml` file:

```
<copyField source="title" dest="title_autocomplete" />
```

The lowercase field type should look like this (just add this to your `schema.xml` file to the types definitions):

```
<fieldType name="lowercase" class="solr.TextField">
  <analyzer>
```



```
<tokenizer class="solr.KeywordTokenizerFactory"/>
<filter class="solr.LowerCaseFilterFactory" />
</analyzer>
</fieldType>
```

The example data looks like this:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="title">Lucene or Solr ?</field>
</doc>
<doc>
  <field name="id">2</field>
  <field name="title">My Solr and the rest of the world</field>
</doc>
<doc>
  <field name="id">3</field>
  <field name="title">Solr recipes</field>
</doc>
<doc>
  <field name="id">4</field>
  <field name="title">Solr cookbook</field>
</doc>
</add>
```

Let's assume that our hypothetical user typed the letters `so` in the search box, and we want to give him the first 10 suggestions with the highest counts. We also want to suggest the whole titles, not just the single words. To do that, we should send the following query to Solr:

```
http://localhost:8983/solr/select?q=*:*&rows=0&facet=true&facet.
field=title_autocomplete&facet.prefix=so
```

As the result on this query, Solr returned the following output:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">16</int>
  <lst name="params">
    <str name="facet">true</str>
    <str name="q">*:*</str>
    <str name="facet.prefix">so</str>
    <str name="facet.field">title_autocomplete</str>
    <str name="rows">0</str>
```

```

    </lst>
  </lst>
  <result name="response" numFound="4" start="0"/>
  <lst name="facet_counts">
    <lst name="facet_queries"/>
    <lst name="facet_fields">
      <lst name="title_autocomplete">
        <int name="solr cookbook">1</int>
        <int name="solr recipes">1</int>
      </lst>
    </lst>
    <lst name="facet_dates"/>
  </lst>
</response>

```

As you can see, we got what we wanted in the faceting results. Now let's see how it works.

### How it works...

You can see that our index structure defined in the `schema.xml` file is pretty simple. Every book is described by two fields—the `id` and the `title`. The additional field will be used to provide the autosuggest feature.

The `copy field` section is there to automatically copy the contents of the `title` field to the `title_autocomplete` field.

The `lowercase` field type is a type we will use to provide the autocomplete feature—the same for lowercase words typed by the users as well as the uppercased. If we want to show different results for uppercased and lowercased letters, then the `string` type will be sufficient.

Now let's take a look at the query. As you can see we are searching the whole index (the parameter `q=*`), but we are not interested in any search results (the `rows=0` parameter). We tell Solr that we want to use the faceting mechanism (`facet=true` parameter) and that it will be field based faceting on the basis of the `title_autocomplete` field (the `facet.field=title_autocomplete` parameter). The last parameter—the `facet.prefix` can be something new. Basically, it tells Solr to return only those faceting results that are beginning with the prefix specified as the value of this parameter, which in our case is the value of `so`. The use of this parameter enables us to show the suggestions that the user is interested in—and we can see from the results that we achieved what we wanted.

### There's more...

There is one more thing.

## Suggesting words, not whole phrases

If you want to suggest words instead of whole phrases, you don't have to change much of the preceding configuration. Just change the type `title_autocomplete` to a type based on `solr.TextField` (like the `text_ws` field type, for example). You should remember though, not to use heavily-analyzed text (like stemmed text), to be sure that your word won't be modified too much.

## How to get the number of documents that don't have a value in the field

Let's imagine we have an e-commerce library where we put some of our books on a special promotion—we give them away for free. We want to share that knowledge with our customers and say: "Hey, you searched for Solr. We found this, but we also have X books that are free!". To do that, we index the books that are free without the price defined. But how to make a query to Solr to retrieve the data we want? This recipe will show you how.

### Getting ready

Before you start reading this recipe, please take a look at the *Getting the number of documents matching the query and sub query* recipe in this chapter.

### How to do it...

For the purpose of the recipe, let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
<field name="price" type="float" indexed="true" stored="true">
```

The example data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Lucene or Solr ?</field>
    <field name="price">11</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">My Solr and the rest of the world</field>
    <field name="price">44</field>
```

```

</doc>
<doc>
  <field name="id">3</field>
  <field name="title">Solr recipes</field>
  <field name="price">15</field>
</doc>
<doc>
  <field name="id">4</field>
  <field name="title">Solr cookbook</field>
</doc>
</add>

```

As you can see, the first three documents have a value in the `price` field, while the last one doesn't. So now, for the purpose of the example, let's assume that our hypothetical user is trying to find books that have `solr` in their `title` field. Besides the search results, we want to show the number of documents that don't have a value in the `price` field. To do that, we send the following query to Solr:

```

http://localhost:8983/solr/select?q=title:solr&facet=true&facet.
query=!price:[* TO *]

```

This query should result in the following output from Solr:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
    <lst name="params">
      <str name="facet">true</str>
      <str name="facet.query">!price:[* TO *]</str>
      <str name="q">title:solr</str>
    </lst>
  </lst>
  <result name="response" numFound="4" start="0">
    <doc>
      <str name="id">3</str>
      <float name="price">15.0</float>
      <str name="title">Solr recipes</str>
    </doc>
    <doc>
      <str name="id">4</str>
      <str name="title">Solr cookbook</str>
    </doc>
  </result>
</response>

```

```
<str name="id">1</str>
<float name="price">11.0</float>
<str name="title">Lucene or Solr ?</str>
</doc>
<doc>
  <str name="id">2</str>
  <float name="price">44.0</float>
  <str name="title">My Solr and the rest of the world</str>
</doc>
</result>
<lst name="facet_counts">
  <lst name="facet_queries">
    <int name="!price:[* TO *]">1</int>
  </lst>
  <lst name="facet_fields"/>
  <lst name="facet_dates"/>
</lst>
</response>
```

As you can see, we got the proper results. Now let's see how it works.

## How it works...

You can see that our index structure defined in the `schema.xml` file is pretty simple. Every book is described by three fields—the `id`, the `title`, and the `price`. Their names speak for the type of information they will hold.

The query is, in most of its parts, something you should be familiar with. First we tell Solr that we are searching for documents that have the word `solr` in the `title` field (the `q=title:solr` parameter). Then we say that we want to have the faceting mechanism enabled by adding the `facet=true` parameter. Then we add a facet query parameter that tells Solr to return the number of documents that don't have a value in the `price` field. We do that by adding the `facet.query=!price:[* TO *]` parameter. How does that work? You should be familiar with how the `facet.query` parameter works; so I'll skip that part. The `price:[* TO *]` expression tells Solr to count all the documents that have a value in the `price` field. By adding the `!` character before the field name, we tell Solr to negate the condition, and in fact we get the number of documents that don't have any value in the specified field.

## How to get all the faceting results, not just the first hundred ones

Let's suppose that we have an e-commerce library and besides the standard search results, we want to show all the categories in which we have books with their numbers. Can Solr do that? Yes it can, and this recipe will show you how to do it.

### Getting ready

Before you start reading this recipe, please take a look at the *Getting the number of documents with the same field value* recipe in this chapter.

### How to do it...

For the purpose of the recipe, let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `category` field to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
<field name="category" type="string" indexed="true" stored="true">
```

The example data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Lucene or Solr ?</field>
    <field name="category">technical</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">My Solr and the rest of the world</field>
    <field name="category">other</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">Solr recipes</field>
    <field name="category">cookbook</field>
  </doc>
  <doc>
    <field name="id">4</field>
```

```
<field name="title">Solr cookbook</field>
<field name="category">cookbook</field>
</doc>
</add>
```

For the purpose of the example, let's assume that our hypothetical user is trying to find books that have `solr` in their `title` field. Besides the search results, we want to show the number of documents that are in each of the category. To do that, we send the following query to Solr:

```
http://localhost:8983/solr/select?q=title:solr&facet=true&facet.
field=category&facet.limit=-1
```

This query resulted in the following response from Solr:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">1</int>
  <lst name="params">
    <str name="facet">true</str>
    <str name="facet.field">category</str>
    <str name="q">title:solr</str>
    <str name="facet.limit">-1</str>
  </lst>
</lst>
<result name="response" numFound="4" start="0">
  <doc>
    <str name="category">cookbook</str>
    <str name="id">3</str>
    <str name="title">Solr recipes</str>
  </doc>
  <doc>
    <str name="category">cookbook</str>
    <str name="id">4</str>
    <str name="title">Solr cookbook</str>
  </doc>
  <doc>
    <str name="category">technical</str>
    <str name="id">1</str>
    <str name="title">Lucene or Solr ?</str>
  </doc>
  <doc>
    <str name="category">other</str>
    <str name="id">2</str>
```

```

    <str name="title">My Solr and the rest of the world</str>
  </doc>
</result>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="category">
      <int name="cookbook">2</int>
      <int name="other">1</int>
      <int name="technical">1</int>
    </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
</response>

```

As you can see, we got all the faceting results. Now let's see how it works.

### How it works...

The index structure is rather simple—the `schema.xml` file contains three fields which describe every book in the index—the `id` field, the `name` field, and the `category` field. Their names speak for themselves, so I'll skip discussing them.

The query is, in most of its parts, what you have seen in the example in the recipe that described faceting by field. First, we tell Solr that we are searching for documents that have the word `solr` in the `title` field (the `q=title:solr` parameter). Then we say that we want to have the faceting mechanism enabled by adding the `facet=true` parameter. Next, we say that we want to get the number of documents in each category by adding the `facet.field=category` parameter to the query. The last parameter tells Solr to get all the faceting results. It is done by adding the negative value of the `facet.limit` parameter (in our case, we send the `-1` value).

### How to have two different facet limits for two different fields in the same query

Imagine a situation where you have a database of cars in your application. Besides the standard search results, you want to show two faceting by field results. First of those two faceting results—the number of cars in each category, should be shown without any limits, while the second faceting result—the one showing the cars by their manufacturer, should be limited to the maximum of 10 results. Can we achieve it in one query? Yes we can, and this recipe will show you how to do it.



## Getting ready

Before you start reading this recipe, please take a look at the *Getting the number of documents with the same field value* and *How to get all the faceting results, not just the first hundred ones* recipes in this chapter.

## How to do it...

For the purpose of the recipe, let's assume that we have the following index structure (just add this to your `schema.xml` file to the field definition section; we will use the `category` and `manufacturer` fields to do the faceting):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="category" type="string" indexed="true" stored="true" />
<field name="manufacturer" type="string" indexed="true" stored="true"
/>
```

The example data looks as follows:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Super Mazda car</field>
    <field name="category">sport</field>
    <field name="manufacturer">mazda</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Mercedes Benz car</field>
    <field name="category">limousine</field>
    <field name="manufacturer">mercedes</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Bentley car</field>
    <field name="category">limousine</field>
    <field name="manufacturer">bentley</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="name">Super Honda car</field>
    <field name="category">sport</field>
    <field name="manufacturer">honda</field>
```

```
</doc>
</add>
```

For the purpose of the example, let's assume that our hypothetical user is trying to search the index for the word *car*. To do that, we should send the following query to Solr:

```
http://localhost:8983/solr/select?q=name:car&facet=true&facet.
field=category&facet.field=manufacturer&f.category.facet.limit=-1&f.
manufacturer.facet.limit=10
```

This query resulted in the following response from Solr:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">1</int>
  <lst name="params">
    <str name="f.manufacturer.facet.limit">10</str>
    <str name="facet">true</str>
    <str name="q">name:car</str>
    <str name="f.category.facet.limit">-1</str>
    <arr name="facet.field">
      <str>category</str>
      <str>manufacturer</str>
    </arr>
  </lst>
</lst>
<result name="response" numFound="4" start="0">
  <doc>
    <str name="category">limousine</str>
    <str name="id">3</str>
    <str name="manufacturer">bentley</str>
    <str name="name">Bentley car</str>
  </doc>
  <doc>
    <str name="category">sport</str>
    <str name="id">1</str>
    <str name="manufacturer">mazda</str>
    <str name="name">Super Mazda car</str>
  </doc>
  <doc>
    <str name="category">limousine</str>
    <str name="id">2</str>
    <str name="manufacturer">mercedes</str>
    <str name="name">Mercedes Benz car</str>
```

```
</doc>
<doc>
  <str name="category">sport</str>
  <str name="id">4</str>
  <str name="manufacturer">honda</str>
  <str name="name">Super Honda car</str>
</doc>
</result>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="category">
      <int name="limousine">2</int>
      <int name="sport">2</int>
    </lst>
    <lst name="manufacturer">
      <int name="bentley">1</int>
      <int name="honda">1</int>
      <int name="mazda">1</int>
      <int name="mercedes">1</int>
    </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
</response>
```

Now let's see how it works.

## How it works...

Our data is very simple. As you can see in the field definition section of the `schema.xml` file and the example data, every document is described by four fields—`id`, `name`, `category`, and `manufacturer`. I think that their names speak for themselves and I don't need to discuss them.

The initial parts of the query are usual. We ask for documents which have the word `car` in their `name` field. Then we tell Solr to enable faceting (the `facet=true` parameter) and we tell which field will be used to calculate the faceting results (the `facet.field=category` and the `facet.field=manufacturer` parameters). Then we specify the limits. By adding the parameter limits in a way shown in the example (`f.FIELD_NAME.facet.limit`), we tell Solr to set the limits for the faceting calculation for the particular field. In our example query, by adding the `f.category.facet.limit=-1` parameter, we told Solr that we don't want any limits on the number of faceting results for the `category` field. By adding the `f.manufacturer.facet.limit=10` parameter, we told Solr that we want a maximum of 10 faceting results for the `manufacturer` field.

Following the pattern, you can specify per field values for faceting properties like sorting, minimum count, and so on.



# 7

## Improving Solr Performance

In this chapter, we will cover:

- ▶ Paging your results quickly
- ▶ Configuring the document cache
- ▶ Configuring the query result cache
- ▶ Configuring the filter cache
- ▶ Improving Solr performance right after the startup or commit operation
- ▶ Setting up a sharded deployment
- ▶ Caching whole result pages
- ▶ Improving faceting performance
- ▶ What to do when Solr slows down during indexing when using Data Import Handler
- ▶ Getting the first top document fast when having millions of them

### Introduction

Performance of the application is one of the most important factors. Of course, there are other factors such as usability, availability, and so on, but one of the most crucial and major among them is performance. Even if our application is perfectly done in terms of usability, the users won't be able to use it if they have to wait long for the search results.

The standard Solr deployment is fast enough, but sooner or later the time will come when you will have to optimize your deployment. This chapter and its recipes will try to help you with the optimization of Solr deployment.

If your business depends on Solr, you should keep monitoring it even after optimization. For that, you can use generic and free tools like Ganglia (<http://ganglia.sourceforge.net/>) or Solr-specific ones like Scalable Performance Monitoring from Sematext (<http://sematext.com/spm/index.html>).

## Paging your results quickly

Imagine a situation where you have a user constantly paging through the search results. For example, one of the clients I was working for was struggling with the performance of his website. His users tend to search for a word and then page through the result pages — the statistical information gathered from the application logs showed that the typical user was changing the page about four to seven times. We decided to optimize paging. How to do that? This recipe will show you how.

### How to do it...

So let's get back to my client deployment. As I mentioned, the typical user typed a word into the search box and then used the paging mechanism to go through a maximum of seven pages. My client's application was showing 20 documents on a single page. So, it can be easily calculated that we need about 140 documents in advance, besides the first 20 documents returned by the query.

So what we did was actually pretty simple. We modified the `queryResultWindowSize` property in the `solrconfig.xml` file. We changed it to the following value:

```
<queryResultWindowSize>160</queryResultWindowSize>
```

Then we changed the maximum number of documents that can be cached for a single query to 160 by adding the following property to the `solrconfig.xml` file:

```
<queryResultMaxDocsCached>160</queryResultMaxDocsCached>
```

We also modified `queryResultCache`, but that's a discussion for another recipe. To learn how to change that cache, please refer to the *Configuring the query result cache* recipe in this chapter.

### How it works...

So how does Solr behave with the preceding proposed changes? First of all, `queryResultWindowSize` tells Solr to store (in `documentCache`) a maximum of 160 documents with every query. Therefore, after doing the initial query, we gather many more documents than we need. Due to this, we are sure that when the user clicks on the **next page** button that is present in our application, the results will be taken from the cache, so there won't be a need for a Lucene query and we will save some CPU time and I/O operations. You must remember that the 160 documents will be stored in cache and won't be visible in the results list as the result size is controlled by the `rows` parameter.

The `queryResultMaxDocsCached` property tells Solr about the maximum number of documents that can be cached for a single query (please remember that in this case, cache stores the document identifiers and not the whole document). We told Solr that we want a maximum of 160 documents for a single query because the statistics showed us that we don't need more, at least for a typical user.

Of course there is another thing that should be done—setting the query result cache size, but that is discussed in another recipe.

## Configuring the document cache

Cache can play a major role in your deployment's performance. One of the caches that you can use for configuration when setting up Solr is the document cache. It is responsible for storing the Lucene internal documents that have been fetched from the disk. The proper configuration of this cache can save precious I/O calls and therefore boost the whole deployment performance. This recipe will show you how to properly configure the document cache.

### How to do it...

For the purpose of the recipe, I assumed that we are dealing with the deployment of Solr where we have about 100,000 documents. In our case, a single Solr instance is getting a maximum of 10 concurrent queries and the maximum number of documents that the query can fetch is 256.

With the preceding parameters, our document cache should look like this (add this to the `solrconfig.xml` configuration file):

```
<documentCache
  class="solr.LRUCache"
  size="2560"
  initialSize="2560"/>
```

Notice that we didn't specify the `autowarmCount` parameter because the document cache uses Lucene's internal ID to identify documents. These identifiers can't be copied between index changes and thus we can't automatically warm this cache.



## How it works...

The document cache configuration is simple. We define it in the `documentCache` XML tag and specify a few parameters that define the document cache behavior. First of all, the `class` parameter tells Solr which Java class to use as the implementation. In our example, we use `solr.LRUCache` because we think that we will be putting more information into the cache than we will get from it. When you see that you are getting more information than you put, consider using `solr.FastLRUCache`. The next parameter tells Solr about the maximum size of the cache (the `size` parameter). As the Solr, wiki says we should always set this value to more than the maximum number of results returned by the query, multiplied by the maximum concurrent queries that we think will be sent to the Solr instance. This will ensure that we always have enough space in the cache, so that Solr will not have to fetch the data from the index multiple times during a single query.

The last parameter tells Solr about the initial size of the cache (the `initialSize` parameter). I tend to set it to the same value as the `size` parameter to ensure that Solr won't be wasting its resources on cache resizing.

There is one thing to remember: The more fields you have marked as 'stored' in the index structure, the higher the memory usage of this cache will be.

Please remember that when using the values shown in the example, you always observe your Solr instance and act when you see that your cache is acting in the wrong way.

## Configuring the query result cache

The major Solr role in a typical e-commerce website is handling the user queries. Of course, users of the site can type multiple queries in the search box and we can't easily predict how many unique queries there may be. However, using the logs that Solr gives us we can check how many different queries there were in the last day, week, month, or year. Using this information, we can configure the query result cache to suit our need in the most optimal way and this recipe will show you how to do it.

## How to do it...

For the purpose of the recipe, let's assume that one Solr instance of our e-commerce website is handling about 10 to 15 queries per second. Each query can be sorted by four different fields (the user can choose by which field). The user can also choose the order of sort. By analyzing the logs for the past three months, we know that there are about 2,000 unique queries that users tend to type in the search box of our application. We also noticed that our users don't usually use the paging mechanism.

On the basis of the preceding information, we configured our query results cache as follows (add this to the `solrconfig.xml` configuration file):

```
<queryResultCache
  class="solr.LRUCache"
  size="16000"
  initialSize="8000"
  autowarmCount="4000"/>
```

## How it works...

Adding the query result cache to the `solrconfig.xml` file is a simple task. We define it in the `queryResultCache` XML tag and specify a few parameters that define the query result cache behavior. First of all, the `class` parameter tells Solr which Java class to use as the implementation. In our example, we use `solr.LRUCache` because we think that we will be putting more information into the cache than we will get from it. When you see that you are getting more information than you put, consider using `solr.FastLRUCache`. The next parameter tells Solr about the maximum size of the cache (the `size` parameter). This cache should be able to store the ordered identifiers of the objects that were returned by the query with its sort parameter and the range of documents requested. This means that we should take the number of unique queries, multiply it by the number of sort parameters and the number of possible orders of sort. So in our example, the size should be at least the result of this equation:

$$\text{size} = 2000 * 4 * 2$$

Which in our case is 16,000.

I tend to set the initial size of this cache to half of the maximum size, so in our case, I set the `initialSize` parameter to a value of 8,000.

The last parameter (the `autowarmCount` parameter) says how many entries should be copied when Solr is invalidating caches (for example, after a commit operation). I tend to set this parameter to a quarter of the maximum size of the cache.

Please remember that when using the values shown in the example, you always observe your Solr instance and act when you see that your cache is acting in the wrong way.

## Configuring the filter cache

Almost every client of mine who uses Solr tends to forget, or simply doesn't know how to use filter queries or simply filters. People tend to add another clause with a logical operator to the main query because they forget how efficient filters can be, at least when used wisely. That's why whenever I can, I tell people using Solr to use filter queries. However, when using filter queries, it is nice to know how to set up a cache that is responsible for holding the filters results—the filter cache. This recipe will show you how to properly set up the filter cache.

## How to do it...

For the purpose of this recipe, let's assume that we have a single Solr slave instance to handle all the queries coming from the application. We took the logs from the last three months and analyzed them. From that we know that our queries are making about 2,000 different filter queries. By getting this information, we can set up the filter cache for our instance. This configuration should look like this (add this to the `solrconfig.xml` configuration file):

```
<filterCache
  class="solr.FastLRUCache"
  size="2000"
  initialSize="1000"
  autowarmCount="1000"/>
```

And that's it. Now let's see what those values mean.

## How it works...

As you may have noticed, adding the filter cache to the `solrconfig.xml` file is a simple task. You just need to know how many unique filters your Solr instance is receiving. We define it in the `filterCache` XML tag and specify a few parameters that define the query result cache behavior. First of all, the `class` parameter tells Solr which Java class to use for implementation. In our example, we use `solr.FastLRUCache` because we think that we will get more information from the cache than we will put into it. The next parameter tells Solr about the maximum size of the cache (the `size` parameter). In our case we said that we have about 2,000 unique filters and we set the maximum size to that value. This is done because each entry of the filter cache stores the unordered sets of Solr document identifiers that match the given filter. This way, after the first use of the filter, Solr can use the filter cache to apply filtering and thus save the I/O operations.

The next parameter, `initialSize`, tells Solr about the initial size of the filter cache. I tend to set it to half the value of the `size` parameter. So in our example, we set it to the value of 1,000.

The last parameter (the `autowarmCount` parameter) says how many entries should be copied when Solr is invalidating caches (for example, after a commit operation). I tend to set this parameter to a quarter of the maximum size of the cache.

Please remember that when using the values shown in the example, you always observe your Solr instance and act when you see that your cache is behaving the wrong way.

## See also

If you want to use the field cache faceting type, please refer to the *Improving faceting performance* recipe in this chapter.

## Improving Solr performance right after the startup or commit operation

Almost everyone who has some experience with Solr would have noticed one thing right after startup—Solr doesn't have such query performance as after running a while. This is happening because Solr doesn't have any information stored in caches, the I/O is not optimized, and so on. Can we do something about it? Of course, we can. And this recipe will show you how.

## How to do it...

First of all, we need to identify the most common and the heaviest queries that we send to Solr. I have two ways of doing this: first, I analyze the logs that Solr produces and see how queries are behaving. I tend to choose those queries that are run often and those that run slowly. The second way of choosing the right queries is analyzing the applications that use Solr and see what queries they produce, what queries will be the most crucial, and so on. Based on my experience, the log-based approach is usually much faster and can be done with the use of self-written scripts.

However, let's assume that we have identified the following queries as good candidates:

```
q=cats&fq=category:1&sort=title+desc,value+desc,score+desc
q=cars&fq=category:2&sort=title+desc
q=harry&fq=category:4&sort=score+desc
```

What we will do is just add the so-called warming queries to the `solrconfig.xml` file. So the listener XML tag definition in the `solrconfig.xml` file should look like this:

```
<listener event="firstSearcher" class="solr.QuerySenderListener">
  <arr name="queries">
    <lst><str name="q">cats</str><str name="fq">category:1</str><str name="sort">title desc,value desc,score desc</str><str name="start">0</str><str name="rows">20</str></lst>
    <lst><str name="q">cars</str><str name="fq">category:2</str><str name="sort">title desc</str><str name="start">0</str><str name="rows">20</str></lst>
    <lst><str name="q">harry</str><str name="fq">category:4</str><str name="sort">score desc</str><str name="start">0</str><str name="rows">20</str></lst>
  </arr>
</listener>
```

Basically, what we did is add the so-called warming queries to the startup of Solr. Now let's see how it works.

## How it works...

By adding the above fragment of configuration to the `solrconfig.xml` file, we told Solr that we want it to run those queries whenever a `firstSearcher` event occurs. The `firstSearcher` event is fired whenever a new searcher object is prepared and there is no searcher object available in the memory. So basically, the `firstSearcher` event occurs right after Solr startup.

So what is happening after the Solr startup? After adding the preceding fragment, Solr is running each of the defined queries. By doing that, the caches are populated with the entries that are significant for the queries that we identified. This means that if we did the job right, we'll have Solr configured and ready to handle the most common and most heavy queries right after its startup.

Maybe a few words about what do all the configuration options mean. The warm-up queries are always defined under the `listener` XML tag. The `event` parameter tells Solr what event should trigger the queries. In our case it is the `firstSearcher` event. The `class` parameter is the Java class that implements the `listener` mechanism. Next, we have an array of queries that are bound together by the `array` tag with the `name="queries"` parameter. Each of the warming queries is defined as a list of parameters that are grouped by the `lst` tag.

## There's more...

There is one more thing.

### Improving Solr performance after commit operations

If you are interested in tuning up your Solr instance performance, you should also look at the `newSearcher` event. This event occurs whenever a commit operation is performed by Solr (for example, after replication). Assuming we identified the same queries as before, as good candidates to warm the caches, we should add the following entries to the `solrconfig.xml` file:

```
<listener event="newSearcher" class="solr.QuerySenderListener">
  <arr name="queries">
    <lst><str name="q">cats</str><str name="fq">category:1</str><str name="sort">title desc,value desc,score desc</str><str name="start">0</str><str name="rows">20</str></lst>
    <lst><str name="q">cars</str><str name="fq">category:2</str><str name="sort">title desc</str><str name="start">0</str><str name="rows">20</str></lst>
    <lst><str name="q">harry</str><str name="fq">category:4</str><str name="sort">score desc</str><str name="start">0</str><str
```

```
name="rows">20</str></lst>
</arr>
</listener>
```

Please remember that the warming queries are especially important for the caches that can't be automatically warmed.

## Setting up a sharded deployment

Imagine that you have to develop an application that will be able to store and search all the books that are available in the libraries across the world. This is the case when a single Solr instance will not be sufficient. Fortunately, there is a Solr mechanism that allows us to have many Solr instances not only in the master-slave architecture, but for the purpose of storing different data sets in different instances. This recipe will show you how to deploy such a setup.

### How to do it...

For the purpose of the recipe, let's assume that we will need two separate Solr instances that will play our shard's role. Those instances will be installed on the same machine. One instance will run on the 8983 port, the second instance will run on the 8984 port.

Both instances will have the same `schema.xml` file. The index structure is defined as follows (just add this to the field definition section of the `schema.xml` file on both the instances):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="title" type="text" indexed="true" stored="true" />
```

Now let's get to our data. For the purpose of this recipe, we will be using small amount of data—two documents per shard. The data that will be sent to the first shard looks similar to the following code:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="title">Book 1</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="title">Book 2</field>
  </doc>
</add>
```

While the data that will be sent to the second shard looks similar to this:

```
<add>
  <doc>
    <field name="id">3</field>
    <field name="title">Book 3</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="title">Book 4</field>
  </doc>
</add>
```

And now, let's test our example. For the purpose of the test, we will ask Solr to return all the indexed documents from all shards. To do that, send the following query to the first or second shard:

```
http://localhost:8983/solr/select/?q=*&shards=localhost:8983/
solr,localhost:8984/solr
```

The results should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">13</int>
    <lst name="params">
      <str name="q">*</str>
      <str name="shards">localhost:8983/solr,localhost:8984/solr</str>
    </lst>
  </lst>
  <result name="response" numFound="4" start="0">
    <doc>
      <str name="id">1</str>
      <str name="title">Book 1</str>
    </doc>
    <doc>
      <str name="id">2</str>
      <str name="title">Book 2</str>
    </doc>
    <doc>
      <str name="id">3</str>
      <str name="title">Book 3</str>
    </doc>
    <doc>
      <str name="id">4</str>
      <str name="title">Book 4</str>
```

```
</doc>
</result>
</response>
```

As you can see, Solr returned four documents which means that we got the data from the first and second shard. Now let's see how it works.

### How it works...

As you can see, the `schema.xml` file is very simple. It just contains two fields. One is (the `id` field) responsible for holding the unique identifier of the document, and the second one (the `title` field) is responsible for holding the title of the document.

For the purpose of the example, I assumed that we can divide the data—two documents for one shard and two documents for the other shard. In real life, it is not so easy. When setting up sharding, you should remember two things:

- ▶ The specific document should always be sent for indexing the same shard. For example, we can send the documents with the even identifiers to the first shard and documents with odd identifiers to the second shard.
- ▶ A single document should always be sent to one instance only.

And the division of data is the responsibility of a data source. Solr won't do that automatically.

The data is very simple, so I decided to skip commenting on it, but notice that we have two data files—one for the first shard and the other for the second shard.

Now let's look at the query. As you may have noticed, we send the query to the first shard, but we could also send it to any of the shards. The first parameter (the `q=*:*` parameter) tells Solr that we want to get all the documents. Then we have the `shards` parameter. It tells Solr what shards should be queried to obtain the results. By adding the `shards=localhost:8983/solr,localhost:8984/solr` parameter, we told Solr that we want to get the results from the two shards. The value of the parameter takes the value of the list of addresses of shards, where each address is followed by a comma character. After getting such a request, Solr will pass the query to each shard, fetch the results, process them, and return the combined list of documents.

You should remember one thing: gathering and processing results from shards is not costless, actually that's a complicated process. Some mechanisms that are available for a standard (not sharded) deployment are not available when using shards (for example, `QueryElevationComponent`). You should always consider the pros and cons about sharding and decide if you really need that kind of deployment.

The preceding results list is nothing unusual. We can see that there were four documents returned which means that the sharding mechanism works as it should.



## There's more...

There is one more thing.

### Dealing with queries taking too much time

One of the common usages of the sharding mechanism is to shard indexes when a single instance of Solr can't handle the query in a reasonable time. The setup is identical—you decide which documents should be indexed at what shard and then you make a query to all shards.

## Caching whole result pages

Imagine a situation where you have an e-commerce library and your data rarely changes. What can you do to take the stress of your search servers? The first thing that comes to mind is caching, for example the HTTP caching. And yes, that is a good point. But do we have to set up external caches prior to Solr? Or, can we tell Solr to use its own caching mechanism? Of course, we can use Solr to cache whole results pages and this recipe will show you how to do it.

## Getting ready

Before you continue reading this recipe, it would be nice for you to know some basics about the HTTP cache headers. To learn more about it, please refer to the RFC document, which can be found on the W3 site: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html>

## How to do it...

So let's configure the HTTP cache. To do this, we need to configure the Solr request dispatcher. Let's assume that our index is changing every 60 minutes. Replace the request dispatcher definition in the `solrconfig.xml` file with the following contents:

```
<requestDispatcher handleSelect="true">
  <httpCaching lastModifiedFrom="openTime" etagSeed="Solr">
    <cacheControl>max-age=3600, public</cacheControl>
  </httpCaching>
</requestDispatcher>
```

After sending a query like this to the local instance of Solr:

```
http://localhost:8983/solr/select?q=book
```

We got the following HTTP headers:

```
Cache-Control:max-age=3600, public
Content-Length:442
Content-Type:text/xml; charset=utf-8
Etag:"YmNlZWZhYjRiNDgwMDAwMFNvbHI="
Expires:Tue, 28 Dec 2010 15:19:19 GMT
Last-Modified:Tue, 28 Dec 2010 14:19:11 GMT
Server:Jetty(6.1.26)
```

And by this we can tell that cache works.

## How it works...

The cache definition is defined inside the `requestDispatcher` XML tag. `handleSelect="true"` tells us about error handling and it should be set to the `true` value. Then we see the `httpCaching` tag (notice the lack of the `<httpCaching` `never304="true">` XML tag), which actually configures the HTTP caching in Solr. The `lastModifiedFrom="openTime"` attribute defines that the Last Modified HTTP header will be relative to when the current searcher object was opened (for example, relative to the last replication execution date). You can also set this parameter value to `dirLastMod` to be relative to when the physical index was modified. Next, we have the `eTagSeed` attribute that is responsible for generating the `ETag` HTTP cache header.

The next configuration tag is the `cacheControl` tag, which can be used to specify the generation of the cache control HTTP headers. In our example, adding the `max-age=3600` parameter tells Solr that he should generate the additional HTTP cache header, which will tell that the cache is valid for a maximum of one hour. The `public` directive means that the response can be cached by any cache type.

As you can see from the response, the headers we got as a part of the results returned by Solr tell us that we got what we wanted.

## Improving faceting performance

Imagine a situation where you stored the data about the companies and their localization in the Solr index. There are two fields that you use to show the faceting results for the user — the city in which the company is located and the state in which the company is located. Is there a way to boost the performance of that kind of a query? Yes there is! And this recipe will show you show to do it.

## Getting ready

Before you continue reading this recipe, please take the time to read the recipe *How to have two different facet limits for two different fields in the same query* discussed in Chapter 6, *Using Faceting Mechanism*.

## How to do it...

Let's assume that besides the search field, we have two additional fields in our index, whose definition looks similar to the following:

```
<field name="facetCity" type="string" indexed="true" stored="true"/>
<field name="facetState" type="string" indexed="true" stored="true"/>
```

So the actual query to show the search results and the faceting results may look like this:

```
http://localhost:8983/solr/select/?q=company&facet=true&facet.
field=facetCity&facet.field=facetState
```

I assume that the number of cities and states in the index is relatively small (not more than 50). Based on this assumption, let's modify the preceding query and let's boost its performance. The modified query looks like this:

```
http://localhost:8983/solr/select/?q=company&facet=true&facet.
field=facetCity&facet.field=facetState&f.facetCity.facet.
method=enum&f.facetState.facet.method=enum
```

Will it boost the query performance? Yes it should.

## How it works...

The index is pretty simple, so I'll skip discussing that.

Let's take a look at the first query. It is a standard query using the field faceting. We have two fields that we want facets to be calculated for and we search for the word `company`.

Note that the second query contains additional fragments: `f.facetCity.facet.method=enum` and `f.facetState.facet.method=enum`. So we specified the additional parameter for both facet fields. Basically, what it does is tell Solr to iterate over the terms (we can say words) that are present in that field to calculate the faceting. To get more into the technical details, Solr runs a filter query for each indexed term in the field. We see the performance improvement when the filter queries are cached. In our case, in the fields that have a small amount of terms present, such as our fields, this method should boost the performance of the facet field query.

The presented method is always used when doing field faceting on the fields that are based on the `Boolean` type.

## What to do when Solr slows down during indexing when using Data Import Handler

One of the most common problems when indexing a vast amount of data is the indexing time. Some of the problems with indexing time are not easily resolvable, but others are. Imagine you need to index about 300,000 documents that are in a single XML file. You run the indexing using Data Import Handler and you wait for a very long time. Something is wrong—when you index 10,000 documents, you need about a minute, but now you are waiting for about an hour and the commit operation didn't take place. Is there something we can do to speed it up? Sure, and this recipe will tell you what.

### How to do it...

The solution to the situation is very simple: just add the commit operation every now and then. However, as you may have noticed, I mentioned that our data is written in a single XML file. So, how to add the commit operation to that kind of data? Send it parallel to the indexing process? No, we need to enable the `autoCommit` mechanism. To do that, let's modify the `solrconfig.xml` file and change the update handler definition to the following:

```
<updateHandler class="solr.DirectUpdateHandler2">
  <autoCommit>
    <maxTime>60000</maxTime>
  </autoCommit>
</updateHandler>
```

If you were to start the indexing described in the indexing process, you would notice that there is a commit command being sent once a minute while the indexing process is taking place. Now let's see how it works.

### How it works...

When using Data Import Handler, Solr tends to slow down the indexing process when indexing a vast amount of data without the commit commands being sent once in a while. This behavior is completely understandable. It is bound to the memory and how much of it Solr can use.

We can avoid the slowing down behavior by adding a commit command after the set amount of time or the set amount of data. In this recipe, we choose the first approach.

We assumed that it would be good to send the commit command once every minute. So we add the `<autoCommit>` section with the `<maxTime>` XML tag set to the value of `60000`. This value is specified in milliseconds. And that's all we need to do. After this change, Solr will send the commit command after every one minute has passed during the indexing operation and we don't have to worry that Solr indexing speed will decrease over time.

## There's more...

### Commit after a set amount of documents

Sometimes there is a need not to rely on the time between the commit operations, but on the amount of the documents that were indexed. If this is the case, we can choose to automatically send the commit command after a set amount of documents being processed. To do this, we add the `<maxDocs>` XML tag with the appropriate amount. For example, if we want to send the commit command after every 50,000 documents, the update handler configuration should look like this:

```
<updateHandler class="solr.DirectUpdateHandler2">
  <autoCommit>
    <maxDocs>50000</maxDocs>
  </autoCommit>
</updateHandler>
```

### Commit within a set amount of time

There may be a situation when you want some of the documents to be committed faster than the `autoCommit` settings. In order to do that, you can add the `commitWithin` attribute to the `<add>` tag of your data XML time. This attribute will tell Solr to commit the documents within the specified time (specified in milliseconds). For example, if we wanted the portion of documents to be indexed within 100 milliseconds, our data file would look like this:

```
<add commitWithin="100">
  <doc>
    <field name="id">1</field>
    <field name="title">Book 1</field>
  </doc>
</add>
```

## Getting the first top documents fast when having millions of them

Imagine a situation where you have millions of documents in the sharded deployment and you are quite satisfied about the response time from Solr when you are using the default, relevancy sorting. But there is another option in your system—sorting on the basis of the author of the document. And when doing that kind of sorting, the performance is not as good as it should be. So, is there something we can do? Yes, we can use the time limiting collector. This recipe will show you how use it.

## Getting ready

Before you continue reading this recipe, please take the time to read the recipe *Getting the most relevant results with early query termination* discussed in *Chapter 1, Apache Solr Configuration* and the *Setting up a sharded deployment* recipe discussed in this chapter.

## How to do it...

For the purpose of this recipe, the actual data doesn't matter. We need to focus on two things:

- ▶ Pre-sorting the index
- ▶ Setting up the time limiting collector

The pre-sorting index technique is described in the recipe I mentioned. So I assume that we have our index ready.

So what's left is the time limiting collector setup. To do that, we can choose two ways: one is to add the appropriate parameter to the request handler definition. The second is to pass the parameter to the query. I choose the second approach: I add the `timeAllowed` parameter with the value of 100 to each query. So my example query can look like this:

```
http://localhost:8983/solr/select?q=financial report&timeAllowed=100
```

And that's all you need to do. Now let's see how it works.

## How it works...

There is a new parameter shown in the recipe, the `timeAllowed` parameter. It allows us to specify the maximum time of execution for a query and is represented in milliseconds. For the purpose of the recipe, I told Solr that we want the query to be running for a maximum of 100 milliseconds. If you don't want any time limits, just don't add this parameter or add it with the value less or equal to 0.

However, getting back to the query: What does it do? We tell Solr to search for the documents that contain two words, `financial` and `report`, in the default search field. Then we set the maximum query execution time by adding the `timeAllowed` parameter. The last parameter is responsible for specifying the shards that will be queried to gather the search results.

Please note that the `timeAllowed` parameter can also be used in a non-sharded deployment.



# 8

## **Creating Applications that use Solr and Developing your Own Solr Modules**

In this chapter, we will cover:

- ▶ Choosing a different response format than the default one
- ▶ Using Solr with PHP
- ▶ Using Solr with Ruby
- ▶ Using SolrJ to query Solr
- ▶ Developing your own request handler
- ▶ Developing your own filter
- ▶ Developing your own search component
- ▶ Developing your own field type

### **Introduction**

Sometimes Solr is just not enough. Well, let's face the truth—you don't deploy Solr and Solr alone. In almost every deployment, Apache Solr is only a piece of the whole system. It's a piece used by other parts to get search results. And this is what this chapter begins with: the information on how to overcome the usual problems and the use of Solr within different applications.



But that's not all. Sometimes, the standard Solr features are just not enough. This chapter will also guide you through the world of Solr module development, to help you deal with the common problems by showing you how to implement the most common Solr features, such as filters or fields.

## Choosing a different response format than the default one

Imagine a simple situation: you fetch so much data from Solr that using the XML format is not an option—it being too heavy. You send hundreds of requests every second and fetch a vast amount of data from Solr and you find yourself in a situation where the network speed becomes an issue. That's where other response formats come in handy, and this recipe will show you how to choose a different response format during query.

### How to do it...

Let's start from the beginning. Suppose we have the following query:

```
http://localhost:8983/solr/select?q=*:*&rows=1
```

This query will result in the following response from Solr:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
    <lst name="params">
      <str name="q">*:*</str>
      <str name="rows">1</str>
    </lst>
  </lst>
  <result name="response" numFound="19" start="0">
    <doc>
      <arr name="cat"><str>electronics</str><str>hard drive</str></arr>
      <arr name="features"><str>7200RPM, 8MB cache, IDE Ultra ATA-133</str><str>NoiseGuard, SilentSeek technology, Fluid Dynamic Bearing (FDB) motor</str></arr>
      <str name="id">SP2514N</str>
      <bool name="inStock">true</bool>
      <str name="manu">Samsung Electronics Co. Ltd.</str>
      <date name="manufacturedate_dt">2006-02-13T15:26:37Z</date>
      <str name="name">Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133</str>
    </doc>
  </result>
</response>
```

```

    <int name="popularity">6</int>
    <float name="price">92.0</float>
  </doc>
</result>
</response>

```

This is the standard XML response from Solr. So, now let's change the response format to JSON. To do that, let's add the `wt=json` parameter to the preceding query. Now the query would look like this:

```
http://localhost:8983/solr/select?q=*&rows=1&wt=json
```

The result of the modified query should look like this:

```

{
  "responseHeader": {
    "status": 0,
    "QTime": 0,
    "params": {
      "wt": "json",
      "q": ":*:*",
      "rows": "1"
    }
  },
  "response": { "numFound": 19, "start": 0, "docs": [
    {
      "id": "SP2514N",
      "name": "Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133",
      "manu": "Samsung Electronics Co. Ltd.",
      "price": 92.0,
      "popularity": 6,
      "inStock": true,
      "manufacturedate_dt": "2006-02-13T15:26:37Z",
      "cat": [
        "electronics",
        "hard drive"
      ],
      "features": [
        "7200RPM, 8MB cache, IDE Ultra ATA-133",
        "NoiseGuard, SilentSeek technology, Fluid Dynamic Bearing (FDB) motor"
      ]
    }
  ]
}

```

As you can see, the format is completely different and that's what we wanted to achieve. Now let's see how it works.

## How it works...

The first query and the first example response are the standard ones from Solr. They use the default response handler and thus the default response format, XML. The second query is what we are interested in. We added a new parameter `wt` with the value `json`. This parameter allows us to specify the name of the response writer that we want to generate. In our example, we have chosen `JSON Response Writer` by passing the `wt=json` parameter to the query. As you can see, the response format is totally different from XML and it's actually the JSON format. So that's what we wanted to achieve. (To learn more about Solr JSON format, please take a look at the following URL address: <http://wiki.apache.org/solr/SolrJSON>.)

## Using Solr with PHP

As you already know, Solr has an API exposed as an HTTP interface. With the use of several provided response writers, we can use virtually any language to connect to Solr, send a query, and fetch the results. In the following recipe, I'll show you how to prepare Solr to use it with PHP, and how to use the PHP language to fetch data from Solr.

## Getting ready

To illustrate how to use Solr within a PHP application, I decided to use `SolrPHPClient` written by Donovan Jimenez. This can be found at the following address: <http://code.google.com/p/solr-php-client/downloads/list>

## How to do it...

Let's start with the `schema.xml` file. We will only need two fields. So the field definition part of the file should look like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
```

Now let's index some data by running the following PHP file:

```
<?php
require_once 'Apache/Solr/Service.php';

$solr = new Apache_Solr_Service("192.168.0.1", "8983", "/solr");

$docs = array(
    'doc1' => array(
        'id' => '1',
        'name' => 'Document one',
    ),
);
```

---

```

        'doc2' => array(
            'id' => 2,
            'name' => 'Document two',
        ),
    );
    $documents = array();
    foreach ( $docs as $item => $fields ) {
        $solrdoc = new Apache_Solr_Document();
        foreach ( $fields as $key => $value ) {
            $solrdoc->$key = $value;
        }
        $documents[] = $solrdoc;
    }
    try {
        $solr->addDocuments( $documents );
        $solr->commit();
        $solr->optimize();
    }
    catch ( Exception $ex ) {
        echo $ex->getMessage();
    }
}
?>

```

I decided to run the following query and fetch the results using PHP:

```
http://192.168.0.1:8983/solr/select?q=document&start=0&rows=10
```

Now we can send an example query to Solr using a simple PHP code. To send the preceding query to Solr and write the name field to the user, I used the following PHP code:

```

<?php
require_once 'Apache/Solr/Service.php';

$solr = new Apache_Solr_Service("192.168.0.1", "8983", "/solr");
$response = $solr->search("document", 0, 10, null);

echo "DOCUMENTS:\n";
foreach ( $response->response->docs as $doc ) {
    echo $doc->id . ":" . $doc->name . "\n";
}
?>

```

The response was as follows:

```

DOCUMENTS:
1:Document one
2:Document two

```

As we can see, everything is working as it should. Now let's take a look at how it works.

## How it works...

As you can see, the index structure is trivial. For the purpose of the recipe, we will use two fields: one to hold the unique identifier of the document (the `id` field) and the other to hold the name of the document (the `name` field).

What you see next is the indexing of the PHP script. We start it with the inclusion of the `SolrPHPClient` library. Then we have the creation of the `Apache_Solr_Service` object that holds the location, port, and context information of Solr. This object will be used to send data to Solr. The next few lines are responsible for the two documents' definition and creation of the array that will be passed to Solr. Finally, in the try-catch block, we send the documents to Solr (`$solr->addDocuments( $documents )`), we run the commit command (`$solr->commit()`), and then we run the optimize command (`$solr->optimize()`). Of course, in the real environment, you would read the documents from the database or some other location (like files). As this was just an example, I decided to use a static document definition. Please remember that optimization should only be used when needed—it's not necessary to use it after every commit operation.

Now, let's head to the query part of the recipe. As you can see, the first two lines are no different from the ones that can be found in the indexing example. The third line is different. We invoke the `search` method from the `$solr` object—it sends a query to Solr, telling that the main query (the `q` parameter) should take the value of `document`, the start parameter should be set to 0, and the `rows` parameter should be set to 10. The last parameter of the `search` method is the array of additional parameters, which in our case is empty. So we pass the `null` value as a parameter. Finally, the `search` method returns the `$response` object that holds the documents fetched from Solr.

The `foreach` loop walks through every document that was fetched from the Solr response and prints their unique identifiers (the ID field read as `$doc->id` from the `$response` object) and their names (the name field read as `$doc->name` from the `$response` object).

## See also

If you want to see alternative ways to connect to Solr using PHP, I suggest visiting the <http://wiki.apache.org/solr/SolrPHP> Solr wiki page.

## Using Solr with Ruby

As you already know, Solr has an API exposed as an HTTP interface. With the use of several provided response writers, we can use virtually any language to connect to Solr, send a query, and fetch the results. In the following recipe, I'll show you how to prepare Solr to use it with Ruby and how to use the Ruby language to fetch data from Solr.

## Getting ready

Before you start with this recipe, please ensure that you have `rsolr` gem installed. Information on how to do that can be found at the following address:

<http://rubygems.org/gems/rsolr>

## How to do it...

Let's start with the `schema.xml` file. We will only need two fields, so the field definition part of the file should look like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
```

Now let's index some data. To do that, I used the following code fragment:

```
require 'rubygems'
require 'rsolr'

solr = RSolr.connect :url => 'http://192.168.0.1:8983/solr'

documents = [
  { :id => 1, :name => 'Document one' },
  { :id => 2, :name => 'Document two' }
]

solr.add documents
solr.update :data => '<commit/>'
```

I decided to run the following query and fetch the documents we indexed. To do that, I typed the following query into the web browser:

```
http://192.168.0.1:8983/solr/select?q=document&start=0&rows=10
```

Now we can send an example query to Solr using a simple Ruby code. To send the preceding query to Solr and write the name field to the user, I used the following portion of the code:

```
require 'rubygems'
require 'rsolr'

solr = RSolr.connect :url => 'http://192.168.0.1:8983/solr'

response = solr.get 'select', :params => {
  :q => 'document',
  :start => 0,
  :rows => 10
}
```

```
puts 'DOCUMENTS'
response["response"] ["docs"].each{|doc| puts doc["id"] + ":" +
doc["name"] }
```

The response of the preceding code was as follows:

```
DOCUMENTS
1:Document one
2:Document two
```

As we have our code working, we can now have a look at how it works.

## How it works...

As you can see, the index structure is trivial. For the purpose of the recipe, we will use two fields: one to hold the unique identifier of the document (the `id` field) and the other to hold the name of the document (the `name` field).

The next thing is to change `solrconfig.xml`. By adding that fragment of configuration, we tell Solr that we want to use the Ruby response format.

Now let's take a look at the first Ruby code. As already mentioned, I use the `rsolr` gem to connect to the Solr instance. The first two lines of the code are the declaration of libraries that we will use in the code. Next, we use the `RSolr` object and its `connect` method to connect to the given URL at which Solr is available. Then we define the `documents` array which will hold the documents that we will send to Solr. The `solr.add documents` line sends the documents to Solr. The last line is responsible for sending the `commit` command. After running this code, we should have two documents in our index.

Now it's time for the query. The first three lines are the same as the ones in the code for updating Solr, so we will skip them. The next block of code is responsible for sending the actual query and fetching the results. We send a query to the `select` handler with the parameters `q=document`, `start=0`, and `rows=10`. Finally, we have the code fragment that is responsible for generating the output. We take each document from the response (`response["response"] ["docs"] .each`) and iterate over it. For each document, we write its unique identifier (the `id` field) and its name (the `name` field) to the standard output (`puts doc["id"] + ":" + doc["name"]`).

## Using SolrJ to query Solr

Let's imagine you have an application written in Java that you want to connect to Solr through an HTTP protocol. With the use of the SolrJ library which is provided with Solr, you can connect to Solr, index data, and fetch results. This recipe will show you how to do it.

## Getting ready

To run the example code provided in this chapter, you will need the following libraries from the standard Solr distribution:

- ▶ `apache-solr-solrj-3.1.1.jar` from the `/dist` directory
- ▶ `commons-codec-1.4.jar`, `commons-httpclient-3.1.jar`, `commons-io-1.4.jar`, `jcl-over-slf4j-1.5.5.jar`, and `slf4j-api-1.5.5.jar` from the `/dist/solrj-lib` directory
- ▶ `slf4j-jdk-1.5.5.jar` from the `/lib` directory

Place these libraries in classpath of your project. You will also need Java 1.6 JDK to compile this example.

## How to do it...

Let's start with the `schema.xml` file. We will only need two fields. So the field definition part of the file should look like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
```

Now let's index some data. To do that, compile and run the following code:

```
package pl.solr;

import java.util.ArrayList;
import java.util.Collection;

import org.apache.solr.client.solrj.impl.CommonsHttpSolrServer;
import org.apache.solr.client.solrj.impl.XMLResponseParser;
import org.apache.solr.common.SolrInputDocument;

public class SolrIndex {
    public static void main(String[] args) throws Exception {
        CommonsHttpSolrServer server = new CommonsHttpSolrServer(
            "http://localhost:8983/solr" );
        server.setParser(new XMLResponseParser());

        SolrInputDocument doc1 = new SolrInputDocument();
        doc1.addField("id", 1);
        doc1.addField("name", "Document one");

        SolrInputDocument doc2 = new SolrInputDocument();
        doc2.addField("id", 2);
        doc2.addField("name", "Document two");

        Collection<SolrInputDocument> docs = new
```



```

ArrayList<SolrInputDocument>() ;
    docs.add(doc1);
    docs.add(doc2);
    server.add(docs);
    server.commit();    }
}

```

Now, as we already indexed our data, we can query Solr to get it. I decided to run the following query and fetch the results:

```
http://192.168.0.1:8983/solr/select?q=document&start=0&rows=10
```

To send the preceding query using SolrJ, I used the following code:

```

package pl.solr;
import java.util.Iterator;
import org.apache.solr.client.solrj.SolrQuery;
import org.apache.solr.client.solrj.impl.CommonsHttpSolrServer;
import org.apache.solr.client.solrj.impl.XMLResponseParser;
import org.apache.solr.client.solrj.response.QueryResponse;
import org.apache.solr.common.SolrDocument;
import org.apache.solr.common.SolrDocumentList;

public class SolrQuerying {
    public static void main(String[] args) throws Exception {
        CommonsHttpSolrServer server = new CommonsHttpSolrServer(
            "http://localhost:8983/solr" );
        server.setParser(new XMLResponseParser());

        SolrQuery query = new SolrQuery();
        query.setQuery("document");
        query.setStart(0);
        query.setRows(10);

        QueryResponse response = server.query( query );
        SolrDocumentList documents = response.getResults();
        Iterator<SolrDocument> itr = documents.iterator();
        System.out.println("DOCUMENTS");
        while (itr.hasNext()) {
            SolrDocument doc = itr.next();
            System.out.println(doc.getFieldValue("id") + ":" + doc.
                getFieldValue("name"));
        }
    }
}

```

The response from running the preceding code is as follows:

```
DOCUMENTS
1:Document one
2:Document two
```

As we have our code working, we can now see how it works.

### How it works...

As you can see, the index structure is trivial. For the purpose of the recipe, we will use two fields: one to hold the unique identifier of the document (the `id` field) and the other to hold the name of the document (the `name` field).

What you see first is the indexing code written in Java. The `main` method starts with the initialization of the connection to Solr with the use of `CommonsHttpSolrServer`. Then we set the connection to use the XML response format (`server.setParser(new XMLResponseParser())`). Of course, we can use the default `javabin` format, but I decided to show you how to change the response format and use the format other than the default one. Next, we create two documents using the `SolrInputDocument` objects. The next operation is the creation of the collection of documents that consists of the two created documents. This newly created collection is then sent to Solr by using the `server.add(docs)` call. The last line of the code sends the `commit` command to the Solr server.

Next, we have the querying code. As with the previous code, we start the `main` method with the connection initialization. Then we create the `SolrQuery` object and set the parameters: the `q` parameter to the document value (`query.setQuery("document")`), the `start` parameter to 0 (`query.setStart(0)`), and the `rows` parameter to 10 (`query.setRows(10)`). Then we send the prepared query to Solr (`server.query(query)`) to get the response. The next lines are responsible for iterating through the document list, and printing the unique identifier of the document and the value of its `name` field to the standard output.

### Developing your own request handler

There are times when the standard Solr features are not enough. For example, you may need to develop your own request handler, such as to add some custom logging to your application. Let's say you want to log additional information about the parameters and the query time. This recipe will show you how to develop your own request handler and use it with Solr.

## Getting ready

To run the example code provided in this chapter, you will need the following libraries from the standard Solr distribution:

- ▶ `apache-solr-solrj-3.1.jar`, `apache-solr-core-3.1.jar`, and `lucene-core-3.1.jar` from the `/dist` directory

Place these libraries in `classpath` of your project. You will also need Java 1.6 JDK to compile this example.

## How to do it...

Let's start with writing a code for our custom request handler. We will add a code that will log some additional things (actually, for the purpose of the recipe, it doesn't matter what things). The custom request handler code looks like this:

```
package pl.solr;

import java.util.logging.Logger;

import org.apache.solr.handler.StandardRequestHandler;
import org.apache.solr.request.SolrQueryRequest;
import org.apache.solr.request.SolrQueryResponse;

public class ExampleRequestHandler extends StandardRequestHandler {
    private static final Logger LOG = Logger.getLogger(
        ExampleRequestHandler.class.toString() );

    public void handleRequestBody(SolrQueryRequest request,
        SolrQueryResponse response)
        throws Exception {
        super.handleRequestBody(request, response);
        LOG.info( "[" + (response.getEndTime() - request.
            getStartTime()) + "]: " + request.getParamString() );
    }
}
```

Now, add the following line to your `solrconfig.xml` file:

```
<requestHandler name="/test" class="pl.solr.ExampleRequestHandler" />
```

Now it's time to add the compiled class to the Solr web application. Of course, you can build a jar library with it, but for the purpose of the recipe, I decided to add just the compiled class file. To do that, I extracted the contents from the `solr.war` file, found in the `webapps` directory of the example deployment, to the `solr` directory. Then I added the compiled `ExampleRequestHandler.class` to the `webapps/solr/WEB-INF/classes/pl/solr` directory.

Now we can run Solr and send queries to our newly developed request handler, which can be found at the following address:

```
http://localhost:8983/solr/test
```

### How it works...

The functionality of the custom code is trivial, but the recipe is all about developing your own request handler, so let's discuss that.

As you can see, the `ExampleRequestHandler` class is extending the `StandardRequestHandler` class. I don't want anything special besides adding the log, so I decided to use that class and extend it. To achieve what we want, we need to override the `handleRequestBody` method and write it so that we can achieve what we want. As you can see, first we call the `handleRequestBody` from the parent class. Then we add the information to the log file through the `LOG.info` call with the appropriate string. And that's basically all that we need to do.

Of course, we could process and do something with the results and so on, but that's not the point here. I suggest you to take a look at the `SearchHandler` class located in the `org.apache.solr.search.component` package to see how that class handles the request and how it constructs the response.

## Developing your own filter

Lucene, and thus Solr, provide a wide variety of filters from simple text analysis to stemming and complicated phonic algorithms. However, sometimes even that can be insufficient. For example, let's assume that you need a filter that will replace the first letter of the token with the second one. To do that, we need to develop our own filter. This recipe will show you how to do that.

### Getting ready

To run the example code provided in this chapter, you will need the following libraries from the standard Solr distribution:

- ▶ `apache-solr-core-3.1.jar` from the `/dist` directory
- ▶ `lucene-analyzers-3.1.jar` and `lucene-core-3.1.jar` from the `/lib` directory

Place these libraries in `classpath` of your project. You will also need Java 1.6 JDK to compile this example.

## How to do it...

Let's start with the code of the actual custom filter. The code should look like this:

```
package pl.solr;

import java.io.IOException;
import org.apache.lucene.analysis.TokenFilter;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.tokenattributes.TermAttribute;

public class ExampleFilter extends TokenFilter {
    private final TermAttribute termAttr = (TermAttribute)
addAttribute(TermAttribute.class);

    public ExampleFilter(TokenStream stream) {
        super(stream);
    }

    @Override
    public boolean incrementToken() throws IOException {
        if (input.incrementToken()) {
            String term = termAttr.term();
            if (term.length() <= 1) {
                return true;
            }
            StringBuffer buffer = new StringBuffer();
            buffer.append(term.charAt(1)).append(term.charAt(0)).
append(term.substring(2));
            termAttr.setTermBuffer(buffer.toString());
            termAttr.setTermLength(buffer.length());
            return true;
        }
        return false;
    }
}
```

Now, let's implement the factory for our custom filter:

```
package pl.solr;

import org.apache.lucene.analysis.TokenStream;
import org.apache.solr.analysis.BaseTokenFilterFactory;

public class ExampleFilterFactory extends BaseTokenFilterFactory {
    @Override
    public TokenStream create(TokenStream stream) {
        return new ExampleFilter(stream);
    }
}
```

Now it's time to add the compiled class to the Solr web application. Of course, you can build a jar library with it, but for the purpose of the recipe, I decided to just add the compiled class file. To do that, I extracted the contents from the `solr.war` file, found in the `webapps` directory of the example deployment, to the `solr` directory. Next I added the compiled `ExampleFilter.class` and `ExampleFilterFactory.class` to the `webapps/solr/WEB-INF/classes/pl/solr` directory.

Next we need to add the newly created filter to our `schema.xml` file. So I've created a custom field type, whose definition looks like this:

```
<fieldtype name="exampleType" stored="true" indexed="true"
class="solr.TextField" >
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory" />
    <filter class="pl.solr.ExampleFilterFactory" />
  </analyzer>
</fieldtype>
```

Now we can test `exampleType` in the Solr administration pages (please refer to the recipe *How to check how the data type or field behave* discussed in *Chapter 4, Solr Administration*). The following screenshot shows the results:

Index Analyzer		
org.apache.solr.analysis.WhitespaceTokenizerFactory {}		
term position	1	2
term text	test	text
term type	word	word
source start,end	0,4	5,9
payload		
pl.solr.ExampleFilterFactory {encoder=float}		
term position	1	2
term text	etst	etxt
term type	word	word
source start,end	0,4	5,9
payload		

As you can see, everything is working as intended. Now let's discuss how it works.

## How it works...

Let's take a look at the `ExampleFilter` class. As you can see, it extends the `TokenFilter` class, which will allow our filter to work. The first thing you see in the code is the `termAttr` property definition. This definition will let us get and set the text of the token that is actually being processed by our filter. Following that, we have the constructor implementation that is doing one thing—it passes the `TokenStream` object to the constructor of its super class. This action is mandatory or the filter won't work.

Now comes the big thing—the `incrementToken` method. Remember that this method will be called for every token in the token stream and you should always make the implementation as efficient as you can. First things first: the return value of the method is responsible for telling Solr if there is a token to process in the token stream or not. If the method returns the `true` value, then Solr knows that there is at least a single token to process. Otherwise Solr knows that it's time to stop the processing. So, we first check if there is another token to process (`if (input.incrementToken())`). If there is, we get on with our logic. First, we get the text of the term (`termAttr.term()`). If the length of the text is one or less, we don't want to process it. We inform Solr that the token can be processed further by returning the `true` value. If the text is longer than one character, we change the first letter with the second, append the rest of the text, and modify the token by setting its new text (`termAttr.setTermBuffer(buffer.toString())`) and new length (`termAttr.setTermLength(buffer.length())`). Finally, we return the `true` value to inform Solr that the modified token can be processed further.

You should be aware of one thing—using the `term()` method of the `TermAttribute` class is not efficient and has a performance penalty because the term is stored in `char[]` and not as `String`. Thus, you should always use the `termBuffer()` and `termLength()` methods whenever possible. The example uses the `String` object to simplify the code and makes it easier to understand.

Now let's take a look at the factory. Factories are responsible for creating the actual instance of the filter that Solr will use. Our factory class is very simple. It extends the `BaseTokenFilterFactory` class. We override just a single method of the superclass: the `create` method that takes a `TokenStream` class object as an argument and returns a `TokenStream` class object. As you can see, we only create and return the instance of our `ExampleFilter` class. And that's all when it comes to the code.

Now let's look at the configuration. You should be familiar with it. As you can see, the `exampleType` field type is pretty simple. Besides the tokenizer, it only consists of our filter. Notice the `class` attribute; we wrote the full package and the class name of the factory, not the filter.

The last thing is the test: if everything is working as intended. I passed the "test text" phrase to the admin panel and the screenshot from Solr administration panel shows that the filter is working as intended.

## See also

If you want to know more about token stream and token stream attributes, please refer to the following URL address:

[http://lucene.apache.org/java/3\\_0\\_0/api/core/org/apache/lucene/analysis/package-summary.html](http://lucene.apache.org/java/3_0_0/api/core/org/apache/lucene/analysis/package-summary.html)

## Developing your own search component

Let's imagine that we want to have a component that will let us check how much memory the fields are given as a query parameter. To do that, you will need to write a custom code, for example, a custom search component. This recipe will show you how to do that.

### Getting ready

To run the example code provided in this chapter, you will need the following libraries from the standard Solr distribution:

- ▶ `apache-solr-core-3.1.jar` and `apache-solr-solrj-3.1.jar` from the `/dist` directory
- ▶ `lucene-core-3.1.jar` from the `/lib` directory

Place these libraries in classpath of your project. You will also need Java 1.6 JDK to compile this example.

### How to do it...

Let's start with the actual search component code. The code will look like this:

```
package pl.solr;

import java.io.IOException;
import org.apache.solr.handler.component.ResponseBuilder;
import org.apache.solr.handler.component.SearchComponent;
import org.apache.solr.request.UnInvertedField;
import org.apache.solr.search.SolrIndexSearcher;

public class ExampleSearchComponent extends SearchComponent {
    String[] fieldNames = null;

    @Override
    public void prepare(ResponseBuilder builder) throws IOException {
        fieldNames = builder.req.getParams().get("exampleFields", "").split(",");
    }

    @Override
    public void process(ResponseBuilder builder) throws IOException {
        if (fieldNames != null) {
            long totalMemorySize = 0;
            SolrIndexSearcher searcher = builder.req.getSearcher();
            for (String fieldName : fieldNames) {
                UnInvertedField field = UnInvertedField.
getUnInvertedField(fieldName, searcher);
                totalMemorySize += field.memSize();
            }
        }
    }
}
```



```

        }
    }
    builder.rsp.add( "example", totalMemorySize );
}

@Override
public String getDescription() {
    return "ExampleSearchComponent";
}

@Override
public String getSource() {
    return "";
}

@Override
public String getSourceId() {
    return "";
}

@Override
public String getVersion() {
    return "1.0";
}
}

```

Now it's time to add the compiled class to the Solr web application. I decided to just add the compiled class file. To do that, I extracted the contents from the `solr.war` file, found in the `webapps` directory of the example deployment, to the `solr` directory. Next, I added the compiled `ExampleSearchComponent.class` to the `webapps/solr/WEB-INF/classes/pl/solr` directory.

Now, let's tell Solr to use this component. To do that, we modify the `solrconfig.xml` file and add the following:

```

<requestHandler name="/example" class="solr.SearchHandler">
    <arr name="components">
        <str>exampleComponent</str>
    </arr>
</requestHandler>

<searchComponent name="exampleComponent" class="pl.solr.
ExampleSearchComponent">
</searchComponent>

```

To use the example component, I indexed the data from the `/exampledocs` directory provided with the example Solr deployment. After that, I run the following query:

```

http://localhost:8983/solr/example?q=*:*&exampleFields=cat,name,popul
arity

```

The response from Solr was as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <long name="example">18208</long>
</response>
```

As you can see, we have our code working. So now let's take a look at how it works.

## How it works...

Let's look at the `ExampleSearchComponent` class code. We extend the `SearchComponent` class to allow Solr to use our component. Then we define a class variable, which we will use to store the name of the fields that we want to calculate the memory usage for.

The `prepare` method of our class is one of the methods that we need to override. In this method, we need to run every code that is request-dependant. For example, we get the fields that we want to calculate the memory usage for. I assumed that the fields will be provided in the query, in the parameter named `exampleFields`, and they will be separated by the comma character. This method is guaranteed to run before any of the search components' process methods.

Next is the `process` method—another one that we need to override. This method is responsible for the actual processing. So we start with the check if we have any fields passed to the component. If so, we get the `SolrIndexSearcher` object from the request (`builder.req.getSearcher()`) and for every field passed to the component, we get the `UnInvertedField` object. This object will allow us to get the memory used by the field, by calling its `memSize` method. We add `memSize` to the `totalMemorySize` variable. After the loop is done, we add the `example` section to the response with the value of the `totalMemorySize` variable. And that's all.

The rest of the methods I override are mandatory, but they only provide information about the class for Solr, so I'll skip discussing them.

Next, we need to tell Solr to use the component. To do that, we define a new request handler called `/example` with one component named `exampleComponent`. Then we add a new search component—`exampleComponent`, by adding the `searchComponent` tag to the `solrconfig.xml` file with the attributes `name` and `class`. The `name` attribute of the `searchComponent` tag specifies the component name and is used to add the component to request handlers, while the `class` attribute specifies which class is responsible for the actual implementation of the component.

Please remember that the code used in this example is not the most efficient one, as we should be using `FieldCache` and so on, but it's sufficient to show you how to develop your own search component.

## Developing your own field type

Imagine a situation where your client tells you that he wants to store a document identifier, name, and price. However, the price he has in the data files are not in a good condition. Some of them have the `,` sign as the delimiter, while some have the `.` sign. The client also wants to sort on the basis of that field. What we can expect is a garbage value in the data. So what can we do besides cleaning that up? One of the many ways of dealing with that is writing your own field type and this recipe will show you how to do that.

### Getting ready

To run the example code provided in this chapter, you will need the following libraries from the standard Solr distribution:

- ▶ `apache-solr-core-3.1.jar` from the `/dist` directory
- ▶ `lucene-core-3.1.jar` from the `/lib` directory

Place the mentioned libraries in `classpath` of your project. You will also need Java 1.6 JDK to compile this example.

### How to do it...

Let's start with the code that implements the example field type. The code looks like this:

```
package pl.solr;

import java.io.IOException;

import org.apache.lucene.document.Fieldable;
import org.apache.solr.request.TextResponseWriter;
import org.apache.solr.request.XMLWriter;
import org.apache.solr.schema.SortableIntField;
import org.apache.solr.util.NumberUtils;

public class ExampleFieldType extends SortableIntField {
    public final String toInternal(final String value) {
        return NumberUtils.int2sortableStr(getInternalValue(value));
    }

    public final void write(final XMLWriter xmlWriter, final String
name, final Fieldable field) throws IOException {
        xmlWriter.writeStr(name, parseFromValue(String.
valueOf(getVal(field))));
    }
}
```

```

    }

    public final void write(final TextResponseWriter writer, final
String name, final Fieldable field) throws IOException {
        writer.writeStr(name, parseFromValue(String.
valueOf(getVal(field))), false);
    }

    protected String getInternalValue( String value ) {
        String internalValue = value.replace(",", ".");
        String[] parts = internalValue.split("\\.");
        internalValue = parts[0];
        if (parts.length > 1 && parts[1] != null) {
            if (parts[1].length() == 2) {
                internalValue += parts[1];
            } else if (parts[1].length() == 1) {
                internalValue += parts[1];
                internalValue += "0";
            } else {
                internalValue += "00";
            }
        } else if (parts.length == 1) {
            internalValue += "00";
        }
        return internalValue;
    }

    protected int getVal( Fieldable field ) {
        String stringValue = field.stringValue();
        return NumberUtils.SortableStr2int(stringValue, 0 ,
stringValue.length());
    }

    protected String parseFromValue(final String value) {
        int length = value.length();
        StringBuffer buffer = new StringBuffer();
        if (length > 2) {
            buffer.append(value.substring(0, length - 2));
            buffer.append(".");
            buffer.append(value.substring(length - 2 , length));
        }
        return buffer.toString();
    }
}

```

The code looks a bit complicated, but let's not worry about that right now.

Now it's time to add the compiled class to the Solr web application. Of course, you can build a jar library with it, but for the purpose of the recipe, I decided to just add the compiled class file. To do that, I extracted the contents from the `solr.war` file, found in the `webapps` directory of the example deployment, to the `solr` directory. Next, I added the compiled `ExampleFieldType.class` to the `webapps/solr/WEB-INF/classes/pl/solr` directory.

Now let's modify our `schema.xml` file to include the new field type. To do that, we add the following field type to the field type section of the `schema.xml` file. The new definition should look like this:

```
<fieldtype name="example" class="pl.solr.ExampleFieldType" />
```

Now let's add a field called `price`, based on the new field type. To do that, add the following field to the field section of the `schema.xml` file:

```
<field name="price" type="example" indexed="true" stored="true"/>
```

So now, to test if the field actually works, I decided to test if the value `12.10`, which was indexed, matched to the value `12,1`, which was passed as a query to the `price` field. Guess what—it worked. I used the analysis section of the Solr administration panel to check it. The result was as follows:

### Field Analysis

Field name	price
Field value (Index)	12.10
verbose output	<input checked="" type="checkbox"/>
highlight matches	<input checked="" type="checkbox"/>
Field value (Query)	12,1
verbose output	<input checked="" type="checkbox"/>
Analyze	

### Index Analyzer

org.apache.solr.schema.FieldType\$DefaultAnalyzer {}

term position	1
term text	1210
raw text	#0;h
term type	word
source start,end	0,5
payload	

### Query Analyzer

org.apache.solr.schema.FieldType\$DefaultAnalyzer {}

term position	1
term text	1210
raw text	#0;h
term type	word
source start,end	0,4
payload	

Now let's see how the code works.

### How it works...

The idea behind the new field type is quite simple: Take the value from the user, change the , character for the . character, and somehow normalize the value. So, let's see how it was done.

As you can see, our `ExampleFieldType` class extends the `SortableIntField` class, which is a class that stores the integer number and allows sorting. The first method named `toInternal` is responsible for parsing the value of the field that was provided in the update file (for example, an XML file) or through the query and preparing the value for storing and indexing. We use the `NumberUtils` class and its `int2sortableString`, which stores the provided value as a sortable string representation (the `getInternalValue` method will be discussed in a moment).

Next, we have two write methods. The first one is responsible for rendering the appropriate XML representation of the value and the second one is responsible for rendering the appropriate text representation. Both methods use `getVal` and `parseFromValue` methods which I'll discuss in a moment.

The three methods that I mentioned are the ones that we had to override to implement the class. Now, let's look at the helper methods.

First we have the `getInternalValue` method, which is responsible for normalization. It basically removes the , and . characters, and adds various amounts of zeros as a postfix value. The number of zeros that are added to the return value is based on the length of the second part of the given number. For example, if the user passed the number `12.1`, there will be an additional zero added. So the return value would be `1210`. If the user provides the value `12.13`, then the returned value would be `1213`. This is done because we want to store the values (and sort them) given by the user as integers, but we must also be able to reverse the process. That's why we add the zeros: so we can always say where the comma character should be placed.

The `getVal` method is responsible for returning the integer value that is stored in the given field. Nothing more, nothing less.

The last method, `parseValueFrom`, is responsible for reversing the process of the `getInternalValue` method and adding the . character to the stored value.

After compiling and placing the class (or jar) file in the appropriate place, we modified the `schema.xml` file and added the new type and field definition to test our newly created type. This part is pretty obvious, so I'll skip commenting on it.

The last thing was the actual test. I decided to use the analysis pages in the Solr administration interface. What I tested was if the two values—`12.10` (written in the index) and `12.1` (given in the query)—will match or not. And that's a match. This means that the type that we created is working as intended.



# 9

## Using Additional Solr Functionalities

In this chapter, we will cover:

- ▶ Getting more documents similar to those returned in the results list
- ▶ Presenting search results in a fast and easy way
- ▶ Highlighting matched words
- ▶ How to highlight long text fields and get good performance
- ▶ Sorting results by a function value
- ▶ Searching words by how they sound
- ▶ Ignoring defined words
- ▶ Computing statistics for the search results
- ▶ Checking user's spelling mistakes
- ▶ Using "group by" like functionalities in Solr

### Introduction

There are many features of Solr that we don't use every day. Highlighting, ignoring words, or statistics computation may not be in everyday use, but they can come in handy in many situations. In this chapter, I'll try to show you how to overcome some typical problems that can be fixed by using some of the Solr functionalities.



## Getting more documents similar to those returned in the results list

Let's imagine a situation where you have an e-commerce library shop and you want to show users the books similar to the ones they found while using your application. This recipe will show you how to do that.

### How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file's fields section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true"
termVectors="true" />
```

The test data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Solr Cookbook first edition</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Solr Cookbook second edition</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Solr by example first edition</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="name">My book second edition</field>
  </doc>
</add>
```

Let's assume that our hypothetical user wants to find books that have `first` in their names. However, we also want to show him the similar books. To do that, we send the following query:

```
http://localhost:8983/solr/select?q=name:edition&mlt=true&mlt.
fl=name&mlt.mintf=1&mlt.mindf=1
```

The results returned by Solr are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">1</int>
  <lst name="params">
    <str name="mlt.mindf">1</str>
    <str name="mlt.fl">name</str>
    <str name="q">name:edition</str>
    <str name="mlt.mintf">1</str>
    <str name="mlt">true</str>
  </lst>
</lst>
<result name="response" numFound="1" start="0">
  <doc>
    <str name="id">3</str>
    <str name="name">Solr by example first edition</str>
  </doc>
</result>
<lst name="moreLikeThis">
  <result name="3" numFound="3" start="0">
    <doc>
      <str name="id">1</str>
      <str name="name">Solr Cookbook first edition</str>
    </doc>
    <doc>
      <str name="id">2</str>
      <str name="name">Solr Cookbook second edition</str>
    </doc>
    <doc>
      <str name="id">4</str>
      <str name="name">My book second edition</str>
    </doc>
  </result>
</lst>
</response>
```

Now let's see how it works.

## How it works...

As you can see, the index structure and the data are really simple. One thing to notice is that the `termVectors` attribute is set to `true` in the `name` field definition. It is a nice thing to have when using more like this component and should be used when possible in the fields on which we plan to use the component.

Now let's take a look at the query. As you can see, we added some additional parameters besides the standard `q` one. The parameter `mlt=true` says that we want to add the `more like this` component to the result processing. Next, the `mlt.fl` parameter specifies which fields we want to use with the `more like this` component. In our case, we will use the `name` field. The `mlt.mintf` parameter tells Solr to ignore terms from the source document (the ones from the original result list) with the term frequency below the given value. In our case, we don't want to include the terms that will have the frequency lower than 1. The last parameter, `mlt.mindf`, tells Solr that the words that appear in less than the value of the parameter documents should be ignored. In our case, we want to consider words that appear in at least one document.

Finally, let's take a look at the search results. As you can see, there is an additional section (`<lst name="moreLikeThis">`) that is responsible for showing us the `more like this` component results. For each document in the results, there is one more similar section added to the response. In our case, Solr added a section for the document with the unique identifier 3 (`<result name="3" numFound="3" start="0">`) and there were three similar documents found. The value of the `id` attribute is assigned the value of the unique identifier of the document that the similar documents are calculated for.

## Presenting search results in a fast and easy way

Imagine a situation where you have to show a prototype of your brilliant search algorithm made with Solr to the client. But the client doesn't want to wait another four weeks to see the potential of the algorithm, he/she wants to see it very soon. On the other hand, you don't want to show the pure XML results page. What to do then? This recipe will show you how you can use the Velocity response writer (a.k.a. Solritas) to present a prototype fast.

## How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file to the fields section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
```

The test data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Solr Cookbook first edition</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Solr Cookbook second edition</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Solr by example first edition</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="name">My book second edition</field>
  </doc>
</add>
```

We need to add the response writer definition. To do this, you should add this to your `solrconfig.xml` file (actually this should already be in the configuration file):

```
<queryResponseWriter name="velocity" class="org.apache.solr.request.
VelocityResponseWriter"/>
```

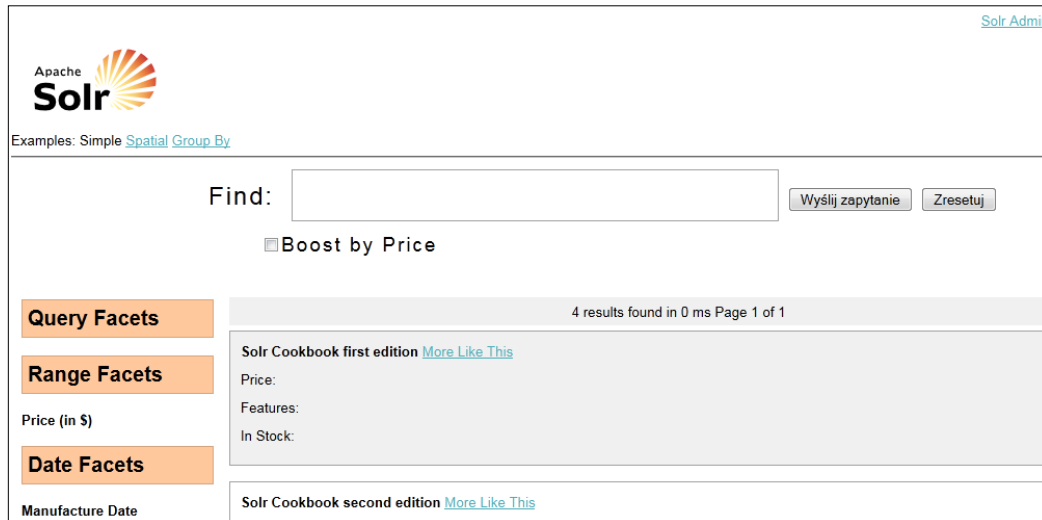
Now let's set up the Velocity response writer. To do that we add the following section to the `solrconfig.xml` file (actually this should already be in the configuration file):

```
<requestHandler name="/browse" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="wt">velocity</str>
    <str name="v.template">browse</str>
    <str name="v.layout">layout</str>
    <str name="title">Solr cookbook example</str>
    <str name="defType">dismax</str>
    <str name="q.alt">*:*</str>
    <str name="rows">10</str>
    <str name="fl">*,score</str>
    <str name="qf">name</str>
  </lst>
</requestHandler>
```

Now you can run Solr and type the following URL address:

```
http://localhost:8983/solr/browse
```

You should see the following page:



The screenshot shows the Apache Solr Admin interface. At the top left is the Apache Solr logo. Below it, there are links for 'Examples: Simple', 'Spatial', and 'Group By'. On the top right is a 'Solr Admin' link. The main search area has a 'Find:' label, a text input field, and two buttons: 'Wyślij zapytanie' and 'Zresetuj'. Below the search bar is a checkbox labeled 'Boost by Price'. On the left side, there are three orange buttons: 'Query Facets', 'Range Facets', and 'Date Facets'. Below these buttons are labels for 'Price (in \$)' and 'Manufacture Date'. The main content area shows search results. At the top, it says '4 results found in 0 ms Page 1 of 1'. The first result is 'Solr Cookbook first edition' with a 'More Like This' link. Below the title are labels for 'Price:', 'Features:', and 'In Stock:'. The second result is 'Solr Cookbook second edition' with a 'More Like This' link.

## How it works...

As you can see, the index structure and the data are really simple, so I'll skip discussing this part of the recipe.

The first thing in configuring the `solrconfig.xml` file is adding the Velocity Response Writer definition. By adding it, we tell Solr that we will be using velocity templates to render the view.

Now we add the search handler to use the Velocity Response Writer. Of course, we could pass the parameters with every query, but we don't want to do that, we want them to be added by Solr automatically. Let's go through the parameters:

- ▶ `wt`: The response writer type; in our case, we will use the Velocity Response Writer.
- ▶ `v.template`: The template that will be used for rendering the view; in our case, the template that Velocity will use is in the `browse.vm` file (the `vm` postfix is added by Velocity automatically). This parameter tells Velocity which file is responsible for rendering the actual page contents.
- ▶ `v.layout`: The layout that will be used for rendering the view; in our case, the template that velocity will use is in the `layout.vm` file (the `vm` postfix is added by velocity automatically). This parameter specifies how all the web pages rendered by Solritas will look like.
- ▶ `title`: The title of the page.
- ▶ `defType`: The parser that we want to use.
- ▶ `q.alt`: Alternate query for the dismax parser in case the `q` parameter is not defined.

- ▶ `rows`: How many maximum documents should be returned.
- ▶ `f1`: Fields that should be listed in the results.
- ▶ `qf`: The fields that we should be searched.

Of course, the page generated by the Velocity Response Writer is just an example. To modify the page, you should modify the Velocity files, but this is beyond the scope of this book.

### There's more...

If you are still using Solr 1.4.1 or 1.4, there is one more thing that can be useful.

### Running Solritas on Solr 1.4.1 or 1.4

Because the Velocity Response Writer is a contrib module in Solr 1.4.1, we need to do the following operations to use it. Copy the following libraries from the `/contrib/velocity/src/main/solr/lib` directory to the `/lib` directory of your Solr instance:

- ▶ `apache-solr-velocity-1.4.dev.jar`
- ▶ `commons-beanutils-1.7.0.jar`
- ▶ `commons-collections-3.2.1.jar`
- ▶ `velocity-1.6.1.jar`
- ▶ `velocity-tools-2.0-beta3.jar`

Then copy the contents of the `/velocity` (with the directory) directory from the code examples provided with this book to your Solr configuration directory.

### See also

If you would like to know more about Velocity, please have a look at the following URL address: <http://velocity.apache.org/>.

## Highlighting matched words

Imagine a situation where you want to show your users which words were matched in the document that is shown in the results list. For example, you may want to show which words in the book name were matched and display that to the user. Do you have to store the documents and do the matching on the application side? The answer is no—we can force Solr to do that for us and this recipe will show you how.

## How to do it...

Let's assume that we have the following index structure (just add this to the fields section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
```

The test data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Solr Cookbook first edition</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Solr Cookbook second edition</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Solr by example first edition</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="name">My book second edition</field>
  </doc>
</add>
```

Let's assume that our user is searching for the word 'book'. To tell Solr that we want to highlight the matches, we send the following query:

```
http://localhost:8983/solr/select?q=name:book&hl=true
```

The response from Solr should be like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">15</int>
    <lst name="params">
      <str name="hl">true</str>
      <str name="q">name:book</str>
    </lst>
```

```

</lst>
<result name="response" numFound="1" start="0">
  <doc>
    <str name="id">4</str>
    <str name="name">My book second edition</str>
  </doc>
</result>
<lst name="highlighting">
  <lst name="4">
    <arr name="name">
      <str>My <em>book</em> second edition</str>
    </arr>
  </lst>
</lst>
</response>

```

As you can see, besides the normal results list, we got the highlighting results (the highlighting results are grouped by the `<lst name="highlighting">` XML tag). The word `book` is surrounded with the `<em>` and `</em>` HTML tags. So everything is working as intended. Now let's see how it works.

### How it works...

As you can see, the index structure and the data are really simple, so I'll skip discussing this part of the recipe. Please note that in order to use the highlighting mechanism, your fields should be stored and not analyzed by aggressive filters (like stemming). Otherwise, the highlighting results can be misleading to the users. The example of such a behavior can be simple—imagine the user types the word `bought` in the search box but Solr highlighted the word `buy` because of the stemming algorithm.

The query is also not complicated. We can see the standard `q` parameter that passes the query to Solr. However, there is also one additional parameter, the `hl` set to `true`. This parameter tells Solr to include the highlighting component results to the results list. As you can see in the results list, in addition to the standard results there is a new section—`<lst name="highlighting">` which contains the highlighting results. For every document, in our case, the only one found (`<lst name="4">` means that the highlighting result is presented for the document with the unique identifier value of 4) there is a list of fields that contains the sample data with the matched words (or words) highlighted. By highlighted, I mean surrounded with the HTML tag, in this case, the `<em>` tag.

You should also remember one thing—if you are using the standard `LuceneQParser` then the default field used for highlighting will be the one set in the `schema.xml` file. If you are using the `DismaxQParser`, then the default fields used for highlighting are the ones specified by the `qf` parameter.



## There's more...

There are a few things that can be useful when using the highlighting mechanism.

### Specifying the fields for highlighting

In many real-life situations, we want to decide which fields we want to show for highlighting. To do that, you must add an additional parameter, `hl.fl` with the list of fields separated by the comma character. For example, if we would like to show the highlighting for the fields `name` and `description` our query should look like this:

```
http://localhost:8983/solr/select?q=name:book&hl=true&hl.fl=name,description
```

### Changing the default HTML tags that surround the matched word

There are situations where you would like to change the default `<em>` and `</em>` HTML tags to the ones of your choice. To do that, you should add the `hl.simple.pre` parameter and the `hl.simple.post` parameter. The first one specifies the prefix that will be added in front of the matched word and the second one specifies the postfix that will be added after the matched word. For example, if you would like to surround the matched word with the `<b>` and `</b>` HTML tags the query would look like this:

```
http://localhost:8983/solr/select?q=name:book&hl=true&hl.simple.pre=<b>&hl.simple.post=</b>
```

## How to highlight long text fields and get good performance

In certain situations, the standard highlighting mechanism may not be performing as well as you would like it to be. For example, you may have long text fields and you want the highlighting mechanism to work with them. This recipe will show you how to do that.

## How to do it...

Let's assume that we have the following index structure (just add this to your `schema.xml` file, to the fields section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true"
termVectors="true" termPositions="true" termOffsets="true" />
```

The test data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Solr Cookbook first edition</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Solr Cookbook second edition</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Solr by example first edition</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="name">My book second edition</field>
  </doc>
</add>
```

Let's assume that our user is searching for the word `book`. To tell Solr that we want to highlight the matches we send the following query:

```
http://localhost:8983/solr/select?q=name:book&hl=true&hl.
useFastVectorHighlighter=true
```

The response from Solr should be like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">15</int>
    <lst name="params">
      <str name="hl">true</str>
      <str name="q">name:book</str>
    </lst>
  </lst>
  <result name="response" numFound="1" start="0">
    <doc>
      <str name="id">4</str>
      <str name="name">My book second edition</str>
    </doc>
  </result>
  <lst name="highlighting">
```

```
<lst name="4">
  <arr name="name">
    <str>My <em>book</em> second edition</str>
  </arr>
</lst>
</lst>
</response>
```

As you can see, everything is working as intended. Now let's see how it works.

### How it works...

As you can see, the index structure and the data are really simple, but there is a difference between using the standard highlighter and the new `FastVectorHighlighting`. To be able to use the new highlighting mechanism, you need to store the information about term vectors, position, and offsets. This is done by adding the following attributes to the field definition or to the type definition: `termVectors="true"`, `termPositions="true"`, `termOffsets="true"`.

Please note that in order to use the highlighting mechanism your fields should be stored and not analyzed by aggressive filters (like stemming). Otherwise, the highlighting results can be misleading to the users. The example of such a behavior can be simple—imagine the user types the word `bought` in the search box but Solr highlighted the word `buy` because of the stemming algorithm.

The query is also not complicated. We can see the standard `q` parameter that passes the query to Solr. However, there is also one additional parameter, `hl`, set to `true`. This parameter tells Solr to include the highlighting component results to the results list. In addition, we add the parameter to tell Solr to use the `FastVectorHighlighting-hl.useFastVectorHighlighter=true`.

As you can see in the results list, in addition to the standard results there is a new section, `<lst name="highlighting">`, which contains the highlighting results. For every document, in our case, the only one found (`<lst name="4">`, means that the highlighting result is presented for the document with the unique identifier value of 4) there is a list of fields that contains the sample data with the matched words (or words) highlighted. By highlighted, I mean surrounded with the HTML tag, in this case, the `<em>` tag.

### Sorting results by a function value

Let's imagine that you have an application that allows the user to search through the companies that are stored in the index. You would like to add an additional feature to your application—to sort the results on the basis of the distance of certain geographical points. Is this possible with Solr? Yes, and this recipe will show you how to do that.

## How to do it..

Let's assume that we have the following index structure (just add this to the fields section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="geoX" type="float" indexed="true" stored="true" />
<field name="geoY" type="float" indexed="true" stored="true" />
```

And the data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Company one</field>
    <field name="geoX">10.1</field>
    <field name="geoY">10.1</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Company two</field>
    <field name="geoX">11.1</field>
    <field name="geoY">11.1</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Company three</field>
    <field name="geoX">12.2</field>
    <field name="geoY">12.2</field>
  </doc>
</add>
```

Let's assume that our hypothetical user searches for the word `company` and the user is in the location which has the geographical point of (13, 13). So, in order to show the results of the query and sort it by the distance from the given point, we send the following query to Solr:

```
http://localhost:8983/solr/select?q=name:company&sort=dist(2,geoX,geoY,13,13)+asc
```

The results list returned by the query looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">10</int>
    <lst name="params">
```

```
<str name="sort">dist(2,geoX,geoY,13,13) asc</str>
<str name="q">name:company</str>
<str name="defType">dismax</str>
</lst>
</lst>
<result name="response" numFound="3" start="0">
  <doc>
    <float name="geoX">12.2</float>
    <float name="geoY">12.2</float>
    <str name="id">3</str>
    <str name="name">Company three</str>
  </doc>
  <doc>
    <float name="geoX">11.1</float>
    <float name="geoY">11.1</float>
    <str name="id">2</str>
    <str name="name">Company two</str>
  </doc>
  <doc>
    <float name="geoX">10.1</float>
    <float name="geoY">10.1</float>
    <str name="id">1</str>
    <str name="name">Company one</str>
  </doc>
</result>
</response>
```

As you can see, everything is working as it should be. So now let's see how it works.

## How it works...

Let's start from the index structure. We have four fields—one for holding the unique identifier (the `id` field), one for holding the name of the company (the `name` field), and two fields responsible for the geographical location of the company (the `geoX` and `geoY` fields). The data is pretty simple so let's just skip discussing that.

Besides the standard `q` parameter responsible for the user query, you can see the `sort` parameter. However, the `sort` parameter is a bit different from the ones you are probably used to. It uses the `dist` function to calculate the distance from the given point and the value returned by the function is then used to sort the documents in the results list. The first argument of the `dist` function (the value `2`) tells Solr to use the Euclidian Distance to calculate the distance. The next two arguments tell Solr which fields in the index hold the geographical position of the company. The last two arguments specify the point from which the distance should be calculated. Of course, as with every sort, we specify the order in which we want to sort. In our case, we want to sort from the nearest to the farthest company (the `asc` value).

As you can see in the results, the documents were sorted as they should be.

## Searching words by how they sound

One day your boss comes to your office and says "Hey, I want our search engine to be able to find the same documents when I enter phone or fone into the search box." You try to reply, but your boss is already on the other side of the door. So, you wonder if this kind of functionality is available in Solr. I think you already know the answer—yes, it is, and this recipe will show you how to configure it and use with Solr.

### How to do it...

Let's assume that we have the following index structure (just add this to the fields section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="phonetic" indexed="true" stored="true" />
```

The phonetic type definition looks like this (this already should be included in the `schema.xml` file of the example Solr instance):

```
<fieldtype name="phonetic" stored="false" indexed="true" class="solr.
TextField" >
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.DoubleMetaphoneFilterFactory" inject="false"/>
  </analyzer>
</fieldtype>
```

And the data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Phone</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Fone</field>
  </doc>
</add>
```

Now let's assume that our user wants to find documents that have the word that sounds like fon. So, we send the following query to Solr:

```
http://localhost:8983/solr/select?q=name:fon
```

The result list returned by the query is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="q">name:fon</str>
    </lst>
  </lst>
  <result name="response" numFound="2" start="0">
    <doc>
      <str name="id">1</str>
      <str name="name">Phone</str>
    </doc>
    <doc>
      <str name="id">2</str>
      <str name="name">Fone</str>
    </doc>
  </result>
</response>
```

So, the filter worked—we got two documents in the results list. Now let's see how it works.

## How it works...

Let's start with the index structure. As you can see, we have two fields—one responsible for holding the unique identifier (the `id` field) of the product and the other responsible for holding the name of the product (the `name` field).

The `name` field is the one that will be used for phonetic search. For that we defined a new field type named `phonetic`. Besides the standard parts (such as `class` and so on) we defined a new filter, `DoubleMetaphoneFilterFactory`. It is responsible for analysis and checking how the words sound like. This filter uses algorithm named double metaphone to analyze the phonetics of the words. The additional attribute `inject="false"` tells Solr to replace the existing tokens instead of inserting additional ones, which means that the original tokens will be replaced by the ones that the filter produces.

As you can see from the query and the data, the `fon` word was matched to the word `phone` and also to the word `fone` which means that the algorithm (and thus the filter) are working quite well. However, take into consideration that this is only an algorithm, so some words that you think should be matched will not match.

## See also

If you would like to know other phonetic algorithms, please take a look at the Solr wiki page that can be found at the following URL address: <http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters>.

## Ignoring defined words

Imagine a situation where you would like to filter the words that are considered vulgar from the data we are indexing. Of course, by accident, such words can be found in your data and you don't want them to be searchable, thus you want to ignore them. Can we do that with Solr? Of course we can and this recipe will show you how.

## How to do it...

Let's start with the index structure (just add this to your `schema.xml` file to the fields section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text_ignored" indexed="true" stored="true" />
```

The `text_ignored` type definition looks like this:

```
<fieldType name="text_ignored" class="solr.TextField"
positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
words="ignored.txt" enablePositionIncrements="true" />
  </analyzer>
</fieldType>
```

The `ignored.txt` file looks like this:

```
vulgar
vulgar2
vulgar3
```

And the data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Company name</field>
  </doc>
</add>
```



Now let's assume that our user wants to find documents that have the words `Company` and `vulgar`. So, we send the following query to Solr:

```
http://localhost:8983/solr/select?q=name:(Company+AND+vulgar)
```

In the standard situation, there shouldn't be any results because we don't have a document that matches the two given words. However, let's take a look at what Solr returned to us as the preceding query result:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="q">name:(Company AND vulgar)</str>
    </lst>
  </lst>
  <result name="response" numFound="1" start="0">
    <doc>
      <str name="id">1</str>
      <str name="name">Company name</str>
    </doc>
  </result>
</response>
```

Hmm... it works. To be perfectly sure, let's see the analysis page found at the administration interface.

Query Analyzer		
org.apache.solr.analysis.WhitespaceTokenizerFactory {}		
term position	1	2
term text	Company vulgar	
term type	word	word
source start,end	0,7	8,14
payload		
org.apache.solr.analysis.StopFilterFactory {words=ignored.txt, ignoreCase=true, enablePositionIncrements=true}		
term position	1	
term text	Company	
term type	word	
source start,end	0,7	
payload		

As you can see, the word `vulgar` was cut and thus ignored.

## How it works...

Let's start with the index structure. As you can see, we have two fields. One responsible for holding the unique identifier (the `id` field) of the product and the other responsible for holding the name of the product (the `name` field).

The `name` field is the one that we will use for the ignore functionality of Solr, `StopFilterFactory`. As you can see, the `text_ignored` type definition is analyzed the same way both in the query and index time. The unusual thing is the new filter, `StopFilterFactory`. The `words` attribute of the filter definition specifies the name of the file, encoded in UTF-8 that consists of words (new word at every file line) that should be ignored. The defined file should be placed in the same directory that we put the `schema.xml` file. The `ignoreCase` attribute set to `true` tells the filter to ignore the case of the tokens and the words defined in the file. The last attribute, `enablePositionIncrements=true`, tells Solr to increment the position of the tokens in the token stream. The `enablePositionIncrements` parameter should be set to `true` if you want to preserve the next token after the discarded one to increment its position in the token stream.

As you can see in the query, our hypothetical user queried Solr for two words with the logical operator `AND` which means that both words must be present in the document. However, the filter we added cut the word `vulgar` and thus the results list consists of the document that has only one of the words. The same is the situation when you are indexing your data. The words defined in the `ignored.txt` file will not be indexed.

If you look at the provided screenshot from the analysis page of the Solr administration interface, you can see that the word `vulgar` was cut during the processing of the token stream in the `StopFilterFactory`.

## Computing statistics for the search results

Imagine a situation where you want to compute some basic statistics about the documents in the results list. For example, you have an e-commerce shop where you want to show the minimum and the maximum price of the documents that were found for a given query. Of course, you could fetch all the documents and count it by yourself, but imagine if Solr can do it for you. Yes it can and this recipe will show you how to use that functionality.

## How to do it...

Let's start with the index structure (just add this to the fields section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="price" type="float" indexed="true" stored="true" />
```

The example data file looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Book 1</field>
    <field name="price">39.99</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Book 2</field>
    <field name="price">30.11</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Book 3</field>
    <field name="price">27.77</field>
  </doc>
</add>
```

Let's assume that we want our statistics to be computed for the price field. To do that, we send the following query to Solr:

```
http://localhost:8983/solr/select?q=name:book&stats=true&stats.
field=price
```

The response Solr returned should be like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
    <lst name="params">
      <str name="q">name:book</str>
      <str name="stats">true</str>
      <str name="stats.field">price</str>
    </lst>
  </lst>
  <result name="response" numFound="3" start="0">
    <doc>
      <str name="id">1</str>
      <str name="name">Book 1</str>
      <float name="price">39.99</float>
    </doc>
    <doc>
```

```

    <str name="id">2</str>
    <str name="name">Book 2</str>
    <float name="price">30.11</float>
  </doc>
</doc>
  <str name="id">3</str>
  <str name="name">Book 3</str>
  <float name="price">27.77</float>
</doc>
</result>
<lst name="stats">
  <lst name="stats_fields">
    <lst name="price">
      <double name="min">27.77</double>
      <double name="max">39.99</double>
      <double name="sum">97.86999999999999</double>
      <long name="count">3</long>
      <long name="missing">0</long>
      <double name="sumOfSquares">3276.9851000000003</double>
      <double name="mean">32.62333333333333</double>
      <double name="stddev">6.486118510583508</double>
    </lst>
  </lst>
</lst>
</response>

```

As you can see, in addition to the standard results list, there was an additional section available. Now let's see how it works.

## How it works...

The index structure is pretty straightforward. It contains three fields—one for holding the unique identifier (the `id` field), one for holding the name (the `name` field), and one for holding the price (the `price` field).

The file that contains the example data is simple too, so I'll skip discussing it.

The query is interesting. In addition to the `q` parameter, we have two new parameters. The first one, `stats=true`, tells Solr that we want to use the `StatsComponent`, the component which will calculate the statistics for us. The second parameter, `stats.field=price`, tells the `StatsComponent` which field to use for the calculation. In our case, we told Solr to use the `price` field.

Now let's look at the result returned by Solr. As you can see, the `StatsComponent` added an additional section to the results. This section contains the statistics generated for the field we told Solr we want statistics for. The following statistics are available:

- ▶ `min`: The minimum value that was found in the field for the documents that matched the query
- ▶ `max`: The maximum value that was found in the field for the documents that matched the query
- ▶ `sum`: Sum of all values in the field for the documents that matched the query
- ▶ `count`: How many non-null values were found in the field for the documents that matched the query
- ▶ `missing`: How many documents that matched the query didn't have any value in the specified field
- ▶ `sumOfSquares`: Sum of all values squared in the field for the documents that matched the query
- ▶ `mean`: The average for the values in the field for the documents that matched the query
- ▶ `stddev`: The standard deviation for the values in the field for the documents that matched the query

You should also remember that you can specify multiple `stats.field` parameters to calculate statistics for different fields in a single query.

Please be careful when using this component on the multivalued fields as it can be a performance bottleneck.

## Checking user's spelling mistakes

Most modern search sites have some kind of mechanism to correct user's spelling mistakes. Some of those sites have a sophisticated mechanism, while others just have a basic one. However, that doesn't matter. If all the search engines have it, then there is a big probability that your client or boss will want one too. Is there a way to integrate such functionality into Solr? Yes there is and this recipe will show you how to do it.

### How to do it...

Let's start with the index structure (just add this to the fields section of your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
```

The example data file looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Solr cookbook</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Mechanics cookbook</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Other book</field>
  </doc>
</add>
```

Our spell-check mechanism will work on the basis of the `name` field. Now, let's add the appropriate search component to the `solrconfig.xml` file.

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
  <lst name="spellchecker">
    <str name="name">default</str>
    <str name="field">name</str>
    <str name="spellcheckIndexDir">./spellchecker</str>
    <str name="buildOnCommit">true</str>
  </lst>
</searchComponent>
```

In addition, we would like to have it integrated into our search handler, so we make the default search handler definition like this (add this to your `solrconfig.xml` file):

```
<requestHandler name="standard" class="solr.SearchHandler"
  default="true">
  <arr name="last-components">
    <str>spellcheck</str>
  </arr>
</requestHandler>
```

Now let's check how it works. To do that, we will send a query that contains a spelling mistake—we will send the words `othar boak` instead of `other book`. The query doing that should look like this:

```
http://localhost:8983/solr/spell?q=name:(other boak)&spellcheck=true&spellcheck.collate=true
```

The Solr response for that query looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">4</int>
  </lst>
  <result name="response" numFound="0" start="0"/>
  <lst name="spellcheck">
    <lst name="suggestions">
      <lst name="other">
        <int name="numFound">1</int>
        <int name="startOffset">6</int>
        <int name="endOffset">11</int>
        <arr name="suggestion">
          <str>other</str>
        </arr>
      </lst>
      <lst name="book">
        <int name="numFound">1</int>
        <int name="startOffset">12</int>
        <int name="endOffset">16</int>
        <arr name="suggestion">
          <str>book</str>
        </arr>
      </lst>
      <str name="collation">name:(other book)</str>
    </lst>
  </lst>
</response>
```

As you can see for the preceding response, Solr corrected the spelling mistake we made. Now let's see how it works.

## How it works...

The index structure is pretty straightforward. It contains two fields: one for holding the unique identifier (the `id` field) and the other for holding the name (the `name` field). The file that contains the example data is simple too, so I'll skip discussing it.

The spellchecker component configuration is something we should look at a bit closer. It starts with the name of the component (the `name` attribute which, in our case, is `spellcheck`). Then we have the `class` attribute which specifies the class that implements the component.

Under the `<lst name="spellchecker">` XML tag, we have the actual component configuration. The name of the spellchecker (`<str name="name">`) is an option which is not mandatory when we use a single spellchecker component. We used the default name. The field parameter (`<str name="field">`) specifies the field on the basis of which we will get the mistakes corrected. The `<str name="spellcheckIndexDir">` tag specifies the directory (relative to the directory where your index directory is stored) in which the spellchecker component index will be held. In our case, the spellchecker component index will be named `spellchecker` and will be written in the same directory as the actual Solr index. The last parameter (`<str name="buildOnCommit">`) tells Solr to build the spellchecker index every time the commit operation is performed. Remember that it is crucial to build the index of the spellchecker with every commit, because the spellchecker is using its own index to generate the corrections.

The request handler we defined will be used by Solr as the default one (attribute `default="true"`). As you can see, we told Solr that we want to use the spellchecker component by adding a single string with the name of the component in the `last-components` array tag.

Now let's look at the query. We send the `boak` and `othar` words in the query parameter (`q`). We also told Solr to activate the spellchecker component by adding the `spellcheck=true` parameter to the query. We also told Solr that we want a new query to be constructed for us by adding the `spellcheck.collate=true` parameter. And that's actually all when it comes to the query.

Finally, we come to the results returned by Solr. As you can see, there were no documents found for the word `boak` and the word `othar`, and that was what we were actually expecting. However, as you can see, there is a spellchecker component section added to the results list (`<lst name="spellcheck">` tag). For each word there is a suggestion returned by Solr (the tag `<lst name="boak">` is the suggestion for the `boak` word). As you can see, the spellchecker component informed us about the number of suggestions found (the `<int name="numFound">`), about the start and end offset of the suggestion (`<int name="startOffset">` and `<int name="endOffset">`) and about the actual suggestions (the `<arr name="suggestion">` array). The only suggestion that Solr returned was the word `book` (`<str>book</str>` under the suggestion array). The same goes for the second word.

There is an additional section in the spellchecker component results generated by the `spellcheck.collate=true` parameter, `<str name="collation">name:(other book)</str>`. It tells us what query Solr suggested to us. We can either show the query to the user or send it automatically to Solr and show our user the corrected results list—this one is up to you.



## See also

If you would like to get deeper into the spellchecker component, please visit the blog written by Emmanuel Espina at <http://emmaespina.wordpress.com/2011/01/31/20/>.

## Using "group by" like functionalities in Solr

Imagine a situation where you have a number of companies in your index. Each company is described by its unique identifier, name, and main office identifier. The problem is that you would like to show only one company with the given main office identifier. In other words, you would like to group by that data. Is this possible in Solr? Yes and this recipe will show you how to do it.

## Getting ready

There is one thing that you need to know. The described functionality is not available in Solr 3.1 and lower. To use this functionality, you need to get Solr from the trunk of the Lucene/Solr SVN repository.

## How to do it...

Let's start with the index structure (just add this to your `schema.xml` file to the fields section):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="mainOfficeId" type="int" indexed="true" stored="true" />
```

The example data file looks like this:

```
<add>
<doc>
  <field name="id">1</field>
  <field name="name">Company 1</field>
  <field name="mainOfficeId">1</field>
</doc>
<doc>
  <field name="id">2</field>
  <field name="name">Company 2</field>
  <field name="mainOfficeId">2</field>
</doc>
<doc>
  <field name="id">3</field>
  <field name="name">Company 3</field>
```

```

    <field name="mainOfficeId">1</field>
  </doc>
</add>

```

Let's assume that our hypothetical user sends a query for the word `company`. In the search results, we want to show only one document with the same `mainOfficeId` field value. To do that, we send the following query to Solr:

```

http://localhost:8983/solr/select?q=name:company&group=true&group.
field=mainOfficeId

```

The response that was returned from Solr was as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="group.field">mainOfficeId</str>
      <str name="group">true</str>
      <str name="q">name:company</str>
    </lst>
  </lst>
  <lst name="grouped">
    <lst name="mainOfficeId">
      <int name="matches">3</int>
      <arr name="groups">
        <lst>
          <int name="groupValue">1</int>
          <result name="doclist" numFound="2" start="0">
            <doc>
              <str name="id">1</str>
              <int name="mainOfficeId">1</int>
              <str name="name">Company 1</str>
            </doc>
          </result>
        </lst>
        <lst>
          <int name="groupValue">2</int>
          <result name="doclist" numFound="1" start="0">
            <doc>
              <str name="id">2</str>
              <int name="mainOfficeId">2</int>
              <str name="name">Company 2</str>
            </doc>
          </result>
        </lst>
      </arr>
    </lst>
  </lst>

```

```
        </result>
      </lst>
    </arr>
  </lst>
</lst>
</response>
```

As you can see, the results list is a little bit different from the one that we are used to. Let's see how it works.

### How it works...

The index structure is pretty straightforward. It contains three fields: one for holding the unique identifier (the `id` field), one for holding the name (the `name` field), and one for holding the identifier of the main office (the `mainOfficeId` field). The file that contains the example data is simple too, so I'll skip discussing it.

Now let's look at the query. We send the `company` word in the query parameter (`q`). In addition to that, we have two new additional parameters. The `group=true` parameter tells Solr that we want to use the grouping mechanism. In addition to that we need to tell Solr what field should be used for grouping—to do that we use the `group.field` parameter, which in our case is set to the `mainOfficeId` field.

So let's have a look at how Solr behaves with the given example query. Take a look at the results list. Instead of the standard search results, we got everything grouped under the `<lst name="grouped">` XML tag. For every field (or query) passed to the grouping component (in our case by the `group.field` parameter) Solr creates an additional section—in our case, the `<lst name="mainOfficeId">` XML tag. Next thing we see is the `<int name="matches">` tag, which tells us how many documents were found for the given query. And finally, we have the grouped results under the `<arr name="groups">` XML tag.

For every unique value of the `mainOfficeId` field we have a group returned by Solr. The `<int name="groupValue">` tells us about the value for which the group is constructed. In our example, we have two groups, one for the 1 value and the other for the 2 value. Documents that are in a group are described by the `<result name="doclist">` XML tag. The `numFound` attribute of that tag tells how many documents are in the given group and the `start` attribute tells us from which index the documents are shown. Then for every document in the group, a `<doc>` XML tag is created. The `<doc>` tag contains the information about the fields just like the usual Solr result list.

One thing you should know when using the grouping functionality. Right now the functionality does not support the distributed search and grouping is not supported for multivalued fields.

## There's more...

### Fetching more than one document in a group

There are situations where you would like to get more than one default document for every group. To do that you should add the `group.limit` parameter with the desired value. For example, if you would like to get a maximum of four documents shown in every group, the following query should be sent to Solr:

```
http://localhost:8983/solr/select?q=name:company&group=true&group.  
field=mainOfficeId&group.limit=4.
```



# 10

## Dealing with Problems

In this chapter, we will cover:

- ▶ How to deal with a corrupted index
- ▶ How to reduce the number of files the index is made of
- ▶ How to deal with a locked index
- ▶ How to deal with too many opened files
- ▶ How to deal with out of memory problems
- ▶ How to sort non-English languages properly
- ▶ How to deal with the infinite loop exception when using shards
- ▶ How to deal with garbage collection running too long
- ▶ How to update only one field in all documents without the need of full indexation
- ▶ How to make your index smaller

### Introduction

Every Solr deployment will, sooner or later, have some kind of problem. It doesn't matter if the deployment is small and simple or if it's big and complicated, containing multiple servers and shards. In this chapter, I'll try to help you with some of the problems that you can run into when running Solr. I hope this will help you make your day easier.

### How to deal with a corrupted index

It is night. The phone is ringing. You answer it and hear, "We've got a problem—the index is corrupted". What can we do? Is there anything besides full indexation or restoring from backup? There is something we can do and this recipe will show that.

## How to do it...

For the purpose of the recipe, let's suppose that we have a corrupted index that we want to check and fix. To use the `CheckIndex` tool, you need to change the working directory to the one containing the Lucene libraries. When in the proper directory, you run the following command:

```
java -cp LUCENE_JAR_LOCATION -ea:org.apache.lucene... org.apache.lucene.index.CheckIndex INDEX_PATH -fix
```

Where `INDEX_PATH` is the path to the index and `LUCENE_JAR_LOCATION` is the path to the Lucene libraries. For example, `/usr/share/solr/data/index` and `/solr/lucene/`. So, with the given index location, the command would look like this:

```
java -cp /solr/lucene/lucene-core-3.1-SNAPSHOT.jar -ea:org.apache.lucene... org.apache.lucene.index.CheckIndex /usr/share/solr/data/index -fix
```

After running the preceding command, you should see a series of information about the process of index repair. In my case, it looked like this:

Opening index @ E:\Solrsolrdata\index

```
Segments file=segments_2 numSegments=1 version=FORMAT_DIAGNOSTICS [Lucene 2.9]
```

```
1 of 1: name=_0 docCount=19
```

```
compound=false
```

```
hasProx=true
```

```
numFiles=11
```

```
size (MB)=0,018
```

```
diagnostics = {os.version=6.1, os=Windows 7, lucene.version=2.9.3 951790  
- 2010-06-06 01:30:55, source=flush, os.arch=x86, java.version=1.6.0_23,  
java.vendor=Sun Microsystems Inc.}
```

```
no deletions
```

```
test: open reader.....FAILED
```

```
WARNING: fixIndex() would remove reference to this segment; full  
exception:
```

```
org.apache.lucene.index.CorruptIndexException: did not read all bytes  
from file "_0.fnm": read 150 vs size 152
```

```
at org.apache.lucene.index.FieldInfos.read(FieldInfos.java:370)
```

```
at org.apache.lucene.index.FieldInfos.<init>(FieldInfos.java:71)
```

```
at org.apache.lucene.index.SegmentReader$CoreReaders.<init>(SegmentReader.java:119)
```

```
at org.apache.lucene.index.SegmentReader.get(SegmentReader.java:652)
```

```

at org.apache.lucene.index.SegmentReader.get(SegmentReader.java:605)
at org.apache.lucene.index.CheckIndex.checkIndex(CheckIndex.java:491)
at org.apache.lucene.index.CheckIndex.main(CheckIndex.java:903)

WARNING: 1 broken segments (containing 19 documents) detected
WARNING: 19 documents will be lost

NOTE: will write new segments file in 5 seconds; this will remove 19 docs
from the index. THIS IS YOUR LAST CHANCE TO CTRL+C!
5...
4...
3...
2...
1...
Writing...
OK
Wrote new segments file "segments_3"

```

And that's all. After that, you should have the index processed and depending on the case, you can have your index repaired. Now let's see how it works.

### How it works...

As you see, the command-line instruction runs the `CheckIndex` class from the `org.apache.lucene.index` package. We also provided the absolute path to the directory containing the index files, the library that contains the necessary classes and the `-fix` parameter, which tells the `CheckIndex` tool to try to repair any errors found in the index structure. In addition, we have provided the `ea` parameter to enable the assertions. We did that to make the test more accurate. Let's look at the response that the `CheckIndex` tool provided. As you can see, we have information about the segments, the number of documents, and the version of Lucene used to build the index. We can also see the number of files that the index consists of, the operating system, and so on. This information may be useful, but is not crucial. The most interesting thing for us is the following information:

```

WARNING: 1 broken segments (containing 19 documents) detected
WARNING: 19 documents will be lost

```

This tells us that the `CheckIndex` tool found one broken segment, which contains 19 documents, and that all the documents will be lost in the repair process. This is not always the case, but it can happen and you should be aware of that.



The next lines of the `CheckIndex` tool response tell us about the process of writing the new segment files which will be repaired. And that's actually all. Of course, when dealing with larger indexes, the response generated by the `CheckIndex` tool will be much larger and will contain information about all the segments of the index. The shown example is simple but illustrates how the tool works.

When using the `CheckIndex` tool, you need to be very careful. There are many situations where the index files can't be repaired and the `CheckIndex` tool will result in the deletion of all the documents in the index. That's not always the case, but you should be aware of that and be extra careful. For example, it is a good practice to make a backup of the existing index before running the `CheckIndex` tool.

### There's more...

There is one more thing worth noting when talking about the `CheckIndex` tool.

### Checking the index without the repair procedure

If you only want to check the index for any errors without the need to repair it, you can run the `CheckIndex` tool in this mode. To do that, you run the command-line fragment shown in the recipe without the `-fix` part. For example:

```
java -ea:org.apache.lucene... org.apache.lucene.index.CheckIndex /usr/
share/solr/data/index
```

## How to reduce the number of files the index is made of

The normal situation after running Solr for a period of time is getting your index split between multiple files, but that's not the only issue. The index split between multiple files causes the performance to drop. Is there anything we can do about it? Yes and it is actually pretty simple. This recipe will show you how to do it.

### How to do it...

In order to make the index more compound and merge all the segment files into one file, we need to run the `optimize` command. To do that, we run the `optimize` command in the following way:

```
curl 'http://localhost:8983/solr/update' --data-binary '<optimize/>' -H
'Content-type:text/xml; charset=utf-8'
```

Depending on your index size and your machine, after some time you should see the following response (this may be different depending on the Solr version that you are using):

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader"><int name="status">0</int><int
name="QTime">73279</int></lst><str name="WARNING">This response format is
experimental. It is likely to change in the future.</str>
</response>
```

## How it works...

The idea behind the `optimize` command is to create a new segment file that will contain all the older segment files and remove the old ones. During the work of the `optimize` command, the index size may double due to the writing of the new segment file, but will return to its normal size (or less) just after the command finishes running.

The logic of the `optimize` command is not visible to the users and thus I'll skip the detailed information about it as it is not crucial to understand the basics of this command.

The command presented in the recipe was run on a Unix based operating system using the `curl` command. Of course, you can use any other operating system and tool where you can send requests using `HTTP POST` method. You can also use the SolrJ library to send the command from a Java application.

It is good to have an `optimize` command running in a set period of time; for example, once a day. But remember to run it only on the master server and during the time when the master server is not used as much as during the peak indexing time. The `optimize` command uses the I/O operations heavily, and this can and will affect the performance of the server you send the `optimize` command to. Also, remember to send the `optimize` command to every core you have deployed on your Solr server.

You should always measure the performance of Solr servers on optimized and non-optimized indexes to ensure that the optimization is really needed.

## How to deal with a locked index

Sometimes bad things can happen during the indexing process. It may be a number of different things: from the failure of the machine to some problems with the virtual machine, and so on. Due to the error, we can have problems with the index—one of which is the lock of the index. This happens when the lock file is left in the index directory and because of that, you can't alter the index. Is there something we can do about the locked index? Yes. It's easy and this recipe will show you what you can do about it.

## How to do it...

First, let's assume that the Java virtual machine that we use had a critical failure and killed our Solr master instance during indexing. The typical exception thrown by the Jetty servlet container would look like this (this happened during the commit operation):

```
SEVERE: Exception during commit/optimize:java.io.IOException: Lock
obtain timed out: SimpleFSLock@/usr/share/solr/data/index/lucene-
fe3fc928a4bbfeb55082e49b32a70c10-write.lock
```

Now the hard... hmmm easy part: If you are 100% sure that there is no indexing process running and the lock is there because of the error, then go to the directory containing the index files. One of the files in the `/usr/share/solr/data/index/` directory is the one we are searching for. That file is:

```
lucene-fe3fc928a4bbfeb55082e49b32a70c10-write.lock
```

So in order to get rid of our problem, we only need to remove that file and restart Jetty. And that's all.

## How it works...

The lock logic is actually pretty simple. When there is a file named `lucene-*-write.lock`, there is an index lock present. When the lock is present, you can't alter the index in any way—you can't add or delete the documents from it. The locking of the index is a good thing—it ensures that no more than one index writer is writing to the segment at the same time. However, on some rare occasions, there is a possibility that the lock was created and not removed, and is thus preventing us from altering the index when it should not do that.

If we are sure that the index lock is there when it shouldn't be, we just need to delete a file named `lucene-*-write.lock` that is present in the index directory and restart Solr. Before removing the lock file, please ensure that the file isn't there because of the indexing process taking place.

## How to deal with too many opened files

Sometimes you may encounter a strange error, a thing that lays on the edge between Lucene and the operating system—the too many files opened exception. Is there something we can do about it? Yes we can! And this recipe will show you what.

## Getting ready

Before reading this recipe further, I encourage you to read *How to reduce the number of files the index is made of* recipe in this chapter.

## How to do it...

For the purpose of the recipe, let's assume that the header of the exception thrown by Solr looks like this:

```
java.io.FileNotFoundException: /use/share/solr/data/index/_1.tii (Too many open files)
```

What to do instead of trying to pull your hair out of your head? First of all, this probably happened on a Unix/Linux-based operating system. Let's start with setting the opened files' limit higher. To do that, we use the `ulimit` command-line utility. For example, I had mine set to 2000 and I wanted to double the value. So the command line should look like this:

```
ulimit -n 4000
```

But that will only make things good till the next time. The probable cause of the too many open files exception is because the index consists of too many segment files. So, the first thing after using the `ulimit` command utility should be index optimization.

The next thing to consider is `mergeFactor` lowering to make things simple—the lower the `mergeFactor` setting, the less files will be used to construct the index. So let's set the `mergeFactor` to 2. We modify the following line in the `solrconfig.xml` file and set it with the appropriate value (2 in our case):

```
<mergeFactor>2</mergeFactor>
```

If that doesn't help, we can always use the compound index structure. To use the compound index structure, we add the following line to the `solrconfig.xml` file:

```
<useCompoundFile>true</useCompoundFile>
```

After we set the configuration value, we need to run the optimization of the index. And that's all.

Now let's see what the options mean.

## How it works...

Let's start with the `ulimit` command-line utility. Basically, this utility lets you set the maximum number of files that can be opened at the same time.

Next, we have `mergeFactor`. This configuration utility lets you determine the number of segments, and thus the number of files, that create the index. In our case, setting the `mergeFactor` to 2 will result in two segments and thus, much less files than the default value of 10. When changing the `mergeFactor` value, you must be aware of a few things: The lower the `mergeFactor` setting, the longer the indexing time will be, and will thus improve the search speed. On the other hand, the higher the `mergeFactor` setting, the less time indexing will take, but search time may degrade as there will be more segments, and thus more files that create the index.

The last thing to discuss is the `useCompoundFile` setting, which we set to `true`. I decided not to get deep into the details. Basically this setting will tell Solr and Lucene to use the compound index structure more and will thus reduce the number of files that create the index. The downside of this is that the replication mechanism will have to fetch data files which are much larger than when using Solr with `useCompoundFile` set to `false`.

## See also

If you want to know more about merge policies, I suggest reading Juan Gran's blog located at the following URL:

<http://juanggrande.wordpress.com/2011/02/07/merge-policy-internals/>

## How to deal with out of memory problems

As with every application written in Java, sometimes memory problems happen. When talking about Solr, those problems are usually related to the heap size getting too low. This recipe will show you how to deal with those problems and what to do to avoid them.

## How to do it...

So, what to do when you see an exception like this:

```
SEVERE: java.lang.OutOfMemoryError: Java heap space
```

First of all, you can do something to make your day a bit easier—you can add more memory for the Java virtual machine to use. To do that, you need to add the `Xmx` and preferably the `Xms` parameter to the start script of your servlet container (Apache Tomcat or Jetty). To do that, I used the default Solr deployment and modified the parameters. This is how Solr was run with more than the default heap size:

```
java -Xmx1024M -Xms512m -jar start.jar
```

That should help you get through the day, but in the end you would like to try to reduce Solr's memory usage. First, look at your index to see if each and every file in the index needs to be there. Remember that each field that builds the documents must fit into the memory.

Then look at your queries. How your queries are built, how you use the faceting mechanism, and so on (`facet.method=fc` tends to use less memory when the field has many unique terms in the index). Remember that fetching too many documents at one time may cause Solr to run out of heap memory (for example, when setting a large value for the query result window). The same may happen when you try to calculate too many faceting results. One last thing—the size of the caches may also be one of the reasons for the memory problems.

## How it works...

So what do the `Xmx` and `Xms` Java virtual machine parameters do? The `Xms` parameter specifies how much heap memory should be assigned by the virtual machine at the start and thus, this is the minimal size of heap memory that will be assigned by the virtual machine. The `Xmx` parameter specifies the maximum size of the heap. The Java virtual machine will not be able to assign more memory for the heap than the `Xmx` parameter.

You should remember one thing—sometimes it's good to set the `Xmx` and `Xms` parameter to the same values. It will ensure that the virtual machine won't resize the heap size during application execution and thus won't lose precious time for heap resizing.

## There's more...

There is one more thing I would like to mention:

### Seeing heap when out of memory error occurs

If the out of memory errors happen even after the actions you've taken, you should start monitoring your heap. One of the easiest ways to do that is by adding the appropriate Java virtual machine parameters. Those parameters are `XX:+HeapDumpOnOutOfMemoryError` and `XX:HeapDumpPath`. These two parameters tell the virtual machine to dump the heap on the out of memory error and write it to a file created in the specified directory. So the default Solr deployment start command would look like this:

```
java -jar -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/var/log/dump/
start.jar
```

## How to sort non-English languages properly

As you probably already know, Solr supports UTF-8 encoding and can thus handle data in many languages. But if you ever needed to sort some languages that have characters specific to them, you would probably know that this doesn't work well on the standard Solr `string` type. This recipe will show you how to deal with sorting and Solr.

## How to do it...

For the purpose of the recipe, I assumed that we will have to sort the text that contains Polish characters. To show the good and bad sorting behavior, I created the following index structure (add this to your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
```

```
<field name="name_sort_bad" type="string" indexed="true" stored="true" />
<field name="name_sort_good" type="text_sort" indexed="true"
stored="true" />
```

I also defined some copy fields to automatically fill the `name_sort_bad` and `name_sort_good` fields. Here is how they are defined (add this after the fields section in the `schema.xml` file):

```
<copyField source="name" dest="name_sort_bad" />
<copyField source="name" dest="name_sort_good" />
```

The last thing about the `schema.xml` file is the new type. So the `text_sort` definition looks like this:

```
<fieldType name="text_sort" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.KeywordTokenizerFactory" />
    <filter class="solr.CollationKeyFilterFactory" language="pl"
country="PL" strength="primary" />
  </analyzer>
</fieldType>
```

The test data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Łaka</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Lalka</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Zab</field>
  </doc>
</add>
```

First, let's take a look at how the incorrect sorting order looks like. To do this, we send the following query to Solr:

```
http://localhost:8983/solr/select?q=*&sort=name_sort_
bad+asc&indent=true
```

The response that was returned for the preceding query was as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">1</int>
  <lst name="params">
    <str name="q">*:*</str>
    <str name="sort">name_sort_bad asc</str>
  </lst>
</lst>
<result name="response" numFound="3" start="0">
  <doc>
    <str name="id">2</str>
    <str name="name">Lalka</str>
    <str name="name_sort_bad">Lalka</str>
    <str name="name_sort_good">Lalka</str>
  </doc>
  <doc>
    <str name="id">3</str>
    <str name="name">Zab</str>
    <str name="name_sort_bad">Zab</str>
    <str name="name_sort_good">Zab</str>
  </doc>
  <doc>
    <str name="id">1</str>
    <str name="name">Łaka</str>
    <str name="name_sort_bad">Łaka</str>
    <str name="name_sort_good">Łaka</str>
  </doc>
</result>
</response>

```

Now let's send the query that should return the documents sorted in the correct order. The query looks like this:

```
http://localhost:8983/solr/select?q=*:*&sort=name_sort_
good+asc&indent=true
```

And the results returned by Solr are as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">6</int>
  <lst name="params">

```



```
<str name="q">*:*</str>
<str name="sort">name_sort_good asc</str>
</lst>
</lst>
<result name="response" numFound="3" start="0">
  <doc>
    <str name="id">2</str>
    <str name="name">Lalka</str>
    <str name="name_sort_bad">Lalka</str>
    <str name="name_sort_good">Lalka</str>
  </doc>
  <doc>
    <str name="id">1</str>
    <str name="name">Łaka</str>
    <str name="name_sort_bad">Łaka</str>
    <str name="name_sort_good">Łaka</str>
  </doc>
  <doc>
    <str name="id">3</str>
    <str name="name">Zab</str>
    <str name="name_sort_bad">Zab</str>
    <str name="name_sort_good">Zab</str>
  </doc>
</result>
</response>
```

As you can see, the order is different and believe me it's correct. Now let's see how it works.

## How it works...

Every document in the index is built on four fields. The `id` field is responsible for holding the unique identifier of the document. The `name` field is responsible for holding the name of the document. The last two fields are used for sorting.

The `name_sort_bad` field is nothing new—it's just a `string` based field which is used to perform sorting. The `name_sort_good` field is based on a new type—the `text_sort` field type. The field is based on the `solr.TextField` type and on `solr.KeywordTokenizerFactory`, which basically means that our text won't be tokenized. We used this trick because we want to sort on that field and thus we don't want the data in it to be tokenized. But on the other hand, we need to use the new filter to be able to sort in the correct order. The filter that allows Solr to sort correctly is the `solr.CollationKeyFilterFactory` filter. We used three attributes of this filter. The first is the `language` attribute, which tells Solr about the language of the field. The second is the `country` attribute, which tells Solr about the country variant (this can be skipped if necessary).

These two attributes are needed by Solr to construct the proper locale. The `strength` attribute informs Solr about the collation strength used. More information about those parameters can be found in the Sun Java Virtual Machine documentation. One thing crucial—you need to create an appropriate field and set the appropriate attributes value for every non-English language you want to sort.

The two queries you can see in the examples differ in only one thing—the field used for sorting. The first query is uses the `string` based field—`name_sort_bad`. When sorting on this field, the document order will be incorrect when there will be non-English characters present. However, when sorting on the `name_sort_good` field, everything will be in the correct order as shown in the example.

## See also

If you want to know more about Java collator mechanism, please visit the following URL:

<http://download.oracle.com/javase/1.5.0/docs/api/java/text/Collator.html>

## How to deal with the infinite loop exception when using shards

One of the inconveniences when using shards is the necessity to add the shards address to every query, so you thought it'll be good to write it in the `solrconfig.xml` file and let Solr add the address of the shards. So, you add it to the default handler and when you run the first query, the infinite loop exception is thrown. So, is there a way to avoid this exception and write shards addresses to the handler? Yes there is! And this recipe will show you how to do it.

## How to do it...

Let's assume that you have the following handler defined in your `solrconfig.xml` file on the Solr server you send queries to (its IP address is `192.168.2.1`):

```
<requestHandler name="standard" class="solr.SearchHandler"
  default="true">
  <lst name="defaults">
    <str
  name="shards">192.168.2.1:8983/solr,192.168.2.2:8983/
  solr,192.168.2.3:8983</str>
    </lst>
  </requestHandler>
```

When you send a query to that handler, you get the infinite loop exception. To avoid getting this exception, instead of adding the shards to the default search handler, you should add a new one. For example:

```
<requestHandler name="shardsearch" class="solr.SearchHandler">
  <lst name="defaults">
    <str
name="shards">192.168.2.1:8983/solr,192.168.2.2:8983/
solr,192.168.2.3:8983</str>
  </lst>
</requestHandler>
```

So now, when sending a query that should use shards instead of using the default search handler, you should use the one named `shardsearch`.

### How it works...

The problem is quite simple. The infinite loop exception is thrown because when querying shards, Solr uses the default request handler. That's why when a shard tries to fetch the results, it uses the handler with the shards defined, and thus is trying to query shards again and again. This causes the infinite loop exception.

When we define the new request handler, which is not default, we send the first query to that handler. Then the sharding mechanism uses the default search handler to fetch the search results. As now the default search handler doesn't have the shards information, it doesn't fall into the infinite loop, and thus we have our query working correctly.

## How to deal with garbage collection running too long

As with all the applications running inside the Java virtual machine, Solr can also have problems with garbage collection running too long. You may never encounter this kind of a problem, or you may suffer from it and don't know what is actually happening. This recipe will show you how to deal with garbage collection running too long.

### How to do it...

For the purpose of this recipe, I assumed that we have a machine with multiple cores (or multiple processor units), Java 6, and run Solr with the following command:

```
java -Xms256M -Xmx2048 -jar start.jar
```

After running for a longer period of time, you notice that Solr starts to hang for short periods of time and doesn't respond during that time. Actually, Jetty doesn't respond either. After that, everything goes back to normal for some time and then it happens again. This usually means that we have a problem with garbage collection running too long.

What can we do about that? Let's modify the Solr start command to the following:

```
java -Xms256M -Xmx2048 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC  
-XX:ParallelGCThreads=4 -XX:SurvivorRatio=2 -jar start.jar
```

If you run some tests, you notice that this does the trick in most cases. But how does it work? Let's see.

### How it works...

So what does the magic option mean? Let's take a look. `-XX:+UseConcMarkSweepGC` tells JVM to use the concurrent mark-sweep collection for the old generation. This basically means that the garbage collection for the old generation will run concurrently. `-XX:+UseParNewGC` will turn on the parallel garbage collection in the young generation. The next parameter, `-XX:ParallelGCThreads`, specifies the number of threads used to perform garbage collection in the young generation. In our case, we set it to the value equal to the number of cores we have on our machine. The last parameter, `-XX:SurvivorRatio`, specifies the percent value of the survivor heap space that must be used before objects are promoted to the old generation. I know it sounds strange. Actually, it is strange.

Remember that sometimes it's useful to add more memory (via the `Xmx` parameter) than play around with the Java Virtual Machine tuning.

### There's more...

When talking about garbage collection, it is sometimes useful to know how to monitor it.

### Monitoring the garbage collector

In case you want to diagnose to see if it's really the garbage collector that causes the problem, you can do that fairly easily. Just add the `-Xloggc:gc.log` parameter to the start command. This option will write the garbage collector output to the `gc.log` file. The command from the recipe with the garbage collector logging would look like this:

```
java -Xms256M -Xmx2048 -Xloggc:gc.log -jar start.jar
```

### See also

If you want to know more about the Java Virtual Machine tuning, please refer to the documentation of the appropriate version of the virtual machine you are using. The documentation can be found at Oracle/Sun websites. For Java 5, the documentation on how to tune garbage collection can be found at the following URL:

<http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html>

## How to update only one field in all documents without the need of full indexation

As you probably know, Solr doesn't allow you to update a single field in the document that is written in the index. The standard way to update some information in the document is by adding the whole document once again, which simply means that Solr will first remove the document and then add the new version to the index. But what if you need to update a single value that you use for boosting (for example, the number of visitors that clicked the product) for millions of documents every day? Is there a more optimal way to do that than full indexation of all the documents? Yes there is! And this recipe will show you how to do it.

### How to do it...

For the purpose of the recipe, I assumed that we will have three fields that describe the document. I created the following index structure (add this to your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="visits" type="visitsFieldType" />
```

Now let's add two types to the `schema.xml` file:

```
<fieldType name="floatOld" class="solr.FloatField" omitNorms="true"/>
<fieldType name="visitsFieldType" keyField="id" defVal="0"
stored="false" indexed="false" class="solr.ExternalFileField"
valType="floatOld" />
```

The test data looks like this:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="name">Book one</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="name">Book two</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="name">Book three</field>
  </doc>
</add>
```

As you noticed, we didn't add the visits data. Now create the file named `external_visits` in the directory where the `index` directory is located (I'm talking about the `data` directory, not the `data/index` directory). The file contains the following data:

```
1=200
2=20
3=50
```

Now, after Solr starts, you can use the data that was written in the `external_visits` file in the boosting functions. For example, the following query would work perfectly fine:

```
http://localhost:8983/solr/select?q=book&bf=log(visits)&qt=dismax
```

So how does that work?

### How it works...

As you can see, every document in the index is described by three fields. One is responsible for the unique identifier (the `id` field), the other is responsible for the name of the document (the `name` field), and the last one is responsible for holding the number of visits (the `visits` field).

As you can see, the `visits` field is not based on the field types you are used to. It is based on `visitsFieldType`, which is based on `solr.ExternalFieldType`. This type allows Solr to fetch data from the external file, which needs to satisfy the following rules:

- ▶ The file needs to be placed in the same directory as the index directory
- ▶ The file needs to be named `external_FIELDNAME`; so in our example, it'll be `external_visits`
- ▶ The data inside the external file needs to be float or Solr won't fetch the data from the file
- ▶ The data from the file can only be used in the function queries

Now let's look at the additional attributes of the `visitsFieldType` type. The `keyField` attribute specifies which field in the index is responsible for holding the unique identifier of the document. The `defVal` attribute specified the default value for the field based on that type. The last attribute we should take a look at is `valType`, which specifies the type of data that is in the external field. Please note that at the moment of writing the book, the only type accepted by Solr for `solr.ExternalFieldType` was `solr.FloatField` and thus I defined the `oldFloat` type, which is based on `solr.FloatField`.

As you can see in the example data, I didn't include the visits data in the standard file. This data is written in the `external_visits` file. The structure of this file is simple. Every line contains the unique identifier of the document, the `=` character, and the field value. For example, the `1=200` line means that the document with the unique identifier `1` will have the visits field value of `200`.

There are two more things you have to know when using the external fields. The first is that you can't actually use the data from the external file; it may only be used as a value source in the function query. This means that you can't search, sort, or display this data—you can only use it inside functions. The second thing is the visibility of the file. After you update the file, you have to send the commit command to Solr. After the commit command, Solr will reload the contents of the file and use it.

## How to make your index smaller

There are situations where you would like to make your index smaller. For example, you would like it to fit into the RAM memory. How to do that? This recipe will try to help you with the process of index slimming.

### How to do it...

For the purpose of the recipe, I assumed that we will have four fields that describe the document. I created the following index structure (add this to your `schema.xml` file):

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="description" type="text" indexed="true" stored="true" />
<field name="price" type="string" indexed="true" stored="true" />
```

Let's assume that our application has the following requirements:

- ▶ We need to search in the name and description fields
- ▶ We show two fields: id and price
- ▶ We don't use highlighting and spellchecker

The first thing we should do is set the `stored="false"` attribute for the name and description fields.

Next, we set `indexed="false"` for the price field.

The last thing is adding the term options. We add `termVectors="false"`, `termPositions="false"`, and `termOffsets="false"` to the name and description fields. The modified schema looks like this:

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="name" type="text" indexed="true" stored="false"
termVectors="false" termPositions="false" termOffsets="false"/>
<field name="description" type="text" indexed="true" stored="false"
termVectors="false" termPositions="false" termOffsets="false"/>
<field name="price" type="string" indexed="false" stored="true" />
```

Let's check the index size now. I've indexed 1,000,000 sample documents with the use of the original `schema.xml` file. The index size was 329 237 331 bytes. After changing the `schema.xml` file and indexing the same data, the index size was 84 301 603 bytes. So, as you see, the index size was reduced.

Of course, you should also perform the optimization of the index to delete unused data from it.

Now let's see why we see reduction of the index size.

### How it works...

The first `schema.xml` file you see is the standard index structure provided with the Solr example deployment (at least when talking about the types). We have four fields. All of them are indexed and stored, which means all of them are searchable and are shown in the result list.

Now, let's look at the requirements. First of all, we only need to search in the `name` and `description` fields, which mean that the rest of the fields can be set up as not indexed (`indexed="false"` attribute). We set that for the `price` field—we set the `id` field to be searchable, just in case. When the `indexed` attribute is set to `false`, the information in that field is not indexed, which basically means that it is written into the index structure, which enables the field to be searchable. This saves the index space. Of course, you can't set this attribute to `false` if you need this field to be searchable.

The second requirement tells us what fields we are obligated to show in the search results. Those fields are the ones that need the `stored` attribute to be set to `true`; the rest can have this attribute set to `false`. When we set this attribute to `false`, we tell Solr that we don't want to store the original value—the one before analysis and thus, we don't want this field to be included in the search results. Setting this attribute to `true` in many fields will increase the index size substantially.

The last requirement is actually an information—we don't need to worry about the highlighting functionality so we can reduce the index size in a greater way. To do that, we add `termVectors="false"`, `termPositions="false"`, and `termOffsets="false"` to the `name` and `description` fields. By doing that, we tell Solr not to store some information about terms in the index. This basically means that we can't use the highlighting functionalities of Solr, but we have reduced our index size substantially and don't need highlighting.

Last few words—every time you think about reducing the index size, first do the optimization, then look at your `schema.xml` file and see if you need all those fields. Then check which fields shouldn't be stored and which you can omit when indexing. The last thing should be removing the information about terms because there may come a time when you will need this information and the only thing you will be able to do is the full indexation of millions of documents.



## See also

If you want to read a deep analysis about index size, please take a look at the following URL:

<http://juanggrande.wordpress.com/2010/12/20/solr-index-size-analysis/>

# Index

## A

### **additional information**

storing, payloads used 64, 65

### **Apache Nutch**

about 28

downloading 28

reference link 28

### **Apache Solr** *See Solr*

### **Apache\_Solr\_Service object 210**

### **Apache Tika**

setting up, with Solr 33

### **Apache Tomcat**

port, changing 11

Solr, running on 9, 10

### **autocommits statistic 102**

### **autosuggest feature**

implementing, faceting used 173-175

### **avgTimePerRequest attribute 100**

## B

### **BaseTokenFilterFactory class 220**

### **Buffer overflow 8**

## C

### **cacheControl tag 199**

### **cache status**

checking 93, 94

### **commit statistic 102**

### **contents**

copying, from dynamic fields to one field 70

copying, from one field to another 68, 69

number of characters copied, limiting 70

### **core admin interface**

reference link 23

### **corrupted index**

dealing with 259-262

### **crawl command 30**

### **crawl-urlfilter.txt file 30**

### **CSV format 36**

### **cumulative\_adds 102**

### **cumulative\_evictions parameter 94**

### **cumulative\_hitratio parameter 94**

### **cumulative\_hits parameter 94**

### **cumulative\_inserts parameter 94**

### **cumulative\_lookups parameter 94**

### **custom field type**

developing 224-227

### **custom filter**

developing 217-220

### **custom request handler**

developing 215-217

### **custom search component**

developing 221-223

## D

### **data**

importing, Data Import Handler and delta query used 55, 56

indexing, from database using Data Import Handler 52-55

indexing, in CSV format 36-38

indexing, in JSON format 40-42

indexing, in XML format 38-40

modifying, while importing with Data Import Handler 60-62

stemming 81, 82

### **data analysis**

about 64

additional information, storing using payloads 64, 65

- contents, copying from one field
    - to another 68, 69
  - data, stemming 81, 82
  - geographical points, storing in index 78-81
  - plural words, making singular 75
  - synonyms, using 70, 71
  - text, preparing for efficient trailing wildcard search 83-85
  - text, splitting by camel case 72, 74
  - text, splitting by numbers and non-white space characters 86, 87
  - text, splitting by whitespace only 74, 75
  - whole string, lowercasing 77, 78
  - words, changing 70-72
  - XML and HTML tags, eliminating 66, 67
  - Data Import Handler**
    - configuring, with JDBC 49-51
    - using, with URL Data Source 57-59
  - defined words**
    - ignoring 245-247
  - defType parameter 126**
  - deltainportQuery 57**
  - details command 115**
  - different facet limits**
    - having, for different fields in same query 181-184
  - different query parser**
    - selecting 125, 126
  - different response format**
    - selecting 206-208
  - disablepoll command 115**
  - docsPending statistic 102**
  - document cache**
    - about 26
    - configuring 189, 190
  - documentCache XML tag 190**
  - documents**
    - excluding, with QueryElevationComponent 135
    - positioning, over others on query 131-134
    - positioning, with words closer to each other 136, 137
    - retrieving, with partial match 141-143
  - documents, matching query and sub query**
    - retrieving 161-163
  - documents, without value in field**
    - retrieving 176-178
  - documents, with same date range**
    - retrieving 155-157
  - documents, with same field value**
    - retrieving 152-154
  - documents, with same value range**
    - retrieving 158-160
  - dynamicField tag 18**
- ## E
- enablepoll command 115**
  - event parameter 194**
  - evictions parameter 94**
  - ExampleFilter class 219**
  - ExampleRequestHandler class 217**
  - Extracting Request Handler**
    - setting up 34
- ## F
- faceting mechanism**
    - about 152
    - autosuggest feature, implementing 173-175
    - facets with counts greater than zero, displaying 154
    - results, sorting lexicographically 154
  - faceting results**
    - filters, removing 164-167
    - naming 167-169
    - retrieving 179, 181
    - sorting, alphabetically 170-172
    - sort order, selecting 173
  - fetchindex command 116**
  - field behavior**
    - checking 95-98
  - fields**
    - indexing, in dynamic way 17-19
  - filter cache**
    - about 26
    - configuring 191, 192
    - using, with faceting 27
  - filterCache XML tag 192**
  - filters**
    - about 64
    - removing, from faceting results 164-167
  - firstSearcher event 194**
  - forEach attribute 59**

## G

### **garbage collection running**

- dealing with 272
- monitoring 273

### **generateNumberParts parameter 87**

### **generateWordParts attribute 74**

### **generateWordParts parameter 87**

### **geographical points**

- storing, in index 78-81

### **getInternalValue method 227**

### **getVal method 227**

## H

### **handleRequestBody method 217**

### **highlight matches attribute 98**

### **hitratio parameter 94**

### **hits parameter 94**

### **httpCaching tag 199**

## I

### **ignoreCase attribute 72**

### **incrementToken method 220**

### **index**

- checking, without repair procedure 262
- multiple files, reducing 262, 263

### **indexing issues**

- solving 201

### **index slimming 276, 277**

### **index structure**

- analyzing 116-119

### **indexversion command 116**

### **infinite loop exception**

- dealing, when using shards 271, 272

### **inserts parameter 94**

## J

### **Java based index replication**

- configuration file names, changing 110
- setting up 108, 109
- Slave and HTTP Basic authorization 110

### **Java based replication status**

- checking 104, 105
- index fetching, aborting 116

managing, HTTP commands used 113-115

replication, disabling 116

replication, enabling 116

### **Jetty**

running, on different port 8

Solr, running on 6, 7

### **JSON format 40**

## L

### **light stemmers 76**

### **Linux curl command 42**

### **listener mechanism 194**

### **listener XML tag 193**

### **locked index**

- dealing with 263, 264

### **long text fields**

- highlighting 238-240

### **lookups parameter 94**

## M

### **master\_data\_dir variable 112**

### **masterUrl variable 110**

### **matched words**

- highlighting 235, 237

### **metadata**

- extracting, from binary files 47-49

### **Microsoft Office files**

- indexing 45, 46

### **multilingual data**

- making searchable with multicore deployment 19-22

### **multiple languages**

- handling, in single index 15-17

### **multiple opened files**

- dealing with 264-266

### **multiple thread crawling 30**

### **multiple values**

- querying, in same field 123

## N

### **newSearcher event 194**

### **non-English languages**

- sorting 267-271

### **NumberUtils class 227**

## O

**optimizes statistic** 102  
**out of memory problems**  
    dealing with 266, 267

## P

**particular field value**  
    asking for 122  
    querying, particular field value used 122  
**PDF files**  
    indexing 43, 44  
**phrase**  
    promoting, over words 128-131  
    promoting, with standard query parser 131  
    searching for 126-128  
**plural words**  
    making singular, without stemming 75  
**prepare method** 223

## Q

**qq parameter** 149  
**queries**  
    nesting 147-149  
**query result cache**  
    about 26  
    configuring 190, 191  
**queryResultCache XML tag** 191  
**queryResultMaxDocsCached property**  
    paging 189  
**query result window** 26, 27

## R

**relevant results**  
    getting, with early query termination 31, 32  
**requestDispatcher XML tag** 199  
**requests attribute** 100  
**result pages**  
    caching 198  
**results**  
    paging 188  
    sorting, by distance from point 138-141  
    sorting, by field value 123-125  
    sorting, by function value 240-242  
**rollbacks statistic** 102

**rsyncd\_port variable** 112

## S

**scoring**  
    affecting, with function 143-146  
**script based replication**  
    setting up 110-112  
**script based replication status**  
    checking 106, 108  
**SearchHandler class** 217  
**search results**  
    presenting, in easy way 232-235  
**serviceUrl attribute** 92  
**sharded deployment**  
    setting up 195-197  
**size parameter** 94  
**snappuller script** 112  
**Solr**  
    autosuggest feature, implementing using  
        faceting 173-175  
    core admin interface 23  
    corrupted index, dealing with 259-262  
    custom field type, developing 224-227  
    custom filter, developing 217-220  
    custom request handler, developing 215-217  
    custom search component,  
        developing 221-223  
    data files without header 38  
    Data Import Handler, configuring with JDBC  
        49-51  
    Data Import Handler, using with URL Data  
        Source 57-59  
    data, importing with Data Import  
        Handler 55-57  
    data indexing 35  
    data, indexing from database using Data  
        Import Handler 52-55  
    data, indexing in CSV format 36-38  
    data, indexing in JSON format 40-42  
    data, indexing in XML format 38-40  
    data, modifying while importing with Data  
        Import Handler 60-62  
    default HTML tags, changing 238  
    defined words, ignoring 245-247  
    different response format, selecting 206-208  
    documents matching query and sub query,  
        retrieving 161-163

- documents similar as returned,
  - retrieving 230-232
- documents without value in field,
  - retrieving 176-178
- documents with same date range,
  - retrieving 155-157
- documents with same field value,
  - retrieving 152-154
- documents with same value range,
  - retrieving 158-160
- faceting mechanism 152
- faceting results, naming 167-169
- faceting results, retrieving 179, 181
- faceting results, sorting
  - alphabetically 170-172
- fields, indexing in dynamic way 17-19
- fields, specifying for highlighting 238
- field, updating in all documents without indexing 274-276
- filters, removing from faceting
  - results 164-167
- garbage collection running, dealing with 272
- group by like functionalities, using 254, 256
- index slimming 276, 277
- infinite loop exception, dealing when using
  - shards 271, 272
- locked index, dealing with 263, 264
- long text fields, highlighting 238-240
- matched words, highlighting 235
- metadata, extracting from binary files 47-49
- Microsoft Office files, indexing 45, 46
- multiple files in index, reducing 262, 263
- multiple languages, handling
  - in single index 15-17
- multiple opened files, dealing with 264-266
- non-English languages, sorting 267-271
- out of memory problems, dealing
  - with 266, 267
- PDF files, indexing 43, 44
- reference link 6
- results, sorting by function value 240-242
- running, on Apache Tomcat 9, 10
- running, on Jetty 6, 7
- search results, presenting 232-235
- statistics, computing 247-250
- suggester component, using 11-13
- text data, analyzing 64

- troubleshooting 259
- user's spelling mistake, checking 250-253
- using, with PHP 208-210
- using, with Ruby 210-212
- web pages, indexing 28, 29
- words, searching by sound 243, 244

## **Solr administration**

- about 89
- cache status, checking 93, 94
- field behavior, checking 95-98
- index structure, analyzing 116-119
- instance logging configuration,
  - changing 102-104
- Java based index replication,
  - setting up 108, 109
- Java based replication status,
  - checking 104, 105
- Java based replication status,
  - managing 113, 115
- JMX agent, connecting to 92
- MBean server, connecting to 92
- monitoring, via JMX 90, 91
- remote JXM server, running 92
- script based replication, setting up 110-112
- script based replication status,
  - checking 106, 108
- Solr query handler usage, checking 98, 100
- Solr update handler usage,
  - checking 100-102

## **Solr caches**

- configuring 24, 25
- document cache 23, 26
- field cache 23
- filter cache 23, 26
- query result cache 23, 26
- query result window 26, 27

## **solrconfig.xml file 90**

### **solr.DelimitedPayloadTokenFilterFactory**

- filter 65**

### **solr.EnglishMinimalStemFilterFactory 76**

## **Solr instance logging configuration**

- changing 102-104

## **Solr instance**

- running, on Solr 1.4.1 235

## **Solr library**

- using 212-215

## **Solr performance**

- improving 187
- improving, after commit operations 194, 195
- improving, after startup operation 193, 194

## **Solr performance, improving**

- after commit operations 194
- after startup operation 193, 194
- document cache, configuring 189, 190
- faceting performance, improving 199, 200
- filter cache, configuring 191, 192
- issues, solving while indexing 201
- query result cache, configuring 190, 191
- result pages, caching 198, 199
- results, paging 188
- sharded deployment, setting up 195-197
- time limiting collector, using 202, 203

## **solr.PointType class 80**

## **solr\_port variable 112**

## **Solr query handler usage**

- checking 98
- working 100

## **Solr, querying**

- about 121
- different query parser. selecting 125, 126
- documents, excluding with
  - QueryElevationComponent 135
- documents, positioning over others 131-134
- documents, positioning with words close to each other 136, 137
- documents, retrieving with partial match 141-143
- multiple values, querying in same field 123
- particular field value, asking for 122
- particular field value, querying using dismax query parser 122
- phrase, searching 126-128
- phrases, promoting over words 128-131
- phrases, promoting with standard query parser 131
- queries, nesting 147, 148
- results, sorting by distance from point 138-141
- results, sorting by field value 123-125
- scoring, affecting with function 143-147

## **solr.TextField class 72**

## **Solr update handler usage**

- checking 100

- working 101, 102

## **SortableIntField class 227**

## **splitOnCaseChange attribute 74**

## **splitOnNumerics parameter 87**

## **SqlEntityProcessor 57**

## **StandardRequestHandler class 217**

## **statistics**

- computing 247-249

## **stats attribute 100**

## **stream attribute 59**

## **subFieldSuffix attribute 81**

## **suggester component**

- about 11
- suggestions, from static dictionary 14
- suggestion word, rebuilding 14
- uncommon words, removing 14, 15
- using 11, 12
- working 13

## **synonyms attribute 72**

## **synonyms setup 72**

# **T**

## **termAttr property 219**

## **term() method 220**

## **text**

- preparing, for efficient trailing wildcard search 83-85
- splitting, by camel case 72, 73
- splitting, by numbers and non-white space characters 86, 87
- splitting, by whitespace only 74, 75

## **time limiting collector**

- using 202, 203

## **timeouts attribute 100**

## **TokenFilter class 219**

## **tokenizers 64**

## **TokenStream object 219**

# **U**

## **user's spelling mistake**

- checking 250-253

# **V**

## **version attribute 100**

## W

**warmupTime parameter** 94

**web pages**

fetching, Apache Nutch used 28, 29

indexing, Apache Nutch used 28, 29

**whole string**

lowercasing 77, 78

**WordDelimiterFilter** 74

**words**

searching, by sound 243, 244

## X

**XML and HTML tags**

eliminating, from text 66-68

**XML format** 38







## **Thank you for buying Apache Solr 3.1 Cookbook**

### **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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

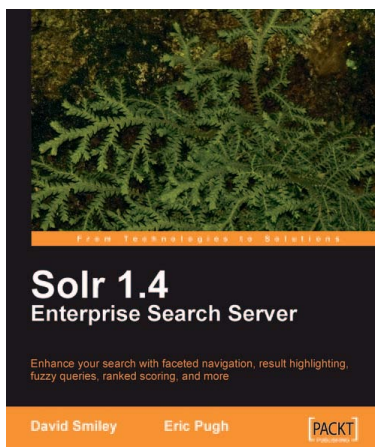
### **About Packt Open Source**

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 licences, 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](mailto: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, 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.

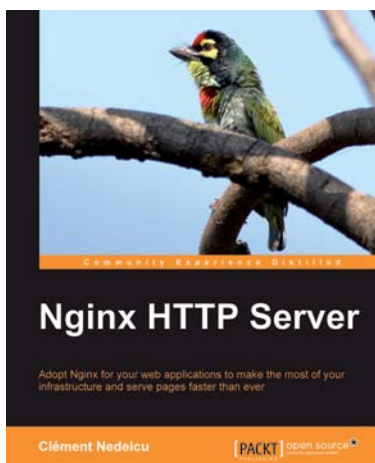


## Solr 1.4 Enterprise Search Server

ISBN: 978-1-847195-88-3      Paperback: 336 pages

Enhance your search with faceted navigation, result highlighting, fuzzy queries, ranked scoring, and more

1. Deploy, embed, and integrate Solr with a host of programming languages
2. Implement faceting in e-commerce and other sites to summarize and navigate the results of a text search
3. Enhance your search by highlighting search results, offering spell-corrections, auto-suggest, finding “similar” records, boosting records and fields for scoring, phonetic matching



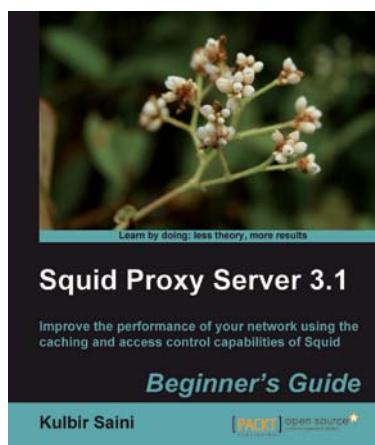
## Nginx HTTP Server

ISBN: 978-1-849510-86-8      Paperback: 348 pages

Adopt Nginx for your web applications to make the most of your infrastructure and serve pages faster than ever

1. Get started with Nginx to serve websites faster and safer
2. Learn to configure your servers and virtual hosts efficiently
3. Set up Nginx to work with PHP and other applications via FastCGI
4. Explore possible interactions between Nginx and Apache to get the best of both worlds

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

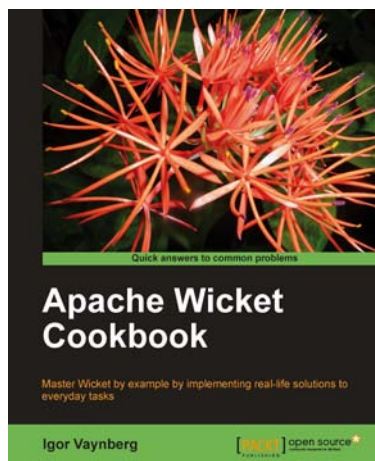


## **Squid Proxy Server 3.1: Beginner's Guide**

ISBN: 978-1-849513-90-6      Paperback: 332 pages

Improve the performance of your network using the caching and access control capabilities of Squid

1. Get the most out of your network connection by customizing Squid's access control lists and helpers
2. Set up and configure Squid to get your website working quicker and more efficiently
3. No previous knowledge of Squid or proxy servers is required



## **Apache Wicket Cookbook**

ISBN: 978-1-849511-60-5      Paperback: 312 pages

Master Wicket by example by implementing real-life solutions to every day tasks

1. The Apache Wicket Cookbook covers the full spectrum of features offered by the Wicket web framework
2. Implement advanced user interactions by following the live examples given in this Cookbook
3. Create reusable components and speed up your web application development

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles