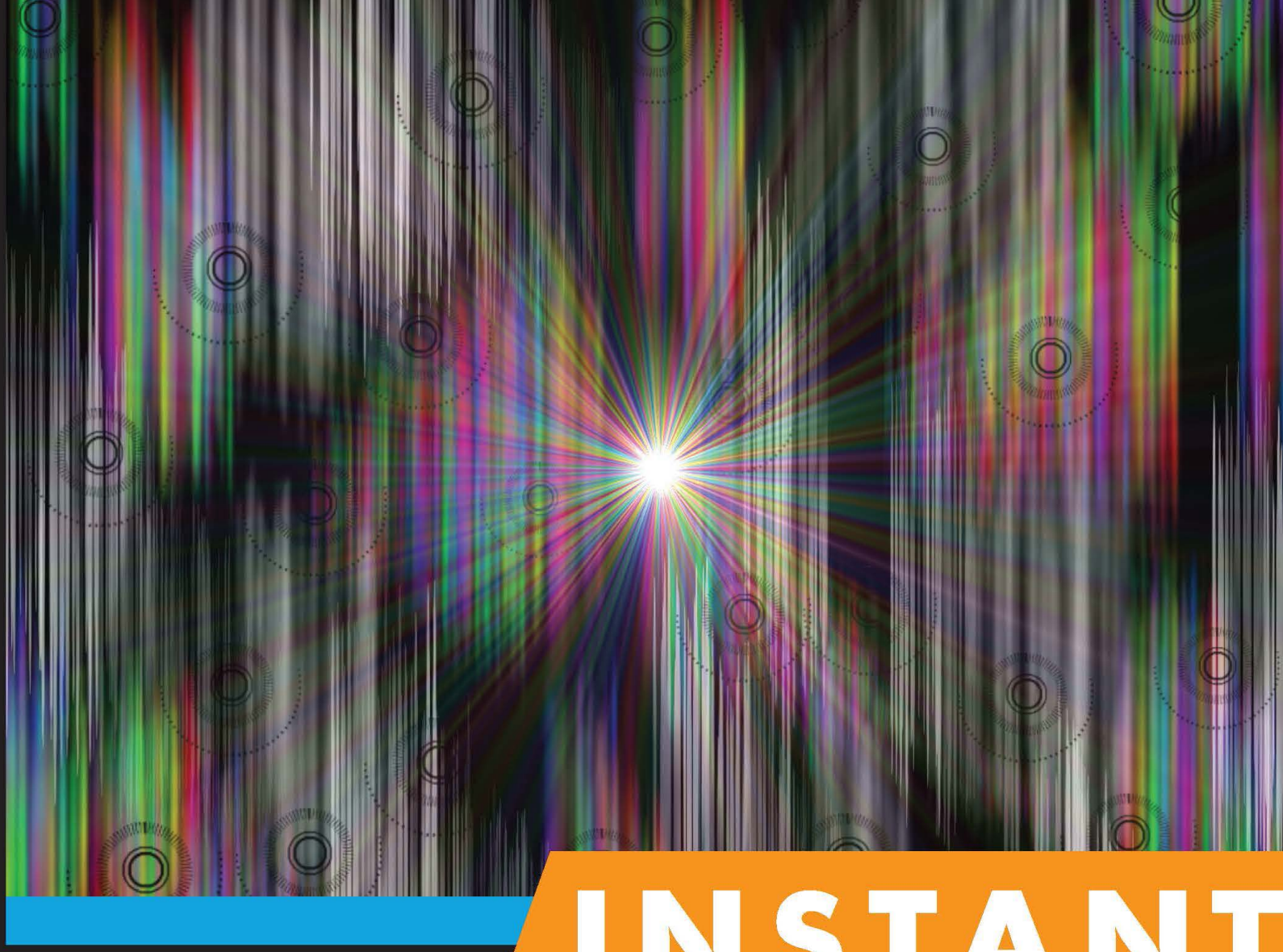


INSTANT

Short | Fast | Focused

Apache Solr for Indexing Data How-to

Learn how to index your data correctly and create better search experiences with Apache Solr

Alexandre Rafalovitch

[PACKT] PUBLISHING

Instant Apache Solr for Indexing Data How-to

Learn how to index your data correctly and create better search experiences with Apache Solr

Alexandre Rafalovitch

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Instant Apache Solr for Indexing Data How-to

First published: June 2013

Production Reference: 1300513

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

ISBN 978-1-78216-484-5

www.packtpub.com

Credits

Author
Alexandre Rafalovitch

Reviewer
Sebastian Saip

Acquisition Editor
Andrew Duckworth

Commissioning Editor
Harsha Bharwani

Technical Editor
Hardik B. Soni

Project Coordinator
Sherin Padayatty

Proofreader
Maria Gould

Production Coordinator
Nitesh Thakur

Cover Work
Nitesh Thakur

Cover Image
Sheetal Aute

About the Author

Alexandre Rafalovitch is an IT professional with more than 20 years of experience. Throughout his career, he has worked as a software developer, as a QA engineer, in a senior tech support role, and as a web master. Alexandre has worked with Java, C#, Python, and even XQuery, building software and websites (both the backend and frontend components). He is familiar with the issues of processing and presenting multilingual content in many languages, including Russian, Chinese, and Arabic.

Alexandre has developed several small open source projects of his own and has contributed to several more, including W3C Jigsaw and Apache Solr. He has published several industrial and academic publications and has presented at JavaOne twice.

Alexandre is currently working for the United Nations; however, the views expressed herein are those of the author and do not necessarily reflect the views of the United Nations.

I would like to thank my family for creating space and time to make writing this book an easier experience.

I would also like to thank the Solr Mailing List community for their active support and quick answers to many—frequently incomprehensible—questions.

About the Reviewer

Sebastian Saip worked as a test automation engineer, where he gained knowledge of the search engine Microsoft FAST. For a private project, he got himself familiar with Solr, which is open source and easier to configure. He taught himself the basics and got help from the Solr Mailing List and `stackoverflow.com`, where he also met the author of this book. Sebastian is currently finishing his Bachelor of Science in Business Informatics and plans to attend a Master's degree program afterwards.

I'd like to thank Alex for the help he's given me and the opportunity to review this book. I truly think that knowledge should be shared and open source is the way to go—I'm glad that I could be a part of it!

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `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` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

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

`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`, 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

Preface

Welcome to *Instant Apache Solr for Indexing Data How-to*. Solr is an amazing product: a search engine with both depth and breadth. Complete books have been written trying to provide an overview of many of its features. This book is not one of them. Instead, it takes one particular aspect of Solr and goes deep into it.

The topic of this book is indexing. Of all the possible deep topics in Solr, this is perhaps the most important one. Solr, amazing as it is, cannot do anything until it has the data indexed. So indexing is what this book is about.

And as indexing is one of the first things you need to learn, this book starts really easy. Then, it will take you on a learning journey. It will also teach you indexing-related skills and strategies to make you familiar with what can be done and how it can be done.

By the end of this book, you will have an in-depth understanding of the possibilities and the capacity to comprehend and integrate the additional information that is easy obtainable from Wikis, mailing lists, and blogs.

What this book covers

Creating your first collection (Simple) will help you create a basic Solr collection and populate it with a simple dataset in CSV format.

Running several collections at once (Simple) will extend your configuration to run several collections within one Solr instance—a very useful setup for experimenting.

Importing multivalued fields (Simple) will show you how to add multivalued fields to Solr and how to represent them correctly in CSV format.

Using Solr's XML format (Simple) will teach you how to provide data in Solr's own XML format.

Indexing text (Intermediate) will help you learn about the real strength of Solr, its text indexing and custom analyzer chains. You will also see how to analyze the same content in several different ways by using the copyField directive.

Indexing text – in depth (Advanced) will dig much deeper into all three analyzer component types based on a complex use-case scenario. You will also get your first glimpse at faceting and how to properly support it through indexing.

Indexing binary content on the server (Intermediate) will switch from indexing text to indexing binary content and will explain what information Solr can and cannot extract usefully.

Pulling data from XML with DataImportHandler (Intermediate) will walk you through pulling data from an external XML source using the DataImportHandler module. Also known as DIH, this is a small world of its own within Solr, well worth exploring.

Pulling data from the database with DIH (Intermediate) will expand your knowledge of DIH to using it for the databases, both for original import and for incremental update.

Commits and near real-time optimizations (Advanced) will cover the features new to Solr 4 that allow the newly indexed data to become searchable much faster than under the less flexible commit process of earlier Solr versions.

Using the UpdateRequestProcessors plugins (Intermediate) will dig into a little known part of Solr's indexing pipeline and will explore recently added components that make it very useful in a range of scenarios.

Client indexing with Java (Intermediate) will show you how to write a Solr client application. This particular example will use a SolrJ library, in both remote and embedded mode.

Atomic updates (Intermediate) will help you learn and explore another long-requested Solr 4 feature that allows updating documents in place. You will understand both preconditions required to enable this feature and different update methods it supports.

Indexing multiple languages (Advanced) will look at a challenging scenario of indexing and searching multilingual content. You will learn not only how to index it, but also how to configure various field aliases to make language-specific searches more transparent to the client applications.

What you need for this book

Obviously, you will need Solr itself running on your system. This book relies on Solr Version 4.3.

As Solr is implemented in Java, you will also need a recent version of Java installed on your machine. Version 1.6 or later of either the runtime or development environment will work fine. You will also need a modern web browser to access the Admin WebUI.

Who this book is for

This book is for you whether you are just starting with Solr or have already built your first collection by copying and modifying examples. This book is definitely for you if you need to take your Solr instance to the next step of complexity and need to understand how different configurations and elements of pipelines will work together to achieve the desired result.

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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In the `schema.xml` file (under the `multivalued/conf` directory), add `multiValued="true"` to the `addr_to` field definition."

A block of code is set as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema version="1.5">
  <fields>
    <field name="id" type="string" indexed="true" stored="true"
required="true"/>
    <field name="addr_from" type="string" indexed="true" stored="true"
required="true"/>
    <field name="addr_to" type="string" indexed="true" stored="true"
required="true"/>
    <field name="subject" type="string" indexed="true" stored="true"
required="true"/>
  </fields>
  <uniqueKey>id</uniqueKey>
  <types>
    <fieldType name="string" class="solr.StrField" />
  </types>
</schema>
```

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

```
email2,kari@acme.example.com,maija@acme.example.com;ivan@acme.example.
com,"Updating vacancy description"
```

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

```
java -Dsolr.solr.home=SOLR-INDEXING -jar start.jar
```

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: "Visit the **Core Admin** section of the Admin WebUI and review various options."

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`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `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. The most recent example source code is also available at the following URL:

`https://github.com/arafalov/solr-indexing-book`

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/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded 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` 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` if you are having a problem with any aspect of the book, and we will do our best to address it.

Disclaimer

The views expressed herein are those of the author and do not necessarily reflect the views of the United Nations.

Instant Apache Solr for Indexing Data How-to

Welcome to *Instant Apache Solr for Indexing Data How-to*. **Solr** is an open source enterprise search platform built on top of the famous **Lucene** search library. While Solr and Lucene are implemented in Java, Solr is designed to be used with many programming languages and even with the pure command line. This book covers Solr 4.3 and takes you on the journey of learning the basics along with tricks for getting your data indexed. Most of the examples will be based around indexing e-mails and other information related to fictional job vacancies.

Creating your first collection (Simple)

You will start this journey by creating your own first collection.

However, even before that, please download the latest 4.*x* Solr distribution from `https://lucene.apache.org/solr/`. Also, go through the tutorial available at `https://lucene.apache.org/solr/tutorial.html`. This will both provide a taster for Solr's capabilities and will make sure that you understand the most basic ways of working with Solr.

Getting ready

Assuming that you have walked through the tutorial, you should be nearly ready with the setup. Still, it does not hurt to go through the checklist:

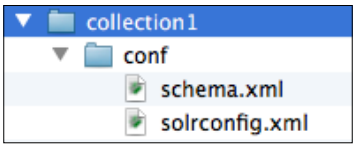
- Be familiar that you know how to start your operating system's shell (`cmd.exe` on Windows, `Terminal/iTerm` on Mac, and `sh/bash/tch/zsh` on Unix).
- Ensure that running the `java -version` command on the shell's prompt returns at least Version 1.6. You may need to upgrade if you have an older version.
- Ensure that you know where you unpacked the Solr distribution and the full path to the `example` directory within that. You needed that directory for the tutorial, but that's also where we are going to start our own Solr instance. That allows us to easily run an embedded Jetty web server and to also find all the additional JAR files that Solr needs to operate properly.

Now, create a directory where we will store our indexes and experiments. It can be anywhere on your drive. As Solr can run on any operating system where Java can run, we will use `SOLR-INDEXING` as a name whenever we refer to that directory. Make sure to use absolute path names when substituting with your real path for the directory.

How to do it...

As our first example, we will create an index that stores and allows for the searching of simplified e-mail information. For now, we will just look at the `addr_from` and `addr_to` e-mail addresses and the `subject` line. You will see that it takes only two simple configuration files to get the basic Solr index working.

1. Under the `SOLR-INDEXING` directory, create a `collection1` directory and inside that create a `conf` directory.
2. In the `conf` directory, create two files: `schema.xml` and `solrconfig.xml`.



3. The `schema.xml` file should have the following content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema version="1.5">
  <fields>
    <field name="id" type="string" indexed="true" stored="true"
required="true"/>
```

```
    <field name="addr_from" type="string" indexed="true"
stored="true" required="true"/>
    <field name="addr_to" type="string" indexed="true"
stored="true" required="true"/>
    <field name="subject" type="string" indexed="true"
stored="true" required="true"/>
  </fields>
  <uniqueKey>id</uniqueKey>
  <types>
    <fieldType name="string" class="solr.StrField" />
  </types>
</schema>
```

4. The `solrconfig.xml` file should have the following content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
  <luceneMatchVersion>LUCENE_43</luceneMatchVersion>
  <requestDispatcher handleSelect="false">
    <httpCaching never304="true" />
  </requestDispatcher>
  <requestHandler name="/select" class="solr.SearchHandler" />
  <requestHandler name="/update" class="solr.UpdateRequestHandler" />
  <requestHandler name="/admin" class="solr.admin.AdminHandlers" />
  <requestHandler name="/analysis/field" class="solr.
FieldAnalysisRequestHandler" startup="lazy" />
</config>
```

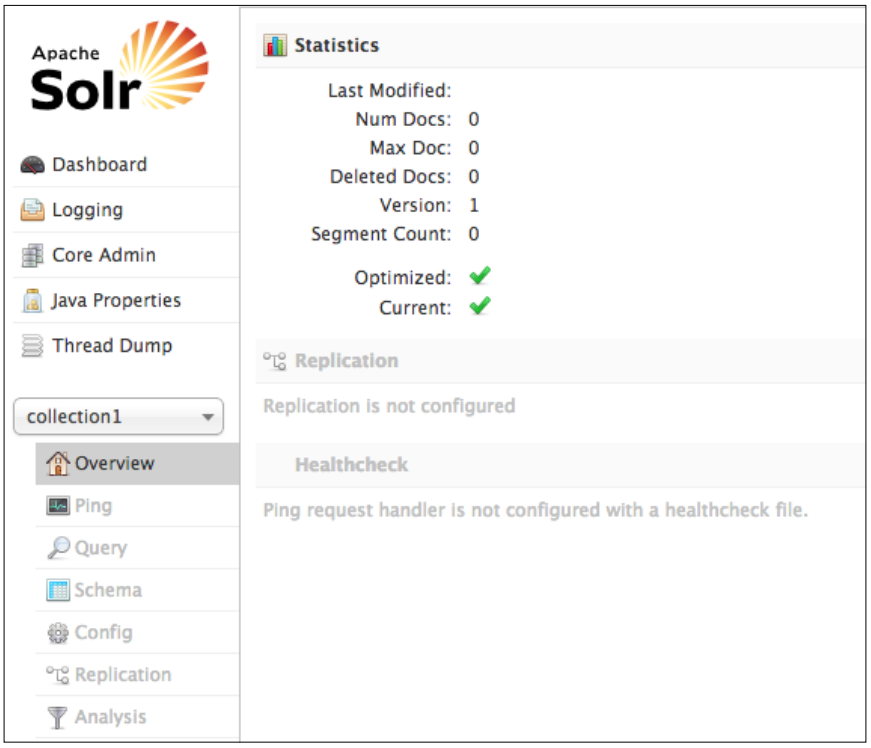
5. That is it. Now, let's start our just-created Solr instance. Open a new shell (we'll need the current one later). On that shell's command prompt, change the directory to the `example` directory of the Solr distribution and run the following command:

```
java -Dsolr.solr.home=SOLR-INDEXING -jar start.jar
```

Notice that `solr.solr.home` is not a typo; you do need the `solr` part twice. And, as always, if you have spaces in your paths (now or later), you may need to escape them in platform-specific ways, such as with backslashes on Unix/Linux or by quoting the whole value.

6. In the window of your shell, you should see a long list of messages that you can safely ignore (at least for now).

7. You can verify that everything is working fine by checking for the following three elements:
    - The long list of messages should finish with a message like `Started SocketConnector@0.0.0.0:8983`. This means that Solr is now running on port `8983` successfully.
    - You should now have a directory called `data`, right next to the directory called `conf` that we created earlier.
    - If you open the web browser and go to the `http://localhost:8983/solr/`, you should see a web-based admin interface that makes testing and troubleshooting your Solr instance much easier. We will be using this interface later, so do spend a couple of minutes clicking around now.



8. Now, let's load some actual content into our collection:
    a. Copy `post.jar` from the Solr distribution's `example/exampledocs` directory to our root `SOLR-INDEXING` directory.
    b. Create a file called `input1.csv` in the `collection1` directory, next to the `conf` and `data` directories with the following three-line content:

```
id,addr_from,addr_to,subject
email1,fulan@acme.example.com,kari@acme.example.com,"Kari,
we need more Junior Java engineers"
email2,kari@acme.example.com,maija@acme.example.
com,"Updating vacancy description"
```

    c. Run the `import` command from the command line in the `SOLR-INDEXING` directory (one long command; do not split it across lines):

```
java -Dauto -Durl=http://localhost:8983/solr/collection1/
update -jar post.jar collection1/input1.csv
```

    d. You should see the following in one of the message lines: `"1 files indexed".`

9. If you now open a web browser and go to `http://localhost:8983/solr/collection1/select?q=*%3A*&wt=ruby&indent=true`, you should see Solr output with all the three documents displayed on the screen in a somewhat readable format.

How it works...

We have created two files to get our example working. Let's review what they mean and how they fit together:

- The `schema.xml` file in the collection's `conf` directory defines the actual shape of data that you want to store and index. The fields define a structure of a record. Each field has a type, which is also defined in the same file. The field defines whether it is stored, indexed, required, multivalued, or a small number of other, more advanced properties. On the other hand, the field type defines what is actually done to the field when it is indexed and when it is searched. We will explore all of these later.
- The `solrconfig.xml` file also in the collection's `conf` directory defines and tunes the components that make up Solr's runtime environment. At the very least, it needs to define which URLs can be called to add records to a collection (here, `/update`), which to query a collection (here, `/select`), and which to do various administrative tasks (here, `/admin` and `/analysis/field`).

Once Solr started, it created a single collection with the default name of `collection1`, assigned an update handler to it at the `/solr/collection1/update` URL and search handler at the `/solr/collection1/select` URL (as per `solrconfig.xml`). At that point, Solr was ready for the data to be imported into the four required fields (as per `schema.xml`).

We then proceeded to populate the index from a CSV file (one of many update formats available) and then verified that the records are all present in an indented Ruby format (again, one of many result formats available).

There's more...

Even with a basic collection, there are some things that are worth keeping in mind.

Default collection

If you have a simple Solr configuration and use only one collection, you do not have to put the collection name in the URL. For this particular example, the following two requests are identical:

- `http://localhost:8983/solr/collection1/select?q=*%3A*&wt=ruby&indent=true`
- `http://localhost:8983/solr/select?q=*%3A*&wt=ruby&indent=true`

CSV and JSON update handlers

If you are reading this book after some experience with an earlier version of Solr, please note that before Solr 4, CSV and JSON had their own handlers. As of Solr 4, the `/update` handler handles those formats directly.

Scripting the Solr server startup

We are going to start and restart Solr all the time. You may find it useful to create a script or a batch file to run Solr with all the parameters and directories included. It will save you a lot of time later. A sample script could be placed in the `SOLR-INDEXING` directory and look something like this:

```
cd /PATH/TO/apache-solr-4.x.x/example
java -Dsolr.solr.home=SOLR-INDEXING -jar start.jar
```

This makes sure that Solr runs from the `example` directory, but points to the directory where our book's exercises are. Remember to give the script execute permissions to run (`chmod +x runSolr.sh` on Unix/Linux/Mac).

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. You can also get the example code files for this book at `https://github.com/arafalov/solr-indexing-book`.

Running several collections at once (Simple)

It is possible to run Solr with only one collection. In fact, most of the small and mid-size production installations do just that. However, we are going to set up Solr in a way that allows us to run several collections at once. This gives us an easier way to compare the content of different examples without having to start and stop different Solr instances or without having multiple server instances running on different ports.

How to do it...

1. Under the `SOLR-INDEXING` directory, create a `solr.xml` file with the following content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<solr>
  <cores adminPath="/admin/cores">
    <core name="collection1" instanceDir="collection1" />
    <core name="multivalued" instanceDir="multivalued" />
  </cores>
</solr>
```

2. Restart Solr (or switch to using the script from the previous example).
3. Visit the **Core Admin** section of the Admin WebUI and review (but do not click on them) the various options at `http://localhost:8983/solr/`. You should see an exception at the top of the screen, which can be ignored for now.

How it works...

The `solr.xml` file is a top-level file that defines what collections Solr is serving. Collection is actually a logical term, which—for our simple examples—maps to a Solr core. A single Solr server can handle many cores simultaneously, each with its own definition and index. At the most basic level, a core has a name and an instance directory. The core's `name` becomes part of the URL. The core's `instanceDir` (instance directory) is where the configuration, index data, and support files reside.

If you create a core that does not actually point anywhere, just as we did here, Solr will complain in every possible way, but will actually operate all the valid cores anyway.

There's more...

Core management is a more complicated topic that this book can cover. However, it is worth keeping a couple of issues at the back of your mind.

Manipulating cores from Admin WebUI

You may have noticed a number of interesting actions in Admin WebUI, such as **Add core**, **Unload**, and **Reload**. We will be using the **Reload** core option quite a lot in this book, but will ignore most of the others.

The main reason we ignore the other options is because to keep Solr remembering those permanently, it needs to be able to write the changes back to the `solr.xml` config file. This has to be enabled in the config file itself (the `persistent="true"` value on the `solr` element) and the formatting and comments in the file will be lost. Just as importantly, Solr needs to have security permissions to be able to write to that file and directory.

Auto-discovery of Solr cores

In the future, `solr.xml` will no longer list cores as they will be auto-discovered. An early mechanism for this has already been implemented in Solr 4.3. However, it is not battle tested yet and is actually not any easier to use for learning purposes. So, for now we will stay with the format that is known to work.

Importing multivalued fields (Simple)

One of the most common issues with importing data is that a field may have more than one value. Accommodating this requirement in a database usually requires a separate table for values with foreign keys and referential constraints. In Solr, on the other hand, having multivalued fields is easy.

In our example, an e-mail can be sent to multiple people at once. Let's see how this can be represented in the input data, as well as what changes need to be done to Solr to make it store and search multivalued fields.

And we are going to keep all our examples in their own collections. While it is possible to just keep adding features to our first collection, it makes it harder to come back and review or redo the earlier lessons.

How to do it...

1. Copy and rename the `collection1` directory to the `multivalued` directory to match the `instanceDir` value from the previous example.
2. Delete the `data` subdirectory under `multivalued`.
3. Rename the `input1.csv` file to `multivalued.csv`.
4. In the `multivalued.csv` file, change the second data line (third file line) to have two e-mail addresses separated by a semi-colon for the `addr_to` column:

   ```
   email2,kari@acme.example.com,maija@acme.example.com;ivan@acme.
   example.com,"Updating vacancy description"
   ```

5. In the `schema.xml` file (under the `multivalued/conf` directory), add `multiValued="true"` to the `addr_to` field definition. It should now look like following code line:

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

6. Restart Solr.
7. Run the indexer (notice the change of the path in the target URL from `collection1` to `multivalued`):

   ```
   java -Dauto -Durl=http://localhost:8983/solr/multivalued/update
   -jar post.jar multivalued/multivalued.csv
   ```

8. Check in the Admin WebUI that we still have only one field (`http://localhost:8983/solr/multivalued/select?q=*%3A*&wt=ruby&indent=true`).
9. Update the import definition by running a different import command:

   ```
   java -Dauto -Durl="http://localhost:8983/solr/multivalued/
   update?f.addr_to.split=true&f.addr_to.separator=;" -jar post.jar
   multivalued/multivalued.csv
   ```

10. Reload the browser window for the previous query and notice that the `addr_to` values are now split into two entries.

How it works...

Solr has a very strong support for multivalued fields. The extra parameter in `schema.xml` is all it takes for Solr to recognize that a field may have multiple values and deal with it accordingly both during import and export.

Then, we needed to tell the CSV handler which fields can have multiple values and how to break them apart. That's what `f.addr_to.split` and `f.addr_to.separator` do, where `addr_to` is the name of the field as defined in the `schema.xml` file. The `f.<fieldname>.<option>` approach is a standard Solr one for per-field settings.

Notice that we just replaced documents, because we have an ID defined. If we did not, Solr would have appended them instead. The field that is used for the ID is defined not in the field definition, but as a separate line in the `schema.xml` file: `<uniqueKey>id</uniqueKey>`.

There's more...

Here are a couple more troubleshooting tips to pay attention to.

Unique document keys

It is not necessary to have an unique document key in simple scenarios. There is a Lucene document ID under the covers, but it is not exposed through Solr. If no unique key is defined, Solr will add documents as new entries. However, some components do require a unique key and it is needed for SolrCloud. Therefore, it is recommended whenever possible.

If a natural unique key is not available, Solr can do deduplication based on the document's content using `SignatureUpdateProcessorFactory` or create a special UUID key using `UUIDUpdateProcessorFactory` (we will look at Update Processors later).

Advanced CSV processing

Solr's CSV import can do more than just standard CSV files. It can also process tab-separated values, which sometimes make it easier to import from a database, such as MySQL.

CSV is also available as an output format. This could be useful for importing results of a search into Excel for post-processing.

Indexing unexpected multivalued fields

If you try to index a document with multivalued fields and the `schema.xml` file does not define that field as multivalued, Solr server will return a `#400 Bad Request` exception and you will see an error message in the **Logging** section of Admin WebUI.

Using Solr's XML format (Simple)

While CSV is perfectly an useful import format for Solr, it starts to get unwieldy very quickly. Fortunately, Solr supports a number of import formats. One of them is Solr's own update XML import format. It is more verbose than CSV, but you will see that it is easier to read and deals better with multivalued fields, optional fields, complex text strings, and other real-life requirements.

How to do it...

1. Copy and rename the `multivalued` directory to a directory named `xml` and delete the `data` subdirectory in the new copy.
2. Add another `collection/core` reference to `solr.xml` in the `SOLR-INDEXING` directory:

   ```
   <core name="xml" instanceDir="xml" />
   ```

3. In `schema.xml` (under the `xml/conf` directory):
   a. Inside the `fields` section, add three new fields for dates, message, and priority:

      ```
      <field name="date" type="string" indexed="true"
      stored="true" required="true" />
      <field name="message" type="string" indexed="true"
      stored="true" required="true" />
      <field name="priority" type="int" indexed="true"
      stored="true" />
      ```

b. Inside the `types` section, add a new type definition required by the `priority` field:

```
<fieldType name="int" class="solr.TrieIntField"
precisionStep="0" positionIncrementGap="0"/>
```

4. Restart Solr for it to find the new core and new definitions.
5. Now, let's create an updated XML in the format that Solr expects for this information. The file should be on the same level as we had CSV import files and we will call it `updatexml.xml` (or download the book's examples at `https://github.com/arafalov/solr-indexing-book`):

```
<add>
  <doc>
    <field name="id">email1</field>
    <field name="addr_from">Fulan AlFulani &lt;fulan@acme.example.
    com&gt;</field>
    <field name="addr_to">Kari Nordmann &lt;kari@acme.example.
    com&gt;</field>
    <field name="subject">Kari, we need more Junior Java
    engineers</field>
    <field name="date">5 Jan 2012</field>
    <field name="message">Kari, some of our Java engineers left at
    the end of last year. Can you please advertise some open
    positions at Junior level. Thanks, Fulan</field>
  </doc>
  <doc>
    <field name="id">email2</field>
    <field name="addr_from"><![CDATA[Kari Nordmann <kari@acme.
    example.com>]]></field>
    <field name="addr_to"><![CDATA[Maija Meikäläinen <maija@acme.
    example.com>]]></field>
    <field name="addr_to"><![CDATA[Ivan Ivanovich Ivanov <ivan@
    acme.example.com>]]></field>
    <field name="subject">Updating vacancy description</field>
    <field name="date">6 Jan 2012</field>
    <field name="message">Maija, Vania I need you to work together
    to update the description for Junior Java programmers. Fulan
    wants to hire some more for both New York and San Francisco
    location. See if we can wrap this up in a week. Thank you,
    Kari</field>
    <field name="priority">2</field>
  </doc>
</add>
```

6. Run the import command (you may want to open yet another shell to avoid changing directories all the time, if you are using command-line editing tools):

```
java -Dauto -Durl="http://localhost:8983/solr/xml/update" -jar
post.jar xml/updatexml.xml
```

7. Look in the browser to see what the two new records look like (`http://localhost:8983/solr/xml/select?q=*%3A*&wt=ruby&indent=true`).

How it works...

There are several new things introduced by this example. Let's start from the changes to the schema needed for the field `priority`:

- We had to define a new field type, `int`. Notice that Solr does not have any hard-coded type names; you have to define which ones you want and what you want to name them. For now, you can ignore the actual type definition details (we will discuss them in another example, shortly).
- Now that we defined a new type, we can refer to it from the field definition with the attribute type. Notice that we have field definitions in the file before the type definitions, though you certainly can swap the order of the sections (Solr does not care either way).
- We did not include the `required="true"` attribute in the field definition for `priority`. This allows the field to be present or absent for any specific document. In our example, one document has that field present and another one does not.

Looking at the submission format, it is very important to realize that it is not just any XML, it is a custom update XML format designed for Solr needs. It is quite flexible and you can see that it has better native support for documents with the missing (`priority`) and multivalued (`addr_to`) fields.

At the same time, it is quite strict on what it expects, which is why we are still indexing the `date` field as a string; Solr expects the real dates to come in precise but over-verbose ISO 8601 format (for example, 2012-01-06T23:59:59Z).

Of course, as with any XML, we need to escape characters such as `<` (less-than sign), `>` (greater-than sign), and `&` (ampersand). We have shown here two ways of dealing with this issue:

- By escaping the individual characters
- By enclosing the whole content of the element in a `CDATA` section

Finally, you will notice that we are using UTF-8 for representing non-ASCII characters in Maija's surname (Meikäläinen). UTF-8 is a modern standard and should be used in the majority of cases. Solr has full support for UTF-8 content.

There's more...

This is a good place to mention a couple of small topics that we will explore in detail later.

Other operations possible in updated XML

So far we have only looked at adding and updating documents. It is also possible to specify deletion of documents. There are two ways to do so:

- Delete a document by its ID:

  ```
  <delete><id>05991</id></delete>
  ```

- Delete possibly multiple documents by a query:

  ```
  <delete><query>priority:2</query></delete>
  ```

It is also possible to have multiple `id` and `query` elements within one `delete` statement.

Notice that, when deleting by ID, the `<id>` element does not actually refer to a field named `id`, but rather refers to whichever field is defined in the `schema.xml` file in the `uniqueKey` element.

The `delete` command can be triggered from a command line:

```
java -Ddata=args -Durl=http://localhost:8983/solr/xml/update -jar post.
jar "<delete><query>*:*</query></delete>"
```

Otherwise, it can be triggered directly from a web browser:

```
http://localhost:8983/solr/xml/update?stream.body=%3Cdelete%3E%3Cquer
y%3E*:*%3C/query%3E%3C/delete%3E&commit=true
```

Using JSON format

Apart from XML format, it is also possible to use JSON format for nearly all the same operations. We will see JSON and XML compared and used side by side in the *Atomic updates* recipe later.

Indexing text (Intermediate)

While we now have data in Solr, so far it is not that different from storing it in a database. We can get data in and we can get data out. The real strength of Solr is in indexing the text, or to be more precise, indexing text is what Lucene does and Solr wraps around Lucene to provide definition management, import, export, caching, and other functions.

Let's have a look at what possibilities we gain by actually starting to use the text handling options Solr and Lucene offer us.

Let's try to index in a way that allows us to find documents for the following scenarios:

- **Scenario A**: Find a message for `Kari`
- **Scenario B**: Find all messages with the word `Kari` in any text field
- **Scenario C**: Find all messages with the phrase `Java engineers` in any text field

How to do it...

We will be running many queries in this example, so let's make it easy to see what is different between different request URLs. We can do so by moving output formatting parameters into the request handler's default.

1. Copy and rename the `xml` directory to a `text1` directory, add the core reference to `text1` in `solr.xml` (see previous recipe for more detailed steps, if required). Do not delete the `data` directory at this point.
2. Modify the request handler in `text1/conf/solrconfig.xml` for `/select` to include the default formatting parameters:

```
<requestHandler name="/select" class="solr.SearchHandler" >
  <lst name="defaults">
    <str name="wt">ruby</str>
    <str name="indent">true</str>
  </lst>
</requestHandler>
```

3. Restart Solr.

Before doing anything special, let's see how far we can get with the definition and content that was used for the previous example.

Let's start with scenario A:

- Try running a query (`addr_to:kari`), `http://localhost:8983/solr/text1/select?q=addr_to%3Akari`, that should get a response with zero results, but notice that the results are in indented Ruby format. If you run the same query against the `xml` core, you will get default XML output format instead.
- Try the same but a case-sensitive one (`addr_to:Kari`), `http://localhost:8983/solr/text1/select?q=addr_to%3AKari`; still no results.
- How about a wildcard case-insensitive search (`addr_to:kari*`), `http://localhost:8983/solr/text1/select?q=addr_to%3Akari*`? Still no results; this is certainly no Google.

- How about a wildcard case-sensitive search (`addr_to:Kari*`), `http://localhost:8983/solr/text1/select?q=addr_to%3AKari*`? This should finally get you one record.
- You can also do a double-wildcard search (`addr_to:*ari*`), `http://localhost:8983/solr/text1/select?q=addr_to%3A*ari*`; this should also give you one record.

What about the scenario B:

- Let's try searching anything for `Kari` (`*:Kari*`), `http://localhost:8983/solr/text1/select?q=*%3AKari*`; that should get an error message as it is actually not a valid Solr query. You may also get scary-looking exception traces in your Solr server logs.
- What about just searching for `Kari` (`Kari`), `http://localhost:8983/solr/text1/select?q=Kari`? That still returns an error message, but one that is a lot more informative (no field name specified in the query and no default specified via the `df` param).

Scenario C is even more complex than scenario B and is not even worth trying.

Let's now extend our current configuration to make the desired scenarios possible:

1. In the `schema.xml` file, add the following type definition in the `types` section:

```
<fieldType name="email" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

   Change the type for both the `addr_from` and `addr_to` fields to `email`:

```
<field name="addr_from" type="email" indexed="true" stored="true"
required="true" />
<field name="addr_to" type="email" multiValued="true"
indexed="true" stored="true" required="true" />
```

2. Reload the `text1` core from the Admin WebUI, under the **Core Admin** screen.
3. Run the indexing process (notice that we are using the unchanged copy of the file from the previous example):

```
java -Dauto -Durl=http://localhost:8983/solr/text1/update -jar
post.jar text1/updatexml.xml
```

4. Now test the first scenario by searching for the string `kari` in the `addr_to` field (`addr_to:kari`), `http://localhost:8983/solr/text1/select?q=addr_to%3Akari`. You should get a single result without the need for any wildcards and regardless of the capitalization.

Let's proceed with implementing scenarios B and C:

1. Modify the request handler in `solrconfig.xml` for `/select` to add another default parameter `df`:

```
<requestHandler name="/select" class="solr.SearchHandler" >
  <lst name="defaults">
    <str name="df">text</str>
  ...
```

2. In the `schema.xml` file, add a couple of field definitions and the `copyField` instructions in the `fields` section:

```
<field name="text" type="text_general" multiValued="true"
indexed="true" stored="false" />
<copyField source="message" dest="text" />
<copyField source="subject" dest="text" />
<copyField source="addr_*" dest="text" />
```

   Add the field type definition now used by the field `text` in the `types` section:

```
<fieldType name="text_general" class="solr.TextField"
positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

3. Reload the `text1` core again in the Admin WebUI. Check that no new error messages showed up in the Solr logfiles due to mistyped XML or other reasons.
4. Re-run the indexing process as we previously did in this recipe.
5. Re-run the queries for scenarios B and C against the new collection:
   - The `http://localhost:8983/solr/text1/select?q=Kari` URL should now get you two results
   - The `http://localhost:8983/solr/text1/select?q=%22Java+engineers%22` URL should now get you one result

How it works...

Before getting into indexing, let's review the default parameters. The parameters are set in the `defaults` section for the `/select` handler, which means that we can override them during a query by explicitly specifying a parameter name. It is possible to define multiple `requestHandlers` with different configurations for different use cases. Solr's main example comes with more than five different `SearchHandler` configurations to demonstrate different capabilities. Also in addition to overridable defaults, Solr allows us to append settings (`<lst name="appends">....`) and make settings that are compulsory (`<lst name="invariants">...`). This allows us to significantly simplify both examples and requests that need to be built by the client applications.

For the indexing, it is important to understand that Lucene does not actually search `text`. It breaks `text` into tokens and then stores those tokens in a specialized (inverted index) structure that allows us to very quickly look them up during the query and find all the places where those tokens come from. While Lucene does support wildcard suffixes (for example, `kari*`) and even wildcard prefixes (`*kari`), these are much less efficient. We may not have noticed this impact with our two-document system but Solr is designed for collections with multiple millions of records, where every millisecond counts.

In our earlier examples, all the fields were defined as type `string` backed by the `solr.StrField` implementation. When such a field was indexed, Solr basically generated a single token for the whole string and stored it. Therefore, when we wanted to search for `Kari`, there was no token it could match against directly, so Solr/Lucene had to examine every single string inefficiently to see whether it was present. And the search text had to exactly match the stored text—upper case and all. This is very similar to how a relational database would handle a `WHERE addr_to LIKE 'kari'` SQL clause.

To allow Solr/Lucene to store and query text efficiently, we need to tell it what processing needs to be done to the text during indexing and during query time to break the text into tokens. This is done by defining a field type and analyzers that break and modify text. We will look into the details of analyzers in the next recipe, so let's just review the basics as they are used in this example.

For scenario A, we knew that the `addr_to` and `addr_from` fields contained people's e-mail IDs, so we could assign a specialized tokenizer that broke the text into words (based on Unicode definition of words), but had special treatment of e-mail addresses and kept those together. We then lower-cased each of those tokens to make sure that both `Kari` and `kari` matched during the query time. You can review how a particular field type processes text by visiting the **Analysis** screen of Admin WebUI using the following URL:

`http://localhost:8983/solr/#/text1/analysis?analysis.fieldvalue=Kari+Nordmann+%3Ckari%40acme.example.com%3E&analysis.query=&analysis.fieldtype=email&verbose_output=0`.

Analyzer	Tokens		
UAXURLET	Kari	Nordmann	`kari@acme.example.com`
LCF	kari	nordmann	`kari@acme.example.com`

For scenarios B and C, we have done something slightly different. We needed to be able to search the content from multiple fields at once, we needed that content to be tokenized for quick search, and we needed to be able to search without naming a specific field.

We started by adding another default parameter `df` (default search field), which tells Solr which field to search against when the field is not explicitly specified.

Then, we defined this default field (which we called `text` here) and declare that its content comes by a copy from the four other text fields we expect people to want to find content in. To copy the content, we use the `copyField` instruction, which tells Solr to copy the content of one field into the other one without needing to provide the content twice during indexing. There are many ways to get data into Solr and `copyField` ensures that we apply the instructions regardless of the source. Notice that for copying e-mail addresses, we used a wildcard version of `copyField` to copy all fields with names starting with `addr_`.

As the text field is an accumulation of the other fields, we need to get a couple of details right. Firstly, it has to be a multivalued field, as it gets content from several fields. Secondly, it does not need to actually store the copied content. It just needs to have it indexed. It is important to realize that stored text is not actually used for search; it is stored separately and is only returned to the client when requested. So, if a particular field is used for search but is never displayed to the client (for example, extracted content of a PDF document), it can have its `stored` attribute safely set to `false`.

Finally, the field definition itself does two particular things. It is very similar to the `email` type definition above it, but uses a more generic `StandardTokenizerFactory`, which has a good general approach on splitting text into tokens (Unicode word-boundary rules once again) suitable for most languages. It also applies the lower-casing filter to making searching case insensitive.

There's more...

Once you get a strong feel for the basics, feel free to come back and review some additional details, as outlined in the following sections.

Defining fields in bulk

Not every field has to be defined individually. Solr supports dynamic fields, which is great for handling metadata during binary indexing. All of the following are valid definitions of dynamic fields:

```
<dynamicField name="*_facet" type="string" indexed="false"
stored="true" />
<dynamicField name="meta_*" type="text" indexed="true" stored="false"
/>
<dynamicField name="*" type="string" indexed="false" stored="false" />
```

The last definition is great if you are somehow getting a lot of fields submitted but you want to ignore everything that is not explicitly defined, not indexed, and not stored (nothing for Solr to worry about).

Versatility of copyField

We have seen one use of the `copyField` directive with explicit source and target field names, as well as with basic wildcards. It is actually a lot more versatile than that. Here are a couple more ways to use it:

Instruction	Effect
`<copyField source="fulltext" dest="excerpt" maxChars="200"/>`	Copies only the first 200 characters.
`<copyField source="*" dest="alltext" />`	Copies all of the fields into one big multivalued field.
`<copyField source="*_t" dest="alltext" />`	Copies only the field with names ending in `_t` to the `alltext` field (prefix wildcard).
`<copyField source="props_*" dest="props_text_*" />`	Copies only the field with names starting from `props_` to match fields (same suffix) starting with `props_text_`. This is a particularly useful command to use with dynamic field definitions; in fact it requires a matching definition like `<dynamicField name="props_text_*"....`

Indexing text – in depth (Advanced)

In the previous example, we had a look at how to stop treating text as a long string and start treating it as a set of tokens. In this example, we will look at more complex processing sequences that Solr can apply to a text.

But before we start digging in, it is very important to understand that indexing is not the ultimate goal. Searching and finding content is the goal. Make sure that this is what drives your decision process by analyzing the searches that real users do and adapting your schema and processes to that. A useful book to read on this subject could be *Search Analytics for Your Site*, by *Louis Rosenfeld*.

For the sake of learning though, let's consider a somewhat complex scenario. We would like to be able to see which people participate in the e-mails and how often. We want to be able to count either by name, regardless of the e-mail, or by e-mail alone. We would also like to be able to search for people's names both with and without accents, and we would like documents where people are senders (in the `addr_from` field) to rank higher than those that are receivers.

Counting values in a particular field based on the current search is called **faceting** and is one of the very popular features of modern search interfaces, as it allows users to do basic search first and then start narrowing it down based on categories and counts returned. It is faster and a lot more flexible than forcing users to select types and categories before they search. Solr has a very strong support for faceting.

How to do it...

1. Copy and rename the `text1` directory to a `text2` directory, delete the `data` directory, and then add core reference to `text2` in `solr.xml`.
2. Let's start by setting up facets for names and e-mails separately. Add another document to `updatexml.xml` within the `add` section:

```
...
  <doc>
    <field name="id">email3</field>
    <field name="addr_from"><![CDATA[Maija Meikäläinen <maija@
acme.example.com>]]></field>
    <field name="addr_to"><![CDATA[Kari Nordmann <kari@acme.
example.com>]]></field>
    <field name="addr_cc"><![CDATA[Ivan Ivanovich Ivanov <vania@
home.example.com>]]></field>
    <field name="subject">Re: Updating vacancy description</field>
    <field name="date">10 Jan 2012</field>
```

```
    <field name="message">Kari, please find the proposed
description attached. Vania is not feeling too well today. He will
check his home email, if something is urgent. Salut, Maija</field>
  </doc>
...
```

3. In the `schema.xml` file:

   a. Replace the definition of `addr_to` in the `schema.xml` field to make it dynamic (leave the `addr_from` definition as is):

   ```
   <dynamicField name="addr_*" type="email" multiValued="true"
   indexed="true" stored="true" />
   ```

   b. In the `types` section, add another two type definitions:

   ```
   <fieldType name="address_noemail" class="solr.TextField">
     <analyzer type="index">
       <charFilter class="solr.PatternReplaceCharFilterFactory"
   pattern=" &lt;.*" replacement="" />
       <tokenizer class="solr.KeywordTokenizerFactory" />
     </analyzer>
     <analyzer type="query">
       <tokenizer class="solr.KeywordTokenizerFactory" />
     </analyzer>
   </fieldType>
   <fieldType name="address_email" class="solr.TextField">
     <analyzer>
       <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
       <filter class="solr.TypeTokenFilterFactory"
   types="filter_email.txt" enablePositionIncrements="true"
   useWhitelist="true"/>
     </analyzer>
   </fieldType>
   ```

   c. Back in the `fields` section, add two new fields using those `types`, as well as the `copyField` instructions to populate them:

   ```
   <field name="facet_contacts" type="address_noemail"
   multiValued="true" indexed="true" stored="false" />
   <field name="facet_emails"   type="address_email"
   multiValued="true" indexed="true" stored="false" />
   <copyField source="addr_*" dest="facet_contacts" />
   <copyField source="addr_*" dest="facet_emails" />
   ```

4. In the same `conf` directory, create a new file `filter_email.txt` and populate it with the following line:

   ```
   <EMAIL>
   ```

5. Restart Solr.
6. Index the updated `updatexml.xml` file now with three e-mail addresses in it:

   ```
   java -Dauto -Durl="http://localhost:8983/solr/text2/update" -jar
   post.jar text2/updatexml.xml
   ```

7. Run the query to get all the e-mails, but also to get facet counts for `facet_contacts` and `facet_emails`:

   ```
   http://localhost:8983/solr/text2/
   select?q=*%3A*&facet=true&facet.field=facet_contacts&facet.
   field=facet_emails
   ```

   You should get the usual list of three e-mails, but at the bottom you should also see an additional section:

   ```
   'facet_counts'=>{
     'facet_queries'=>{},
     'facet_fields'=>{
       'facet_contacts'=>[
         'Kari Nordmann',3,
         'Ivan Ivanovich Ivanov',2,
         'Maija Meikäläinen',2,
         'Fulan AlFulani',1],
       'facet_emails'=>[
         'kari@acme.example.com',3,
         'maija@acme.example.com',2,
         'fulan@acme.example.com',1,
         'ivan@acme.example.com',1,
         'vania@home.example.com',1]},
     'facet_dates'=>{},
     'facet_ranges'=>{}}}
   ```

   Notice that the name counts do not match the e-mail counts, as `Ivan` has used two different e-mails.

8. Run another search, this time for Fulan:

   ```
   http://localhost:8983/solr/text2/
   select?q=fulan&facet=true&facet.field=facet_contacts&facet.
   field=facet_emails
   ```

   You should only get two e-mail IDs (the second e-mail ID matches against the message) and facet counts should now be lower so that it matches only the messages found.

   Now, let's improve the users' ability to search for accented fields with a special boost for names found in the `addr_to` field.

9. In `solrconfig.xml`, add more default parameters for the select handler:

```
...
  <lst name="defaults">
    <str name="defType">edismax</str>
    <str name="qf">addr_from^10 addr_to addr_cc subject message</
str>
...
```

10. In the `schema.xml` file, replace the type for subject and message fields from string to `text_general`.
11. Reload the `text2` core in the Solr Admin WebUI.
12. Re-index `updatexml.xml` as in step 6.
13. We are done with preferential searches. Let's compare searching for `Kari` and `Maija`:
    - `http://localhost:8983/solr/text2/select?q=kari`
    - `http://localhost:8983/solr/text2/select?q=maija`

    You should see that the e-mail IDs show up in a different order, bringing Kari's and Maija's e-mail IDs to the top respectively.

    The final task is to improve searching for names with accents and complex characters.

14. In the `schema.xml` file, replace the `LowerCaseFilterFactory` filter with the `ICUFoldingFilterFactory` filter for both the `email` and `text_general` types:

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

15. In the `solrconfig.xml` file, add the library references needed for `ICUFoldingFilterFactory` just at the top of the file (relative to `${user.dir}`, which points at Solr's `example` directory and not—as possibly expected—to our own `SOLR-INDEXING` directory):

```
<config>
  <lib dir="${user.dir}/../contrib/analysis-extras/lucene-libs/" />
  <lib dir="${user.dir}/../contrib/analysis-extras/lib/" />
```

16. Reload the `text2` core in the Solr Admin WebUI.
17. Re-index `updatexml.xml` again as in step 6.

18. Let's now compare searches between the old `text1` implementation of accent processing and the new one:

    - `http://localhost:8983/solr/text1/select?q=meikalainen`
    - `http://localhost:8983/solr/text2/select?q=meikalainen`

    You should only get the results for the second query and it will match documents with Meikäläinen in them.

How it works...

We started by adding another e-mail document to our indexable set. Notice that the e-mail has another field that we did not use before: `addr_cc`. As the field is identical in every way to the `addr_to` field, we have replaced the static field definition with the dynamic one. Notice also that the dynamic definition does not support the required attribute, so we have removed it. We also kept the `addr_from` definition separate, as it is not multivalued.

To support our facet requirements, we have explored quite a bit of the functionality around the analyzers. Let's look at the overall picture, before seeing the individual implementation choices that we made.

The role of the text analyzer, as we discussed earlier, is to map between the source string of text and the final list (actually graph) of tokens. This happens both during indexing and during query time.

To do this mapping, Solr allows us to define an analyzer. It is possible to define a single analyzer class that will do all the work, but more frequently, an implementation will use an analyzer chain composed from standard blocks.

The analyzer chain can consist of the following components:

- **Char filters**: These work right on the character level with the original text stream and can modify it in any way required. Char filters can be chained and run one after another.
- **Tokenizers**: There can only be one tokenizer in the chain, as it transforms a stream of characters into distinct tokens.
- **Token filters**: These work on the stream of tokens and can add, change, or remove tokens. Just as char filters, token filters can be chained as well.

It is possible to have different analyzer chains for indexing and a query, but usually they are the same or very similar. After all, if the query tokens do not match the index tokens, the search will just fail.

In implementing faceting, the most important thing to remember is that the facet values come from the *indexed* and not the *stored* values of the field. This usually translates into having fields used for faceting defined with the `string` type, as that generates a single token. However, in our case, we wanted to separate names from e-mails and facet them separately. This requires some analyzer tricks and we use different ones for names and for e-mails.

For names, we are relying on an e-mail coming at the end of the field value and being in the form of `<Name> <space> <less-than-sign> <email>...`. So, we use a char filter to find the beginning of the e-mail and remove it completely. So, what we have left is just a name, which we feed to the `solr.KeywordTokenizerFactory` filter. The keyword tokenizer is somewhat misnamed, as all it does is take the whole text content and produce a single token out of it. So, it is very similar to `solr.StrField`, but allows us to do some pre- and post-processing of the text. The end result is a single token that contains the full name in it. We are not expecting anybody to query this field, but in case they do, there is no need to strip any e-mails. Therefore, we defined a query analyzer chain separately and with fewer components.

For e-mail, we re-use `solr.UAX29URLEmailTokenizerFactory`, which we know splits the names and drops the punctuation (such as `<` and `>`) but keeps the e-mail address as a single token. What we did not show before is that the tokenizer also assigns different token types and we can use `solr.TypeTokenFilterFactory` to keep or get rid of particular types. In our case, we whitelist `<EMAIL>` as the only type we want to let through and we are left over with a single token containing the e-mail.

The best place to see what happens with the content as it progresses through the analyzer chain is in the **Analyzer** screen of the Admin WebUI. Enable the **Verbose Output** flag to get advanced details such as token types and offset positions:

```
http://localhost:8983/solr/#/text2/analysis?analysis.fieldvalue=Ka
ri+Nordmann+%3Ckari%40acme.example.com%3E&analysis.query=&analysis.
fieldname=facet_emails&verbose_output=1
```

For the fields themselves, we defined them as usual and set up the `copyField` instructions to populate them from all fields starting with `addr_` (which translates to `addr_from`, `addr_to`, and `addr_cc`). Notice that we set the stored attribute to false, as faceting does not need that information and—similarly to a catch-all text field—we are not planning to display these copied fields back to the user.

Now that all the definitions are set up—and after reindexing—Solr automatically takes care of the hard work of doing the counts. The counts for names and e-mails do not match because we have Mr. Ivanov showing up with both his work and his home e-mail IDs.

To improve our search experience and give preference to names found in the `addr_from` field, we need to switch the way we actually do the query. Up until now, we have been using the default query parser, which allows us to make queries against specific fields (fielded query, such as `addr_to:kari`). Unfielded searches use the default field, which we defined to be text earlier.

Our scenario requires more fine-tuned search strategy however, so we switched to the query parser **eDisMax** (which stands for **Extended Disjunction Max**). eDisMax is a lot more user-friendly than the default (Lucene) query parser and does a lot more advanced processing to try and figure out what a search query could actually mean. One of the many features it supports is giving a boost to a document when the query matches the content of its boosted field.

To get eDisMax working, we set it up as a default query processor in the `solrconfig.xml` file and defined which fields the non-fielded query runs against. We have defined all our original address fields as well as the `subject` and `message` fields. For the `addr_from` field, we used the boost factor of `10`, and the rest got the default boost factor of `1`.

However, now that we are explicitly naming the fields we are searching against, we are ignoring the default field setting, which is pointing at the field text. That means that we need to fix the definitions for the `subject` and `message` fields, which were still stored as non-tokenized strings. This is another advantage of eDisMax over the default query parser, as the catch-all field obviously was only of a single type, so all the copied content was parsed in an identical way. With eDisMax, every field is parsed the way its type is defined.

Using eDisMax does not mean we can no longer run queries against specific fields outside of the previous list. Try, for example, a query against the field `priority` not listed in eDisMax's `qf` parameter:

```
http://localhost:8983/solr/text2/select?q=priority:2
```

Finally, to deal with accents and complex characters, the usual strategy is to fold them into the non-accented equivalents. As long as this is done during both indexing and query time, the content will be found whether it is searched with or without the accents.

We were already doing some case folding and mapping upper-case characters to the lower-case. So, we substitute that step with the more elaborate mapping. As our names come from all over the world, we will use an advanced Unicode-aware implementation that does several steps at once (namely the **Unicode NFKC** normalization, Unicode-aware case folding, and accent mapping).

This token filter, however, is not a part of the core Solr libraries. It does come with Solr's binary distribution we downloaded, but is part of the `contrib/analysis-extras/` library. We had to add a reference to the JARs found in those directories as per the `README.txt` file there. To do so, we used the `lib` directive in the `solrconfig.xml` file. Notice that we used a path relative to the `user.dir` substitution variable, which maps to Solr's `example` directory as that is where we start our Solr server for this book. If you are running the Solr server in a different way, you may need to adjust the directive to use an absolute path instead.

Finally, let's review in how many fields a particular address from `updatexml.xml` will show up:

- It will get stored and indexed in its own field, whether `addr_from`, `addr_to`, or `addr_cc`, where it will get treated as the type `email` and searched by eDisMax
- It will get copied to a catch-all field text, where it will be treated as a general text and—now that we use eDisMax—ignored
- It will get copied to the `facet_contacts` field, where it will get its e-mail removed by a char filter and the remaining name component indexed (but not stored) as a single-token string for faceting
- It will finally also get copied to `facet_emails` where it will be tokenized, but all tokens other than e-mail will get thrown away, leaving a single e-mail token for faceting

There's more...

The previous information should be quite a lot to process. Still, there is always more depth in advanced topics. The following are a couple of more ideas for independent study.

The gory details

Even though we have gone into detail about how the text gets processed, we have just scratched the surface. If you are interested in the full comprehension of the topic, read some of the Lucene books. It is also worth looking at Lucene to appreciate just how many small annoying details Solr deals with internally, to simplify bringing a new search-based application to market.

Many types of boosting

We have used one type of boosting here: a **per-field query time** boosting. But there are many ways boosting can be done. It can be on a document or field level during indexing, it can be defined as we did in the eDisMax, or as a value function or even as in a `QueryElevation` component. Come back and research this topic once you understand what queries your system is running and what kind of help they need.

More about facets

This book is about indexing, so we are not talking in depth about facets. That could take up a small book of its own. I just want to say that if it is worth setting up Solr, it is worth setting up and learning about facets. Please refer to the Solr Wiki or other Solr books about the following topics:

- Returning non-zero facet counts
- Paging facet counts

- Faceting by range (including date range)
- Multiselect facets
- Decision-tree pivots' facets

Adding libraries to Solr

Solr supports a number of pluggable component types, therefore it supports a number of mechanisms to load libraries for those components.

Solr will automatically load any JAR found in a core's `lib` directory, if one exists on the same level as the core's `conf` and `data` directories.

Libraries common to all cores can be loaded with a `sharedLib` attribute in the `solr.xml` file and the path is related to that (`solr.solr.home`) directory:

```
<solr sharedLib="lib">
```

It is also possible—as we have done—to define libraries in `solrconfig.xml` for each core. In which case, the library can be anywhere and pointed at with the absolute path. Or, a path relative to the core's directory can be used. Finally, Solr supports variable substitution format, `${property}`, where the property can be defined by Java runtime, on a command line, in the `solr.xml` file, and so on.

Indexing binary content on the server (Intermediate)

If Solr could only index structured documents, it would be leaving vast majority of possible content untouched. Fortunately, with the help of another Apache open source project—Apache Tika—Solr can also index binary content. Whether it is a PDF document, an MS Word or OpenOffice document, an image, or even a song, it can be indexed into Solr.

Of course, it makes no sense to just load binary content into Solr. Instead, Tika parses binary formats, extracts available metadata and, in some cases, textual content, and makes it available to Solr. In case of pseudo-binary documents, such as the latest MS Word or OpenOffice formats, quite a considerable amount of information is available. For images and music, it is mostly metadata only.

The Solr part of this process is called Solr Cell or—more descriptively—`ExtractingRequestHandler`.

Let's see what we can extract.

How to do it...

As the binary files we will work with have no relation to our ongoing example, we are going to build a clean minimal collection just for that.

1. Create a new Solr collection directory called `cell`, and add the core reference to it in `solr.xml`.
2. Inside the `cell` directory, create a directory `conf` and copy into that the `solrconfig.xml` file from the first example, `collection1/conf/solrconfig.xml`.
3. Inside the `conf` directory, create a `schema.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema version="1.5">
  <fields>
    <dynamicField name="s_*" type="string" indexed="true"
stored="true" multiValued="true" />
    <dynamicField name="t_*" type="text_stemmed" indexed="true"
stored="true" />
    <copyField source="t_content" dest="t_teaser" maxChars="2000" />

    <field name="d_timestamp"  type="date" indexed="true"
stored="true"  multiValued="false" default="NOW" />
    <dynamicField name="d_*" type="date" indexed="true"
stored="true" multiValued="false" />
  </fields>
<types>
  <fieldType name="string" class="solr.StrField" />
  <fieldType name="date" class="solr.TrieDateField"
precisionStep="6" positionIncrementGap="0"/>
  <fieldType name="text_stemmed" class="solr.TextField"
positionIncrementGap="100">
    <analyzer>
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.EnglishMinimalStemFilterFactory"/>
    </analyzer>
  </fieldType>
</types>
</schema>
```

4. Edit the `solrconfig.xml` file, to add libraries and an additional request handler:

```
<lib dir="${user.dir}/../dist/" regex="solr-cell-\d.*\.jar" />
<lib dir="${user.dir}/../contrib/extraction/lib" regex=".*\.jar" />
....
<requestHandler name="/update/extract" class="solr.extraction.
ExtractingRequestHandler" >
  <lst name="defaults">
    <str name="uprefix">s_meta_</str>
    <str name="fmap.content">t_content</str>
    <str name="fmap.dcterms:created">d_created</str>
  </lst>
</requestHandler>
```

5. Restart Solr to recognize the new collection. We are going to index two files from the Solr binary distribution's `docs` directory: `tutorial.html` and `solr.png`.
6. Run the commands from Solr's `example/exampledocs` directory:

```
java -Dtype=text/html -Durl=http://localhost:8983/solr/cell/
update/extract -jar post.jar ../../docs/tutorial.html


java -Dtype=image/png -Durl=http://localhost:8983/solr/cell/
update/extract -jar post.jar ../../docs/solr.png
```

7. We are also going to index a publicly available book, *Open Government*, *O'Reilly Media*. You can download the PDF version of the book from `https://github.com/oreillymedia/open_government`.

```
java -Dtype=application/pdf -Durl="http://localhost:8983/solr/
cell/update/extract" -jar post.jar <PATH_TO>/open_government.pdf
```

8. Run a couple of queries and see what information we were able to capture:
   - Everything (quite messy)

     `http://localhost:8983/solr/cell/select?q=*%3A*&wt=ruby&indent=true`

   - Metadata and teaser (`fl=d_*`, `s_*`, and `t_teaser`)

     `http://localhost:8983/solr/cell/select?q=*%3A*&fl=d_*%2C+s_*%2C+t_teaser&wt=ruby&indent=true`

- Individual fields of interest (`d_timestamp`, `s_meta_title`, `d_created`, and `s_meta_stream_content_type`)

  ```
  http://localhost:8983/solr/cell/select?q=*%3A*&fl=d_
  timestamp%2Cs_meta_title%2Cd_created%2Cs_meta_stream_
  content_type&wt=ruby&indent=true
  ```

It is also possible—in secure environments—to request Solr to load the file directly from the filesystem or even the Internet using remote streaming. This can be done for any format, not just binary.

9. Delete all current index content (`<delete><query>*:*</query></delete>`):

   ```
   http://localhost:8983/solr/cell/update?stream.body=%3Cdelete%3E
   %3Cquery%3E*:*%3C/query%3E%3C/delete%3E&commit=true
   ```

10. Update the `requestDispatcher` definition in `solrconfig.xml` to enable remote streaming:

    ```
    <requestDispatcher handleSelect="false">
      <requestParsers enableRemoteStreaming="true"
    multipartUploadLimitInKB="2048000" />
      <httpCaching never304="true" />
    </requestDispatcher>
    ```

11. Reload the `cell` core in the Solr Admin WebUI.
12. Run the update with remote indexing of a file relative to the `${user.dir}` location (the `example` directory for us):

    ```
    http://localhost:8983/solr/cell/update/extract?stream.file=../
    docs/tutorial.html&stream.contentType=text/html&commit=true
    ```

13. Run the update with remote indexing of a URL:

    ```
    http://localhost:8983/solr/cell/update/extract?stream.
    url=http://www.example.com&stream.contentType=text/
    html&commit=true
    ```

14. Run the queries as the previous one in step 8 and confirm that the content has been indexed.

How it works...

Let's repeat once again that Solr does not actually index binary content. Instead Tika (which we need to load from external libraries) extracts all the content-specific metadata as individual metadata fields and also all available textual content into the content field.

We have defined a basic scheme to receive three types of content:

- `s_*`: Any field that starts from `s_` will be a string
- `t_*`: Any field that starts from `t_` will be a text
- `d_*`: Any field that starts from `d_` will be a date

The field `t_content` will be copied to a field `t_teaser`, but only up to the first 2,000 characters.

The field `d_timestamp` will be automatically initialized to `NOW`—a special date math expression that evaluates to the indexing time.

Now that we have our schema, we set up the rules on what happens with the metadata and content. We decide:

- All fields that do not have matching schema fields will be mapped to names `s_meta_XYZ`, where `XYZ` is the original name as decided by Tika parsers; we will see that some of those names are quite hard to read
- The special field content (where all text goes) will be mapped to the field `t_content` (and copied to `t_teaser`)
- If there is a metadata field, `dcterms:created`, it will be mapped to the field `d_created` (and treated as a date)

With all that, we can proceed to index an HTML, a PNG, and a PDF file. Notice that the path is `/update/extract` as was defined in the request handler; if you forget and use `/update`, you may get an error message that your type is not on the approved list.

Notice also that it takes a while to load a PDF. That's because we are uploading the whole PDF to the Solr server, where it then throws most of it away during the parsing. Not a particularly effective approach, if you have to index a lot of binary content.

Uploading files by posting directly to `ExtractingRequestHandler` is not the only way to get bulk content in. The other ways are:

- Sending a file (or files) as part of a web form posting to Solr (with content type of `application/x-www-form-urlencoded`)
- Sending the content as an inline value in the `stream.body` parameter—that's how we sent the "delete by query" command
- Asking Solr to load a file from (Solr's) local filesystem using the `stream.file` parameter pointing to an absolute or relative path
- Asking Solr to download a file from a URL using the `stream.url` parameter

Both `stream.file` and `stream.url` require enabling remote streaming in `solrconfig.xml` and are dangerous as anybody who can access Solr can use this syntax to ask Solr to index a local file.

Once you have the content indexed and you run the test queries, you should notice straight away that there is no real consistency between metadata field names or formats. Some names have spaces, while some have dashes. Some have titles, and some do not.

In general, indexing the binary content with Tika from inside Solr is mostly for prototypes. In real-time scenarios, usually a dedicated client is required. This allows us to run Tika extraction locally, do some caching and post-production and, only then, send relevant content and metadata to the Solr server.

You also would not just store and index random metadata in real life. Don't index what you will not search, don't store what you will not display.

There's more...

The `date` types support additional functionality. The following is a sampler of why dates are an important type to understand.

Date math

You may have noticed that the `d_timestamp` field got initialized with a magical `NOW` value. This is an example of date math and it can be used during indexing, searching, and faceting wherever dates are used. Other examples of valid date math expressions are:

- `NOW/HOUR` – This specifies the current time, round down to the nearest hour
- `NOW-1DAY` – This specifies exactly one day ago
- `NOW/MONTH+1DAY` – This specifies the second day of this month at 0:00

All the time expressions use UTC rather than local timezones.

Pulling data from XML with DataImportHandler (Intermediate)

There are two ways to get structured data into Solr: push and pull. We have already explored pushing the data; let's have a look at pulling the data.

Data Import Handler (**DIH**) is a specialized handler that is able to connect to multiple types of external data sources and then import, normalize, map, and post-process data to make it ready for Solr. It handles both full and incremental (delta) import.

DIH—and Solr itself—can scale enough to import large datasets, such as English Wikipedia (about 20 GB of structured text at the last recorded attempt). However, most of the time DIH is used for small to medium size imports. It is often a reasonable way to get an initial dataset into Solr for experiments or initial deployment. As the dataset grows and requirements become more specialized, the normalization and mapping usually requires migrating to custom-coded clients.

Our first DIH example will show how to import content from an XML file. While DIH can support many different schemas, we are going to use exactly the same update XML schema and content we used in previous examples.

How to do it...

1. Copy and rename the `text2` directory to a `dihxml` directory, delete the `data` directory, and add the core reference to `dihxml` in `solr.xml`.
2. In the `solrconfig.xml` file:

   Add the `lib` section to load the DIH libraries:

   ```
   <lib dir="${user.dir}/../dist/" regex="solr-dataimporthandler-.*\.
   jar" />
   ```

   Then, add a `requestHandler` section for the DIH section with the name of the configuration file:

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

3. Create a file named `dih-definition.xml` in the same (`conf`) directory:

   ```
   <dataConfig>
     <dataSource name="fileDS" type="FileDataSource"
   encoding="UTF-8"/>
     <document>
       <entity name="emails"
          dataSource="fileDS"
          processor="XPathEntityProcessor"
          url="${solr.solr.home}/dihxml/updatexml.xml"
          forEach="/add/doc"
          transformer="TemplateTransformer, DateFormatTransformer"
          preImportDeleteQuery="type:acme"
       >
   ```

```
      <field column="id" xpath="/add/doc/field[@name='id']"/>
      <field column="addr_from" xpath="/add/doc/field[@name='addr_
from']"/>
      <field column="addr_to" xpath="/add/doc/field[@name='addr_
to']"/>
      <field column="addr_cc" xpath="/add/doc/field[@name='addr_
cc']"/>
      <field column="subject" xpath="/add/doc/field[@
name='subject']"/>
      <field column="message" xpath="/add/doc/field[@
name='message']"/>
      <field column="date" xpath="/add/doc/field[@name='date']"
dateTimeFormat="dd MMM yyyy"/>
      <field column="priority"  xpath="/add/doc/field[@
name='priority']"/>
      <field column="type" template="acme"/>
    </entity>
  </document>
</dataConfig>
```

4. In the `schema.xml` file:
   a. Add the date type definition inside the `types` section:

      ```
      <fieldType name="fulldate" class="solr.TrieDateField"
      precisionStep="6" positionIncrementGap="0"/>
      ```

   b. Change the `date` field to be our newly created type, `fulldate`, instead of the type `string`:

      ```
      <field name="date" type="fulldate" indexed="true"
      stored="true" required="true" />
      ```

   c. Add another field to store the e-mail type:

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

5. Restart Solr to recognize the new collection.
6. Run Data Import Handler by triggering it in the Admin WebUI (the *dataimport* section of the `dihxml` collection); make sure to select the **Commit** button and to enable the **Auto-Refresh Status** checkbox. You can also trigger the import by directly calling the handler, which we defined previously in `solrconfig.xml`:

   ```
   http://localhost:8983/solr/dihxml/dataimport?command=full-
   import&commit=true
   ```

7. We should now see the same content as in the `text2` exercise, except with the `date` field showing a full date in the standard format:
   - `http://localhost:8983/solr/dihxml/select?q=*%3A*`
   - `http://localhost:8983/solr/text2/select?q=*%3A*`
8. Run a query showing that we can now facet on dates (make sure it is one long URL with no spaces):

```
http://localhost:8983/solr/dihxml/select
  ?q=*%3A*
  &facet=true&facet.range=date
  &facet.range.gap=%2B1MONTH
  &facet.range.start=NOW/MONTH-10YEAR
  &facet.range.end=NOW
  &facet.mincount=1
```

You should see the following in the `facet_counts` section of the response:

```
…facet_ranges'=>{
      'date'=>{
         'counts'=>[
           '2012-01-01T00:00:00Z',3],
         'gap'=>'+1MONTH',
…
```

How it works...

Data Import Handler is not part of Solr core libraries, so we have to add a library reference to it. Once that's done, we can define it using the standard request handler format, just like with `/select` and `/update` request handlers.

We define the configuration file as a config parameter. Notice that it is set as a default parameter. That means you can override it by just providing a different config parameter either in a URL request or in the Admin WebUI's **Custom Parameters** section of the dataimport section. The config file is also reloaded on every request, so it does not require core reloading when the file is changed.

In the configuration file, we declared that our content comes from a file (`FileDataSource`) and that we want to process it as an XML file that contains entries under the `/add/doc` XPath (using `XPathEntityProcessor`). Then, for each entry, we define how we map specific XPaths to columns. The columns here correspond to fields we already defined in the `schema.xml` file.

Mapping the processor to the source is done by the `dataSource` property on the entity definition (`entity/@dataSource`) that refers to the corresponding `name` property on the `dataSource` element definition (`datasource/@name`). If there is only one data source, both of these attributes can be omitted.

After initial mapping is done, entity transformers are called. This is something that was not available with the Update XML format, as Update Request Handler expected the content in its own strictly defined format. For `DataImportHandler`, the content comes from external sources and may need to be massaged into a shape suitable for Solr. Transformers need to be declared in the `transformer` attribute of the entity and the names can be comma-separated to create transformer chains. Transformers have full access to the content and can re-arrange it in any way desired. Therefore, the `field` elements often have custom attributes that provide additional configuration parameters to transformers.

We are using the following two transformers:

- The first one is a `TemplateTransformer`, which allows us to populate a field with static text. It also provides variable substitution with values that can come from the current entity or global variable definitions, such as `${solr.solr.home}`. We are using it here to assign a type to all the e-mails we get through this particular DIH entity definition.
- The second transformer is `DateFormatTransformer`, which allows us to convert date strings into real Solr dates by providing a date-parsing pattern. We have provided such a format for the date field in our `dateTimeFormat` custom attribute.

With the date format issue resolved, we can now finally create a `fulldate` field type and change the definition of the date field to use that.

Since the documents in the system now use real dates, we can use advanced operators on that field, such as range faceting. The example uses the previously discussed date math expressions and shows how to count the documents for each month separately (`facet.range.gap` set to encoded `+1MONTH`) for the last 10 years (`NOW/MONTH-10YEAR - NOW`) only showing the months that actually have documents with them (`facet.mincount=1`).

Once you trigger DIH, it runs asynchronously. The Admin WebUI will periodically poll Solr to see whether the operation is complete. If you triggered the import directly, you can check the progress via the URL as well (`http://localhost:8983/solr/dihxml/dataimport?command=status`). In our case, the import is near instantaneous. In real scenarios, however, DIH may be running for minutes or even for hours. This means the status command is quite useful.

There's more...

The following are a couple of tips that may save you some long troubleshooting sessions.

Ignoring XML namespaces

One of the issues with complicated XML files is that they often contain namespaces and all XPath references become extra complicated with all the namespace prefixes and declarations.

DIH will completely ignore the namespace prefixes. So, you don't need to do any namespace stripping or pre-processing. Of course, if you do need namespaces to work (for example, with the same element name in two different namespaces), you are out of luck.

Clearing the old documents

If DIH runs with the `clear` flag set to `true`, by default it runs the full delete command (`*:*`). This is useful when the collection content comes only from DIH and only from one entity type.

For more complex scenarios, it is useful to be able to clear only those documents that will be re-populated by that specific DIH import.

This can be achieved by using `preImportDeleteQuery` and `postImportDeleteQuery` in the entity definition, which will run correspondingly before and after the import:

```
<entity name="emails"
…
  preImportDeleteQuery="type:acme"
>
```

Pulling data from the database with DIH (Intermediate)

Solr is not a database, even though some of the features introduced in Version 4 make it closer to one. Therefore, many production configurations involve having the master content stored in a database, and then copied into Solr and re-organized for faster searching.

`DataImportHandler` has strong support for importing data from the databases, both as a complete set and in incremental-delta mode.

Let's see how DIH makes importing data from a database easy.

Getting ready

We need to create a database first. We are going to use DerbyDB, which comes bundled with Java (and is called JavaDB), as the database is filesystem-based and also includes a built-in client. If your system does not have Derby installed, you can always download it from `http://db.apache.org/derby/`. That may happen if you only have Java Runtime Environment (JRE) installed rather than the full Java Development Kit (JDK).

As with all examples, you can download the database from `https://github.com/arafalov/solr-indexing-book` or you can create it yourself by following the following instructions:

1. Copy and rename the `dihxml` directory to a `dihdb` directory, delete the `data` directory, and add the core reference to `dihdb` in `solr.xml`.
2. Create a file named `alerts.csv` in the `dihdb` directory; this is what we will use to populate the database from. The order of the fields in the CSV file is `ID`, `FROM`, `TO`, `SUBJECT`, `DATE`, and `MESSAGE`.

```
vacalert1,WV Alerts <alerts@jobs.example.com>,John Doe <john@
work.example.com>,(vac1) Junior Java engineer position in United
States,10 Feb 2012,New vacancy at http://jobs.example.com/
vacancies/vac1.html
vacalert2,WV Alerts <alerts@jobs.example.com>,Juan Pérez <juan@
work.example.com>,(vac1) Junior Java engineer position in United
States,10 Feb 2012,New vacancy at http://jobs.example.com/
vacancies/vac1.html
vacalert3,WV Alerts <alerts@jobs.example.com>,Wang Wu <wu@work.
example.com>,(vac2) Senior Ruby on Rails engineer position in
United States,20 Feb 2012,New vacancy at http://jobs.example.com/
vacancies/vac2.html
vacalert4,WV Alerts <alerts@jobs.example.com>,Jean Dupont <jean@
work.example.com>,(vac2) Senior Ruby on Rails engineer position in
United States,20 Feb 2012,New vacancy at http://jobs.example.com/
vacancies/vac2.html
vacalert5,WV Alerts <alerts@jobs.example.com>,Jean Dupont <jean@
work.example.com>,(vac3) Ruby on Rails engineer position in
United States,29 Feb 2012,New vacancy at http://jobs.example.com/
vacancies/vac3.html
vacalert6,WV Alerts <alerts@jobs.example.com>,Wang Wu <wu@work.
example.com>,(vac3) Ruby on Rails engineer position in United
States,29 Feb 2012,New vacancy at http://jobs.example.com/
vacancies/vac3.html
vacalert7,WV Alerts <alerts@jobs.example.com>,Alexandre
Rafalovitch <alex@work.example.com>,(vac4) Senior Solr performance
engineer position in Australia,5 Mar 2012,New vacancy at http://
jobs.example.com/vacancies/vac4.html
```

```
vacalert8,WV Alerts <alerts@jobs.example.com>,Erika Mustermann
<erika@work.example.com>,(vac4) Senior Solr performance engineer
position in Australia,5 Mar 2012,New vacancy at http://jobs.
example.com/vacancies/vac4.html
vacalert9,WV Alerts <alerts@jobs.example.com>,Max Mustermann <max@
work.example.com>,(vac5) Senior Ruby on Rails engineer position
in Australia,10 Mar 2012,New vacancy at http://jobs.example.com/
vacancies/vac5.html
```

3. Find where `derbyrun.jar` is located. It is usually somewhere under Java installation in some sort of Derby's `lib` directory.
4. From the command line in the `dihdb` directory, start the Derby client in interactive console mode:

```
java -jar <PATH_TO_DERBY>/derbyrun.jar ij
```

5. In the interactive console, run the table creation and import commands; *do not forget* the semi-colons at the end:

```
CONNECT 'jdbc:derby:dihdb-content;create=true';
CREATE TABLE ALERTS("ID" varchar(30), "FROM" varchar(60), "TO"
varchar(90), "SUBJECT" varchar(90), "DATE" varchar(20), "MESSAGE"
varchar(300));
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE(NULL, 'ALERTS', 'alerts.csv',
NULL, NULL, 'UTF-8', 1);
SELECT COUNT(*) FROM ALERTS;
SELECT * FROM ALERTS;
EXIT;
```

Derby gives somewhat confusing messages about 0 rows inserted/updated for some of the previous commands. As long as you are getting the results in `SELECT` commands, everything went well.

How to do it...

1. Create a `dihdb` directory based on `dihxml`, if you haven't done it already.
2. Update the `dih-definition.xml` file:
    a. Add a second data source, `dbDS`, at the top of the file:

```
<dataSource name="dbDS"
  type="JdbcDataSource"
  driver="org.apache.derby.jdbc.EmbeddedDriver"
  url="jdbc:derby:${solr.solr.home}/dihdb/dihdb-content"
  user="app" password="app" />
```

b. Add a second entity definition within the `document` section, on the same level as the e-mails entity:

```
<entity name="vacancy-alerts"
  dataSource="dbDS"
  query="select * from ALERTS"
  transformer="TemplateTransformer,DateFormatTransformer"
  preImportDeleteQuery="type:vacancy-alert" >
  <field name="id" column="ID" />
  <field name="addr_from" column="FROM" />
  <field name="addr_to" column="TO" />
  <field name="subject" column="SUBJECT" />
  <field name="message" column="MESSAGE" />
  <field name="date" column="DATE" dateTimeFormat="
  dd MMM yyyy"/>
  <field column="type" template="vacancy-alert"/>
</entity>
```

3. Add a `lib` directive to `solrconfig.xml` to Derby's JARs:

```
<lib path="<PATH_TO_DERBY>/derbyrun.jar" />
```

4. Restart Solr.
5. Run Data Import Handler (DIH) for the `vacancy-alert` entity only in one of two ways:
   - Run the **Execute** command from Admin WebUI, and make sure to check the **Clean** and **Commit** checkboxes and select **vacancy-alert** from the **Entity** dropdown
   - Trigger it by visiting URL directly `http://localhost:8983/solr/dihdb/dataimport?command=full-import&commit=true&clean=true&entity=vacancy-alerts&debug=true` and follow its progress with `http://localhost:8983/solr/dihdb/dataimport?command=status`
6. Check that you loaded nine entities and preview the first entry:

```
http://localhost:8983/solr/dihdb/select?q=*%3A*&rows=1
```

7. Run DIH from Admin WebUI for the second entity, still present from the previous exercise, by selecting **emails** from the **Entity** drop-down.
8. Check the number of entities we have now, their types, and preview the top items from each category:

```
http://localhost:8983/solr/dihdb/
select?q=*%3A*&group=true&group.field=type
```

How it works...

We have already done a full `DataImportHandler` (DIH) setup in the previous example. Review that, if you don't understand the basics.

In this example, we have used an additional data source that connects to the database and maps the database columns to the Solr schema. Our entity processor here is `SqlEntityProcessor`. It is the default processor and does not have to be explicitly declared.

Notice that, for historical reasons, `SqlEntityProcessor` differs from `XPathEntityProcessor` in the meaning of the `name` and `column` attributes. Let's review the mapping of the `add_from` field:

Processor	Example
`XPathEntityProcessor`	`<field column="**addr_from**" xpath="/add/doc/field[@name='addr_from']"/>`
`SqlEntityProcessor`	`<field name="**addr_from**" column="FROM" />`

For `XPathEntityProcessor`, the name of the field that corresponds to the definition in the `schema.xml` file is in the `column` attribute. For `SqlEntityProcessor`, that name is in the `name` attribute and the `column` attribute stores the database column name instead. This may cause some serious confusion when copying examples, so please take care.

The `column` attribute is compulsory in the entity definition. We can deal with it by declaring a non-existent column such as `column="FAKE"`, but that looks confusing. The other way is to use Solr's obscure trick and only have the column name with our field name (which is what we did for the type). Solr will then figure it out behind the scenes when it does not find a `name` attribute.

When running DIH import, we can choose to run one, several, or all entities. Therefore, with multiple entities, it is very important to have properly-scoped delete queries (`preImportDeleteQuery`). This will help to avoid deleting too many documents when running a single-entity import with the `clean` flag enabled. Obviously, this also applies when some of the data is populated from outside of DIH. In our example, we have first imported the `vacancy-alert` documents, then we imported the e-mails documents and—because of scoped deletes—we ended up with a set of documents containing both types, which we were able to confirm by using Result Grouping / Field Collapsing (another recent feature of Solr search).

There's more...

While not useful for learning purposes, you will want to pay attention to the next set of tips if you start using DIH in production.

Automatic field mapping

If the column names in the database match the names defined in `schema.xml`, you do not need to define a `field` element for it. Of course, you still need to do that if you are defining any transformers for that field and need to provide additional attributes.

Incremental update

Apart from full imports, `SqlEntityProcessor` also supports incremental-delta import. This allows us to add/update only changed rows in the database.

Enabling incremental updates requires a couple of preconditions in place:

- Some way to identify new/updated entries. DIH does not change the source database in any way, so the usual way is to have a last-modified timestamp field in the source definition. DIH records the last import timestamp (in the `dataimport.properties` file) and it is accessible via the `${dataimporter.last_index_time}` variable expansion.
- Some form of primary key/ID field.

Once this works, you need to define two additional queries on the entity:

- `deltaQuery`: This is the SQL query to get the IDs of records that need to be added/updated
- `deltaImportQuery`: This is the SQL query that uses set of IDs retrieved by the `deltaQuery` SQL query to do the actual import

A fictitious example would be:

```
<entity …
 deltaQuery="select id from item where last_modified > '${dih.last_
index_time}'"
 deltaImportQuery="select * from item where ID='${dih.delta.id}'"
…
>
```

Incremental delete

Synchronization of deleted entries is always a complex topic. Solr provides several ways of dealing with deleted entries.

The most basic way is to do periodic re-indexing of the whole content combined with `preImportDeleteQuery`.

Another way is to have records in the database marked in some way and kept for a while. Then, it is possible to use yet another query attribute defined on the entity level: `deletedPkQuery`. This SQL query is run before `deltaQuery`. For example, if we have a column that we populate with a timestamp when the record is *deleted*, we can use:

```
deletedPkQuery="SELECT id FROM item WHERE deleted_at > '${dataimporter.
last_index_time}'"
```

Similarly, if records have to be deleted, it might be possible to have database triggers that populate a separate table with deleted IDs and use `deletedPkQuery` to get IDs from there.

Notice that the IDs returned by the query have to match the field defined as `uniqueKey` in `schema.xml`. This may cause problems if that key is auto-generated or is somehow transformed from what is returned by the database.

Finally, there are two special column names that will cause already-committed documents to be deleted:

- `$deleteDocById`, which has to be populated with the document's `uniqueKey`.
- `$deleteDocByQuery`, which deletes using the Solr query, similar to `preImportDeleteQuery`. Both of these, though advanced, use cases which are often easier to achieve with `deletedPkQuery`. There is also a special column `$skipDoc`, which—if set to `true`—will skip the document during import and therefore avoid indexing it in the first place.

Enterprise data sources

Data Import Handler supports databases, XML files, the Web, and other data sources. However, for more enterprise-level data sources, DIH falls short. Fortunately, there is a sister, **Apache project ManifoldCF**, that integrates with Solr and is specifically designed to connect to data sources such as EMC Documentum, Microsoft SharePoint, and IBM FileNet.

Commits and near real-time optimizations (Advanced)

You may have noticed that through the previous examples we have set the `commit` flag or mentioned implicit commits. That allowed us to concentrate on the effects of our changes and ignore Solr optimizations that are important for production environments.

However, the optimizations around commits are important and the core issues need to be understood.

In this example we will look at the lifecycle that a document change (create, update, delete, and so on) goes through and how different commit settings affect the speed with which these changes become visible to end users.

How to do it...

1. Copy and rename the `collection1` directory to a `commit` directory, delete the `data` directory, and add the core reference to `commit` in `solr.xml`.
2. In `schema.xml`:
    a. Add a definition for the type `long`:

```
<fieldType name="long" class="solr.TrieLongField"
precisionStep="0" positionIncrementGap="0"/>
```

    b. Add a new special field `_version_`:

```
<field name="_version_" type="long" indexed="true"
stored="true"/>
```

    c. Add another field and the `copyField` definition to have a stored copy of the subject field:

```
<field name="subject_copy" type="string" indexed="true"
stored="true" required="true"/>
<copyField source="subject" dest="subject_copy"/>
```

3. In `solrconfig.xml`:
    a. Add a new request handler for `/get`:

```
<requestHandler name="/get" class="solr.RealTimeGetHandler">
  <lst name="defaults">
    <str name="omitHeader">true</str>
    <str name="wt">json</str>
    <str name="indent">true</str>
  </lst>
</requestHandler>
```

    b. Add the `updateHandler` sub-section within the top-level `config` section:

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

  <autoSoftCommit>
    <maxTime>60000</maxTime>
  </autoSoftCommit>

  <autoCommit>
    <maxTime>300000</maxTime>
     <openSearcher>false</openSearcher>
  </autoCommit>
</updateHandler>
```

4. Restart Solr to pick up a new collection.
5. We are going to repeatedly compare the next two commands. And as this example is time sensitive, we may need to delete and reload records. To get ready, set up three browser windows/tabs with the following queries:

Query	URL
get	http://localhost:8983/solr/commit/get?id=email1
select	http://localhost:8983/solr/commit/select?q=id%3Aemail1&wt=json&indent=true&omitHeader=true
delete	http://localhost:8983/solr/commit/update?stream.body=%3Cdelete%3E%3Cquery%3E*:*%3C/query%3E%3C/delete%3E&commit=true

6. Import the records from the command line with an additional flag, `commit=no`:

```
java -Dcommit=no -Dauto -Durl=http://localhost:8983/solr/commit/
update -jar post.jar commit/input1.csv
```

7. Immediately, run the `get` and `select` queries in separate windows. You should get one record returned by the `get` query, but no records returned by the `select` query:

```
{
  "doc":
  {
    "id":"email1",
    "addr_from":"fulan@acme.example.com",
    "addr_to":"kari@acme.example.com",
    "subject":"Kari, we need more Junior Java engineers",
    "_version_":1426523988528463872}}
```

8. Wait about a minute without doing anything. You can watch the Solr server log window for a message that mentions "Registered new searcher".
9. After a minute and the message showing up, re-run the `get` and `select` queries. There should still be one result for the `get` query, but the `select` query should now return a result as well:

```
{
  "response":{"numFound":1,"start":0,"docs":[
    {
      "id":"email1",
      "addr_from":"fulan@acme.example.com",
      "addr_to":"kari@acme.example.com",
      "subject":"Kari, we need more Junior Java engineers",
      "subject_copy":"Kari, we need more Junior Java
       engineers",
      "_version_":1426523988528463872}]
  }}
```

10. You should notice a couple of significant points to be discussed later:
    - The output format for the `get` and `select` queries are somewhat different
    - The value for `_version_` is the same for both results (but different from the one in the previous sample)
    - The result for the `select` queries includes a value for the `subject_copy` field, but the result for the `get` query does not
11. If you got delayed or want to re-run the example, run the delete command, which will delete all documents and lets you start again.
12. Once you have the documents shown for both queries, run a different update command and check the `get` and `select` queries again for the value change of the `_version_` value (you have just under 10 seconds to notice a temporary difference):

```
java -Dcommit=no -Dauto -Durl="http://localhost:8983/solr/commit/
update?commitWithin=10000" -jar post.jar commit/input1.csv
```

How it works...

An important thing to understand about Solr's commit is that it is different from a standard database transaction commit. For a database, a series of update instructions are issued in a transaction and—at the end—that whole transaction has to be committed or rolled back. Once the transaction is committed, its changes are written out, becoming the new true state of a database and these changes become visible to all new queries.

In Solr, the commit is handled on the server, which means we can add or update a number of documents in separate transactions and in parallel threads without causing a commit. The uncommitted queue of transactions will accumulate in memory and those changes will not be visible to the searchers. This does not affect the system that does changes once in a while. On the other hand, for a constantly updated system, discrepancy between what is in the system and what search engines show could be a cause of confusion or annoyance.

Unfortunately, a full-blown hard commit is expensive, as it requires waiting for the changes to be saved to disk in a guaranteed fashion (`flushed`, `fsynced`, and so on), then all the searchers need to be reopened (to see the changes), all the caches get cleared, and so on.

In Solr 4, a new concept of near real-time search was introduced by creating two types of commits. A normal (hard) commit is mostly the same as previously described. It writes all the changes out and waits for the filesystem to catch up before returning a success.

On the other hand, a soft commit sends the information to be saved but does not wait. Instead, it immediately re-opens the searchers. The result is much faster visibility of new changes, but at the cost of possibly losing the latest state of the system if the Solr server crashes.

The second new feature is that it is now possible to peek even at the records before they are committed at all. It is only possible to retrieve records by their ID, not queries. Getting these live records requires a new handler, `RealTimeGetHandler`, which we set up in `solrconfig.xml`. That, in turn, depends on two more features being present: `updateLog` and a special internal field, `_version_`.

In the same section where we set up `updateLog`, we can specify the automatic commit behavior. Without those sections, Solr will wait for commit commands from the client (such as a `commit=true` flag).

In our example, we have set up a soft commit to allow for at most 60 seconds (60,000 milliseconds) of uncommitted changes. So, when a change is requested, Solr will wait up to 1 minute and then will apply that change and all other changes that also came within that 60 seconds. Even though the changes are not yet fully on disk, they become visible to the new queries (a searcher is reopened).

We also set up `autoCommit` for hard commits to be 5 minutes (5 x 60 x 1000 milliseconds). At that point, all the committed changes are fully stable on disk. And, as we have `autoSoftCommits` every minute, there is no point in reopening the searchers and redoing all the caches, so we set `openSearcher` to `false`.

The changes are tracked by the `_version_` field, which will get updated even if the document fields are identical to what was in the system before.

This means that once we submit a document, for the first minute it is only visible via the `get` query with the known ID. After that minute, it gets soft committed and is visible to the normal `select` queries as well. Eventually, after five minutes, Solr does the additional (hard commit) work, but it is invisible to us and happens behind the scenes.

It is possible to override commit settings during the update request by either specifying `commit=true` (which is what we used to do) or even by specifying an alternative waiting period with the `commitWithin` parameter. With `commitWithin`, Solr will trigger commit operation at the latest moment possible that still honors all the requested timeout periods..

It is reasonable to have a soft commit happen as often as once a second (`maxTime=1000`), which should give Solr a very up-to-date feel.

Finally, there are some limitations to what is returned with the `get` query. Specifically, the targets of the `copyField` operations are not returned, even if they are marked `stored`. This is something to be aware of but it's not a big issue, as the best practice is for copied fields to be index only in the first place.

There's more...

The following are some advanced tips, which are useful to fine-tune the desired behavior.

Multiple entries with the get query

It is possible to get several entries with one `get` query by providing the `id` parameter multiple times:

```
http://localhost:8983/solr/commit/get?id=email1&id=email2
```

When that happens, the handler uses the more verbose result format, which matches the standard query's result format better:

```
{
  "response":{"numFound":2,"start":0,"docs":[
      {
        "id":"email1",
        "addr_from":"fulan@acme.example.com",
        "addr_to":"kari@acme.example.com",
        "subject":"Kari, we need more Junior Java engineers",
        "_version_":1426530974339956736},
      {
        "id":"email2",
        "addr_from":"kari@acme.example.com",
        "addr_to":"maija@acme.example.com",
        "subject":"Updating vacancy description",
        "_version_":1426530974343102464}]
  }}
```

Commit by document count

It is possible to specify that a commit (either hard or soft) should happen based on the number of changed documents in the queue, rather than based on amount of time. This is useful when the update pattern is unpredictable and a very large number of documents can be added without warning. It is possible to specify both document-count and time-based commits:

```
<autoSoftCommit>
  <maxDocs>10000</maxDocs>
  <maxTime>15000</maxTime>
</autoSoftCommit>
```

Per-document commitWithin

All the update request handlers support the per-request `commitWithin` parameter. Some of the handlers (such as, XML and JSON) also allow to specify settings on a per-document setting.

Using the UpdateRequestProcessor plugins (Intermediate)

So far, we looked at many ways of getting documents into the system. They can come in multiple formats such as CSV, XML, and JSON. They can be pushed to Solr in message bodies or query strings and pulled by Solr from an external XML file or a database. They can come pre-structured or extracted and mapped from binary files.

But whichever way the documents arrive in Solr, they have something in common. They have to go through one or another Update Request Handler.

And at that point, we have one more chance to modify and normalize the content of the document before it gets sent to the collection using the UpdateRequestProcessor plugins and update chains.

The `UpdateRequestProcessor` plugins are somewhat special by being in this position of control. They also have access to the full document and can create, delete, combine, or update any and all the fields. They can even access external systems to do complex adjustments. For example, an **Apache UIMA** plugin integrates external **Natural Language Processing** services with Solr for the extraction of names, places, and dates during document indexing.

The processors can be chained to either execute multiple updates or to do complex operations by combining several simple ones.

For this example, we are going to do some pre-counting of the e-mail ID's distribution (number of recipients). We are also going to improve the consistency by setting up a default priority value where it is not supplied for the `priority` field.

How to do it...

1. Copy and rename the `text2` directory to a `processors` directory, delete the `data` directory, and add the core reference to `processors` in `solr.xml`.
2. Update `schema.xml` to add a new integer field `recipients_count`; notice that we already have the `int` field type definition:

   ```
   <field name="recipients_count" type="int" indexed="true"
   stored="true" />
   ```

3. In `solrconfig.xml`:
   a. Add the Update Request Processor chain definition at the bottom of the file (but still within the `config` section):

      ```
      <updateRequestProcessorChain name="processors" >
        <processor class="solr.CloneFieldUpdateProcessorFactory">
          <lst name="source">
      ```

```
      <str name="fieldRegex">addr_.*</str>
      <lst name="exclude"> <str name="fieldName">addr_from</
str> </lst>
    </lst>
    <str name="dest">recipients_count</str>
  </processor>
  <processor class="solr.
CountFieldValuesUpdateProcessorFactory">
    <str name="fieldName">recipients_count</str>
  </processor>
  <processor class="solr.
DefaultValueUpdateProcessorFactory">
    <str name="fieldName">recipients_count</str>
    <int name="value">0</int>
  </processor>
  <processor class="solr.
DefaultValueUpdateProcessorFactory">
    <str name="fieldName">priority</str>
    <int name="value">1</int>
  </processor>
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

b. Edit the Update Request Handler for `/update` to set the default update chain:

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

4. Restart Solr to recognize the new collection.
5. Run the indexer:

```
java -Dauto -Durl=http://localhost:8983/solr/processors/update
-jar post.jar processors/updatexml.xml
```

6. Run the query and observe that we now have values for both the `recipients_count` and `priority` fields:

```
http://localhost:8983/solr/processors/
select?q=*%3A*&sort=recipients_count+desc&wt=ruby&indent=true
```

How it works...

We start by creating another field in our schema. It is a second field of type `int`, so no extra work is required.

Next, we define the update chain with the name `processors`. The most important thing to remember about the update chain is that the last item in the chain should be `solr.RunUpdateProcessorFactory`. This is where the actual index-updating code is. If you miss that class, your index will silently fail to update, as Solr will assume that one of the other processors is doing the update in some other way.

The processors have to be in a sequence, so we are actually doing two different things in this chain:

- `solr.CloneFieldUpdateProcessorFactory` and `solr.CountFieldValuesUpdateProcessorFactory` together create a count of recipients and `solr.DefaultValueUpdateProcessorFactory` ensures that if we get no recipients at all, the field is set to `0`. As our `schema.xml` file has a dynamic definition for `addr_*`, it makes sense to use dynamic field names matching when we clone fields as well. At the same time, as we don't want the sender inflating the count, we exclude `addr_from` from copying.
- Another `solr.DefaultValueUpdateProcessorFactory` for the `priority` field ensures that it is set to `1` if it is not set to anything else. This could be important if the field was used in some multiplication function query.

The following are couple more subtle points to keep in mind when creating update chains:

- The parameters that can be provided to the individual processors depend on each provider. Check the Javadoc pages for each class and for the class' parent to see the available syntax. Many of the processors, including the previous processor `CloneFieldUpdateProcessorFactory`, inherit from `FieldMutatingUpdateProcessorFactory`, which provides a rich set of inclusion and exclusion parameters. Notice that `fieldRegex` in the example uses full-strength Java regular expressions, With commitWithin, Solr will trigger commit operation at the latest moment possible that still honors all the requested timeout periods.
- Restrictions and type definitions from `schema.xml` are applied in the `RunUpdateProcessorFactory`, which gives some freedom to the intermediate processors. That's why we were able to clone a number of text fields into one destination, effectively creating a multivalued field. By the time Solr tries to map it to a singlevalued int field, the intermediate results were replaced with a single number by the `CountFieldValuesUpdateProcessorFactory` processor.

Once we have the update chain defined, we need to make sure it is used. It can be provided as any other parameter during the request or, as we have done here, we can provide it as a default value that can be overridden. If you don't want it overridden, use invariants instead of a defaults section.

There's more...

The following are a couple of advanced tips, which may come useful one day in a complex import scenario.

Data Import Handler and Update Request Processors

You may have noticed the similarity between transformers in DIH and Update Request Processors. Both of them can change document content and have access to all fields within a document.

Nevertheless, it is possible to define an update chain on a `DataImportHandler` request handler as well and the processors will run after DIH is all finished and all the DIH transformers have been executed.

This double pipeline allows more flexibility, as the functions currently provided by the DIH transformers and by processors are somewhat different. At the same time, you have to be careful combining them, especially if the DIH transformers convert some of the fields into non-textual representation.

Script Update Request Processor

A new functionality in Solr 4 is `solr.StatelessScriptUpdateProcessorFactory`, which allows us to execute a script in JavaScript (using Rhino), JRuby, Groovy, Jython, or any other JVM-based implementation that can be configured to plug into the Java scripting engine framework.

This is great for quick custom processing or prototypes. Of course, for a very large number of documents or more complex requirements, a compiled processor is still a much better option.

Client indexing with Java (Intermediate)

In a production system, it is a bad idea to do complex indexing on the server, as it is too busy actually serving queries. In addition, while Solr has a very large number of extensibility points, sometimes it is easier to do the heavy pre-processing outside of Solr, in a long running batch or even completely offline. Then, the final results can be sent to Solr in the format it expects.

And in some cases, it may make sense to create a standalone collection, populate it in the most efficient way, and only then have it added to the production server by copying over the `data` directory or swapping out a collection.

Solr supports both of these scenarios with the Java client library SolrJ. It can be run as a pure client access library while talking to the remote Solr server. This allows SolrJ to use additional libraries, have access to protected systems, or do complex caching of content in preparation for indexing it to Solr.

However, SolrJ can also embed a Solr instance right inside the runtime, which allows for more efficient access. It is also a reasonable base for custom applications that use Solr in their core without exposing Solr directly to users.

We are going to use SolrJ in both of these ways in this example.

How to do it...

1. Copy and rename the `multivalued` directory to a `solrj` directory, add the core reference in `solr.xml`, and restart Solr. Do *not* delete the `data` directory, as we actually do want the collection content.
2. Check that the content is present in Admin WebUI or directly use the URL `http://localhost:8983/solr/solrj/select?q=*%3A*&wt=ruby&indent=true` (you should have two records).
3. Create a file called `ClientSolrj.java` in the `solrj` directory.

```
import org.apache.solr.common.*;
import org.apache.solr.core.CoreContainer;
import org.apache.solr.client.solrj.*;
import org.apache.solr.client.solrj.beans.Field;
import org.apache.solr.client.solrj.embedded.EmbeddedSolrServer;
import org.apache.solr.client.solrj.impl.HttpSolrServer;
import org.apache.solr.client.solrj.response.QueryResponse;
import java.io.FileNotFoundException;

import java.io.File;

public class ClientSolrj {

    public static void main(String[] args) {
        if (args.length == 0){
            runRemoteExample(); //connect to localhost
        } else{
            runEmbeddedExample(args[0]);
        }
    }

    private static void runRemoteExample() {
        SolrServer solr = new HttpSolrServer("http://
localhost:8983/solr/solrj");
        listCollectionContent(solr, "Running in remote mode");

        SolrInputDocument doc = new SolrInputDocument();
        doc.addField("id", "email3");
```

```
            doc.addField("addr_from", "maija@acme.example.com");
            doc.addField("addr_to","kari@acme.example.com");
            doc.addField("addr_to","vania@home.example.com");
            doc.addField("subject","Re: Updating vacancy description");
            try {
                solr.add(doc);
                solr.commit();
            } catch (Exception e) {
            //Catch and ignore over-generic exception. Do NOT do
            this in production
                e.printStackTrace();
            }
            listCollectionContent(solr, "Added record in remote mode");
            solr.shutdown();
        }

        public static class Email{
            @Field String id, addr_from, subject;
            @Field String[] addr_to;
        }
        private static void runEmbeddedExample(String solrHomePath) {
            File coreConfig = new File(solrHomePath, "solr-solrj.xml"
    );
            CoreContainer container = null;
            try {
                container = new CoreContainer( solrHomePath, coreConfig);
            } catch (FileNotFoundException ex) {throw new
              RuntimeException(ex);}
            EmbeddedSolrServer solr = new EmbeddedSolrServer(
            container, "solrj" );
            listCollectionContent(solr, "Running in embedded mode");
            Email doc = new Email();
            doc.id = "email4";
            doc.addr_from = "kari@acme.example.com";
            doc.addr_to = new String[]{"maija@acme.example.com",
            "vania@home.example.com"};
            doc.subject = "Thanks (Was Re: Updating vacancy
            description)";
            try {
                solr.addBean(doc);
                solr.commit();
```

```
            } catch (Exception e) {
            //Catch and ignore over-generic exception. Do NOT do
            this in production
                e.printStackTrace();
            }
            listCollectionContent(solr, "Added record in embedded mode");

            solr.shutdown();
        }

        private static void listCollectionContent(SolrServer solr,
    String message) {
            System.out.println("\n------------------\n");
            System.out.println(message);
            SolrQuery parameters = new SolrQuery();
            parameters.set("q", "*:*");
            QueryResponse response = null;
            try {
                response = solr.query(parameters);
            } catch (SolrServerException e) {
                e.printStackTrace();
            }
            SolrDocumentList list = response.getResults();
            for (SolrDocument doc: list)
            {
                System.out.println(doc);
            }
            System.out.println("\n------------------\n");
        }
    }
```

4. From the command line in the `solrj` directory, set the `CLASSPATH` environment variable to add the servlet JAR, the content of Solr webapps's `lib` directory, the content of Solrj's `lib` directory and current directory. The actual command line is different for different operating systems. Here it is for Unix/Linux/Mac:

   `export CLASSPATH=$CLASSPATH:"<SOLR_INSTALL_PATH>/example/lib/servlet-api-3.0.jar:<SOLR_INSTALL_PATH>/example/solr-webapp/webapp/WEB-INF/lib/*:<SOLR_INSTALL_PATH>/dist/solrj-lib/*."`

   `echo $CLASSPATH`

5. Compile the client class, making sure you include libraries from the Solr installation:

   `javac ClientSolrj.java`

6. Run the client application, which by default goes against the live Solr server and adds a new `email3` document:

   ```
   java ClientSolrj
   ```

7. Re-check that the previous query now returns all three records:

   ```
   http://localhost:8983/solr/solrj/select?q=*%3A*&wt=ruby&indent
   =true
   ```

8. Shut down Solr, as we are now going to access the `collection` directly.
9. Copy `solr.xml` to `solr-solrj.xml` and in the new file remove all core definitions except for `solrj`.
10. Run the client application and pass it the path to our Solr home installation. Either an absolute or relative path can be used:

    ```
    java ClientSolrj ..
    ```

    You should be able to see (hidden in the sea of Solr's own messages) that we were able to list three existing documents and then add yet another one, `email4`.

11. Start Solr.
12. Re-check that the previous query now returns all the four records.

How it works...

Running SolrJ in remote-client mode is fairly simple. It is very similar to what any other client has to do; it can send a query and materialize resulting documents. Or it can create documents with associated fields, serialize them to some wire protocol, and send them over the Internet to Solr. The main advantage of SolrJ is that it allows us to do so under the full control of the code and is not restricted by the extension points the Solr framework provides.

In the embedded mode, SolrJ has to work a bit harder. It does not need to serve content over HTTP, but it does need everything else related to actually loading, searching, and indexing the collections. It has to be aware of what collections/cores are available, how they are defined, and where they are located. Therefore, it needs to be given information to where `solr.xml` is. As we are running a multi-collection setup but only care about one collection, we can create an alternative configuration file, `solr-solrj.xml`, with just one collection in it.

Notice that at the end of our queries and updates, we explicitly shut down the server (`solr.shutdown()`). When Solr runs within a servlet container such as Jetty, it has the shutdown hooks that ensure that Solr gracefully shuts down and writes out all outstanding changes. We do not have shutdown hooks, so we need to be explicit about a graceful exit.

For the example, we are using a command-line argument (path to the Solr configuration) as a decision factor on whether we run in remote-client mode (when no path given) or in the embedded mode.

Notice that SolrJ allows several ways to create a document object. There is a generic `SolrInputDocument` way to which we just add fields. Alternatively, it is possible to create a **Plain Old Java Object** (**POJO**) with field names matching those in `schema.xml` and with field types reflecting the field definitions. For example, the multivalued `addr_to` field maps to a Java string array (collections such as `List<String>` are also acceptable). SolrJ will then do the mapping of the values automatically when the document is submitted. Having strongly-typed fields allows us to improve the developer's experience and reduces late-stage validation errors due to mistyped names or confused types. It also allows us to deal with the denormalized nature of Solr schemas by creating several types that have different fields but map behind the scenes to the same schema. Then, business-level validation can be enforced on the object level.

The support libraries' requirements for the SolrJ are also different based on whether it is run in remote-client or embedded mode. For remote-client mode, it only needs a couple of libraries. For embedded mode, it needs pretty much everything that Solr has apart from Jetty libraries. This includes some Lucene libraries that are not even available outside of the Solr's packaged webapp. To make the example simpler, we are using a unified classpath and we load most of our libraries from the unpacked copy of the Solr's webapp as created by Jetty on the first run of a standalone Solr server. We also added the current directory (`.`) to the classpath to make it easier to compile and run our custom SolrJ client.

There's more...

It is well worth exploring the SolrJ Javadoc (linked from Solr's release notes page). This helps to discover additional methods such as the one described here.

Working in batches

As with other update formats and methods, SolrJ supports sending more than one document at once. This is much more efficient when SolrJ runs in remote-client mode. The `SolrServer.add()` method supports an alternative signature to pass in a collection of the `SolrInputDocument` instances. Similarly, you can call `SolrServer.addBeans()` and pass in a collection of POJO instances.

In remote-client mode (when using `HttpSolrServer`), you can also provide an iterator of documents or POJOs, and it will request the documents in a pulling fashion.

Atomic updates (Intermediate)

Solr did not used to be at all like a database. It was designed to excel at searching, leaving data storage and rapid in-place updates to regular databases. However, with new features in Version 4, such as soft commits and other Near Real Time optimizations, Solr can substitute a database in previously impossible scenarios. Partial record (Atomic) update is also one of these features. It allows sending only changed document fields to Solr as part of the add/update command and having them merged with already known values for other fields of the same document.

How to do it...

1. Copy and rename the `commit` directory to the `update` directory, delete the `data` directory, and add the core reference to `update` in `solr.xml`.
2. In `schema.xml`, add a couple of optional fields, one of type `long` and one of type `string`:

```
<field name="priority" type="long" indexed="true" stored="true" />
<field name="addr_cc" type="string" indexed="true" stored="true"
multiValued="true" />
```

3. Restart Solr to recognize the new collection and the latest configuration files.
4. Populate the collection:

```
java -Dauto -Durl=http://localhost:8983/solr/update/update -jar
post.jar update/input1.csv
```

5. Check in the Admin WebUI or browser that we have two records as expected (`http://localhost:8983/solr/update/select?q=*%3A*`); neither should have either the `priority` or `add_cc` field present. Also watch the `_version_` field values as we reload this query over and over again.
6. Set those fields using the XML Update format for the `email1` document:

```
java -Ddata=args -Durl=http://localhost:8983/solr/update/update
-jar post.jar '<add><doc><field name="id">email1</field><field
name="addr_cc" update="set">user1@update.example.com</field><field
name="priority" update="set">1</field></doc></add>'
```

   Re-running the query in step 5, you should now see that the `email1` document now has `user1@update.example.com` as the value of the `addr_cc` field and the `priority` field is set to `1`.

7. Add more values to those fields:

```
java -Ddata=args -Durl=http://localhost:8983/solr/update/update
-jar post.jar '<add><doc><field name="id">email1</field><field
name="addr_cc" update="add">user2@update.example.com</field><field
name="priority" update="inc">2</field></doc></add>'
```

   Re-running the query, you should now see that `addr_cc` has two values and that `priority` has been incremented to `3`.

8. Now delete those extra fields:

```
java -Ddata=args -Durl=http://localhost:8983/solr/update/update
-jar post.jar '<add><doc><field name="id">email1</field><field
name="addr_cc" update="set" null="true" /><field name="priority"
update="set" null="true"/></doc></add>'
```

   We should be back to the original state of the record, except for the value of the `_version_` field that changed with each operation on the `email1` document.

9. Now, let's do exactly the same using the JSON Update format instead of XML on the `email2` document:

   a. Set

```
java -Dtype=application/json -Ddata=args -Durl=http://
localhost:8983/solr/update/update -jar post.jar
'[{"id":"email2", "addr_cc":{"set":"user1@update.example.
com"},"priority":{"set":1}}]'
```

   b. Add or increment

```
java -Dtype=application/json -Ddata=args -Durl=http://
localhost:8983/solr/update/update -jar post.jar
'[{"id":"email2", "addr_cc":{"add":"user2@update.example.
com"},"priority":{"inc":2}}]'
```

   c. Delete

```
java -Dtype=application/json -Ddata=args -Durl=http://
localhost:8983/solr/update/update -jar post.jar
'[{"id":"email2", "addr_cc":{"set":null},"priority":{"set":
null}}]'
```

How it works...

There are a couple of preconditions that have to be fulfilled before atomic update:

- The Update Log needs to be enabled and the special field called `_version_` needs to be present. Review the *Commits and near real-time optimizations (Advanced)* recipe for the full explanation of this.
- All the fields (apart from the `copyField` destinations) need to be configured as `stored="true"`. This is because updating under the hood actually needs to retrieve the original document, update it, and then save it back. That requires access to the original stored content, as the indexed fields may have no resemblance to the original content after tokenization, synonyms, and stop-words exclusion.

With those preconditions in place, we can do one of the following four types of operations:

- Set or replace a particular value (the `set` modifier)
- Delete a value as a special case of set; for multivalued fields this removes all values
- Add an extra value to a multivalued field (the `add` modifier)
- Increment a numeric value by a specific amount (the `inc` modifier)

At the minimum, the document describing the update needs to have a way to identify the original (usually by configuring the `uniqueKey` setting in the `schema.xml` file) and to have at least one update operation present. If no update operations are present, Solr will assume that it is just a normal full-document update and will complain about all the missing required fields (if any).

For now, only XML and JSON support Atomic Update modifiers. The syntax is mostly similar:

Operation	XML examples	JSON examples
Normal (non-atomic) set	`<field name="addr_cc">user1@update.example.com</field>`	`"addr_cc":"user1@update.example.com"` `"priority":1`
Atomic Set/ Update	`<field name="addr_cc" update="set">user1@update.example.com</field>`	`"addr_cc":{"set":"user1@update.example.com"}` `"priority":{"set":1}`
Delete	`<field name="addr_cc" update="set" null="true" />`	`"addr_cc":{"set":null}`
Add (multivalued)	`<field name="addr_cc" update="add">user2@update.example.com</field>`	`"addr_cc":{"add":"user2@update.example.com"}`
Increment (numeric)	`<field name="priority" update="inc">2</field>`	`"priority":{"inc":2}`

There are two noticeable differences between the XML and JSON formats:

- JSON does not require quoting numeric values, but does require quoting strings.
- For delete operations, in XML we explicitly mark the element as null (`null="true"`), but in JSON we can just use the reserved keyword `null`. That's because XML does not directly support null values, only empty ones. And setting a field to the empty value does not delete it, but rather just sets an empty string as the field's value. That's possible in JSON as well, by explicitly passing the empty string `""` instead of null.

Indexing multiple languages (Advanced)

We have seen in previous examples that Solr can support issues with languages beyond English by, for example, enabling non-accented searches for words that originally have accents and complex characters.

In this example, we will look at deeper language support that Solr provides by automatically detecting the language used, by allowing the text processing of different languages and by hiding the implementation details from users and client applications.

As Solr is quite flexible with its language support, let's consider and implement one scenario:

- An e-mail may arrive in one of the two languages: English or Russian
- The language-specific content will be in the `subject` and `message` fields with both fields assumed to be of the same language
- We want to be able to search English and Russian content separately
- When we do a search, we want to indicate that the language and results of search with as little effort as possible with the rest of the user experience to be as identical as possible

How to do it...

1. Copy and rename the `text2` directory to a `languages` directory, delete the `data` directory, and add the core reference to `languages` in `solr.xml`.
2. Add a new e-mail to `updatexml.xml` with some Russian text. Remember you can download all the examples at `https://github.com/arafalov/solr-indexing-book`.

```
<doc>
  <field name="id">email4</field>
  <field name="addr_from"><![CDATA[Alexandre Rafalovitch <alex@
home.example.com>]]></field>
  <field name="addr_to"><![CDATA[Maria Rafalovitch <masha@home.
example.com>]]></field>
```

```
  <field name="subject">Интересная работа в Австралии</field>
  <field name="date">6 Mar 2012</field>
  <field name="message"> <![CDATA[Маша,
Вчера по электроной почте мне прислали очень интересную вакансию. И
прямо по специальности. Но в Австралии! Хочешь пожить в тёплой стране
несколько лет? Давай обговорим всё это вечером.
Целую, Саша.
]]> </field>
</doc>
```

3. In `schema.xml`, add support for language-based fields:
   a. Add two new types that apply English- and Russian-specific processing analyzers to its text (based on the schema from Solr's `example` directory):

```
<fieldType name="text_en" class="solr.TextField"
positionIncrementGap="100" autoGeneratePhraseQueries="true">
  <analyzer>
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.StopFilterFactory"
ignoreCase="true" words="stopwords_en.txt"
enablePositionIncrements="true" />
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.EnglishPossessiveFilterFactory"/>
      <filter class="solr.PorterStemFilterFactory"/>
  </analyzer>
</fieldType>

<fieldType name="text_ru" class="solr.TextField"
positionIncrementGap="100">
    <analyzer>
     <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.StopFilterFactory"
ignoreCase="true" words="stopwords_ru.txt" format="snowball"
enablePositionIncrements="true"/>
      <filter class="solr.RussianLightStemFilterFactory"/>
    </analyzer>
</fieldType>
```

   b. Remove the `required="true"` flag from the subject and message fields.
   c. Add a field for storing the identified language of the text as well as dynamic fields for English and Russian text types:

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

```
<dynamicField name="*_en" type="text_en" stored="true"
indexed="true" />
<dynamicField name="*_ru" type="text_ru" stored="true"
indexed="true" />
```

4. Copy the `stopwords_en.txt` and `stopwords_ru.txt` file to the `languages/conf` directory from the `solr/collection1/conf/lang` directory of Solr's `example` directory.
5. In `solrconfig.xml`, add libraries and an update chain to identify languages and route text to the appropriate fields (the sections can go anywhere in the file as long as they are directly within the `config` element):
    a. Add the library statements to load optional language components:

```
<lib dir="${user.dir}/../dist/" regex="solr-langid-.*\.jar"
/>
<lib dir="${user.dir}/../contrib/langid/lib/" />
```

    b. Create an update chain to catch and process the incoming documents according to the desired scenario:

```
<updateRequestProcessorChain name="languages">
  <processor class="solr.
LangDetectLanguageIdentifierUpdateProcessorFactory">
    <lst name="invariants">
      <str name="langid.fl">subject,message</str>
      <str name="langid.whitelist">en,ru</str>
      <str name="langid.fallback">en</str>
      <str name="langid.langField">language</str>
      <bool name="langid.map">true</bool>
      <bool name="langid.map.keepOrig">false</bool>
    </lst>
  </processor>
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

    c. Modify the existing `/update` request handler to use the just-created update chain:

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

6. Restart Solr to recognize the new collection and the latest configuration files.

7. Populate the collection:

```
java -Dauto -Durl="http://localhost:8983/solr/languages/update"
-jar post.jar languages/updatexml.xml
```

8. Check in Admin WebUI (or load `http://localhost:8983/solr/languages/select?q=*%3A*`) and verify that all four documents are returned. You should see that for `email1`, `email2`, and `email3`, the language field is set to `en` and the subject and message are in the `subject_en` and `message_en` fields correspondingly. For `email4`, you should see the language field set to `ru` and the fields with content should be `subject_ru` and `message_ru`:

```
{
        'id'=>'email3',
        'addr_from'=>'Maija Meikäläinen <maija@acme.example.com>',
        'addr_to'=>['Kari Nordmann <kari@acme.example.com>'],
        'addr_cc'=>['Ivan Ivanovich Ivanov <vania@home.example.
com>'],
        'date'=>'10 Jan 2012',
        'language'=>'en',
        'message_en'=>'Kari, please find the proposed description
attached. Vania is not feeling too well today. He will check his
home email, if something is urgent. Salut, Maija',
        'subject_en'=>'Re: Updating vacancy description'},
      {
        'id'=>'email4',
        'addr_from'=>'Alexandre Rafalovitch <alex@home.example.
com>',
        'addr_to'=>['Maria Rafalovitch <masha@home.example.com>'],
        'date'=>'6 Mar 2012',
        'language'=>'ru',
        'message_ru'=>'Маша,
Вчера по электроной почте мне прислали очень интересную вакансию. И
прямо по специальности. Но в Австралии! Хочешь пожить в тёплой стране
несколько лет? Давай обговорим всё это вечером.
Целую, Саша.
 ',
        'subject_ru'=>'Интересная работа в Австралии'}]
  }
```

9. Edit `solrconfig.xml` again and create language-specific request handlers:

    a. Copy the `/select` request handler and create an English request handler by renaming the copy to `/selectEN`.

b. In the handler definition, add an `invariants` section after the `defaults` section and set filters and aliases to map the `subject` and `message` fields:

```
<requestHandler name="/selectEN" class="solr.SearchHandler" >
  <lst name="defaults">
    <str name="defType">edismax</str>
    <str name="qf">addr_from^10 addr_to addr_cc subject
message</str>
    <str name="df">text</str>
    <str name="wt">ruby</str>
    <str name="indent">true</str>
  </lst>
  <lst name="invariants">
    <str name="fq">language:en</str>
    <str name="f.subject.qf">subject_en</str>
    <str name="f.message.qf">message_en</str>
    <str name="uf">* -subject_* -message_*</str>
    <str name="fl">*,subject:subject_en,message:message_en</
str>
  </lst>
</requestHandler>
```

c. Repeat the process for Russian request handler and use `ru` instead of `en` in all definitions:

```
<requestHandler name="/selectRU" class="solr.SearchHandler" >
  <lst name="defaults">
    <str name="defType">edismax</str>
    <str name="qf">addr_from^10 addr_to addr_cc subject
message</str>
    <str name="df">text</str>
    <str name="wt">ruby</str>
    <str name="indent">true</str>
  </lst>
  <lst name="invariants">
    <str name="fq">language:ru</str>
    <str name="f.subject.qf">subject_ru</str>
    <str name="f.message.qf">message_ru</str>
    <str name="uf">* -subject_* -message_*</str>
    <str name="fl">*,subject:subject_ru,message:message_ru</
str>
  </lst>
</requestHandler>
```

10. Reload the `language` core from the Admin WebUI's **Core Admin** screen.
11. You can now see that making requests to new handlers only return English and Russian results correspondingly:
    - `http://localhost:8983/solr/languages/selectEN?q=*%3A*`
    - `http://localhost:8983/solr/languages/selectRU?q=*%3A*`
    - The full results are still visible by accessing the original search handler: `http://localhost:8983/solr/languages/select?q=*%3A*`
12. If you look at the results, you will also notice that even though the subject and message are stored in fields with language suffixes in their names, they are returned as subject and message fields without language suffixes for both English and Russian searches in our new request handlers.
13. You can also do fielded and non-fielded searches against the language-suffixed fields without mentioning the suffixes:
    - тёплая – `http://localhost:8983/solr/languages/selectRU?q=%D1%82%D1%91%D0%BF%D0%BB%D0%B0%D1%8F` (1 result)
    - message:тёплая – `http://localhost:8983/solr/languages/selectRU?q=message:%D1%82%D1%91%D0%BF%D0%BB%D0%B0%D1%8F` (1 result)
    - subject:тёплая – `http://localhost:8983/solr/languages/selectRU?q=subject:%D1%82%D1%91%D0%BF%D0%BB%D0%B0%D1%8F` (0 result, no matches in subject)
    - Vacancies – `http://localhost:8983/solr/languages/selectEN?q=Vacancies` (2 results)
    - subject:Vacancies – `http://localhost:8983/solr/languages/selectEN?q=subject:Vacancies` (2 results)
    - message:Vacancies – `http://localhost:8983/solr/languages/selectEN?q=message:Vacancies` (0 results)
14. Finally, notice that even though we know that the text is stored in, say, the `subject_en` field, we cannot query it directly with the English request handler and have to use the unsuffixed subject field. We can, of course, still use the un-aliased field name against the original search handler:
    - `http://localhost:8983/solr/languages/selectEN?q=subject_en:Vacancies` (0 results)
    - `http://localhost:8983/solr/languages/select?q=subject_en:Vacancies` (2 results)

How it works...

To understand how this example works, we need to break it into parts, which we will cover in the following sections.

Getting the text into the right fields

When the text comes in, we need to know in which fields Solr needs to look for the text, what languages to look for, and where to put the results and the discovered language. The following flags on the Update Request Processor achieve that:

Field	Value	Meaning
`langid.fl`	subject, message	Check at the source fields subject and message to identify the language.
`langid.whitelist`	en,ru	Only chose between English and Russian—no other language options allowed.
`langid.fallback`	en	If, somehow, another language is detected, treat it as English.
`langid.langField`	language	Put the detected language code into the language field.
`langid.map`	true	Use fields with language suffixes as the destination (default mapping is `<field>_<lang>`).
`langid.map.keepOrig`	false	Do not keep original field values, which means all language-specific content ends up in one of the suffixed field names and the original subject and message fields are always empty.

Since we now know that Russian text will be in the `*_ru` fields and English text in the `*_en` fields, we can adjust `schema.xml` to declare it that way and provide language-specific processing, including grammar-based stemming, lower-casing, and stop-word removals. And since the original subject and message fields are now empty, we need to make them non-required.

Notice that later, when we actually run the queries, the word we use to test Russian search is `тёплая` when the actual text is `тёплой` (different declension of the same word) and we are relying on Russian grammar rules to ensure the match. The same thing-with searching English for `Vacancies` while the text only contains `vacancy`. You can explore these analysis chains in detail in the Admin WebUI's **Analysis** screen by testing the text against the `text_en` or `text_ru` types.

Searching the right fields

We have chosen to use custom request handlers to simplify the client interface. All our client application needs to do is to choose a correct URL endpoint and the language-specific properties will be mapped automatically.

We are doing a couple of things to ensure correct searching:

- We are using filter queries (the `fq` parameter) to limit the search only to the documents with appropriate language values. This does not conflict with whatever the user wants to put in the query (the `q` parameter) and Solr has internal optimizations for enforcing the `fq` limits.
- We are mapping any queries done against the fields `subject` and `message` to the corresponding language-suffixed fields. The basic format of the field alias parameter is `f.<aliasField>.qf=<realField>`. The more complicated format allows us to alias to multiple fields and to provide boosts. Notice that the alias affects even field names in the already existing global query field's (`qf`) configuration, so we do not need to change the names there. After the alias is defined, there is no longer a way to query the original subject and message. That's not a problem, as we don't have any values there anymore.
- To complete the alias mapping, we are also hiding language-specific fields from direct access by specifying which user fields (`uf`) we are exposing to the query. Our configuration (`* -subject_* -message_*`) allows all (`*`) fields to be queried except (represented by the minus sign in the example) for fields starting from `subject_` or `message_`, which are the language-specific fields.

Displaying the right fields

Now that we found the documents, we want to map language-specific fields to the base names. Just as with searches, it simplifies iterating and mapping the results.

To do so, we use a different kind of field aliasing that can be applied when selecting what fields to display (the `fl` parameter). The format of display aliasing is `<aliasField>:<realField>`, so our Russian configuration (`*,subject:subject_ru,message:message_ru`) says that we want to return all fields, but under the name `subject` we should return the content of the field `subject_ru`. It is the same for `message` and `message_ru`.

Final touches

As the language configuration is complicated and requires various settings and aliases to work together, we don't want a user to override those settings either accidentally or on purpose.

Therefore, we have put all the language-specific settings in the `invariants` sections, instead of customary `defaults` section. Of course, the other settings such as the return format (`wt`) are still in the `defaults` section and can be overridden with a query parameter.

There's more...

We have just scratched the surface of language processing. The following are a couple more directions worth exploring.

Language identification algorithms

Language identification is a complex, non-precise world. There is no magic algorithm that will identify any language in a world from a short snippet. Therefore, there are different language algorithms and different sets of languages supported by them.

We have used `solr.LangDetectLanguageIdentifierUpdateProcessorFactory`, which uses Google's `LangDetect` library. Solr can also use Tika's `solr.TikaLanguageIdentifierUpdateProcessorFactory`, which uses a different algorithm and supports a different set of languages (depending on which version of Tika is bundled with your version of Solr).

You may need to experiment with different algorithms and different thresholds to achieve the best results for your production scenario.

Enhancing the default search handler

In a production environment, you will probably delete the default search handler, as it is exposing language-related implementation details. However, you may want to keep it around for now, for debugging purposes.

If you do keep it around, you will notice that non-field (for example, `Vacancies`) searches will no longer match against our `message` and `subject` fields. The reason for that is because the `qf` definition still has the `subject` and `message` fields in it, but without alias definitions like in other handlers, these refer to the original—now always empty—fields. To fix that, add the following lines either in `defaults` or in a separate `invariants` section of the `/select` handler:

```
<str name="f.subject.qf">subject_en subject_ru</str>
<str name="f.message.qf">message_en message_ru</str>
```

Now, after reloading the core in Admin WebUI, all of the following queries will produce matches:

- `http://localhost:8983/solr/languages/select?q=subject_en:Vacancies`
- `http://localhost:8983/solr/languages/select?q=subject:Vacancies`
- `http://localhost:8983/solr/languages/select?q=Vacancies`
- `http://localhost:8983/solr/languages/select?q=%D1%82%D1%91%D0%BF%D0%BB%D0%B0%D1%8F` (тёплая)

You may need to force-reload the web page if you are still getting no results. Sometimes browsers will cache the old results too aggressively.

Removing the catch-all (text) field

You may have noticed that since we started using the eDisMax handler, we have no real use for the catch-all text field. In fact, the current configuration does not even copy language-specific fields into text.

It would make sense to remove the text field, all the related `copyField` instructions and other references from the configuration files.

We are leaving this as a reader exercise. We have added the field in the *Indexing text (Intermediate)* recipe; you should be able to follow the instructions to reverse the effect.

Thank you for buying
Instant Apache Solr for Indexing Data How-to

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.

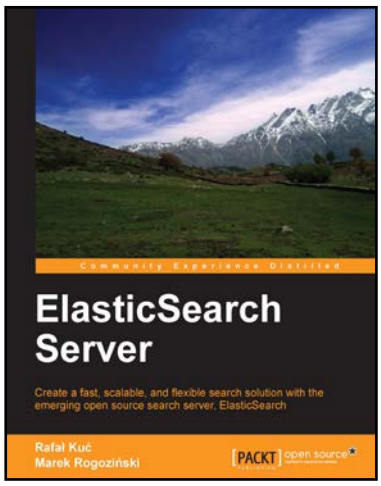
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, 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.

[PACKT] PUBLISHING



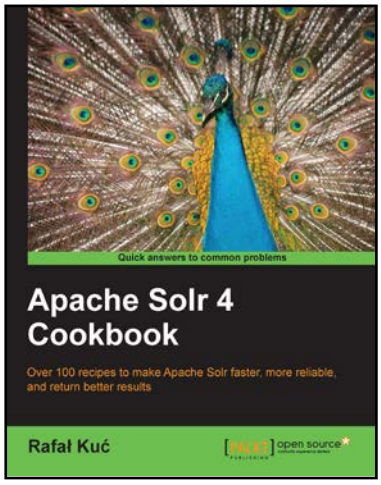
ElasticSearch Server

ISBN: 978-1-84951-844-4 Paperback: 318 pages

Create a fast, scalable, and flexible search solution with the emerging open source search server, ElasticSearch

1. Learn the basics of ElasticSearch like data indexing, analysis, and dynamic mapping
2. Query and filter ElasticSearch for more accurate and precise search results
3. Learn how to monitor and manage ElasticSearch clusters and troubleshoot any problems that arise

---



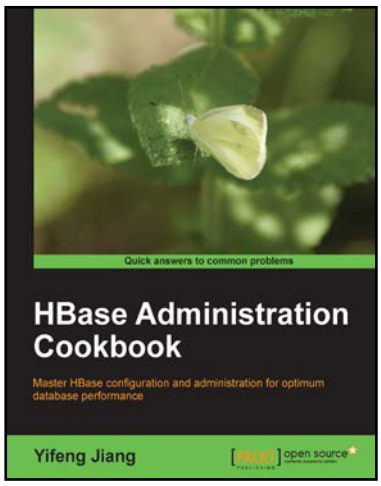
Apache Solr 4 Cookbook

ISBN: 978-1-78216-132-5 Paperback: 328 pages

Over 100 recipes to make Apache Solr faster, more reliable, and return better results

1. Learn how to make Apache Solr search faster, more complete, and comprehensively scalable
2. Solve performance, setup, configuration, analysis, and query problems in no time
3. Get to grips with, and master, the new exciting features of Apache Solr 4

[PACKT] PUBLISHING



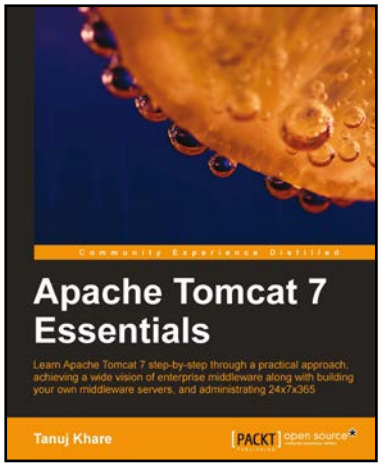
HBase Administration Cookbook

ISBN: 978-1-84951-714-0 Paperback: 332 pages

Master HBase configuration and administration for optimum database performance

1. Move large amounts of data into HBase and learn how to manage it efficiently
2. Set up HBase on the cloud, get it ready for production, and run it smoothly with high performance
3. Maximize the ability of HBase with the Hadoop eco-system including HDFS, MapReduce, Zookeeper, and Hive

---



Apache Tomcat 7 Essentials

ISBN: 978-1-84951-662-4 Paperback: 294 pages

Learn Apache Tomcat 7 step-by-step through a practical approach,achieving a wide vision of enterprise middleware along with building your own middleware servers, and administrating 24x7x365

1. Readymade solution for web technologies for migration/hosting and supporting environment for Tomcat 7
2. Tips, tricks, and best practices for web hosting solution providers for Tomcat 7
3. Content designed with practical approach and plenty of illustrations