Solr 1.4 Enterprise Search Server

Enhance your search with faceted navigation, result highlighting, fuzzy queries, ranked scoring, and more

David Smiley

Eric Pugh

BIRMINGHAM - MUMBAI

www.allitebooks.com

Solr 1.4 Enterprise Search Server

Copyright © 2009 Packt Publishing

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

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

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

First published: August 2009

Production Reference: 1120809

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

ISBN 978-1-847195-88-3

www.packtpub.com

Cover Image by Harmeet Singh (singharmeet@yahoo.com)

www.allitebooks.com

Credits

Authors
David Smiley
Eric Pugh

Reviewers
James Brady
Jerome Eteve

Acquisition Editor
Rashmi Phadnis

Development Editor
Darshana Shinde

Technical Editor
Pallavi Kachare

Copy Editor
Leonard D'Silva

Indexer
Monica Ajmera

Production Editorial Manager
Abhijeet Deobhakta

Editorial Team Leader
Akshara Aware

Project Team Leader
Priya Mukherji

Project Coordinator
Leena Purkait

Proofreader
Lynda Sliwoski

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

www.allitebooks.com

About the Authors

Born to code, **David Smiley** is a senior software developer and loves programming. He has 10 years of experience in the defense industry at MITRE, using Java and various web technologies. David is a strong believer in the opensource development model and has made small contributions to various projects over the years.

David began using Lucene way back in 2000 during its infancy and was immediately excited by it and its future potential. He later went on to use the Lucene based "Compass" library to construct a very basic search server, similar in spirit to Solr. Since then, David has used Solr in a major search project and was able to contribute modifications back to the Solr community. Although preferring open source solutions, David has also been trained on the commercial Endeca search platform and is currently using that product as well as Solr for different projects.

www.allitebooks.com

Most, if not all, authors seem to dedicate their book to someone. As simply a reader of books, I have thought of this seeming prerequisite as customary tradition. That was my feeling before I embarked on writing about Solr, a project that has sapped my previously "free" time on nights and weekends for a year. I chose this sacrifice and would not change it, but my wife, family, and friends did not choose it. I am married to my lovely wife Sylvie who has sacrificed easily as much as I have to complete this book. She has suffered through this time with an absentee husband while bearing our first child—Camille. She was born about a week before the completion of my first draft and has been the apple of my eye ever since. I officially dedicate this book to my wife Sylvie and my daughter Camille, whom I both lovingly adore. I also pledge to read book dedications with newfound firsthand experience at what the dedication represents.

I would also like to thank others who helped bring this book to fruition. Namely, if it were not for Doug Cutting creating Lucene with an open source license, there would be no Solr. Furthermore, CNet's decision to open source what was an in-house project, Solr itself in 2006, deserves praise. Many corporations do not understand that open source isn't just "free code" you get for free that others wrote; it is an opportunity to let your code flourish on the outside instead of it withering inside. Finally, I thank the team at Packt who were particularly patient with me as a first-time author writing at a pace that left a lot to be desired.

Last but not least, this book would not have been completed in a reasonable time were it not for the assistance of my contributing author, Eric Pugh. His perspectives and experiences have complemented mine so well that I am absolutely certain the quality of this book is much better than what I could have done alone.

Thank you all.

www.allitebooks.com

Fascinated by the 'craft' of software development, **Eric Pugh** has been heavily involved in the open source world as a developer, committer, and user for the past five years. He is an emeritus member of the Apache Software Foundation and lately has been mulling over how we move from the read/write Web to the read/write/share Web.

In biotech, financial services, and defense IT, he has helped European and American companies develop coherent strategies for embracing open source software. As a speaker, he has advocated the advantages of Agile practices in software development.

Eric became involved with Solr when he submitted the patch SOLR-284 for Parsing Rich Document types such as PDF and MS Office formats that became the single most popular patch as measured by votes! The patch was subsequently cleaned up and enhanced by three other individuals, demonstrating the power of the open source model to build great code collaboratively. SOLR-284 was eventually refactored into Solr Cell as part of Solr version 1.4.

He blogs at `http://www.opensourceconnections.com/blog/`.

www.allitebooks.com

Throughout my life I have been helped by so many people, but all too rarely do I get to explicitly thank them. This book is arguable one of the high points of my career, and as I wrote it, I thought about all the people who have provided encouragement, mentoring, and the occasional push to succeed. First off, I would like to thank Erik Hatcher, author, entrepreneur, and great family man for introducing me to the world of open source software. My first hesitant patch to Ant was made under his tutelage, and later my interest in Solr was fanned by his advocacy. Thanks to Harry Sleeper for taking a chance on a first time conference speaker; he moved me from thinking of myself as a developer improving myself to thinking of myself as a consultant improving the world (of software!). His team at MITRE are some of the most passionate developers I have met, and it was through them I met my co-author David. I owe a huge debt of gratitude to David Smiley. He has encouraged me, coached me, and put up with my lack of respect for book deadlines, making this book project a very positive experience! I look forward to the next one. With my new son Morgan at home, I could only have done this project with a generous support of time from my company, OpenSource Connections. I am incredibly proud of what o19s is accomplishing!

Lastly, to the all the folks in the Solr/Lucene community who took the time to review early drafts and provide feedback: Solr is at the tipping point of becoming the "it" search engine because of your passion and commitment

I am who I am because of my wife, Kate. Schweetie, real life for me began when we met. Thank you.

www.allitebooks.com

About the Reviewers

James Brady is an entrepreneur and software developer living in San Francisco, CA. Originally from England, James discovered his passion for computer science and programming while at Cambridge University. Upon graduation, James worked as a software engineer at IBM's Hursley Park laboratory—a role which taught him many things, most importantly, his desire to work in a small company.

In January 2008, James founded WebMynd Corp., which received angel funding from the Y Combinator fund, and he relocated to San Francisco. WebMynd is one of the largest installations of Solr, indexing up to two million HTML documents per day, and making heavy use of Solr's multicore features to enable a partially active index.

Jerome Eteve holds a BSC in physics, maths and computing and an MSC in IT and bioinformatics from the University of Lille (France). After starting his career in the field of bioinformatics, where he worked as a biological data management and analysis consultant, he's now a senior web developer with interests ranging from database level issues to user experience online. He's passionate about open source technologies, search engines, and web application architecture. At present, he is working since 2006 for Careerjet Ltd, a worldwide job search engine.

www.allitebooks.com

Table of Contents

Preface **1**

Chapter 1: Quick Starting Solr **7**

An introduction to Solr **7**
Lucene, the underlying engine 8
Solr, the Server-ization of Lucene 8
Comparison to database technology **9**
Getting started **10**
The last official release or fresh code from source control 11
Testing and building Solr 12
Solr's installation directory structure 13
Solr's home directory 15
How Solr finds its home 15
Deploying and running Solr 17
A quick tour of Solr! **18**
Loading sample data 20
A simple query 22
Some statistics 24
The schema and configuration files **25**
Solr resources outside this book **26**
Summary **27**

Chapter 2: Schema and Text Analysis **29**

MusicBrainz.org **30**
One combined index or multiple indices **31**
Problems with using a single combined index 33
Schema design **34**
Step 1: Determine which searches are going to be powered by Solr 35
Step 2: Determine the entities returned from each search 35

www.allitebooks.com

Step 3: Denormalize related data 36
Denormalizing—"one-to-one" associated data 36
Denormalizing—"one-to-many" associated data 36
Step 4: (Optional) Omit the inclusion of fields only used in search results 38
The schema.xml file 39
Field types 40
Field options 40
Field definitions 42
Sorting 44
Dynamic fields 45
Using copyField 46
Remaining schema.xml settings 47
Text analysis 47
Configuration 48
Experimenting with text analysis 50
Tokenization 52
WorkDelimiterFilterFactory 53
Stemming 54
Synonyms 55
Index-time versus Query-time, and to expand or not 57
Stop words 57
Phonetic sounds-like analysis 58
Partial/Substring indexing 60
N-gramming costs 61
Miscellaneous analyzers 62
Summary 63
Chapter 3: Indexing Data 65
Communicating with Solr 65
Direct HTTP or a convenient client API 65
Data streamed remotely or from Solr's filesystem 66
Data formats 66
Using curl to interact with Solr 66
Remote streaming 68
Sending XML to Solr 69
Deleting documents 70
Commit, optimize, and rollback 70
Sending CSV to Solr 72
Configuration options 73
Direct database and XML import 74
Getting started with DIH 75
The DIH development console 76

www.allitebooks.com

DIH documents, entities 78
DIH fields and transformers 79
Importing with DIH 80
Indexing documents with Solr Cell 81
Extracting binary content 81
Configuring Solr 83
Extracting karaoke lyrics 83
Indexing richer documents 85
Summary 88
Chapter 4: Basic Searching 89
Your first search, a walk-through 89
Solr's generic XML structured data representation 92
Solr's XML response format 93
Parsing the URL 94
Query parameters 95
Parameters affecting the query 95
Result paging 96
Output related parameters 96
Diagnostic query parameters 98
Query syntax 99
Matching all the documents 99
Mandatory, prohibited, and optional clauses 99
Boolean operators 100
Sub-expressions (aka sub-queries) 101
Limitations of prohibited clauses in sub-expressions 102
Field qualifier 102
Phrase queries and term proximity 103
Wildcard queries 103
Fuzzy queries 105
Range queries 105
Date math 106
Score boosting 107
Existence (and non-existence) queries 107
Escaping special characters 108
Filtering 108
Sorting 109
Request handlers 110
Scoring 112
Query-time and index-time boosting 113
Troubleshooting scoring 113
Summary 115

Chapter 5: Enhanced Searching **117**
Function queries **117**
An example: Scores influenced by a lookupcount 118
Field references 120
Function reference 120
Mathematical primitives 121
Miscellaneous math 121
ord and rord 122
An example with scale() and lookupcount 123
Using logarithms 123
Using inverse reciprocals 124
Using reciprocals and rord with dates 126
Function query tips 128
Dismax Solr request handler **128**
Lucene's DisjunctionMaxQuery 130
Configuring queried fields and boosts 131
Limited query syntax 131
Boosting: Automatic phrase boosting 132
Configuring automatic phrase boosting 133
Phrase slop configuration 134
Boosting: Boost queries 134
Boosting: Boost functions 137
Min-should-match 138
Basic rules 139
Multiple rules 139
What to choose 140
A default search 140
Faceting **141**
A quick example: Faceting release types 142
MusicBrainz schema changes 144
Field requirements 146
Types of faceting 146
Faceting text 147
Alphabetic range bucketing (A-C, D-F, and so on) 148
Faceting dates 149
Date facet parameters 151
Faceting on arbitrary queries 152
Excluding filters 153
The solution: Local Params 155
Facet prefixing (term suggest) 156
Summary **158**

Chapter 6: Search Components 159
About components 159
The highlighting component 161
A highlighting example 161
Highlighting configuration 163
Query elevation 166
Configuration 167
Spell checking 169
Schema configuration 169
Configuration in solrconfig.xml 171
Configuring spellcheckers (dictionaries) 173
Processing of the q parameter 175
Processing of the spellcheck.q parameter 176
Building the dictionary from its source 176
Issuing spellcheck requests 177
Example usage for a mispelled query 178
An alternative approach 180
The more-like-this search component 182
Configuration parameters 183
Parameters specific to the MLT search component 183
Parameters specific to the MLT request handler 184
Common MLT parameters 185
MLT results example 186
Stats component 189
Configuring the stats component 189
Statistics on track durations 190
Field collapsing 191
Configuring field collapsing 192
Other components 193
Terms component 194
termVector component 194
LocalSolr component 194
Summary 195
Chapter 7: Deployment 197
Implementation methodology 197
Questions to ask 198
Installing into a Servlet container 199
Differences between Servlet containers 199
Defining solr.home property 199

Logging **201**
HTTP server request access logs 201
Solr application logging 203
Configuring logging output 203
Logging to Log4j 204
Jetty startup integration 205
Managing log levels at runtime 205
A SearchHandler per search interface **207**
Solr cores **208**
Configuring solr.xml 208
Managing cores 209
Why use multicore 210
JMX **212**
Starting Solr with JMX 212
Take a walk on the wild side! Use JRuby to extract JMX information 215
Securing Solr **217**
Limiting server access 217
Controlling JMX access 220
Securing index data 220
Controlling document access 221
Other things to look at 221
Summary **222**
Chapter 8: Integrating Solr **223**
Structure of included examples **223**
Inventory of examples 224
SolrJ: Simple Java interface **224**
Using Heritrix to download artist pages 226
Indexing HTML in Solr 227
SolrJ client API 230
Indexing POJOs 234
When should I use Embedded Solr 235
In-Process streaming 236
Rich clients 237
Upgrading from legacy Lucene 237
Using JavaScript to integrate Solr **238**
Wait, what about security? 239
Building a Solr powered artists autocomplete widget with
jQuery and JSONP 240
SolrJS: JavaScript interface to Solr 245
Accessing Solr from PHP applications **247**
solr-php-client 248
Drupal options 250
Apache Solr Search integration module 251

Hosted Solr by Acquia 252
Ruby on Rails integrations 253
acts_as_solr 254
Setting up MyFaves project 255
Populating MyFaves relational database from Solr 256
Build Solr indexes from relational database 258
Complete MyFaves web site 260
Blacklight OPAC 263
Indexing MusicBrainz data 263
Customizing display 267
solr-ruby versus rsolr 269
Summary 270
Chapter 9: Scaling Solr 271
Tuning complex systems 271
Using Amazon EC2 to practice tuning 273
Firing up Solr on Amazon EC2 274
Optimizing a single Solr server (Scale High) 276
JVM configuration 277
HTTP caching 277
Solr caching 280
Tuning caches 281
Schema design considerations 282
Indexing strategies 283
Disable unique document checking 285
Commit/optimize factors 285
Enhancing faceting performance 286
Using term vectors 286
Improving phrase search performance 287
The solution: Shingling 287
Moving to multiple Solr servers (Scale Wide) 289
Script versus Java replication 289
Starting multiple Solr servers 290
Configuring replication 291
Distributing searches across slaves 291
Indexing into the master server 292
Configuring slaves 292
Distributing search queries across slaves 293
Sharding indexes 295
Assigning documents to shards 296
Searching across shards 297
Combining replication and sharding (Scale Deep) 298
Summary 300
Index 301

Preface

Text search has been around for perhaps longer than we all can remember. Just about all systems, from client installed software to web sites to the web itself, have search. Yet there is a big difference between the best search experiences and the mediocre, unmemorable ones. If you want the application you're building to stand out above the rest, then it's got to have great search features. If you leave this to the capabilities of a database, then it's near impossible that you're going to get a great search experience, because it's not going to have features that users come to expect in a great search. With Solr, the leading open source search server, you'll tap into a host of features from highlighting search results to spell-checking to faceting.

As you read *Solr Enterprise Search Server* you'll be guided through all of the aspects of Solr, from the initial download to eventual deployment and performance optimization. Nearly all the options of Solr are listed and described here, thus making this book a resource to turn to as you implement your Solr based solution. The book contains code examples in several programming languages that explore various integration options, such as implementing query auto-complete in a web browser and integrating a web crawler. You'll find these working examples in the online supplement to the book along with a large, real-world, openly available data set from `MusicBrainz.org`. Furthermore, you will also find instructions on accessing a Solr image readily deployed from within Amazon's Elastic Compute Cloud.

Solr Enterprise Search Server targets the Solr 1.4 version. However, as this book went to print prior to Solr 1.4's release, two features were not incorporated into the book: search result clustering and trie-range numeric fields.

What this book covers

Chapter 1, Quick Starting Solr introduces Solr to the reader as a middle ground between database technology and document/web crawlers. The reader is guided through the Solr distribution including running the sample configuration with sample data.

Chapter 2, The Schema and Text Analysis is all about Solr's schema. The schema design is an important first order of business along with the related text analysis configuration.

Chapter 3, Indexing Data details several methods to import data; most of them can be used to bring the MusicBrainz data set into the index. A popular Solr extension called the DataImportHandler is demonstrated too.

Chapter 4, Basic Searching is a thorough reference to Solr's query syntax from the basics to range queries. Factors influencing Solr's scoring algorithm are explained here, as well as diagnostic output essential to understanding how the query worked and how a score is computed.

Chapter 5, Enhanced Searching moves on to more querying topics. Various score boosting methods are explained from those based on record-level data to those that match particular fields or those that contain certain words. Next, faceting is a major subject area of this chapter. Finally, the term auto-complete is demonstrated, which is implemented by the faceting mechanism.

Chapter 6, Search Components covers a variety of searching extras in the form of Solr "components", namely, spell-check suggestions, highlighting search results, computing statistics of numeric fields, editorial alterations to specific user queries, and finding other records "more like this".

Chapter 7, Deployment transits from running Solr from a developer-centric perspective to deploying and running Solr as a deployed production enterprise service that is secure, has robust logging, and can be managed by System Administrators.

Chapter 8, Integrating Solr surveys a plethora of integration options for Solr, from supported client libraries in Java, JavaScript, and Ruby, to being able to consume Solr results in XML, JSON, and even PHP syntaxes. We'll look at some best practices and approaches for integrating Solr into your web application.

Chapter 9, Scaling Solr looks at how to scale Solr up and out to avoid meltdown and meet performance expectations. This information varies from small changes of configuration files to architectural options.

Who this book is for

This book is for developers who would like to use Solr to implement a search capability for their applications. You need only to have basic programming skills to use Solr; extending or modifying Solr itself requires Java programming. Knowledge of Lucene, the foundation of Solr, is certainly a bonus.

Conventions

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

Code words in text are shown as follows: "These are essentially defaults for searches that are processed by Solr request handlers defined in `solrconfig.xml`."

A block of code is set as follows:

```
<uniqueKey>id</uniqueKey>
<!-- <defaultSearchField>text</defaultSearchField>
<solrQueryParser defaultOperator="AND"/> -->
<copyField source="r_name" dest="r_name_sort" />
```

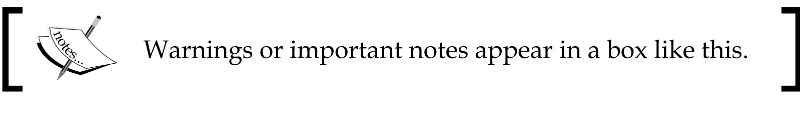
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<arr name="id">
  <str>mccm.pdf</str>
</arr>
```

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

```
>> curl http://localhost:8983/solr/karaoke/update/ -H "Content-Type:
text/xml" --data-binary '<commit waitFlush="false"/>'
```

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: "Take for example the **Top Voters** section ".



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 email to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, 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 for the book

Visit `http://www.packtpub.com/files/code/5883_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

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 to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. 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 web site 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.

1
Quick Starting Solr

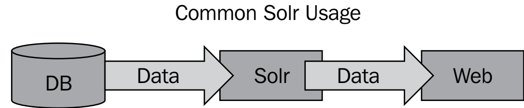
Welcome to Solr! You've made an excellent choice in picking a technology to power your searching needs. In this chapter, we're going to cover the following topics:

- An overview of what Solr and Lucene are all about
- What makes Solr different from other database technologies
- How to get Solr, what's included, and what is where
- Running Solr and importing sample data
- A quick tour of the interface and key configuration files

An introduction to Solr

Solr is an open source enterprise search server. It is a mature product powering search for public sites like CNet, Zappos, and Netflix, as well as intranet sites. It is written in Java, and that language is used to further extend/modify Solr. However, being a server that communicates using standards such as HTTP and XML, knowledge of Java is very useful but not strictly a requirement. In addition to the standard ability to return a list of search results for some query, it has numerous other features such as result highlighting, faceted navigation (for example, the ones found on most e-commerce sites), query spell correction, auto-suggest queries, and "more like this" for finding similar documents.

Common Solr Usage



Lucene, the underlying engine

Before describing Solr, it is best to start with Apache Lucene, the core technology underlying it. Lucene is an open source, high-performance text search engine library. Lucene was developed and open sourced by Doug Cutting in 2000 and has evolved and matured since then with a strong online community. Being just a code library, Lucene is not a server and certainly isn't a web crawler either. This is an important fact. There aren't even any configuration files. In order to use Lucene directly, one writes code to store and query an index stored on a disk. The major features found in Lucene are as follows:

- A text-based inverted index persistent storage for efficient retrieval of documents by indexed terms
- A rich set of text analyzers to transform a string of text into a series of terms (words), which are the fundamental units indexed and searched
- A query syntax with a parser and a variety of query types from a simple term lookup to exotic fuzzy matches
- A good scoring algorithm based on sound **Information Retrieval (IR)** principles to produce the more likely candidates first, with flexible means to affect the scoring
- A **highlighter** feature to show words found in context
- A query spellchecker based on indexed content

For even more information on the query spellchecker, check out the *Lucene In Action* book (LINA for short) by Erik Hatcher and Otis Gospodnetić.

Solr, the Server-ization of Lucene

With the definition of Lucene behind us, Solr can be described succinctly as the server-ization of Lucene. However, it is definitely not a thin wrapper around the Lucene libraries. Most of Solr's features are distinct from Lucene, such as faceting, but not far into the implementation. The line is often blurred as to what is Solr and what is Lucene. Without further adieu, here is the major feature-set in Solr:

- HTTP request processing for indexing and querying documents.
- Several caches for faster query responses.
- A web-based administrative interface including:
    - Runtime performance statistics including cache hit/miss rates.

    - A query form to search the index.
    - A schema browser with histograms of popular terms along with some statistics.
    - Detailed breakdown of scoring mathematics and text analysis phases.
- Configuration files for the schema and the server itself (in XML).
    - Solr adds to Lucene's text analysis library and makes it configurable through XML.
    - Introduces the notion of a field type (this is important yet surprisingly not in Lucene). Types are present for dates and special sorting concerns.
- The **disjunction-max** query handler is more usable by end user queries and applications than Lucene's underlying raw queries.
- Faceting of query results.
- A **spell check** plugin used for making alternative query suggestions (that is, "did you mean ___")
- A **more like this** plugin to list documents that are similar to a chosen document.
- A distributed Solr server model with supporting scripts to support larger scale deployments.

These features will be covered in more detail in later chapters.

Comparison to database technology

Knowledge of relational databases (often abbreviated RDBMS or just database for short) is an increasingly common skill that developers possess. A database and a [Lucene] search index aren't dramatically different conceptually. So let's start off by assuming that you know database basics, and I'll describe how a search index is different.

> This comparison puts aside the possibility that your database has built-in text indexing features. The point here is only to help you understand Solr.

This biggest difference is that a Lucene index is like a single-table database without any support for relational queries (JOINs). Yes, it sounds crazy, but remember that an index is usually only there to support search and not to be the primary source of the data. So your database may be in "third normal form" but the index will be completely de-normalized and contain mostly just the data needed to be searched. One redeeming aspect of the single table schema is that fields can be multi-valued.

Other notable differences are as follows:

- **Updates**: Entire documents can be deleted and added again but not updated.
- **Substring Search versus Text Search**: Using a database, the poor man's search would be a substring search such as `SELECT * FROM mytable WHERE name LIKE '%Books%'`. That would match "CookBooks" as well as "My Books". Lucene instead fundamentally searches on terms (words). Depending on analysis configuration, this can mean that various forms of the word (example: book, singular) are found too, even phonetic (sounds-like) matches are possible. Using advanced **ngram** analysis techniques, it can do partial words too, although this is uncommon.
- **Scored Results and Boosting**: Much of the power of Lucene is in its ability to score each matched document according to how well the search matched it. For example, if multiple words are searched for and are optional (a boolean `OR` search), then Lucene scores documents that matched more terms higher than those that just matched one. There are a variety of other factors too, and it's possible to adjust weightings of different fields. By comparison, a database has no concept of this, a record either matched or not. Of course, Lucene can sort on field values if that is needed.
- **Slow commits**: Solr is highly optimized for search speed, and that speed is largely attributable to caches. When a commit is done to finalize documents that were just added, all of the caches need to be rebuilt, which could take between seconds and a minute, depending on various factors.

Getting started

Solr is a Java based web application, but you don't need to be particularly familiar with Java in order to use it. With most topics, this book assumes little to no such knowledge on your part. However, if you wish to extend Solr, then you will definitely need to know Java. I also assume a basic familiarity with the command line, whether it is DOS or any Unix shell.

Before truly getting started with Solr, let's get the prerequisites out of the way. Note that if you are using Mac OS X, then you should have the needed pieces already (though you may need the developer tools add-on). If any of the `-version` test commands mentioned as follows fail, then you don't have it. URLs are provided for convenience, but it is up to you to install the software according to instructions provided at the relevant sites.

- A **Java Development Kit (JDK)** v1.5 or later: You can download the JDK from `http://java.sun.com/javase/`. Typing `java -version` will tell you which version of Java you are using if any, and you should type `javac -version` to ensure that you have the development kit too. You only need the JRE to run Solr, but you will need the JDK to compile it from source and to extend it.
- **Apache Ant**: Any recent version should do and is available at `http://ant.apache.org/`. If you never modify Solr and just stick to a recent official release, then you can skip this. Note that the software provided with this book uses Ant as well. Therefore, you'll want Ant if you wish to follow along. Typing `ant -version` should demonstrate that you have it installed.
- **Subversion or Git for source control of Solr**: `http://subversion.tigris.org/getting.html` or `http://git-scm.com/`. This isn't strictly necessary, but it's recommended for working with Solr's source code. If you choose to use a command line based distribution of either, then `svn -version` or `git --version` should work. Further instructions in this book are based on the command line, because it is a universal access method.
- **Any Java EE servlet engine app-server**: This is a Java web server. Solr includes one already, Jetty, and we'll be using this throughout the book. In a later chapter, "Solr in the real world", deploying to an alternative is discussed.

The last official release or fresh code from source control

Let's finally get started and get Solr running. The official site for Solr is at `http://lucene.apache.org/solr`, where you can download the latest official release. Solr 1.3 was released on September 15th, 2008. Solr 1.4 is expected around the same time a year later and thus is probably available as you read this. This book was written in-between these releases and so it contains many but not all of 1.4's features. An alternative to downloading an official release is getting the latest code from source control (that is version control). In either case, the directory structure is conveniently identical and both include the source code. For many open source projects, the choice is almost always the last official release and not the latest source.

However, Solr's committers have made unit and integration testing a priority, evident by the testing infrastructure and test code-coverage of over 70 percent (`http://hudson.zones.apache.org/hudson/view/Solr/job/Solr-trunk/clover/`), which is very good. Many projects have none at all. As a result, the latest source release is very stable, and it also makes changes to Solr easier, given that so many tests are in place to give confidence that Solr is working properly—so far as the tests test it, of course. And unlike a database, which is almost never modified to suit the needs of a project, Solr is modified often. Also note that there are a good many feature additions provided as source code patches within Solr's JIRA (its issue tracking system). The decision is of course up to you. If you are satisfied with the feature-set in the latest release and/or you don't think you'll be modifying Solr at all, then the latest release is fine. One way to gauge what (completed) features are not yet in the latest official release is to visit Solr's JIRA at `http://issues.apache.org/jira/browse/SOLR`, and then click on **Roadmap**. Also, the Wiki at `http://wiki.apache.org/solr/` should have features that are not yet in the latest release version marked as such.

Choose to get Solr through source control even if you are going to stick with the last official release. When/if you make changes to Solr, it will then be easier to see what those differences are. Switching to a different release becomes much easier too.

We're going to get the code through a subversion and check out the `trunk` (a source control term for the latest code). If you are using an IDE or some GUI tool for subversion, then feel free to use that. The command line will suffice too. You should be able to successfully execute the following:

```
svn co http://svn.apache.org/repos/asf/lucene/solr/trunk/ solr_svn
```

That will result in Solr being checked out into the `solr_svn` directory. If you prefer one of the official releases, then use one of the following URLs, instead of the one above: `http://svn.apache.org/repos/asf/lucene/solr/tags/` (put that into your web browser to see the choices). So called nightlies are also available if you don't want to use a subversion but want recent code.

Testing and building Solr

If you prefer a downloadable pre-built Solr, instead of using a subversion, then you can skip this section.

Ant basics

Apache ant is a cross-platform build scripting tool specified with XML. It is largely Java oriented. An ant script is assumed to be named `build.xml` in the root of a project. It contains a set of named ant `targets` that you can run. In order to list them while including description, type `ant -p` to get a nice report. In order to run a target, simply supply it to ant as the first argument such as `ant compile`. Targets often internally invoke other targets, and you'll see this in the output. In the end, ant should report **BUILD SUCCESSFUL** if successful and **BUILD FAILED** if not. Note that ant's use of the term 'build' is universal in ant, even if 'build' is not an apt description of what a target performed.

Testing and building Solr is easy. Before we build Solr, we're going to test it first to ensure that there are no failing tests. Simply execute the `test` target in Solr's installation directory like `ant test`. That should have executed without any errors. On my old machine, it took about ten minutes to run. If there were errors (extremely rare), then you'll have to switch to a different version or wait shortly for it to be fixed. Now to build a ready-to-install Solr, just type `ant dist`. This is going to fill the `dist` directory with some JAR files and a WAR file. If you are not familiar with Java, these files are a packaging mechanism for compiled code and related resources. These files are technically ZIP files but with a different file extension, and so you can use any ZIP file tools to view their contents. The most important one is the WAR file which we'll be using next.

Solr's installation directory structure

In this section, we'll orient you to Solr's directory structure. This is not Solr's home directory, but a different place that we'll mention after this.

- `build`: Only appears after Solr is built to house compiled code before being packaged. You won't need to look in here.
- `client`: Contains convenient language-specific APIs for talking to Solr as an alternative to using your own code to send XML over HTTP. As of this writing, this only contains a couple of Ruby choices. The Java client called SolrJ is actually in `src/solrj`. More information on using clients to communicate with Solr is in Chapter 8.
- `dist`: The built Solr JAR files and WAR file are here, as well as the dependencies. This directory is created and filled when Solr is built.

- `example`: This is an installation of the Jetty servlet engine (a Java web server) including some sample data and Solr configuration. The interesting child directories are:
    - `example/etc`: Jetty's configuration. Among other things, here you can change the web port used from the pre-supplied 8983 to 80 (HTTP default).
    - `example/multicore`: Houses multiple Solr home directories in a Solr `multicore` setup. This will be discussed in Chapter 7.
    - `example/solr`: A Solr `home` directory for the default setup that we'll be using.
    - `example/webapps`: Solr's WAR file is deployed here.
- `lib`: All of Solr's API dependencies. The larger pieces are Lucene, some Apache commons utilities, and Stax for efficient XML processing.
- `site`: This is for managing what is published on the Solr web site. You won't need to go in here.
- `src`: Various source code. It's broken down into a few notable directories:
    - `src/java`: Solr's source code, written in Java.
    - `src/scripts`: Unix bash shell scripts, particularly useful in larger production deployments employing multiple Solr servers.
    - `src/solrj`: Solr's Java client.
    - `src/test`: Solr's test source code and test files.
    - `src/webapp`: Solr's web administration interface, including Java Servlets (source code form) and JSPs. This is mostly what constitutes the WAR file. The JSPs for the admin interface are under here in `web/admin/`, if you care to tweak any to your needs.

If you are a Java developer, you may have noticed that the Java source in Solr is not located in one place. It's in `src/java` for the majority of Solr, `src/common` for the parts of Solr that are common to both the server side and Solrj client side, `src/test` for the test code, and `src/webapp/src` for the servlet-specific code. I am merely pointing this out to help you find code, not to be critical. Solr's files are well organized.

www.allitebooks.com

Solr's home directory

A Solr home directory contains Solr's configuration and data (a Lucene Index) for a running Solr instance. Solr includes a sample, one at `example/solr`, which we'll be using in-place throughout most of the book. Technically, `example/multicore` is also a valid Solr home but for a multi-core setup, which will be discussed much later. You know you're looking at a Solr home directory when it contains either a `solr.xml` file (formerly `multicore.xml` in Solr 1.3), or if it contains both a `conf` and a `data` directory, though strictly speaking these might not be the actual requirements.

`data` might not yet be present because you haven't started Solr yet, which will create it if it's not present and assuming it's not configured to be named differently.

Solr's home directory is laid out like this:

- `bin`: Suggested directory to place Solr replication scripts, if you have a more advanced setup.
- `conf`: Configuration files. The two I mention below are very important, but it will also contain some other `.txt` and `.xml` files, which are referenced by these two files for different things such as special text analysis steps.
- `conf/schema.xml`: This is the schema for the index including field type definitions with associated analyzer chains.
- `conf/solrconfig.xml`: This is the primary Solr configuration file.
- `conf/xslt`: This directory contains various XSLT files that can be used to transform Solr's XML query responses into formats such as Atom/RSS.
- `data`: Contains the actual Lucene index data. It's binary data, so you won't be doing anything with it except perhaps deleting it occasionally.
- `lib`: Optional placement of extra Java JAR files that Solr will load on startup, allowing you to externalize plugins from the Solr distribution (the WAR file) for convenience. If you extend Solr without modifying Solr itself, then those modifications can be deployed in a JAR file here.

It's really important to know how Solr finds its home directory. This is covered next.

How Solr finds its home

In the next section, you'll start Solr. When Solr starts up, about the first thing it does is load its configuration from its home directory. Where that is exactly can be specified in several different ways.

Solr first checks for a Java system property named `solr.solr.home`. There are a few ways to set a Java system property, but a universal one, no matter which servlet engine you use, is through the command line where Java is invoked. You could explicitly set Solr's home like so when you start Jetty: `java -Dsolr.solr.home=solr/ -jar start.jar`, or you could use **Java Naming and Directory Interface (JNDI)** to bind the directory path to `java:comp/env/solr/home`. As with Java system properties, there are multiple ways to do this. Some are app-server dependent, but a universal one is to add the following to the WAR file's `web.xml` located in `src/web-app/web/WEB-INF` (you'll find this there already but commented out).

```
<env-entry>
  <env-entry-name>solr/home</env-entry-name>
  <env-entry-value>solr/</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

As this is a change to `web.xml`, you'll need to re-run `ant dist-war` to repackage it, and only then you'll redeploy it. Doing this with Jetty supplied with Solr is insufficient because JNDI itself isn't set up. I'm not going to get into this further, because if you know what JNDI is and want to use it, then you'll surely figure out how to do it for your particular app-server.

Finally, if Solr's home isn't configured as a Java system property or through JNDI, then it defaults to `solr/`. In the examples above, I used that particular path too. We're going to simply stick with this path for the rest of this book, because this is a development, not production, setting.

In a production environment, you will almost certainly configure Solr's home rather than let it fall back to the default `solr/`. You will also probably use an absolute path instead of a relative one, which wouldn't work if you accidentally start your app-server from a different directory.

When troubleshooting setting Solr's home, be sure to look at the very first Solr log messages when Solr starts:

Aug 7, 2008 4:59:35 PM org.apache.solr.core.Config getInstanceDir

INFO: Solr home defaulted to 'null' (could not find system property or JNDI)

Aug 7, 2008 4:59:35 PM org.apache.solr.core.Config setInstanceDir

INFO: Solr home set to 'solr/'

This shows that Solr was left to default to `solr/`. You'll see this output when you start Solr, as described in the next section.

Deploying and running Solr

The file we're going to deploy is the file ending in `.war` in the `dist` directory (`dist/apache-solr-1.4.war`). The WAR file in particular is important, because this single file represents an entire Java web application. It includes Solr's JAR file, all of Solr's dependencies (which amount to other JAR files), **Java Server Pages (JSPs)** (which are rendered to a web browser when the WAR is deployed), and various configuration files and other web resources. It does not include Solr's home directory, however.

How one deploys a WAR file to a Java servlet engine depends on that servlet engine, but it is common for there to be a directory named something like `webapps`, which contains WAR files optionally in an expanded form. By expanded, I mean that the WAR file may be uncompressed and thus a directory by the same name. This can be a convenient deployed form in order to make changes in-place (such as to JSP files and static web files) without requiring rebuilding a WAR file and replacing an existing one. The disadvantage is that changes are not directly tracked by source control (example: Subversion). Another thing to note about the WAR file is that by convention, its name (without the `.war` extension, if present) is the path portion of the URL where the web server mounts the web application. For example, if you have an `apache-solr-1.4.war` file, then you would access it at `http://localhost:8983/apache-solr-1.4/`, assuming it's on the local machine and running at that default port.

We're going to deploy this WAR file into the Jetty servlet engine included with Solr. If you are using a pre-built downloaded Solr distribution, then Solr is already deployed into Jetty as `solr.war`. Solr has an ant target that does this (and some other things we don't care about) called `example`, so you can simply run it like `ant example`. This target didn't keep the original WAR filename when copying it. It abbreviated it to simply `solr.war`. This means that the URL path is just `solr`. By the way, because ant targets generally call other necessary ant targets, it was technically not necessary to run `ant dist` earlier in order for this step to work. This would not have run the tests, however.

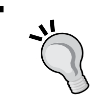
Now we're going to start up Jetty and finally see Solr running (albeit without any data to query yet). First go to the `example` directory, and then run Jetty's `start.jar` file by typing the following command:

```
cd example
java -jar start.jar
```

You'll see about a page of output including references to Solr. When it is finished, you should see this output at the very end of the command prompt:

```
2008-08-07 14:10:50.516::INFO: Started SocketConnector @ 0.0.0.0:8983
```

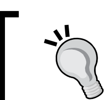
The `0.0.0.0` means it's listening to connections from any host (not just localhost, notwithstanding potential firewalls) and 8983 is the port. If Jetty reports this, then it doesn't necessarily mean that Solr was deployed successfully. You might see an error such as a stack trace in the output, if something went wrong. Even if it did go wrong, you should be able to access the web server at this address: `http://localhost:8983`. It will show you a list of links to web applications which will just be Solr for this setup. Solr should have this link: `http://localhost:8983/solr`, and if you go there, then you should either see details about an error if Solr wasn't loaded correctly, or a simple page with a link to Solr's admin page, which should be `http://localhost:8983/solr/admin/`. You'll be visiting that link often.



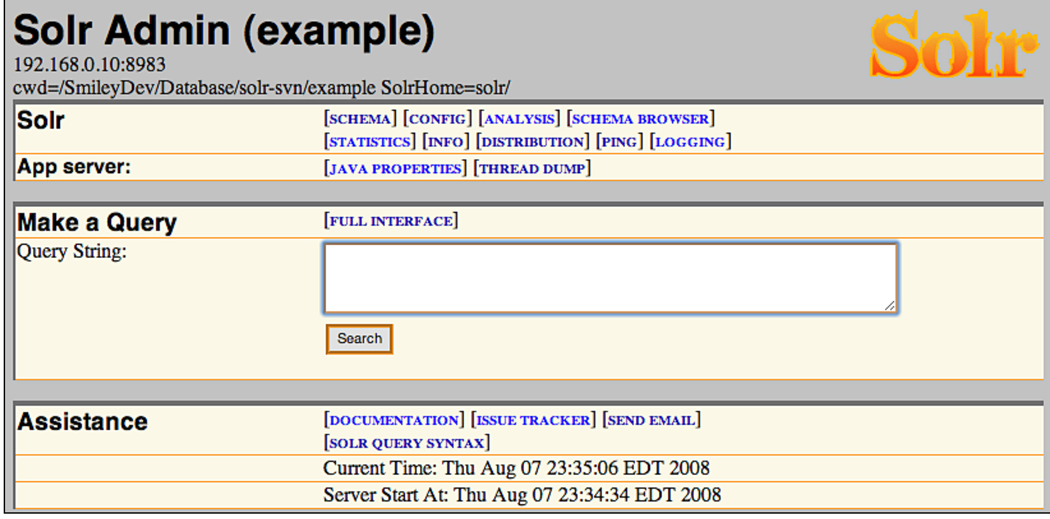
To quit Jetty (and many other command line programs for that matter), hit *Ctrl-C* on the keyboard.

A quick tour of Solr!

Start up Jetty if it isn't already up and point your browser to the admin URL: `http://localhost:8983/solr/admin/`, so that we can get our bearings on this interface that is not yet familiar to you. We're not going to discuss any page in any depth at this point.



This part of Solr is somewhat rough and is subject to change more than any other part of Solr.



The top gray area in the previous screenshot is a header that is on every page. When you start dealing with multiple Solr instances (development machine versus production, multicore, Solr clusters), it is important to know where you are. The IP and port are obvious. The **(example)** is a reference to the name of the schema. That's just a simple label at the top of the schema file to name the schema. If you have multiple schemas for different data sets, then this is a useful differentiator. Next is the current working directory **cwd**, and Solr's home.

The block below this is a navigation menu to the different admin screens and configuration data. The navigation menu is explained as follows:

- **SCHEMA**: This downloads the schema configuration file (XML) directly to the browser.

> Firefox conveniently displays XML data with syntax highlighting. Safari, on the other hand, tries to render it and the result is unusable. Your mileage will vary depending on the browser you use. You can always use your browser's view source command if needed.

- **CONFIG**: It is similar to the **SCHEMA** choice, but this is the main configuration file for Solr.
- **ANALYSIS**: It is used for diagnosing potential query/indexing problems having to do with the text analysis. This is a somewhat advanced screen and will be discussed later.
- **SCHEMA BROWSER**: This is a neat view of the schema reflecting various heuristics of the actual data in the index. We'll return here later.
- **STATISTICS**: Here you will find stats such as timing and cache hit ratios. In Chapter 9, we will visit this screen to evaluate Solr's performance.
- **INFO**: This lists static versioning information about internal components to Solr. Frankly, it's not very useful.
- **DISTRIBUTION**: It contains Distributed/Replicated status information, only applicable for such configurations. More information on this is in Chapter 9.
- **PING**: Ignore this, although it can be used for a health-check in distributed mode.
- **LOGGING**: This allows you to adjust the logging levels for different parts of Solr at runtime. For Jetty as we're running it, this output goes to the console and nowhere else.

Solr uses SLF4j for its logging, which in Solr, is by default configured to use Java's built-in logging (that is JUL or JDK14 Logging). If you're more familiar with another framework like Log4J, then you can do this by simply removing the slf4j-jdk14 JAR file and adding slf4j-log4j12 (not included). If you're using Solr 1.3, then you're stuck with JUL.

- **JAVA PROPERTIES**: It lists Java system properties.
- **THREAD DUMP**: This displays a Java thread dump useful for experienced Java developers in diagnosing problems.

After the main menu is the **Make a Query** text box where you can type in a simple query. There's no data in Solr yet, so there's no point trying that right now.

- **FULL INTERFACE**: As you might guess, it brings you to a form with more options, especially useful when diagnosing query problems or if you forget what the URL parameters are for some of the query options. The form is still very limited, however, and only allows a fraction of the query options that you can submit to Solr.

Finally, the bottom **Assistance** area contains useful information for Solr online. The last section of this chapter has more information on such resources.

Loading sample data

Solr happens to come with some sample data and a loader script, found in the `example/exampledocs` directory. We're going to use that, but just for the remainder of this chapter so that we can explore Solr more without getting into schema decision making and deeper data loading options. For the rest of the book, we'll base the examples on the supplemental files, which are provided online.

Firstly, ensure that Solr is running. You should assume that it is always in a running state throughout this book to follow any example. Now go into the `example/exampledocs` directory, and run the following:

```
exampledocs$ java -jar post.jar *.xml
SimplePostTool: version 1.2
SimplePostTool: WARNING: Make sure your XML documents are encoded in UTF-
8, other encodings are not currently supported
SimplePostTool: POSTing files to http://localhost:8983/solr/update..
SimplePostTool: POSTing file hd.xml
SimplePostTool: POSTing file ipod_other.xml
SimplePostTool: POSTing file ipod_video.xml
SimplePostTool: POSTing file vidcard.xml
SimplePostTool: COMMITting Solr index changes..
```

Or if you are using a Unix-like environment, you have the option of using the `post.sh` shell script, which behaves similarly. What this does is it invokes the Java program embedded in `post.jar` with each file in the current directory ending in `.xml`. `post.jar` is a simple program that iterates over each argument given (a file reference), and HTTP posts it to Solr running on the current machine at the example server's default configuration (being `http://localhost:8983/solr/update`). I recommend examining the contents of the `post.sh` shell script for illustrative purposes. As seen above, the command will mention the files it is sending. Finally it will send a `commit` command, which will cause documents that were posted prior to the last commit to be saved and visible.

> The `post.sh` and `post.jar` programs could theoretically be used in a production scenario, but they are intended just for demonstration of the technology with the example data.

Let's take a look at one of these documents like `monitor.xml`:

```
<add>
  <doc>
    <field name="id">3007WFP</field>
    <field name="name">Dell Widescreen UltraSharp 3007WFP</field>
    <field name="manu">Dell, Inc.</field>
    <field name="cat">electronics</field>
    <field name="cat">monitor</field>
    <field name="features">30" TFT active matrix LCD, 2560 x 1600,
                 .25mm dot pitch, 700:1 contrast</field>
    <field name="includes">USB cable</field>
    <field name="weight">401.6</field>
    <field name="price">2199</field>
    <field name="popularity">6</field>
    <field name="inStock">true</field>
  </doc>

</add>
```

The schema for the XML files that are posted to Solr are very simple. This one here doesn't demonstrate all of it, but this is most of what matters. Multiple documents (represented by the `<doc>` tag) can be present in series within the `<add>` tag, which is recommended in bulk data loading scenarios for performance. Remember that Solr gets a `<commit/>` tag sent to it in a separate POST. This syntax and command-set may very well be all that you use. More about these options and other data loading choices will be discussed in Chapter 3.

A simple query

On the main admin page, let's run a simple query searching for `monitor`.

When using Solr's search form, don't hit the return key. It would be nice if it submits the form, but it adds a carriage return to the search box instead. If you leave this carriage return there and hit **Search**, then you'll get an error. Perhaps this will be fixed at some point.

Before we go over the XML output, I want to point out the URL and its parameters, which you will become very familiar with: `http://localhost:8983/solr/select/?q=monitor&version=2.2&start=0&rows=10&indent=on`.

The form (whether the basic one or the **Full Interface** one) simply constructs a URL with appropriate parameters, and your browser sees the XML results. It is convenient to use the form at first, but then subsequently make direct modifications to the URL in the browser instead of returning to the form. The form only controls a basic subset of all possible parameters. The main benefit to the form is that it applies the URL escaping for special characters in the query, and for some basic options, you needn't remember what the parameter names are.

Solr's search results from its web interface are in XML. As suggested earlier, you'll probably find that using the Firefox web browser provides the best experience due to the syntax coloring. Internet Explorer displays XML content well too. If you, at some point, want Solr to return a web page to your liking or an alternative XML structure, then that will be covered later. Here is the XML response with my comments:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">3</int>
  <lst name="params">
    <str name="indent">on</str>
    <str name="rows">10</str>
    <str name="start">0</str>
    <str name="q">monitor</str>
    <str name="version">2.2</str>
  </lst>
</lst>
```

The first section of the response, which precedes the `<result>` tag that is about to follow, indicates how long the query took (measured in milliseconds), as well as listing the parameters that define the query. Solr has some sophisticated caching, and you will find that your queries will often complete in a millisecond or less, if you've run the query before. In the `params` list, `q` is clearly your query. `rows` and `start` have to do with paging. Clearly you wouldn't want Solr to always return all of the results at once, unless you really knew what you were doing. `indent` indents the XML output, which is convenient for experimentation. `version` isn't used much, but if you start building clients that interact with Solr, then you'll want to specify the version to reduce the possibility of things breaking, if you were to upgrade Solr. These parameters in the output are convenient for experimentation but can be configured to be omitted. Next up is the most important part, the results.

```
<result name="response" numFound="2" start="0">
```

The `numFound` number is self explanatory. `start` is the index into the query results that are returned in the XML. Often, you'll want to see the score of the documents. However, the very basic query performed from the front Solr page doesn't include the score, despite the fact that it's sorted by it (Solr's default). The full interface form includes the score by default. Queries that include the score will include a `maxScore` attribute in the result tag. The `maxScore` for the query is independent of any paging, so that no matter which part of the result set you've paged into (by using the `start` parameter), the `maxScore` will be the same. The content of the result tag is a list of documents that matched the query in a score sorted order. Later, we'll do some sorting by specified fields.

```
<doc>
    <arr name="cat"><str>electronics</str><str>monitor</str></arr>
    <arr name="features"><str>30" TFT active matrix LCD, 2560 x 1600,
               .25mm dot pitch, 700:1 contrast</str></arr>
    <str name="id">3007WFP</str>
    <bool name="inStock">true</bool>
    <str name="includes">USB cable</str>
    <str name="manu">Dell, Inc.</str>
    <str name="name">Dell Widescreen UltraSharp 3007WFP</str>
    <int name="popularity">6</int>
    <float name="price">2199.0</float>
    <str name="sku">3007WFP</str>
    <arr name="spell"><str>Dell Widescreen UltraSharp 3007WFP</str>
    </arr>
    <date name="timestamp">2008-08-09T03:56:41.487Z</date>
    <float name="weight">401.6</float>
</doc>
<doc>
...
</doc>
</result>
</response>
```

The document list is pretty straightforward. By default, Solr will list all of the stored fields, plus the score if you asked for it (we didn't in this case). Remember that not all of the fields are necessarily stored (that is, you can query on them but not store them for retrieval—an optimization choice). Notice that basic data types `str`, `bool`, `date`, `int`, and `float` are used. Also note that certain fields are multi-valued, as indicated by an `arr` tag.

This was a basic query. As you start adding more query options like faceting, highlighting, and so on, you will see additional XML following the `result` tag.

Some statistics

Let's take a look at the statistics page: `http://localhost:8983/solr/admin/stats.jsp`. Before we loaded data into Solr, this page reported that `numDocs` was `0`, but now it should be `26`. If you're wondering what `maxDocs` is and the difference, `maxDocs` reports a number that is in some situations higher due to documents that have been deleted but not yet committed. That can happen either due to an explicit delete posted to Solr or by adding a document that replaces another in order to enforce a unique primary key. While you're at this page, notice that the query handler named `/update` has some stats too:

name	**/update**
class	org.apache.solr.handler.XmlUpdateRequestHandler
version	$Revision: 679936 $
description	Add documents with XML
stats	handlerStart: 1218253728453
	requests: 19
	errors: 4
	timeouts: 0
	totalTime: 1392
	avgTimePerRequest: 73.26316
	avgRequestsPerSecond: 2.850955E-4

In my case, as seen above, there are some errors reported because I was fooling around, posting all of the files in the `exampledocs` directory, not just the XML ones. Another Solr handler name you'll want to examine is `standard`, which has been processing our queries.

These statistics are as up-to-date as Solr is running, they are not stored to disk. As such, you cannot use them for long-term statistics.

The schema and configuration files

Solr's configuration files are extremely well documented. We're not going to go over the details here but this should give you a sense of what is where.

The schema (defined in `schema.xml`) contains field type definitions (defined within the `<types>` tag) and lists the fields that make up your schema (within the `<fields>` tag), which references a type. The schema contains other information too such as the primary key (the field that uniquely identifies each document—a constraint that Solr enforces) and the default search field. The sample schema in Solr uses the field named `text`, confusingly, there is a field type named `text` too. But remember that the `monitor.xml` document we reviewed earlier had no field named `text`, right? It is common for the schema to call out for certain fields to be copied to other fields—particularly fields not in input documents. So, even though the input documents don't have a field named `text`, there are `<copyField>` tags in the schema, which call for the fields named `cat`, `name`, `manu`, `features`, and `includes` to be copied to `text`. This is a popular technique to speed up queries, so that queries can search over a small number of fields rather than a long list of them. Such fields used this way are rarely `stored`, as they are just needed for querying and so are `indexed`. There is a lot more we could talk about in the schema, but we're going to move on for now.

Solr's `solrconfig.xml` file contains lots of parameters that can be tweaked. At the moment, we're just going to take a peak at the request handlers that are defined with `<requestHandler>` tags. They make up about half of the file. In our first query, we didn't specify any request handler, so we got the default one. It's defined here:

```
<requestHandler name="standard" class="solr.SearchHandler"
    default="true">
<!-- default values for query parameters -->
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <!--
    <int name="rows">10</int>
    <str name="fl">*</str>
    <str name="version">2.1</str>
    -->
  </lst>
</requestHandler>
```

When you POST commands to Solr (such as to index a document) or query Solr (HTTP GET), it goes through a particular request handler. Handlers can be registered against certain URL paths. When we uploaded the documents earlier, it went to the handler defined like this:

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

The request handlers oriented to querying using the class `solr.SearchHandler` are much more interesting.

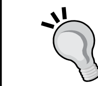
> The important thing to realize about using a request handler is that they are nearly completely configurable through URL parameters or POST'ed form parameters. They can also be specified in `solrconfig.xml` within either `default`, `appends`, or `invariants` named `lst` blocks, which serve to establish defaults. More on this is in Chapter 4. This arrangement allows you to set up a request handler for a particular application that will be querying Solr without forcing the application to specify all of its query options.

The `standard` request handler defined previously doesn't really define any defaults other than the parameters that are to be echoed in the response. Remember its presence at the top of the XML output? By changing `explicit` to `none` you can have it omitted, or use `all` and you'll potentially see more parameters, if other defaults happened to be configured in the request handler. This parameter can alternatively be specified in the URL through `echoParams=none`. Remember to separate URL parameters with ampersands.

Solr resources outside this book

The following are some prominent Solr resources that you should be aware of:

- Solr's Wiki: `http://wiki.apache.org/solr/` has a lot of great documentation and miscellaneous information. For a Wiki, it's fairly organized too. In particular, if you are going to use a particular app-server in production, then there is probably a Wiki page there on specific details.
- Within the Solr installation, you will also find that there are `README.txt` files in many directories within Solr and that the configuration files are very well documented.
- Solr's mailing lists contain a wealth of information. If you have a few discriminating keywords then you can find nuggets of information in there with a search engine. The mailing lists of Solr and other Lucene sub-projects are best searched at: `http://www.lucidimagination.com/search/` or `Nabble.com`.



It is highly recommended to subscribe to the Solr-users mailing list. You'll learn a lot and potentially help others too.

- Solr's issue tracker, a JIRA installation at `http://issues.apache.org/jira/browse/SOLR` contains information on enhancements and bugs. Some of the comments for these issues can be extensive and enlightening. JIRA also uses a Lucene-powered search.

Notation convention: Solr's JIRA issues are referenced like this: SOLR-64. You'll see such references in this book and elsewhere. You can easily look these up at Solr's JIRA. You may also see issues for Lucene that follow the same convention, for example, LUCENE-1215.

Summary

This completes a quick introduction to Solr. In the ensuing chapters, you're really going to get familiar with what Solr has to offer. I recommend you proceed in order from the next chapter through Chapter 6, because these build on each other and expose nearly all of the capabilities in Solr. These chapters are also useful as a reference to Solr's features. You can of course skip over sections that are not interesting to you. Chapter 8, is one you might peruse at any time, as it may have a section particularly applicable to your Solr usage scenario.

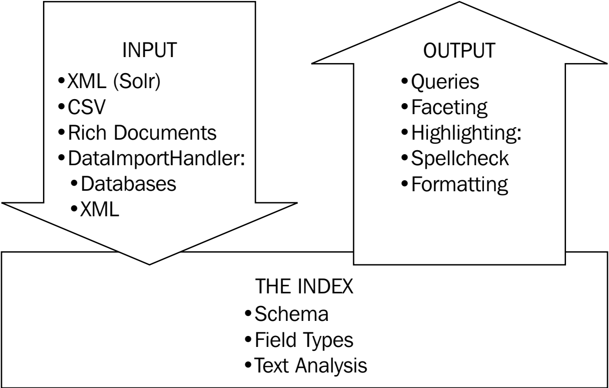
Accompanying the book at PACKT's web site is both source code and data to be indexed by Solr. In order to try out the same examples used in the book, you will have to download it and run the provided ant task, which prepares it for you. This first chapter is the only one that is not based on that supplemental content.

2
Schema and Text Analysis

The foundation of Solr is based on Lucene's index—the subject of this chapter. You will learn about:

- Schema design decisions in which you map your source data to Lucene's limited structure. In this book we'll consider the data from `MusicBrainz.org`.
- The structure of the `schema.xml` file wherein the schema definition is defined. Within this file are both the definition of field types and the fields of those types that store your data.
- Text analysis—the configuration of how text is processed (tokenized and so on) for indexing. This configuration affects whether or not a particular search is going to match a particular document.

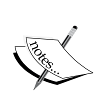
Observe the following diagram:



The next chapter will cover importing data into Solr. The later chapters cover all of the ways in which the data can be queried.

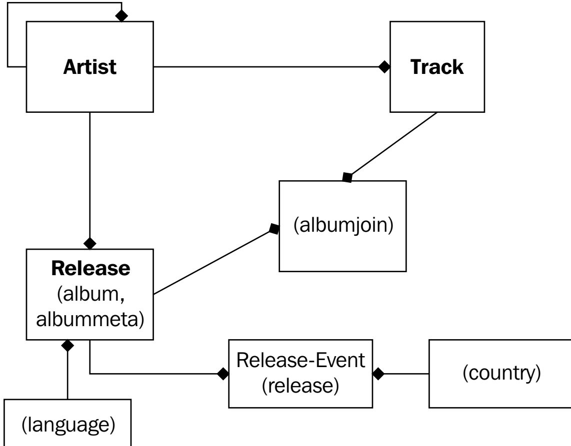
MusicBrainz.org

Instead of continuing to work with the sample data that comes along with Solr, we're going to use a large database of music metadata from the MusicBrainz project at `http://musicbrainz.org`. The data is free and is submitted by a large community of users. MusicBrainz offers nightly snapshots of this data in an SQL file that can be imported into a PostgreSQL database. There are instructions on the MusicBrainz web site on how to do this. Unfortunately, it requires some non-trivial setup and additional software. In order to make it easier for you to play with this data, this book comes with pre-queried data in formats that can readily be imported into Solr. See the files accompanying this book for instructions on how to load it. Alternatively, I recommend that you work with your data, using this book as a guide.

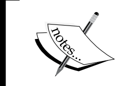


I chose the MusicBrainz data set because it has a lot of data, is freely available, and has an innovative approach to collaborative data collection. The MusicBrainz web site enforces a democratic process in which the community presides over proposed data additions and changes. This is in contrast to Wikipedia in which anyone can make an immediate change without it being approved by anyone else.

The `MusicBrainz.org` database is highly relational. Therefore, it will serve as an excellent instructional data set to discuss Solr schema choices. The MusicBrainz schema is actually quite a complex beast, and it would be a distraction to go over even half of it (not that I could do it). Instead, I'm going to use a subset of it and express it in a way that has a more straightforward mapping than the user interface seen on the web site. Each of these tables depicted below can be easily constructed through SQL sub-queries or views from the actual MusicBrainz tables.



I'll be using some examples here from my favorite band, the *Smashing Pumpkins*. This is an artist with a type of "group" (a band). Some artists (groups in particular) have members who are also other artists of type "person". So this is a self-referential relationship. The *Smashing Pumpkins* has Billy Corgan, Jimmy Chamberline, and others as members. An artist is attributed as the creator of a *release*. The most common type of release is an "album" but there are also singles, EPs, compilations, and others. Furthermore, releases have a "status" property that is either official, promotional, or bootleg. Associated with a release is some number of "events". The terminology is confusing, and perhaps a better word is "distribution" (which is admittedly still vague). Each event contains the date, country, music label, format (CD or tape), and some other uninteresting details that the release was distributed as. A popular official album from this band is titled "Siamese Dream". A release is composed of one or more tracks. Siamese Dream has 13 tracks starting with "Cherub Rock" and ending with "Luna". Note that a track is part of just one release and so it is not synonymous with a song. For example, the song "Cherub Rock" is not only a track on this release but also on a "Greatest Hits" release, as well as quite a few others in the database. MusicBrainz doesn't really have the notion of a song. However, one can make reasonable assumptions based on the track's name being unique, at least among those produced by the same artist. Additionally, there is a track's **PUID** (**PRONOM Unique Identifier**), an audio fingerprinting technology quasi-identifier based on the actual sound on a track. It's not foolproof as there are collisions, but they are rare. Another interesting bit of data MusicBrainz stores is the PUID "lookup count", which is how often it has been requested by their servers—a decent measure of popularity.



MusicBrainz uses the proprietary "MusicDNS" audio fingerprinting technology, which was donated to the project by MusicIP. It is not open source.

One combined index or multiple indices

As mentioned in the first chapter, technically, an index is basically like a single-table database schema. Imagine a massive spreadsheet, if you will. Inspite of this limitation, there is nothing to stop you from putting different types of data (say, artists and tracks from MusicBrainz) into a single index, thereby, in effect mitigating this limitation. All you have to do is use different fields for the different document types, and use a field to discriminate between the types. An identifier field would need to be unique across all documents in this index, no matter the type, so you could easily do this by concatenating the field type and the entity's identifier. This may appear really ugly from a relational database design standpoint, but this isn't a database. More importantly, unlike a database, there is no overhead

whatsoever for a document to have no data in a field. On the other hand, databases usually set aside space for data in a row whether it is filled or not. This is where the spreadsheet metaphor can break down, because a blank cell in a spreadsheet takes up space, but not in an index. Here's a sample `schema.xml` snippet of a single combined index approach:

```
<field name="id" ... /> <!-- example:  "artist:534445"  -->
<field name="type" ... /> <!-- example: "artist", "track", "release",
... -->
<field name="name" ... /> <!-- common to various types -->
<!-- track fields: -->
<field name="PUID" ... />
<field name="num" ... /> <!-- i.e. the track # on the release -->
<!-- ... -->
<!-- artist fields: -->
<field name="startDate" ... /> <!-- date of first release -->
<field name="endDate" ... /> <!-- date of last release -->
<field name="homeCountry" ... />
<!-- etc. -->
```

One combined index is often easier to use when compared to using a different index for each entity type. It's just one configuration to manage. A key deciding factor is if you need to query across them—an easy task with one index. Solr can search across multiple indices (a so-called "distributed search") too, however, it is a relatively new capability that has limitations.

"Distributed search" is mainly used for scaling a large index by breaking it up. See Chapter 9 for more information. However, large or not, it can also be used to search across heterogeneous indices.

The approaches can be mixed and matched for different entities if that suits you.

Before complicating your schema index strategy with more than one index, start with one index. If some of the issues (mentioned next) apply to you, then expand to multiple indices. In the example configuration using MusicBrainz, each entity type gets its own index, though one schema is used to ease configuration.

Problems with using a single combined index

Some problems that you may face while using a single combined index are as follows:

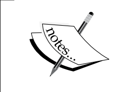
- There may be namespace collision problems unless you prefix the field names by type such as: `artist_startDate` and `track_PUID`. In the example that we just saw, most entity types have a `name`. Therefore, it's straightforward for all of them to have this common field. If the type of the fields were different, then you would be forced to name them differently.
- If you share the same field for different things (like the `name` field in the example that we have just seen), then there are some problems that can occur when using that field in a query and while filtering documents by document type. These caveats do not apply when searching across all documents.
    - You will get scores that are of lesser quality. The explanation for this is a little complicated, and you may need to read up on *Lucene scoring* to understand it better. One component of a score is the **IDF** (**Inverse Document Frequency**) of a term in the search. In other words, documents matching rare words get scored higher. IDF is based on all of the field values in the index (no matter what type of document it is). For example, if you put different types of things into the same field, then what could be a rare word for a track name might not be for an artist name. Therefore, searching for only tracks or only artists would not make use of good IDF factors when computing the score.
    - *Prefix*, *wildcard*, and *fuzzy* queries will take longer and will be more likely to reach internal scalability thresholds. These query types require scanning for all of the indexed terms used in a field to see if they match the queried term. If you share a field with different types of documents, then the total number of terms to search over is going to be larger, which takes longer to scan over. It will also match more terms than it would otherwise, while possibly generating a query that exceeds the `maxBooleanClauses` threshold (configurable in `solrconfig.xml`).
- With or without sharing field names, the IDF component of the score calculation is diluted, because the ratio is based on the total number of documents indexed, not just those of the type being queried.

- If you do not share field names and instead prefix the field names with a short type identifier like `track_name` instead of just `name`, then you may find it inconvenient or awkward, if the users are exposed to this.
- For a large number of documents, a strategy using multiple indices will prove to be more scalable. Only testing will indicate what "large" is for your data and your queries, but less than a million documents will not likely benefit from multiple indices. Ten million documents have been suggested as a reasonable maximum number for a single index. There are seven million tracks in MusicBrainz, so we'll definitely have to put tracks in its own index.
- Committing changes to a Solr index invalidates the caches used to speed up querying. If this happens often, and the changes are usually to one type of entity in the index, then you will get better query performance by using separate indices.

We've chosen to highlight four main entities in MusicBrainz: Artists, Releases, Tracks, and Labels. The web interface offers a search against the names of each of these entities.

Schema design

A key thing to come to grips with is that a Solr schema strategy is driven by how it is queried and not by a standard third normal form decomposition of the data. This isn't to say that all databases are pure third normal form and that they aren't influenced by queries. But in an index, the queries you need to support *completely drive* the schema design. This is necessary, as you can't perform relational queries on an index. Consequently all the data needed to match a document must be in the document matched. To satisfy that requirement, data that would otherwise exist in one place (like an artist's name in MusicBrainz, for example) is inlined into related entities that need it to support a search. This may feel dirty but I'll just say "get over it". Besides your data's gold source most likely is not in Solr.



Even if you're not working with a database as your source data, these concepts still apply. So pay close attention to this important subject in any case.

www.allitebooks.com

Step 1: Determine which searches are going to be powered by Solr

Any text search capability is going to be Solr powered. At the risk of stating the obvious, I'm referring strictly to those places where a user types in a bit of text and subsequently gets some search results. On the MusicBrainz web site, the main search function is accessed through the form that is always present on the left. There is also a more advanced form that adds a few options but is essentially the same capability, and I treat it as such from Solr's point of view. We can see the MusicBrainz search form in the next screenshot:

Once we look through the remaining steps, we may find that Solr should additionally power some faceted navigation in areas that are not accompanied by a text search (that is the facets are of the entire data set, not necessarily limited to the search results of a text query alongside it). An example of this at MusicBrainz is the "Top Voters" tally, which I'll address soon.

Step 2: Determine the entities returned from each search

For the MusicBrainz search form, this is easy. The entities are: Artists, Releases, Tracks, Labels, and Editors. It just so happens that in MusicBrainz, a search will only return one entity type. However, that needn't be the case. Note that internally, each result from a search corresponds to a distinct document in the Solr index and so each entity will have a corresponding document. This entity also probably corresponds to a particular row in a database table, assuming that's where it's coming from.

Step 3: Denormalize related data

For each entity type, find all of the data in the schema that will be needed across all searches of it. By "all searches of it," I mean that there might actually be multiple search forms, as identified in Step 1. Such data includes any data queried for (that is, criteria to determine whether a document matches or not) and any data that is displayed in the search results. The end result of denormalization is to have each document sufficiently self-contained, even if the data is duplicated across the index. Again, this is because Solr does not support relational joins. Let's see an example. Consider a search for tracks matching **Cherub Rock**:

The following tracks matched your query

Page 1 of 1114 Go!

Score	Num	Track	Duration	Type	Artist	Album	Tracks
100	1	Cherub Rock	4:58	album	The Smashing Pumpkins	Siamese Dream	13
100	1	Cherub Rock	4:59	single	The Smashing Pumpkins	Cherub Rock	3
100	4	Cherub Rock	4:57	compilation	The Smashing Pumpkins	Greatest Hits (Rotten Apples)	18
100	14	Cherub Rock	4:29	live	The Smashing Pumpkins	Turpentine Kisses	14
100	11	Cherub Rock	5:55	live	The Smashing Pumpkins	Squashed Zucchini	13

Denormalizing—"one-to-one" associated data

The track's name and duration are definitely in the track table, but the artist and album names are each in their own tables in the MusicBrainz schema. This is a relatively simple case, because each track has no more than one artist or album. Both the artist name and album name would get their own field in Solr's flat schema for a track. They also happen to be elsewhere in our Solr schema, because artists and albums were identified in Step 2. Since the artist and album names are not unambiguous references, it is useful to also add the IDs for these tables into the track schema to support linking in the user interface, among other things.

Denormalizing—"one-to-many" associated data

One-to-many associations can be easy to handle in the simple case of a field requiring multiple values. Unfortunately, databases make this harder than it should be if it's just a simple list. However, Solr's schema directly supports the notion of multiple values. Remember in the MusicBrainz schema that an artist can have some number of other artists as members. Although MusicBrainz's current search capability doesn't leverage this, we'll capture it anyway because it is useful for more interesting searches. The Solr schema to store this would simply have a member name field that is multi-valued (the syntax will come later). The `member_id` field alone would be insufficient, because denormalization requires that the member's name be inlined into the artist. This example is a good segue to how things can get a little more

complicated. If we only record the name, then it is problematic to do things like have links in the UI from a band member to that member's detail page. This is because we don't have that member's artist ID, only their name. This means that we'll need to have an additional multi-valued field for the member's ID. Multi-valued fields maintain ordering so that the two fields would have corresponding values at a given index. Beware, there can be a tricky case when one of the values can be blank, and you need to come up with a placeholder. The client code would have to know about this placeholder.

What you should *not* do is try to shove different types of data into the same field by putting both the artist IDs and names into one field. It could introduce text analysis problems, as a field would have to satisfy both types, and it would require the client to parse out the pieces. The exception to this is when you are not indexing the data and if you are merely storing it for display then you can store whatever you want in a field.

What about the track count of the corresponding album for this track? We'll use the same approach that MusicBrainz' relational schema does—inline this total into the album information, instead of computing it on the fly. Such an "on the fly" approach with a relational schema would involve relating in a tracks table and doing an SQL `group by` with a count. In Solr, the only way to compute this on the fly would be by submitting a second query, searching for tracks with album IDs of the first query, and then faceting on the album ID to get the totals. Faceting is discussed in Chapter 4.

Note that denormalizing in this way may work most of the time, **but there are limitations** in the way you query for things, which may lead you to take further steps. Here's an example. Remember that releases have multiple "events" (see my description earlier of the schema using the Smashing Pumpkins as an example). It is impossible to query Solr for releases that have an event in the UK that were over a year ago. The issue is that the criteria for this hypothetical search involves multi-valued fields, where the index of one matching criteria needs to correspond to the same value in another multi-valued field in the same index. You can't do that. But let's say that this crazy search example was important to your application, and you had to support it somehow. In that case, there is exactly one release for each event, and a query matching an event shouldn't match any other events for that release. So you could make event documents in the index, and then searching the events would yield the releases that interest you. This scenario had a somewhat easy way out. However, there is no general step-by-step guide. There are scenarios that will have no solution, and you may have to compromise. Frankly, Solr (like most technologies) has its limitations. Solr is not a general replacement for relational databases.

Step 4: (Optional) Omit the inclusion of fields only used in search results

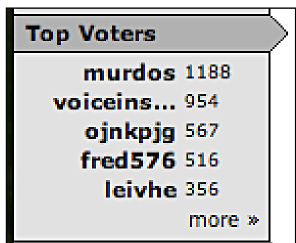
It's not likely that you will actually do this, but it's important to understand the concept. If there is any data shown on the search results that is not queryable, not sorted upon, not faceted on, nor are you using the highlighter feature for, and for that matter are not using any Solr feature that uses the field except to simply return it in search results, then it is not necessary to include it in the schema for this entity. Let's say, for the sake of the argument, that the only information queryable, sortable, and so on is a track's name, when doing a query for tracks. You can opt not to inline the artist name, for example, into the track entity. When your application queries Solr for tracks and needs to render search results with the artist's name, the onus would be on your application to get this data from somewhere—it won't be in the search results from Solr. The application might look these up in a database or perhaps even query Solr in its own artist entity if it's there or somewhere else.

This clearly makes generating a search results screen more difficult, because you now have to get the data from more than one place. Moreover, to do it efficiently, you would need to take care to query the needed data in bulk, instead of each row individually. Additionally, it would be wise to consider a caching strategy to reduce the queries to the other data source. It will, in all likelihood, slow down the total render time too. However, the benefit is that you needn't get the data and store it into the index at indexing time. It might be a lot of data, which would grow your index, or it might be data that changes often, necessitating frequent index updates.

If you are using distributed search (discussed in Chapter 9), there is some performance gain in not sending too much data around in the requests. Let's say that you have the lyrics to the song, it is distributed on 20 machines, and you get 100 results. This could result in 2000 records being sent around the network. Just sending the IDs around would be much more network efficient, but then this leaves you with the job of collecting the data elsewhere before display. The only way to know if this works for you is to test both scenarios. However, I have found that even with the very little overhead in HTTP transactions, if the record is not too large then it is best to send the 2000 records around the network, rather than make a second request.

Why not power all data with Solr?

It would be an interesting educational exercise to do so, but it's not a good idea to do so in practice (presuming your data is in a database too). Remember the "lookup versus search" point made earlier. Take for example the **Top Voters** section. The account names listed are actually editors in MusicBrainz terminology. This piece of the screen tallies an edit, grouped by the editor that performed the edit. It's the edit that is the entity in this case. The following screenshot is that of the **Top Voters** (aka editors), which is tallied by the number of edits:



This data simply doesn't belong in an index, because there's no use case for searching edits, only lookup when we want to see the edits on some other entity like an artist. If you insisted on having the voter's tally (seen above) powered by Solr, then you'd have to put all this data (of which there is a lot!) into an index, just because you wanted a simple statistical list of top voters. It's just not worth it! One objective guide to help you decide on whether to put an entity in Solr or not is to ask yourself if users will ever be doing a text search on that entity—a feature where index technology stands out from databases. If not, then you probably don't want the entity in your Solr index.

The schema.xml file

Let's get down to business and actually define our Solr schema for MusicBrainz.

We're going to define one index to store artists, releases (example albums), and labels. The tracks will get their own index, leveraging the **SolrCore** feature. This is because they are separate indices, and they don't necessarily require the same schema file. However, we'll use one because it's convenient. There's no harm in a schema defining fields which don't get used.

Before we continue, find a `schema.xml` file to follow along. This file belongs in the `conf` directory in a Solr home directory. In the example code distributed with the book, available online, I suggest looking at `cores/mbtracks/conf/schema.xml`. If you are working off of the Solr distribution, you'll find it in `example/solr/conf/schema.xml`. The example `schema.xml` is loaded with useful field types, documentation, and field definitions used for the sample data that comes with Solr. I prefer to begin a Solr index by copying the example Solr home directory and modifying it as needed, but some prefer to start with nothing. It's up to you.

At the start of the file is the schema opening tag:

```
<schema name="musicbrainz" version="1.1">
```

We've set the name of this schema to `musicbrainz`, the name of our application. If we use different schema files, then we should name them differently to differentiate them.

Field types

The first section of the schema is the definition of the field types. In other words, these are the data types. This section is enclosed in the `<types/>` tag and will consume lots of the file's content. The field types declare the types of fields, such as booleans, numbers, dates, and various text flavors. They are referenced later by the field definitions under the `<fields/>` tag. Here is the field type for a boolean:

```
<fieldType name="boolean" class="solr.BoolField"
           sortMissingLast="true" omitNorms="true"/>
```

A field type has a unique name and is implemented by a Java class specified by the `class` attribute.

Abbreviated Java class names

A fully qualified classname in Java looks like `org.apache.solr.schema.BoolField`. The last piece is the simple name of the class, and the part preceding it is called the package name. In order to make configuration files in Solr more concise, the package name can be abbreviated to just `solr` for most of Solr's built-in classes. Nearly all of the other XML attributes in a field type declaration are options, usually boolean, that are applied to the field that uses this type by default. However, a few are not overridable by the field. They are not specific to the field type and/or its class. For example, `sortMissingLast` and `omitNorms`, as seen above, are not `BoolField` specific configuration options, they are applicable to every field. Aside from the field options, there is the text analysis configuration that is only applicable to text fields. That will be covered later.

Field options

The options of a field specified using XML attributes are defined as follows:

These options are assumed to be boolean (`true/false`) unless indicated, otherwise `indexed` and `stored` default to `true`, but the rest default to `false`. These options are sometimes specified at the field *type* definition, which is inherited sometimes at the field definition. The indented options defined below, underneath `indexed` (and `stored`) imply `indexed` (`stored`) must be `true`.

- `indexed`: Indicates that this data should be searchable or sortable. If it is not `indexed`, then `stored` should be `true`. Usually fields are `indexed`, but sometimes if they are not, then they are included only in search results.
    - `sortMissingLast`, `sortMissingFirst`: Sorting on a field with one of these set to `true` indicates on which side of the search results to put documents that have no data for the specified field, regardless of the sort direction. The default behavior for such documents is to appear first for ascending and last for descending.
    - `omitNorms`: (advanced) Basically, if the length of a field does not affect your scores for the field, and you aren't doing index-time document boosting, then enable this. Some memory will be saved. For typical general text fields, you should not set `omitNorms`. Enable it if you aren't scoring on a field, or if the length of the field would be irrelevant if you did so.
    - `termVectors`: (advanced) This will tell Lucene to store information that is used in a few cases to improve performance. If a field is to be used by the *MoreLikeThis* feature, or if you are using it and it's a large field for highlighting, then enable this.
- `stored`: Indicates that the field is eligible for inclusion in search results. If it is not `stored`, then `indexed` should be `true`. Usually fields are stored, but sometimes the special fields that hold copies of other fields are not stored. This is because they need to be analyzed differently, or they hold multiple field values so that searches can search only one field instead of many to improve performance and reduce query complexity.
    - `compressed`: You may want to reduce the storage size at the expense of slowing down indexing and searching by compressing the field's data. Only the fields with a class of `StrField` or `TextField` are compressible. This is usually only suitable for fields that have over 200 characters, but it is up to you. You can set this threshold with the **compressThreshold** option in the field type, not the field definition.
- `multiValued`: Enable this if a field can contain more than one value. Order is maintained from that supplied at index-time.

> This is internally implemented by separating each value with a configurable amount of whitespace—the `positionIncrementGap`.

- `positionIncrementGap`: (advanced) For a `multiValued` field, this is the number of (virtual) spaces between each value to prevent inadvertent querying across field values. For example, A and B are given as two values for a field, which prevents A and B from matching.

Field definitions

The definitions of the fields in the schema are located within the `<fields/>` tag. In addition to the field options defined above, a field has these attributes:

- `name`: Uniquely identifies the field.
- `type`: A reference to one of the field types defined earlier in the schema.
- `default`: (optional) The default value, if an input document doesn't specify it. This is commonly used on schemas that record the time of indexing a document by specifying **NOW** on a date field.
- `required`: (optional) Set this to `true` if you want Solr to fail to index a document that does not have a value for this field.

The default precision of dates is to the millisecond. You can improve the date query performance and reduce the index size by rounding to a lesser precision such as **NOW/SECOND**. Date/time syntax is discussed later.

Solr comes with a predefined schema used by the sample data. Delete the field definitions as they are not applicable, but leave the field types at the top. Here's a first cut of our MusicBrainz schema definition. You can see the definition of the `name`, `type`, `indexed`, and `stored` attributes in a few pages under the *Field options* heading. Note that some of these types aren't in Solr's default type definitions, but we'll define them soon enough.

In the following code, notice that I chose to prefix the various document types (`a_`, `r_`, `l_`), because I'd rather not overload the use of any field across entity types (as explained previously). I also use this abbreviation when I'm inlining relationships like in `r_a_name` (a release's artist's name).

```
<!-- COMMON TO ALL TYPES: -->
<field name="id" type="string" required="true" />
       <!-- Artist:11650 -->
<field name="type" type="string" required="true" />
       <!-- Artist | Release | Label -->
<!-- ARTIST: -->
<field name="a_name" type="title" /><!-- The Smashing Pumpkins -->
```

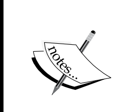
```
<field name="a_name_sort" type="string" stored="false" />
       <!-- Smashing Pumpkins, The -->
<field name="a_type" type="string" /><!-- group | person -->
<field name="a_begin_date" type="date" />
<field name="a_end_date" type="date" />
<field name="a_member_name" type="title" multiValued="true" />
       <!-- Billy Corgan -->
<field name="a_member_id" type="title" multiValued="true" />
       <!-- 102693 -->
<!-- RELEASE -->
<field name="r_name" type="title" /><!-- Siamese Dream -->
<field name="r_name_sort" type="title_sort" /><!-- Siamese Dream -->
<field name="r_a_name" type="title" /><!-- The Smashing Pumpkins -->
<field name="r_a_id" type="string" /><!-- 11650 -->
<field name="r_type" type="string" />
       <!-- Album | Single | EP |... etc. -->
<field name="r_status" type="string" />
       <!-- Official | Bootleg | Promotional -->
<field name="r_lang" type="string" indexed="false" /><!-- eng /
        latn -->
<field name="r_tracks" type="integer" indexed="false" />
<field name="r_event_country" type="string" multiValued="true" />
        <!-- us -->
<field name="r_event_date" type="date" multiValued="true" />
<!-- LABEL -->
<field name="l_name" type="title" /><!-- Virgin Records America -->
<field name="l_name_sort" type="string" stored="false" />
<field name="l_type" type="string" />
       <!-- Distributor, Orig. Prod., Production -->
<field name="l_begin_date" type="date" />
<field name="l_end_date" type="date" />
<!-- TRACK -->
<field name="t_name" type="title" /><!-- Cherub Rock -->
<field name="t_num" type="integer" indexed="false" /><!-- 1 -->
<field name="t_duration" type="integer" indexed="false"/>
       <!-- 298133 -->
<field name="t_a_name" type="title" /><!-- The Smashing Pumpkins -->
<field name="t_r_type" type="string" />
       <!-- album | single | compilation -->
<field name="t_r_name" type="title" /><!-- Siamese Dream -->
<field name="t_r_tracks" type="integer" indexed="false" /><!-- 13 -->
```

Put some sample data in your schema comments.

You'll find the sample data helpful and anyone else working on your project will thank you for it. In the examples above, I sometimes use actual values and on other occasions I list several possible values separated by |, if there is a predefined list.

Although it is not required, you should define a unique ID field. A unique ID allows specific documents to be updated or deleted, and it enables various other miscellaneous Solr features. If your source data does not have an ID field that you can propagate, Solr can generate one by simply having a field with a field type and with a class of `solr.UUIDField`. At a later point in the schema, we'll tell Solr which field is our unique field. In our schema, the ID includes the type so that it's unique across the whole index. Also, note that the only fields that we can mark as required are those common to all, which are ID and type, because we're doing a combined index approach. This isn't a big deal though.

One thing I want to point out is that in our schema we're choosing to index most of the fields, even though MusicBrainz's search doesn't require more than the name of each entity type. We're doing this so that we can make the schema more interesting to demonstrate more of Solr's capabilities. As it turns out, some of the other information in MusicBrainz's query results actually are queryable if one uses the advanced form, checks **use advanced query syntax**, and your query uses those fields (example: artist: "Smashing Pumpkins").



At the time of writing this, MusicBrainz used Lucene for its text search and so it uses Lucene's query syntax. `http://wiki.musicbrainz.org/TextSearchSyntax`. You'll learn more about the syntax in another chapter.

Sorting

Usually, search results are sorted by their score (how well the document matched the query), but it is common to need to support the sorting of supplied data too. It just happens that MusicBrainz already supplies alternative artist and label names for sorting, which is perhaps unusual, but it makes little difference to us. When different from the original name, these sortable versions move words like "The" from the beginning to the end after a comma. The MB search results actually displays this sort-specific field, which I think is very unusual. Hence, we're not going to do that (not that it really matters). Ironically, the search results page doesn't let you use it for sorting either (though I'm sure it's used elsewhere), but we're going to support that. Therefore, we've marked the sort names as not `stored` but indexed, instead of the other way around. Remember that `indexed` and `stored` are `true` by default.

Sorting limitations: A field needs to be indexed, not be multi-valued, and it should not have multiple tokens (either there is no text analysis or it yields just one token).

Because of the special text analysis restrictions of fields used for sorting, text fields in your schema that need to be sortable will usually be copied into another field and analyzed differently (more on text analysis is explained later). The copyField directive in the schema facilitates this task. For non-text fields, this tends not to be an issue, but pay attention to the predefined types in Solr's schema and choose appropriately. Some are explicitly for sorting purposes and are documented as such. The string type is a type that has no text analysis and so it's perfect for our MusicBrainz case. As we're getting a sort-specific value from MB, we don't need to derive something ourselves. However, note that in the MusicBrainz schema there are no sort-specific release names. We could opt to not support sorting by release name, but we're going to anyway. One option is to use the string type again. That's fine, but you may want to lowercase the text, remove punctuation, and collapse multiple spaces into one (if the data isn't clean). It's up to you. For the sake of variety, we'll be taking the latter route, and we're using a type title_sort that does these kinds of things, which is defined later.

By the way, Lucene sorts text by the internal Unicode code point. For most users, this is just fine. Internalization sensitive users may want a locale specific option. The latest development in this area is a patch to the latest Lucene in LUCENE-1435. It can easily be exposed for use by Solr, if the reader has the need and some Java programming experience.

Dynamic fields

The very notion of the feature about to be described, highlights the flexibility of Lucene's index, as compared to typical database technology. Not only can you explicitly name fields in the schema, but you can also have some defined on the fly based on the name used. Solr's sample schema.xml file contains some examples of this, such as:

```
<dynamicField name="*_dt" type="date" indexed="true"  stored="true"/>
```

If at index-time a document contains a field that isn't matched by an explicit field definition, but does have a name matching this pattern (that is, ends with _dt such as updated_dt), then it gets processed according to that definition. This also applies to searching the index. A dynamic field is declared just like a regular field in the same section. However, the element is named dynamicField, and it has a name attribute that must start or end with an asterisk (the wildcard). If the name is just *, then it is the final fallback.

Using dynamic fields is most useful for the * fallback if you decide that all fields attempted to be stored in the index should succeed, even if you didn't know about the field when you designed the schema. It's also useful if you decide that instead of it being an error, such unknown fields should simply be ignored (that is, not indexed and not stored).

Using copyField

Closely related to the field definitions are copyField directives, which are specified at some point after the fields element, not within it. A copyField directive looks like this:

```
<copyField source="r_name" dest="r_name_sort" />
```

These are really quite simple. At index-time, each copyField is evaluated for each input document. If there is a value for the field referenced by the source of this directive in the input document (r_name in this case), then it is copied to the destination field referenced (r_name_sort in this case). Perhaps appendField might have been a more suitable name, because the copied value(s) will be in addition to any existing values if present. If by any means a field contains more than one value, be sure to declare it multi-valued since you will get an error at index-time if you don't. Both fields must be defined, but they may be dynamic fields and so need not be defined explicitly. You can also use a wildcard in the source such as * to copy every field to another field. If there is a problem resolving a name, then Solr will display an error when it starts up.

This directive is useful when a value needs to be stored in additional field(s) to support different indexing purposes. Sorting is a common scenario since there are some constraints on the field to sort on it, as well as for faceting. Another is a common technique in indexing technologies in which many fields are copied to a common field that is indexed without norms and not stored. This permits searches, which would otherwise search many fields, to search one instead, thereby drastically improving performance at the expense of reducing score quality. This technique is usually complemented by searching some additional fields with higher boosts. The dismax request handler, which is described in a later chapter, makes this easy.

Finally, note that copying data to additional fields necessitates, that indexing time will be longer and the index's disk size will be greater. It is a consequence that is unavoidable.

Remaining schema.xml settings

Following the definition of the fields are some more configuration settings. As with the other parts of the file, you should leave the helpful comments in place. For the MusicBrainz schema, this is what remains:

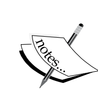
```
<uniqueKey>id</uniqueKey>
<!-- <defaultSearchField>text</defaultSearchField>
<solrQueryParser defaultOperator="AND"/> -->
<copyField source="r_name" dest="r_name_sort" />
```

The `uniqueKey` is straightforward and is analogous to a database primary key. This is optional, but it is likely that you have one. We have discussed the unique IDs earlier.

The `defaultSearchField` declares the particular field that will be searched for queries that don't explicitly reference one. And the `solrQueryParser` setting allows one to specify the default search operator here in the schema. These are essentially defaults for searches that are processed by Solr request handlers defined in `solrconfig.xml`. I recommend you explicitly configure these there, instead of relying on these defaults as they are search-related, especially the default search operator. These settings are optional here, and I've commented them out.

Text analysis

Text analysis is a topic that covers tokenization, case normalization, stemming, synonyms, and other miscellaneous text processing used to process raw input text for a field, both at index-time and query-time. This is an advanced topic, so you may want to stick with the existing analyzer configuration for the field types in Solr's default schema. However, there will surely come a time when you are trying to figure out why a simple query isn't matching a document that you think it should, and it will quite often come down to your text analysis configuration.



> This material is almost completely Lucene-centric and so also applies to any other software built on top of Lucene. For the most part, Solr merely offers XML configuration for the code in Lucene that provides this capability. For information beyond what is covered here, including writing your own analyzers, read the *Lucene In Action* book.

The purpose of text analysis is to convert text for a particular field into a sequence of terms. It is often thought of as an index-time activity, but that is not so. At index-time, these terms are indexed (that is, recorded onto a disk for subsequent querying) and at query-time, the analysis is performed on the input query and then the resulting terms are searched for. A term is the fundamental unit that Lucene actually stores and queries. If every user's query is always searched for the identical text that was put into Solr, then there would be no text analysis needed other than tokenizing on whitespace. But people don't always use the same capitalization, nor the same identical words, nor do documents use the same text among each other even if they are similar. Therefore, text analysis is essential.

Configuration

Solr has various field types as we've previously explained, and one such type (perhaps the most important one) is `solr.TextField`. This is the field type that has an analyzer configuration. Let's look at the configuration for the `text` field type definition that comes with Solr:

```
<fieldType name="text" class="solr.TextField"
           positionIncrementGap="100">
    <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <!-- in this example, we will only use synonyms at query time
        <filter class="solr.SynonymFilterFactory"
                synonyms="index_synonyms.txt" ignoreCase="true"
                          expand="false"/>
        -->
        <!-- Case insensitive stop word removal.
        enablePositionIncrements=true ensures that a 'gap' is left to
        allow for accurate phrase queries.
        -->
        <filter class="solr.StopFilterFactory" ignoreCase="true"
           words="stopwords.txt"  enablePositionIncrements="true" />
        <filter class="solr.WordDelimiterFilterFactory"
            generateWordParts="1" generateNumberParts="1"
            catenateWords="1" catenateNumbers="1" catenateAll="0"
            splitOnCaseChange="1"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.EnglishPorterFilterFactory"
                protected="protwords.txt"/>
        <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
```

```
        <filter class="solr.SynonymFilterFactory"
            synonyms="synonyms.txt" ignoreCase="true" expand="true"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true"
                words="stopwords.txt"/>
        <filter class="solr.WordDelimiterFilterFactory"
                generateWordParts="1" generateNumberParts="1"
                catenateWords="0" catenateNumbers="0" catenateAll="0"
                splitOnCaseChange="1"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.EnglishPorterFilterFactory"
                protected="protwords.txt"/>
        <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
    </analyzer>
</fieldType>
```

There are two **analyzer chains**, each of which specifies an ordered sequence of processing steps that convert the original text into a sequence of terms. One is of the index type, while the other is query type. As you might guess, this means the contents of the index chain apply to index-time processing, whereas the query chain applies to query-time processing. Note that the distinction is optional and so you can opt to specify just one analyzer element that has no type, and it will apply to both. When both are specified (as in the example above), they usually only differ a little.

Analyzers, Tokenizers, Filters, oh my!

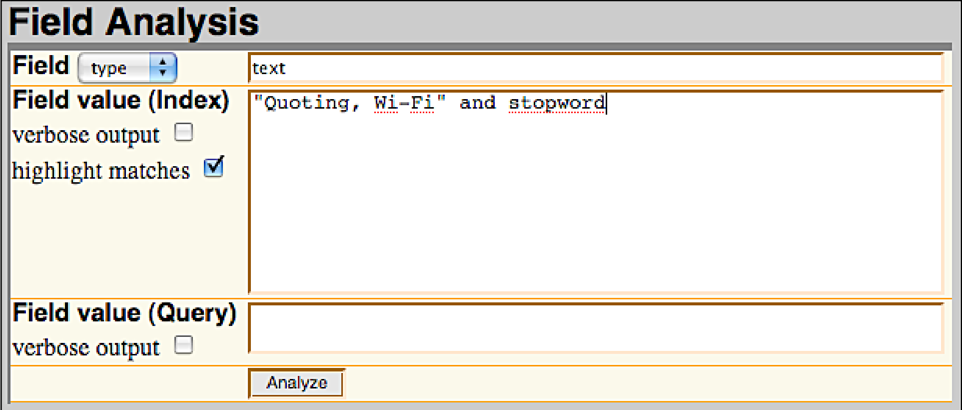
The various components involved in text analysis go by various names, even across Lucene and Solr. In some cases, their names are not intuitive. Whatever they go by, they are all conceptually the same. They take in text and spit out text, sometimes filtering, sometimes adding new terms, sometimes modifying terms. I refer to the lot of them as **analyzers**. Also, **term**, **token**, and **word** are often used interchangeably.

An analyzer chain can optionally begin with a `CharFilterFactory`, which is not really an analyzer but something that operates at a character level to perform manipulations. It was introduced in Solr 1.4 to perform tasks such as normalizing characters like removing accents. For more information about this new feature, search Solr's Wiki for it, and look for the example of it that comes with Solr's sample schema.

The first analyzer in a chain is always a **tokenizer**, which is a special type of analyzer that tokenizes the original text, usually with a simple algorithm such as splitting on whitespace. After this tokenizer is configured, the remaining analyzers are configured with the `filter` element in sequence.(*These analyzers don't necessarily filter* – it was a poor name choice). What's important to note about the configuration is that an analyzer is either a tokenizer or a filter, not both. Moreover, the analysis chain must have only one tokenizer, and it always comes first. There are a handful of tokenizers available, and the rest are filters. Some filters actually perform a tokenization action such as `WordDelimeterFilterFactory`. However, you are not limited to do all tokenization at the first step.

Experimenting with text analysis

Before we dive into the details of particular analyzers, it's important to become comfortable with Solr's analysis page, which is an experimentation and a troubleshooting tool that is absolutely indispensable. You'll use this to try out different analyzers to verify whether you get the desired effect, and you'll use this when troubleshooting to find out why certain queries aren't matching certain text you think they should. In Solr's admin pages, you'll see a link at the top that looks like this:**[ANALYSIS].**



The first choice at the top of the page is required. You pick whether you want to choose a field type based on the name of one, or if you want to indirectly choose it based on the name of a field. Either way you get the same result, and it's a matter of convenience. In this example, I'm choosing the **text** field type that has some interesting text analysis. This tool is mainly for the text oriented field types, not boolean, date, and numeric oriented types. You may get strange results if you try those.

At this point you can analyze index and/or query text at the same time. Remember that there is a distinction for some field types. You activate that analysis by putting some text into the text box, otherwise it won't do that phase. If you are troubleshooting why a particular query isn't matching a particular document's field value, then you'd put the field value into the **Index** box and the query text into the **Query** box. Technically that might not be the same thing as the original query string, because the query string may use various operators to target specified fields, do fuzzy queries, and so on. You will want to check off **verbose output** to take full advantage of this tool. However, if you only care about which terms are emitted at the end, you can skip it. The **highlight matches** is applicable when you are doing both query and index analysis together and want to see matches in the index part of the analysis.

The output after clicking **Analyze** on the **Field Analysis** is a bit verbose so I'm not repeating it here verbatim. I encourage you to try it yourself. The output will show one of the following grids after the analyzer is done:

org.apache.solr.analysis.WordDelimiterFilterFactory {catenateWords=1, catenateNumbers=1, splitOnCaseChange=1, catenateAll=0, generateNumberParts=1, generateWordParts=1}

term position	1	2	3	5
term text	Quoting	Wi	Fi WiFi	stopword
term type	word	word	word word	word
source start,end	1,8	10,12	13,15 10,15	22,30
payload				

The most important row and that which is least technical to understand is the second row, which is **term text**. If you recall, terms are the atomic units that are actually stored and queried. Therefore, a matching query's analysis must result in a term in common with that of the index phase of analysis. Notice that at position **3** there are two terms. Multiple terms at the same position can occur due to synonym expansion and in this case due to alternate tokenizations introduced by `WordDelimeterFilterFactory`. This has implications with phrase queries. Other things to notice about the analysis results (not visible in this screenshot) is that **Quoting** ultimately became **quot** after stemming and lowercasing. **and** was omitted by the **StopFilter**. Keep reading to learn more about specific text analysis steps such as stemming and synonyms.

Tokenization

A tokenizer is an analyzer that takes text and splits it into smaller pieces of the original whole, most of the time skipping insignificant bits like whitespace. This must be performed as the first analysis step and not done thereafter. Your tokenizer choices are as follows:

- `WhitespaceTokenizerFactory`: Text is tokenized by whitespace (that is, spaces, tabs, carriage returns). This is usually the most appropriate tokenizer and so I'm listing it first.
- `KeywordTokenizerFactory`: This doesn't actually do any tokenization or anything at all for that matter! It returns the original text as one term. There are cases where you have a field that always gets one word, but you need to do some basic analysis like lowercasing. However, it is more likely that due to sorting or faceting requirements you will require an indexed field with no more than one term. Certainly a document's identifier field, if supplied and not a number, would use this.
- `StandardTokenizerFactory`: This analyzer works very well in practice. It tokenizes on whitespace, as well as at additional points. Excerpted from the documentation:
    - Splits words at punctuation characters, removing punctuations. However, a dot that's not followed by whitespace is considered part of a token.
    - Splits words at hyphens, unless there's a number in the token. In that case, the whole token is interpreted as a product number and is not split.
    - Recognizes email addresses and Internet hostnames as one token.
- `LetterTokenizerFactory`: This tokenizer emits each contiguous sequence of letters (only A-Z) and omits the rest.
- `HTMLStripWhitespaceTokenizerFactory`: This is used for HTML or XML that need not be well formed. Essentially it omits all tags altogether, except the contents of tags, skipping script, and style tags. Entity references (example: `&`) are resolved. After this processing, the output is internally processed with `WhitespaceTokenizerFactory`.
- `HTMLStripStandardTokenizerFactory`: Like the previous tokenizer, except the output is subsequently processed by `StandardTokenizerFactory` instead of just whitespace.

- `PatternTokenizerFactory`: This one can behave in one of two ways:
    - To split the text on some separator, you can use it like this: `<tokenizer class="solr.PatternTokenizerFactory" pattern=";*" />`. Pattern is a regular expression. This example would be good for a semi-colon separated list.
    - To match only particular patterns and possibly use only a subset of the pattern as the token. Example: `<tokenizer class="solr.PatternTokenizerFactory" pattern="\'([^\']+)\'" group="1" />`. If you had input text like `'aaa' 'bbb' 'ccc'`, then this would result in tokens `bbb` and `ccc`.

The regular expression specification supported by Solr is the one that Java uses. It's handy to have this reference bookmarked: http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html

WorkDelimiterFilterFactory

I have mentioned earlier that tokenization only happens as the first analysis step. That is true for those tokenizers listed above, but there is a very useful and configurable Solr `filter` that is essentially a tokenizer too:

```
<filter class="solr.WordDelimiterFilterFactory"
        generateWordParts="1" generateNumberParts="1"
        catenateWords="1" catenateNumbers="1"
        catenateAll="0" splitOnCaseChange="1"/>
```

The purpose of this analyzer is to both split and join compound words with various means of defining compound words. This one is often used with a basic tokenizer, not a `StandardTokenizer`, which removes the intra-word delimiters, thereby defeating some of this processing. The options to this analyzer have the values `1` to enable and `0` to disable.

The `WordDelimiter` analyzer will tokenize (aka split) in the following ways:

- split on intra-word delimiters: `Wi-Fi` to `Wi, Fi`
- split on letter-number transitions: `SD500` to `SD, 500`
- omit any delimiters: `/hello--there, dude` to `hello, there, dude`
- if `splitOnCaseChange`, then it will split on lower to upper case transitions: `WiFi` to `Wi, Fi`

The splitting results in a sequence of terms, wherein each term consists of only letters or numbers. At this point, the resulting terms are filtered out and/or catenated (that is combined):

- To filter out individual terms, disable `generateWordParts` for the alphabetic ones or `generateNumberParts` for the numeric ones. Due to the possibility of catenation, the actual text might still appear in spite of this filter.
- To concatenate a consecutive series of alphabetic terms, enable `catenateWords` (example: `wi-fi` to `wifi`). If the `generateWordParts` is enabled, then this example would also generate `wi` and `fi` but not otherwise. This will work even if there is just one term in the series, thereby emitting a term that disabling `generateWordParts` would have omitted. `catenateNumbers` works similarly but for numeric terms. `catenateAll` will concatenate all of the terms together. The concatenation process will take care to not emit duplicate terms.

Here is an example exercising all options:

`WiFi-802.11b` to `Wi, Fi, WiFi, 802, 11, 80211, b, WiFi80211b`

Solr's out-of-the-box configuration for the `text` field type is a reasonable way to use the `WordDelimiter` analyzer: generation of word and number parts at both index and query-time, but concatenating only at index-time (query-time would be redundant).

Stemming

Stemming is the process for reducing inflected (or sometimes derived) words to their stem, base, or root form. For example, a stemming algorithm might reduce riding and rides, to just ride. Most stemmers in use today exist thanks to the work of Dr. Martin Porter. There are a few implementations to choose from:

- `EnglishPorterFilterFactory`: This is an English language stemmer using the Porter2 (aka Snowball English) algorithm. Use this if you are targeting the English language.
- `SnowballPorterFilterFactory`: If you are not targeting English or if you wish to experiment, then use this stemmer. It has a `language` attribute in which you make an implementation choice. Remember the initial caps, and don't include my parenthetical remarks: Danish, Dutch, Kp (a Dutch variant), English, Lovins (an English alternative), Finnish, French, German, German2, Italian, Norwegian, Portuguese, Russian, Spanish, or Swedish.
- `PorterStemFilterFactory`: This is the original Porter algorithm. It is for the English language.

- `KStem`: An alternative to the Porter's English stemmer that is less aggressive. This means that it will not stem in as many cases as Porter will in an effort to reduce false-positives at the expense of missing stemming opportunities.

You have to download and build `KStem` yourself due to licensing issues. See `http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters/Kstem`

Example:

```
<filter class="solr.EnglishPorterFilterFactory"
        protected="protwords.txt"/>
```

Algorithmic stemmers such as these are fundamentally imperfect and will stem in ways you do not want on occasion. But on the whole, they help. If there are particularly troublesome exceptions, then you can specify those words in a file and reference them as in the example (the `protected` attribute references a file in the `conf` directory). If you are processing general text, then you will most likely improve search results with stemming. However, if you have text that is mostly proper nouns (such as an artist's name in MusicBrainz) then stemming will only hurt the results.

Remember to apply your stemmer at both index-time and query-time or else few stemmed words will match the query. Unlike Synonym processing, the stemmers in Lucene do not have the option of **expansion**.

Synonyms

The purpose of synonym processing is straightforward. Someone searches using a word that wasn't in the original document but is synonymous with a word that is indexed, so you want that document to match the query. Of course, the synonym need not be strictly those identified by a Thesaurus, and they can be whatever you want including terminology specific to your application's domain.

The most widely known free Thesaurus is WordNet: `http://wordnet.princeton.edu/`. There isn't any Solr integration with that data set yet. However, there is some simple code in the Lucene sandbox for parsing WordNet's `prolog` formatted file into a Lucene index. A possible approach would be to modify that code to instead output the data into a text file formatted in a manner about to be described—a simple task. Then, Solr's `SynonymFilterFactory` can make use of it.

Here is a sample analyzer configuration line for synonym processing:

```
<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
        ignoreCase="true" expand="true"/>
```

The synonyms reference is to a file in the `conf` directory. Set `ignoreCase` to `true` if the case of the terms in `synonyms.txt` might not be identical, however, they should match anyway. Before describing the `expand` option, let's consider an example.
The synonyms file is processed line-by-line. Here is a sample line with an **explicit mapping** that uses the arrow `=>`:

```
i-pod, i pod => ipod
```

This means that if either `i-pod` or `ipod` (source terms) are found, then they are replaced with `ipod` (a replacement term). There could have been multiple replacement terms but not in this example. Also notice that commas are what separates each term.

Alternatively you may have lines that look like this:

```
ipod, i-pod, i pod
```

These lines don't have a `=>` and are interpreted differently according to the `expand` parameter. If `expand` is `true`, then it is translated to this explicit mapping:

```
ipod, i-pod, i pod => ipod, i-pod, i pod
```

If `expand` is `false` then it becomes this explicit mapping:

```
ipod, i-pod, i pod => ipod
```

By the way, it's okay to have multiple lines that reference the same terms. If a source term in a new rule is already found to have replacement terms from another rule, then those replacements are merged.

Multi-word (aka Phrase) synonyms

For multi-word synonyms to work, the analysis must be applied at index-time and with expansion so that both the original words and the combined word get indexed. The next section elaborates on why this is so. Also, be aware that the tokenizer and previous filters can affect the terms that the `SynonymFilter` sees. Thus depending on the configuration, hyphens and other punctuations may or may not be stripped out.

Index-time versus Query-time, and to expand or not

If you are doing synonym expansion (have any source terms that map to multiple replacement terms), then do synonym processing at either index-time or query-time, but not both, as that would be redundant. For a variety of reasons, it is usually better to do this at index-time:

- A synonym containing multiple words (example: i pod) isn't recognized correctly at query-time because the query parser tokenizes on whitespace.
- The IDF component of Lucene's scoring algorithm will be much higher for documents matching a synonym appearing rarely, as compared to its equivalents that are common. This reduces the scoring effectiveness.
- Prefix, wildcard, and fuzzy queries aren't analyzed, and thus won't match synonyms.

However, any analysis at index-time is less flexible, because any changes to the synonyms will require a complete re-index to take effect. Moreover, the index will get larger if you do index-time expansion. It's plausible to imagine the issues above being rectified at some point. However, until then, index-time is usually best.

Alternatively, you could choose not to do synonym expansion. This means that for a given synonym term, there is just one term that should replace it. This requires processing at both index-time and query-time to effectively normalize the synonymous terms. However, since there is query-time processing, it suffers from the problems mentioned above with the exception of poor scores, which isn't applicable. The benefit to this approach is that the index size would be smaller, because the number of indexed terms is reduced.

You might also choose a blended approach to meet different goals. For example, if you have a huge index that you don't want to re-index often but you need to respond rapidly to new synonyms, then you can put new synonyms into both a query-time synonym file and an index-time one. When a re-index finishes, you empty the query-time synonym file. You might also be fond of the query-time benefits, but due to the multiple word term issue, you decide to handle those particular synonyms at index-time.

Stop words

There is a simple filter called `StopFilterFactory` that filters out certain so-called **stop words** specified in a file in the `conf` directory, optionally ignoring case. Example usage:

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

This is usually incorporated into both index and query analyzer chains.

For indexes with lots of text, common uninteresting words like "the", "a", and so on, make the index large and slow down phrase queries. To deal with this problem, it is best to remove them from fields where they show up often. Fields likely to contain more than a sentence are ideal candidates. Our MusicBrainz schema does not have content like this. The trade-off when omitting stop words from the index is that those words are no longer query-able. This is usually fine, but in some circumstances like searching for **To be or not to be**, it is obviously a problem. Chapter 9 discusses a technique called **shingling** that can be used to improve phrase search performance, while keeping these words.

Solr comes with a decent set of stop words for the English language. You may want to supplement it or use a different list altogether if you're indexing non-English text. In order to determine which words appear commonly in your index, access the **SCHEMA BROWSER** menu option in Solr's admin interface. A list of your fields will appear on the left. In case the list does not appear at once, be patient. For large indexes, there is a considerable delay before the field list appears because Solr is analyzing the data in your index. Now, choose a field that you know contains a lot of text. In the main viewing area, you'll see a variety of statistics about the field including the top-10 terms appearing most frequently.

Phonetic sounds-like analysis

Another useful text analysis option to enable searches that sound like a queried word is phonetic translation. A filter is used at both index and query-time that phonetically encodes each word into a **phoneme**. There are four phonetic encoding algorithms to choose from: `DoubleMetaphone`, `Metaphone`, `RefinedSoundex`, and `Soundex`. Anecdotally, `DoubleMetaphone` appears to be the best, even for non-English text. However, you might want to experiment in order to make your own choice. `RefinedSoundex` declares itself to be most suitable for spellcheck applications. However, Solr can't presently use phonetic analysis in its spellcheck component (described in a later chapter).

Solr has three tools at its disposal for more aggressive in-exact searching: phonetic sounds-like, query spellchecking, and fuzzy searching. These are all employed a bit differently.

The following is a suggested configuration for phonetic analysis in the `schema.xml`:

```
<!-- for phonetic (sounds-like) indexing -->
<fieldType name="phonetic" class="solr.TextField"
      positionIncrementGap="100" stored="false" multiValued="true">
    <analyzer>
```

```
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.WordDelimiterFilterFactory"
             generateWordParts="1" generateNumberParts="0"
             catenateWords="1" catenateNumbers="0" catenateAll="0"/>
        <filter class="solr.DoubleMetaphoneFilterFactory"
             inject="false" maxCodeLength="8"/>
    </analyzer>
</fieldType>
```

Note that the encoder options internally handle both upper and lower case.

In the MusicBrainz schema that is supplied with the book, a field named a_phonetic is declared to use this field type, and it has the artist name copied into it through a copyField directive. In a later chapter, you will read about the dismax search handler that can conveniently search across multiple fields with different scoring boosts. Such a handler might be configured to search not only the artist name (a_name) field, but also a_phonetic with a low boost, so that regular exact matches will come above those that match phonetically.

Using Solr's analysis admin page, it can be shown that this field type encodes Smashing Pumpkins as SMXNK|XMXNK PMPKNS. The use of a vertical bar | here indicates both sides are alternatives for the same position. This is not supposed to be meaningful, but it is useful for comparing similar spellings to detect its effectiveness.

The example above used the DoubleMetaphoneFilterFactory analysis filter, which has these two options:

- inject: A boolean defaulting to true that will cause the original words to pass through the filter. It might interfere with other filter options, querying, and potentially scoring. Therefore, it is preferred to disable this, and use a separate field dedicated to phonetic indexing.
- maxCodeLength: The maximum phoneme code (that is Phonetic character, or syllable) length. It defaults to 4. Longer code are truncated. Only DoubleMetaphone supports this option.

In order to use one of the other three phonetic encoding algorithms, you must use this filter:

```
<filter class="solr.PhoneticFilterFactory" encoder="RefinedSoundex"
        inject="false"/>
```

The encoder attribute must be one of those algorithms listed in the first paragraph of this section.

Partial/Substring indexing

Usually, text indexing technology is employed to search entire words. Occasionally however, there arises a need for a search to match an arbitrary substring of a word or across them. Lucene supports leading and trailing wildcards (example: `*`) on queries. However, only the latter is supported by Solr without internal modification. Moreover, this approach only scales for very small indices before it gets very slow and/or results in an error. The right way to solve this is to venture into the black art of **n-grams**.

> Before employing this approach, consider if what you really need is better tokenization for special code. For example, if you have a long string code that internally has different parts that users might search on separately, then you can use a `PatternReplaceFilterFactory` with some other analyzers to split them up.

N-gram analysis slices text into many smaller substrings ranging between a minimum and maximum configured size. For example, consider the word **Tonight**. An `NGramFilterFactory` configured with `minGramSize` of 2 and `maxGramSize` of 5 would yield all of the following indexed terms: (2-grams:) To, on, ni, ig, gh, ht, (3-grams:) Ton, oni, nig, igh, ght, (4-grams:) Toni, onig, nigh, ight, (5-grams:) Tonig, onigh, night. Note that **Tonight** fully does not pass through because it has more characters than the `maxGramSize`. N-gram analysis can be used as a filter for processing on a term-by-term basis, and it can also be used as a tokenizer with `NGramTokenizerFactory`, which will emit n-grams spanning across the words of the entire source text.

The following is a suggested analyzer configuration using n-grams to match substrings:

```
<fieldType name="nGram" class="solr.TextField"
        positionIncrementGap="100" stored="false" multiValued="true">
    <analyzer type="index">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <!-- potentially word delimiter, synonym filter, stop words,
             NOT stemming -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.NGramFilterFactory" minGramSize="2"
                      maxGramSize="15"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <!-- potentially word delimiter, synonym filter, stop words,
             NOT stemming -->
        <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
</fieldType>
```

Notice that the n-gramming only happens at index-time. The range of gram sizes goes from the smallest number of characters you wish to enable substring searches on (2 in this example), to the maximum size permitted for substring searches (15 in this example).

This analysis would be applied to a field created solely for the purpose of matching substrings. Another field would exist for typical searches, and a dismax handler (described in a later chapter) would be configured for searches to use both fields using a smaller boost for this field.

Another variation is `EdgeNGramTokenizerFactory` and `EdgeNGramFilterFactory`, which emit n-grams that are adjacent to either the start or end of the input text. For the filter-factory, this input-text is a term, and the tokenizer is the entire input. In addition to `minGramSize` and `maxGramSize`, these analyzers take a `side` argument that is either `front` or `back`. If only prefix or suffix matching is needed instead of both, then an `EdgeNGram` analyzer is for you.

N-gramming costs

There is a high price to be paid for n-gramming. Recall that in the earlier example, **Tonight** was split into 15 substring terms, whereas typical analysis would probably leave only one. This translates to greater index sizes, and thus a longer time to index. Let's look at the effects of this in the MusicBrainz schema. The `a_name` field, which contains the artist name, is indexed in a typical fashion and is `stored`. The `a_ngram` field is fed by the artist name and is indexed with n-grams ranging from 2 to 15 characters in length. It is not a `stored` field because the artist's name is already stored in `a_name`.

	`a_name`	**`a_name + a_ngram`**	**Increase**
Indexing Time	46 seconds	479 seconds	> 10x
Disk Size	11.7 MB	59.7 MB	> 5x
Distinct Terms	203,431	1,288,720	> 6x

The table above shows a comparison of index statistics of an index with just `a_name` versus both `a_name` and `a_ngram`. Note the ten-fold increase in indexing time for the artist name, and a five-fold increase in disk space. Remember that this is just one field!

Given these costs, n-gramming, if used at all, is generally only done on a field or two of small size where there is a clear requirement for substring matches.

The costs of n-gramming are lower if `minGramSize` is raised and to a lesser extent if `maxGramSize` is lowered. Edge n-gramming costs less too. This is because it is only based on one side. It definitely costs more to use the tokenizer-based n-grammers instead of the term-based filters used in the example before, because terms are generated that include and span whitespace. However, with such indexing, it is possible to match a substring spanning words.

Miscellaneous analyzers

There are some Solr filters that have not been mentioned yet that we can go over quickly:

- `StandardFilterFactory`: Works in conjunction with `StandardTokenizer`. It will remove periods inbetween acronyms and `s` at the end of terms:

  ```
  "I.B.M. cat's" => "IBM", "cat"
  ```

- `LowerCaseFilterFactory`: Simply lowercases all text. Don't put this before `WordDelimeterFilterFactory` if you want to split on case transitions.
- `KeepWordFilterFactory`: Omits all of the words, except those in the specified file:

  ```
  <filter class="solr.KeepWordFilterFactory" words="keepwords.txt"
            ignoreCase="true"/>
  ```

  If you want to ensure a certain vocabulary of words in a special field, then you might enforce it with this.
- `LengthFilterFactory`: Filters out the terms that do not have a length within an inclusive range.

  ```
  <filter class="solr.LengthFilterFactory" min="2" max="5" />
  ```

- `RemoveDuplicatesTokenFilterFactory`: Ensures that no duplicate terms appear at the same position. This can happen, for example, when synonyms stem to a common root. It's a good idea to add this to your last analysis step, if you are doing a fair amount of other analysis.
- `ISOLatin1AccentFilterFactory`: This will normalize accented characters such as `é` to the unaccented equivalent `e`. An alternative and more customizable mechanism introduced in Solr 1.4 is a `CharFilterFactory`, which is something that actually comes before the lead tokenizer in the analysis chain. For more information about this approach, search Solr's Wiki for `MappingCharFilterFactory`.

- `CapitalizationFilterFactory`: This capitalizes each word according to the rules that you specify. For more information, see the Javadocs at `http://lucene.apache.org/solr/api/org/apache/solr/analysis/CapitalizationFilterFactory.html`.
- `PatternReplaceFilterFactory`: Takes a regular expression and replaces the matches. Example:

  ```
  <filter class="solr.PatternReplaceFilterFactory" pattern=".*@(.*)"
          replacement="$1" replace="first" />
  ```

  This replacement happens to be a reference to a `regexp` group, but it might be any old string. The replace attribute is either `first` to only apply to the first occurrence, or `all`. This example is for processing an email address field to get only the domain of the address.
- **Write your own**: Writing your own filter is of course an option if the existing ones don't suffice. Crack open the source code to Solr for one of these to get a handle on what's involved. Before you head down this path though, you'd be surprised at what a little creativity with `PatternReplaceFilterFactory` and some of the others can offer you. For starters, check out the `rType` field type in the `schema.xml` that is supplied online with this book.

There are some other miscellaneous Solr filters not mentioned, and a few stemmers are not of the Snowball variety. See the known implementing classes listed at the top of this URL: `http://lucene.apache.org/solr/api/org/apache/solr/analysis/TokenFilterFactory.html`

Summary

At this point, you should have a schema that you believe will suit your needs—for now anyway. But do expect to revisit the schema. It is quite normal to start with something workable, and then subsequently make modifications to address issues, and implement features that require changes. The only irritant with changing the schema is that you probably need to re-index all of the data. The only exception to this would be an analysis step applied only at query-time. In the next chapter, you'll learn about the various ways to import data into the index.

3
Indexing Data

With a first cut of the schema defined, it's time to get data into the index. In this chapter, we're going to review the four main mechanisms that Solr offers:

- Solr's native XML
- CSV (Character Separated Value)
- Direct Database and XML Import through Solr's `DataImportHandler`
- Rich documents through Solr Cell

You will also find some options in Chapter 8 that have to do with language bindings and framework integration. All of them generally use Solr's native XML format, which we'll get to right away.

Communicating with Solr

There are a few dimensions to the options available for communicating with Solr:

Direct HTTP or a convenient client API

Applications interact with Solr over HTTP. This can either be done directly (by hand, but by using an HTTP client of your choice), or it might be facilitated by a Solr integration API such as SolrJ or Solr Flare, which in turn use HTTP. Such APIs are discussed in Chapter 8.

An exception to HTTP is offered by SolrJ, which can optionally be used in an embedded fashion with Solr (so-called Embedded Solr) to avoid network and interprocess communication altogether. However, unless you are sure you really want to embed Solr within another application, this option is discouraged in favor of writing a custom Solr updating request handler. More information about SolrJ and EmbeddedSolr is in Chapter 8.

Data streamed remotely or from Solr's filesystem

Even though an application will be communicating with Solr over HTTP, it does not have to send Solr data over this channel. Solr supports what it calls remote streaming. Instead of giving Solr the data directly, it is given a URL that it will resolve. It might be an HTTP URL, but more likely it is a filesystem based URL, applicable when the data is already on Solr's machine. Finally, in the case of Solr's `DataImportHandler`, the data can be fetched from a database.

Data formats

The following are the different data formats:

- **Solr-XML**: Solr has a specific XML schema it uses to specify documents and their fields. It supports instructions to delete documents and to perform optimizes and commits too.
- **Solr-binary**: Analogous to Solr-XML, it is an efficient binary representation of the same structure. This is only supported by the SolrJ client API.
- **CSV**: CSV is a character separated value format (often a comma).
- Rich documents like PDF, XLS, DOC, PPT to Solr: The text data extracted from these formats is directed to a particular field in your Solr schema.
- Finally, Solr's DIH `DataImportHandler` contrib add-on is a powerful capability that can communicate with both databases and XML sources (for example: web services). It supports configurable relational and schema mapping options and supports custom transformation additions if needed. The DIH uniquely supports delta updates if the source data has modification dates.

We'll use the XML, CSV, and DIH options in bringing the MusicBrainz data into Solr from its database to demonstrate Solr's capability. Most likely, an application would use just one format.

Before these approaches are described, we'll discuss **curl** and **remote streaming**, which are foundational topics.

Using curl to interact with Solr

Solr receives commands (and possibly the associated data) through HTTP POST.

Solr lets you use HTTP GET too (for example, through your web browser). However, this is an inappropriate HTTP verb if it causes something to change on the server, as happens with indexing. For more information on this concept, read about REST at `http://en.wikipedia.org/wiki/Representational_State_Transfer`

One way to send an HTTP POST is through the Unix command line program `curl` (also available on Windows through Cygwin). Even if you don't use `curl`, it is very important to know how we're going to use it, because the concepts will be applied no matter how you make the HTTP messages.

There are several ways to tell Solr to index data, and all of them are through HTTP POST:

- Send the data as the entire POST payload (only applicable to Solr's XML format). `curl` does this with `data-binary` (or some similar options) and an appropriate content-type header reflecting that it's XML.
- Send some name-value pairs akin to an HTML form submission. With `curl`, such pairs are proceeded by `-F`. If you're giving data to Solr to be indexed (as opposed to it looking for it in a database), then there are a few ways to do that:
    - Put the data into the `stream.body` parameter. If it's small, perhaps less than a megabyte, then this approach is fine. The limit is configured with the `multipartUploadLimitInKB` setting in `solrconfig.xml`.
    - Refer to the data through either a local file on the Solr server using the `stream.file` parameter or a URL that Solr will fetch it from through the `stream.url` parameter. These choices are a feature that Solr calls remote streaming.

Here is an example of the first choice. Let's say we have an XML file named `artists.xml` in the current directory. We can post it to Solr using the following command line:

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

If it succeeds, then you'll have output that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
    <int name="status">0</int><int name="QTime">128</int>
</lst>
</response>
```

To use the `solr.body` feature for the example above, you would do this:

```
curl http://localhost:8983/solr/update -F solr.body=@artists.xml
```

In both cases, the `@` character instructs `curl` to get the data from the file instead of being `@artists.xml` literally. If the XML is short, then you can just as easily specify it literally on the command line:

```
curl http://localhost:8983/solr/update -F stream.body=' <commit />'
```

Notice the leading space in the value. This was intentional. In this example, `curl` treats `@` and `<` to mean things we don't want. In this case, it might be more appropriate to use `form-string` instead of `-F`. However, it's more typing, and I'm feeling lazy.

Remote streaming

In the examples above, we've given Solr the data to index in the HTTP message. Alternatively, the POST request can give Solr a pointer to the data in the form of either a file path accessible to Solr or an HTTP URL to it.

The file path is accessed by the Solr server on its machine, not the client, and it must also have the necessary operating system file permissions too.

However, just as before, the originating request does not return a response until Solr has finished processing it. If you're sending a large CSV file, then it is practical to use remote streaming. Otherwise, if the file is of a decent size or is already at some known URL, then you may find remote streaming faster and/or more convenient, depending on your situation.

Here is an example of Solr accessing a local file:

```
curl http://localhost:8983/solr/update -F stream.file=/tmp/artists.xml
```

To use a URL, the parameter would change to `stream.url`, and we'd specify a URL. We're passing a name-value parameter (`stream.file` and the path), not the actual data.

Remote streaming must be enabled

In order to use remote streaming (`stream.file` or `stream.url`), you must enable it in `solrconfig.xml`. It is disabled by default and is configured on a line that looks like this:

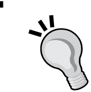
```
<requestParsers enableRemoteStreaming="true"
    multipartUploadLimitInKB="2048" />
```

Sending XML to Solr

Solr's native XML syntax is very simple. You can tell Solr to add documents to an index, to commit changes, to optimize the index, and to delete documents. Here is a sample XML file you can HTTP POST to Solr:

```
<add allowDups="false">
<doc boost="2.0">
<field name="id">5432a</field>
<field name="type" ...</field>
<field name="a_name" boost="0.5"></field>
<!-- the date/time syntax MUST look just like this (ISO-8601)-->
<field name="begin_date">2007-12-31T09:40:00Z</field>
</doc>
<doc>
<doc>
<field name="id">5432a</field>
<field name="type" ...
<field name="begin_date">2007-12-31T09:40:00Z</field>
</doc>
<!-- more here as needed -->
</add>
```

The `allowDups` defaults to `false` to guarantee the uniqueness of values in the field that you have designated as the unique field in the schema (assuming you have such a field). If you were to add another document that has the same value for the unique field, then this document would override the previous document, whether it is pending a commit or it's already committed. You will not get an error.



If you are sure that you will be adding a document that is not a duplicate, then you can set `allowDups` to `true` to get a performance improvement.

Boosting affects the scores of matching documents in order to affect ranking in score-sorted search results. Providing a boost value, whether at the document or field level, is optional. The default value is `1.0`, which is effectively a non-boost. Technically, documents are not boosted, only fields are. The effective boost value of a field is that specified for the document multiplied by that specified for the field.

> Specifying boosts here is called *index-time boosting*, which is rarely done as compared to the more flexible *query-time boosting*. Index-time boosting is less flexible because such boosting decisions must be decided at index-time and will apply to all of the queries.

Deleting documents

You can delete a document by its unique field (we delete two documents here):

```
<delete><id>artist:11604</id><id>artist:11603</id></delete>
```

Or, you can delete all of the documents that match a particular Lucene/Solr query (the query syntax is not discussed in this chapter):

```
<delete><query>timestamp:[* TO NOW-12HOUR]</query></delete>
```

The contents of the delete tag can be any number of ID and query tags if you want to batch many deletions into one message to Solr.

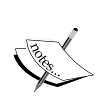
The query syntax is not discussed in this chapter, but I'll explain this somewhat complicated query anyway. Let's suppose that all of your documents had a timestamp field with a value of the time it was indexed, and you have an update strategy that bulk loads all of the data on a daily basis. If the loading process results in documents that shouldn't be in the index anymore, then we can delete them immediately after a bulk load. This query would delete all of the documents not indexed within the last 12 hours. Twelve was chosen somewhat arbitrarily, but it needs to be less than 24 (the update process interval) and greater than the longest time it might conceivably take to bulk load all the data.

Commit, optimize, and rollback

Data sent to Solr is not immediately searchable, nor do deletions take immediate effect. Like a database, changes must be committed first. Unlike a database, there are no distinct sessions (that is transactions) between each client, and instead there is in-effect one global modification state. This means that if more than one Solr client were to submit modifications and commit them at similar times, it is possible for part of one client's set of changes to be committed before that client told Solr to commit. Usually, you will have just one process responsible for updating Solr. But if not, then keep this in mind.

To commit changes using the XML syntax, simply send this to Solr:

```
<commit />
```



Depending on Solr's auto-warming configuration and cache state prior to committing (a Chapter 9 topic), a commit can take a non-trivial amount of time, in the order of seconds, perhaps up to a minute or longer in extreme cases. The amount of data committed has little impact on this delay. Generally, databases commit very fast. The last chapter deals with performance.

All uncommitted changes can be withdrawn by sending Solr the `rollback` command:

```
<rollback />
```

Lucene's index is internally composed of one or more segments. Modifications get committed to the last segment. Lucene will on occasions either start a new segment or merge them all together into one. When Lucene has just one segment, it is in an **optimized** state, because each segment degrades query performance. It is recommended to explicitly optimize the index at an opportune time like after a bulk load of data and/or a daily interval in off-peak hours, if there are sporadic updates to the index. You can do this by simply sending this XML:

```
<optimize />
```

Both `commit` and `optimize` take two additional boolean options that default to `true`:

```
<commit waitFlush="true" waitSearcher="true">
```

If you were to set these to `false`, then commit and optimize return immediately, even though the operation hasn't actually finished yet. So if you wrote a script that committed with these at their false values and then executed a query against Solr, then you may find that the query will not reflect the changes (yet). By waiting for the data to flush to the disk (`waitFlush`) and waiting for a new searcher to be ready to respond to changes (`waitSearcher`), this circumstance is avoided.

A convenient alternative to send these commands through XML is to simply add `commit`, `optimize`, or `rollback` as boolean request parameters when communicating with Solr. You'll see an example of this with CSV in the next section. Request parameters can be put on the URL and/or as form parameters, if applicable. These three request parameters are honored by Solr whether you send Solr its native XML format, CSV, or rich documents. `waitFlush` and `waitSearcher` are not supported in this manner.

Sending CSV to Solr

If you have data in a CSV format or if it is more convenient for you to get CSV than XML, then you may prefer the CSV option to the XML format. Solr's CSV options are fairly flexible.

To get some CSV data out of a local PostgreSQL database for the MusicBrainz tracks, I ran this command:

```
psql -U postgres -d musicbrainz_db -c "COPY (\
select 'Track:' || t.id as id, 'Track' as type, t.name as t_name,
t.length/1000 as t_duration, a.id as t_a_id, a.name as t_a_name,
albumjoin.sequence as t_num, r.id as t_r_id, r.name as t_r_name, array_
to_string(r.attributes,' ') as t_r_attributes, albummeta.tracks as t_r_
tracks \
from (track t inner join albumjoin on t.id = albumjoin.track \
 inner join album r on albumjoin.album = r.id left join albummeta on
albumjoin.album = albummeta.id) inner join artist a on t.artist = a.id \
) to '/tmp/tracks' CSV HEADER"
```

And it generated output that looks like this (first three lines):

```
id,type,t_name,t_duration,t_a_id,t_a_name,t_num,t_r_id,t_r_name,t_r_
attributes,t_r_tracks
Track:183326,Track,In the Arms of Sleep,254,11650,The Smashing
Pumpkins,4,22471,Mellon Collie and the Infinite Sadness (disc 2: Twilight
to Starlight),0 1 100,14
Track:183328,Track,Tales of a Scorched Earth,228,11650,The Smashing
Pumpkins,6,22471,Mellon Collie and the Infinite Sadness (disc 2: Twilight
to Starlight),0 1 100,14
```

To get Solr to import the CSV file, I typed this at the command line:

```
curl http://localhost:8983/solr/update/csv -F f.t_r_attributes.split=true
-F f.t_r_attributes.separator=' ' -F overwrite=false -F commit=true -F
stream.file=/tmp/tracks
```

When I actually did this I had PostgreSQL on one machine and Solr on another. I used the Unix `mkfifo` command to create an in-memory data pipe mounted at `/tmp/tracks`. This way, I didn't have to actually generate a huge CSV file. I could essentially stream it directly from PostgreSQL into Solr. Details on this approach and PostgreSQL are out of the scope of this book.

Configuration options

The configuration options to Solr's CSV capability are set through HTTP posting name-value pairs in the same format that HTML forms post their data. As explained earlier, technically you could use a URL through HTTP GET with a `stream.url` or `stream.file` parameter. However, this is a bad practice. Also note that Solr's CSV capability doesn't support index-time boosting, but that is an uncommon requirement.

The following are the names of each configuration option with an explanation. For the MusicBrainz track CSV file, I was able to use the defaults with the exception of specifying how to parse the multi-valued `t_r_attributes` field and disabling unique key processing for performance.

- **separator**: The character that separates each value on a line. Defaults to `,`.
- **header**: Is set to `true` if the first line lists the field names (the default).
- **fieldnames**: If the first line doesn't have the field names, then you'll have to use this instead to indicate what they are. They are comma separated. If no name is specified for a column, then its data is skipped.
- **skip**: The fields to not import in the CSV file.
- **skipLines**: The number of lines to skip in the input file. Defaults to `0`.
- **trim**: If `true`, then removes leading and trailing whitespace as a final step, even if quoting is used to explicitly specify a space. Defaults to `false`. Solr already does an initial pass trim, but quoting may leave spaces.
- **encapsulator**: This character is used to encapsulate (that is surround, quote) values in order to preserve the field separator as a field value instead of mistakenly parsing it as the next field. This character itself is escaped by doubling it. It defaults to the double quote, unless escape is specified. Example:

  ```
  11604, foo, "The ""second"" word is quoted.", bar
  ```

- **escape**: If this character is found in the input text, then the next character is taken literally in place of this escape character, and it isn't otherwise treated specially by the file's syntax. Example:

  ```
  11604, foo, The second\, word is followed by a comma., bar
  ```

- **keepEmpty**: Specified whether blank (zero length) fields should be indexed as such or omitted. It defaults to `false`.
- **overwrite**: It indicates whether to enforce the unique key constraint of the schema by overwritting existing documents with the same ID. It defaults to `true`. Disable this to increase performance, if you are sure you are passing new documents.

- **split**: This is a field-specific option used to split what would normally be one value into multiple values. Another set of CSV configuration options (separator, and so on) can be specified for this field to instruct Solr on how to do that. See the previous tracks MusicBrainz example on how this is used.
- **map**: This is another field-specific option used to replace input values with another. It can be used to remove values too. The value should include a colon which separates the left side which is replaced with the right side. If we were to use this feature on the tracks of the MusicBrainz data, then it could be used to map the numeric code in `t_r_attributes` to more meaningful values. Here's an example of such an attempt:

  ```
  -F keepEmpty=false -F f.t_r_attributes.map=0:
    -F f.t_r_attributes.map=1:Album -F f.t_r_attributes.map=2:Single
  ```

  This causes `0` to be removed, because it seems to be useless data, as nearly all tracks have it, and we map `1` to `Album` and `2` to `Single`.

Direct database and XML import

The capability for Solr to get data directly from a database or HTTP GET accessible XML is distributed with Solr as a contrib module, and it is known as the `DataImportHandler` (DIH in short). The complete reference documentation for this capability is here at `http://wiki.apache.org/solr/DataImportHandler`, and it's rather thorough. In this chapter, we'll only walk through an example to see how it can be used with the MusicBrainz data set.

In short, the DIH offers the following capabilities:

- Imports data from databases through **JDBC** (**Java Database Connectivity**)
- Imports XML data from a URL (HTTP GET) or a file
- Can combine data from different tables or sources in various ways
- Extraction/Transformation of the data
- Import of updated (delta) data from a database, assuming a last-updated date
- A diagnostic/development web page
- Extensible to support alternative data sources and transformation steps

As the MusicBrainz data is in a database, the most direct method to get data into Solr is definitely through the DIH using JDBC.

Getting started with DIH

DIH is not a direct part of Solr. Hence it might not be included in your Solr setup. It amounts to a JAR file named something like `apache-solr-dataimporthandler-1.4.jar`, which is probably already embedded within the `solr.war` file. You can use `jar -tf solr.war` to see. Alternatively, it may be placed in `<solr-home>/lib`, which is alongside the `conf` directory we've been working with. For database connectivity, we need to ensure that the JDBC driver is on the Java classpath. Placing it in `<solr-home>/lib` is a convenient way to do this.

The DIH needs to be registered with Solr in `solrconfig.xml`. Here is how it is done:

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

`mb-dih-artists-jdbc.xml` (**mb** being short for MusicBrainz) is a file in `<solr-home>/conf`, which is used to configure DIH. It is possible to specify some configuration aspects in this request handler configuration instead of the dedicated configuration file. However, I recommend that it all be in the DIH `config` file, as in our example here.

Given below is an `mb-dih-artists-jdbc.xml` file with a rather long SQL query:

```
<dataConfig>
    <dataSource name="jdbc" driver="org.postgresql.Driver"
        url="jdbc:postgresql://localhost/musicbrainz_db"
        user="musicbrainz" readOnly="true" autoCommit="false" />
    <document>
        <entity name="artist" dataSource="jdbc" pk="id" query="
                select
              a.id as id,
              a.name as a_name, a.sortname as a_name_sort,
              a.begindate as a_begin_date, a.enddate as a_end_date,
              a.type as a_type
              ,array_to_string(
                  array(select aa.name from artistalias aa
                         where aa.ref = a.id )
                  , '|') as a_alias
              ,array_to_string(
```

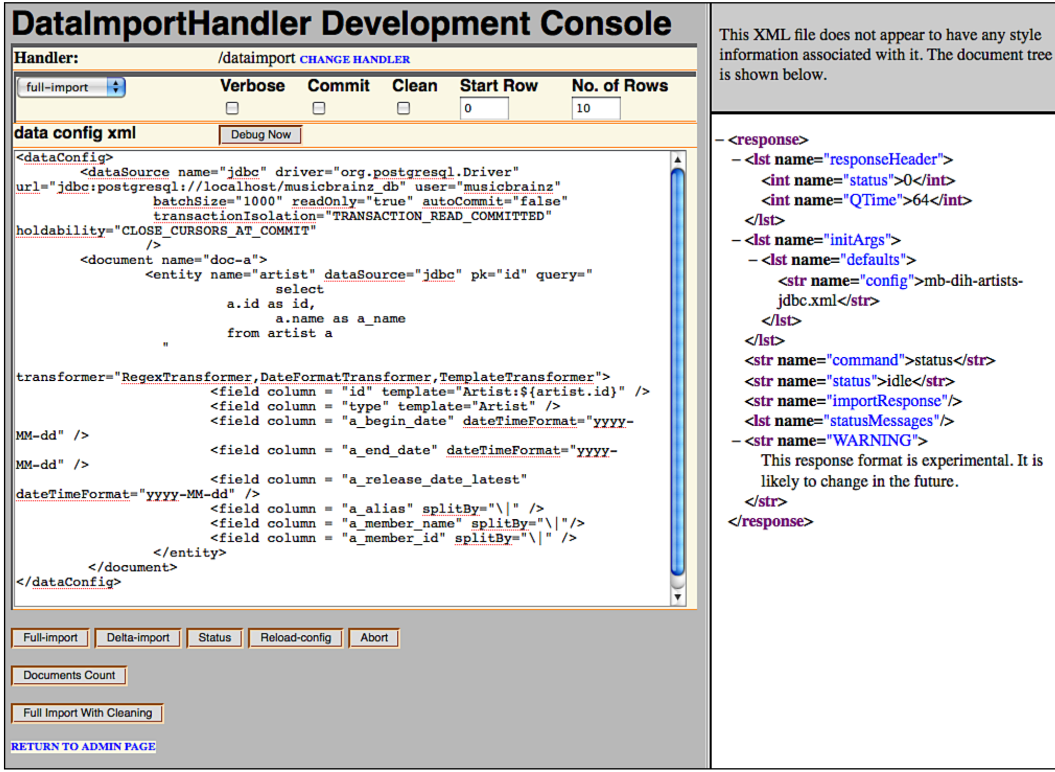
```
                    array(select am.name from v_artist_members am
                          where am.band = a.id order by am.id)
                   , '|') as a_member_name
                ,array_to_string(
                    array(select am.id from v_artist_members am
                          where am.band = a.id order by am.id)
                  , '|') as a_member_id,
                  (select re.releasedate from release re inner join
                   album r on re.album = r.id where r.artist = a.id
                   order by releasedate desc limit 1) as
                   a_release_date_latest
                from artist a
            "
      transformer="RegexTransformer,DateFormatTransformer,
                   TemplateTransformer">
              <field column = "id" template="Artist:${artist.id}" />
              <field column = "type" template="Artist" />
              <field column = "a_begin_date"
                                dateTimeFormat="yyyy-MM-dd" />
              <field column = "a_end_date"
                                dateTimeFormat="yyyy-MM-dd" />
              <field column = "a_alias" splitBy="\|" />
              <field column = "a_member_name" splitBy="\|"/>
              <field column = "a_member_id" splitBy="\|" />
          </entity>
      </document>
  </dataConfig>
```

The DIH development console

Before describing the configuration details, we're going to take a look at the DIH development console. It is accessed by going to this URL (modifications may be needed for your host, port, core, and so on):

```
http://localhost:8983/solr/admin/dataimport.jsp
```

The development console looks like the following screenshot:



The screen is divided into two panes: on the left is the DIH control form, which includes an editable version of the DIH configuration file and on the right is the command output as raw XML. The screen works quite simply. The form essentially results in submitting a URL to the right pane. There's no real server-side logic to this interface beyond the standard DIH command invocations being executed on the right. The last section on DIH in this chapter goes into more detail on submitting a command to the DIH.

DIH DataSources of type JdbcDataSource

The DIH configuration file starts with the declaration of one or more data sources using the element `<dataSource/>`, which refers to either a database, a file, or an HTTP URL, depending on the `type` attribute. It defaults to a value of `JdbcDataSource`. Those familiar with JDBC should find the `driver` and `url` attributes with accompanying `user` and `password` straightforward—consult the documentation for your driver/database for further information. `readOnly` is a boolean that will set a variety of other JDBC options appropriately when set

to `true`. And `batchSize` is an alias for the JDBC `fetchSize` and defaults to `500`. There are numerous JDBC oriented attributes that can be set as well. I would not normally recommend learning about a feature by reading source code, but this is an exception. For further information, read `org.apache.solr.handler.dataimport.JdbcDataSource.java`

Efficient JDBC configuration

Many database drivers in the default configurations (including those for PostgreSQL and MySQL) fetch all of the query results into the memory instead of on-demand or using a batch/fetch size. This may work well for typical database usage like **OLTP (Online Transaction Processing systems)**, but is completely unworkable for **ETL (Extract Transform and Load)** usage such as this. Configuring the driver to stream the data requires driver-specific configuration options. You may need to consult relevant documentation for the JDBC driver. For PostgreSQL, set `autoCommit` to `false`. For MySQL, set `batchSize` to `-1` (The DIH detects the `-1` and replaces it with Java's `Integer.MIN_VALUE`, which triggers the special behavior in MySQL's JDBC driver). For Microsoft SQL Server, set `responseBuffering` to `adaptive`. Further information about specific databases is at `http://wiki.apache.org/solr/DataImportHandlerFaq`.

DIH documents, entities

After the declaration of `<dataSource/>` element(s) is the `<document/>` element. In turn, this element contains one or more `<entity/>` elements. In this sample configuration, we're only getting artists. However, if we wanted to have more than one type in the same index, then another could be added. The `dataSource` attribute references a correspondingly named element earlier. It is only necessary if there are multiple to choose from, but we've put it here explicitly anyway.

The main piece of an entity used with a JDBC data source is the `query` attribute, which is the SQL query to be evaluated. You'll notice that this query involves some sub-queries, which are made into arrays and then transformed into strings joined by spaces. The particular functions used to do these sorts of things are generally database specific. This is done to shoe-horn multi-valued data into a single row in the results. It may create a more complicated query, but it does mean that the database does all of the heavy lifting so that all of the data Solr needs for an artist is in the row. An alternative with DIH is to declare other entities within the entity. If you aren't using a database or if you wish to mix in another data source (even if it's of a different type), then you will be forced to do that. See the Solr DIH Wiki page for examples: `http://wiki.apache.org/solr/DataImportHandler`.

The DIH also supports a delta query, which is a query that selects time-stamped data with dates after the last queried date. This won't be covered here, but you can find more information at the previous URL.

DIH fields and transformers

Within the `<entity/>` are some `<field/>` elements that declare how the columns in the query map to Solr. The field element must have a `column` attribute that matches the corresponding named column in the SQL query. The `name` attribute is the Solr schema field name that the column is going into. If it is not specified (and it never is for our example), then it defaults to the column name.

Use the SQL `as` a keyword as we've done to use the same names as the Solr schema instead of the database schema. This reduces the number of explicit mappings needed in `<field/>` elements and shortens existing ones.

When a column in the result can be placed directly into Solr without further processing, there is no need to specify the field declaration, because it is implied.

An attribute of the entity declaration that we didn't mention yet is `transformer`. This declares a comma-separated list of transformers that manipulate the transfer of data from the JDBC `resultset` into a Solr field. These transformers evaluate a field, if it has an attribute it uses to do its job. More than one might operate on a given field. Therefore, the order in which the transformers are declared in matters. Here are the attributes we've used:

- `template`: It is used by `TemplateTransformer` and declares text, which might include variable name substitutions using `${name}` syntax. To access an existing field, use the `entityname.columnname` syntax.
- `splitBy`: It is used by `RegexTransformer` and splits a single string value into a multi-value by looking for the specified character.
- `dateTimeFormat`: It is used by `DateFormatTransformer`. This is a Java date/time format pattern (`http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html`). If the type of the field in the schema is a date, then it is necessary to ensure Solr can interpret the format. Alternatively, ensure that the string matches the ISO-8601 format, which looks like this: `1976-10-23T23:59:59.000Z`. As in all cases in Solr, when specifying dates you can use its so-called "DateMath" syntax (described in the next chapter) such as appending `/DAY` to tell Solr to round the date to a day.

Importing with DIH

Unlike the other importing mechanisms, the DIH returns immediately, while the import continues asynchronously. To get the current status of the DIH, go to this URL `http://localhost:8983/solr/dataimport`, and you'll get output like the following:

```
<response>
    <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">15</int>
    </lst>
    <lst name="initArgs">
        <lst name="defaults">
            <str name="config">mb-dih-artists-jdbc.xml</str>
        </lst>
    </lst>
    <str name="status">idle</str>
    <str name="importResponse"/>
    <lst name="statusMessages"/>
    <str name="WARNING">This response format is experimental. It is
              likely to change in the future.</str>
</response>
```

Commands are given to DIH as request parameters just as everything else is in Solr. We could tell the DIH to do a full-import just by going to this URL: `http://localhost:8983/solr/dataimport?command=full-import`. On the command line we would use:

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

It uses HTTP POST, which is better, as discussed much earlier.

Other boolean parameters named `clean`, `commit`, and `optimize` may accompany the parameter that defaults to `true`, if not present. Clean is specific to DIH, and it means that before running the import, it will remove all of the existing data.

Two other useful commands are `reload-config` and `abort`. The first will reload the DIH configuration file, which is useful for picking up small changes. The second will cancel any existing imports in progress.

Indexing documents with Solr Cell

While most of this book assumes that the content you want to index in Solr is in a neatly structured data format of some kind, such as in a database table, a selection of XML files, or CSV, the reality is that we store information in the much messier world of binary formats such as PDF, Microsoft Office, or even images and music files. Your author Eric Pugh, became involved with the Solr community when a client required a search system that could ingest the thousands of PDF and Microsoft Word documents that they had produced over the years. The outgrowth of that effort, **Solr Cell**, distributed as a `contrib` module to Solr, provides a very powerful framework for indexing various binary formats.

Solr Cell is technically called the Extracting Request Handler, however the name came about because:

> *Grant [Ingersoll] was writing the javadocs for the code and needed an entry for the <title> tag and wrote out "Solr Content Extraction Library", as the contrib directory is named "extraction". This then lead to an "acronym": Solr CEL, which then gets mashed to: Solr Cell! Hence, the project name is "Solr Cell"! It's also appropriate because a Solar Cell's job is to convert the raw energy of the Sun to electricity, and this contrib's module is responsible for converting the "raw" content of a document to something usable by Solr.*

We'll look at how to leverage Solr Cell for extracting karaoke song lyrics from MIDI files. Just think, you can build a Solr powered index of all your favorite karaoke songs! The complete reference material for Solr Cell is available at `http://wiki.apache.org/solr/ExtractingRequestHandler`.

Extracting binary content

Every binary format is different, and all of them provide different types of metadata, as well as different methods of extracting content. The heavy lifting of providing a single API to an ever expanding list of binary/structured formats is left up to **Apache Tika**:

> *Apache Tika is a toolkit for detecting and extracting metadata and structured text content from various documents using existing parser libraries.*

Tika supports a wide variety of formats, from the predictable to the unexpected. Some of the key formats supported are Adobe PDF, Microsoft Office including Word, Excel, PowerPoint, and Visio, and Outlook. Other formats that are supported include extracting metadata from images such as JPG, GIF, and PNG, as well as from various audio formats such as MP3, MIDI, and Wave audio. Tika itself does not attempt to parse the individual binary formats. Instead, it delegates the parsing to various third party libraries, while providing a high level stream of SAX events as the documents are parsed.

Solr Cell is a fairly thin wrapper consisting of a `SolrContentHandler` that consumes the SAX events and builds the `SolrInputDocument` from the fields that are specified to be extracted from the binary document.

There are some not so obvious things to keep in mind when indexing binary documents:

- You can supply any kind of supported document to Tika, and the `AutoDetectParser` will attempt to discover the correct MIME type of the document. Alternatively, you can supply a `stream.type` parameter to specify which parser to use.
- The default `SolrContentHandler` that is used by Solr Cell is fairly simplistic. You may find that you need to perform extra massaging of the data being indexed beyond what Solr Cell offers to reduce the junk data being indexed. Subclass `createFactory()` method of `ExtractingRequestHandler` to return your own custom `SolrContentHandler`.
- Remember that as you are indexing you are potentially sending large binary files over the wire that must then be parsed in server memory, which can be very slow. If you are looking to only index metadata, then it may make sense to write your own parser using Tika directly, extract the metadata, and post that across to the server. See the *Indexing HTML in Solr* section in Chapter 8 for an example of parsing out metadata from an archive of a web site and posting the data through SolrJ.
- You need to supply to Solr the various dependent JAR files that Tika requires to parse the documents. Put them with the Solr Cell JAR (named something like `apache-solr-cell-1.4.jar`) in `<solr-home>/lib`.

Tika has only recently become a full fledged project and has already had a couple of releases. You can learn more about Tika from the web site at `http://lucene.apache.org/tika/`.

Configuring Solr

In /examples/cores/karaoke/conf/solrconfig.xml lies the request handler for parsing binary documents:

```
<requestHandler name="/update/extract"
 class="org.apache.solr.handler.extraction.ExtractingRequestHandler">
    <lst name="defaults">
      <str name="map.Last-Modified">last_modified</str>
      <str name="uprefix">metadata_</str>
    </lst>
</requestHandler>
```

Here we can see that the Tika metadata attribute Last-Modified is being mapped to the Solr field last_modified, assuming we are provided that Tika attribute. The parameter uprefix is specifying the prefix to use when storing any Tika fields that don't have a corresponding matching Solr field.

In order to use Solr Cell, we placed the Solr Cell JAR in the ./examples/cores/karaoke/lib/ directory, because it is not included in solr.war. The JAR files placed in this lib directory are available only to the karaoke core. To share across all cores add to ./examples/cores/lib/ and by specifying it as the shared lib in ./examples/cores/solr.xml:

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

For this example, we are parsing .kar karaoke files that are recorded in the MIDI format using the standard Java package javax.audio.midi. However, we have also put other JAR dependencies of Solr Cell such as pdfbox, poi, and icu4j in ./lib.

Extracting karaoke lyrics

We are now ready to extract karaoke lyrics by posting MIDI files to our Solr /karaoke/update/extract handler. Some classic ABBA tunes for your enjoyment are available in the ./examples/appendix/karaoke/songs/ directory, gratefully sourced from FreeKaraoke at http://www.freekaraoke.com/. In order to index the song *Angel Eyes* from the command line using curl, the simplest command to run is:

```
>> curl 'http://localhost:8983/solr/karaoke/update/extract?map.
content=text' -F "file=@angeleyes.kar"
```

Don't forget to commit your changes:

```
>> curl http://localhost:8983/solr/karaoke/update/ -H "Content-Type:
text/xml" --data-binary '<commit waitFlush="false"/>'
```

You can also trigger a commit when indexing content by appending `commit=true` to the URL, however this is an inefficient approach if you are indexing many documents

We have a single `map.content=text` parameter that specifies the default field for content extracted from the source. In this case, `angeleyes.kar` should be stored in the Solr field `text`. Now go look for the results at `http://localhost:8983/solr/karaoke/select/?q=*:*`. You should see:

```
<result name="response" numFound="1" start="0">
  <doc>
    <arr name="text">
      <str>
    Angel Eyes by Abba sequenced by Michael Boyce
    tinker@worldnet.att.netSoft karaoke@KMIDI KARAOKE
    FILEWords@LENGL@TAngel Eyes@TABBA\Last night I was taking a walk
    along the river/And I saw him together with a young girl/And the
    look that he gave made me shiver/'Cause he always used ...
      </str>
    </arr>
  </doc>
</result>
```

You've now indexed information about the song and the lyrics in the text field that forms the textual content of the MIDI file. However, what about the metadata, for the MIDI file that Tika also exposes? Well, this is where dynamic fields come in very handy. Every binary format has a set of metadata that to a varying extent overlaps with other formats. Fortunately it is very easy to specify to Solr Cell how you would want to map metadata by using the `uprefix` property. We specify that all of the `metadata_*` fields should be created using dynamic fields in `schema.xml`:

```
<dynamicField name="metadata_*"  type="string"  indexed="true"
              stored="true" multiValued="false"/>
```

Since handling metadata properly is something we want to standardize on, we add to the configuration element in `solrconfig.xml`:

```
<str name="map.Last-Modified">last_modified</str>
<str name="uprefix">metadata_</str>
```

Notice that the `&` values in the URL are escaped with backslashes: `\`. Forgetting to escape special characters is a common issue when working with `curl`.

www.allitebooks.com

When you search for all documents, you should see indexed metadata for *Angel Eyes*, prefixed with `metadata_`:

```
<str name="metadata_Content-Type">audio/midi</str>
<str name="metadata_divisionType">PPQ</str>
<str name="metadata_patches">0</str>
<str name="metadata_stream_content_type">
        application/octet-stream</str>
<str name="metadata_stream_name">angeleyes.kar</str>
<str name="metadata_stream_size">55677</str>
<str name="metadata_stream_source_info">file</str>
<str name="metadata_tracks">16</str>
```

Obviously, in most use cases, every time you index the same file you don't want to get a new document. If your schema has a `uniqueKey` field defined such as `id`, then you can provide a specific ID by passing a literal value using `literal.id=34`. Each time you index the file using the same ID, it will delete and insert that document. However, that implies that you have the ability to manage IDs through some third party system like a database. If you want to use the metadata, such as the `stream_name` provided by Tika to provide the key, then you just need to map that field using `map.stream_name=id`. To make the example work, update `./examples/cores/karaoke/schema.xml` to specify `<uniqueKey>id</uniqueKey>`.

```
>> curl 'http://localhost:8983/solr/karaoke/update/extract?map.
content=text&map.stream_name=id'  -F "file=@angeleyes.kar"
```

This of course assumes that you've defined `<uniqueKey>id</uniqueKey>` to be of type string, not a number.

Indexing richer documents

Indexing karaoke lyrics from MIDI files is also a fairly trivial example. We basically just strip out all of the contents, and store them in the Solr text field. However, indexing other types of documents, such as PDFs, can be a bit more complicated. Let's look at *Take a Chance on Me,* a complex PDF file that explains what a Monte Carlo simulation is, while making lots of puns about the lyrics and titles of songs from ABBA. View `./examples/appendix/karaoke/mccm.pdf`, and you will see a complex PDF document with multiple fonts, background images, complex mathematical equations, Greek symbols, and charts. However, indexing that content is as simple as the prior example:

```
>> curl 'http://localhost:8983/solr/karaoke/update/extract?map.
content=text&map.stream_name=id&commit=true'  -F "file=@mccm.pdf"
```

If you do a search for the document using the filename as the `id` via `http://localhost:8983/solr/karaoke/select/?q=id:mccm.pdf`, then you'll also see that the `last_modified` field that we mapped in `solrconfig.xml` is being populated. Tika provides a `Last-Modified` field for PDFs, but not for MIDI files:

```
<doc>
  <arr name="id">
    <str>mccm.pdf</str>
  </arr>
  <arr name="last_modified">
    <str>Sun Mar 03 15:55:09 EST 2002</str>
  </arr>
  <arr name="text">
    <str>
      Take A Chance On Me
```

So with these richer documents, how can we get a handle on the metadata and content that is available? Passing `extractOnly=true` on the URL will output what Solr Cell has extracted, including metadata fields, without actually indexing them:

```
<response>
...
<str name="mccm.pdf">&lt;?xml version="1.0" encoding="UTF-8"?&gt;
  &lt;html xmlns="http://www.w3.org/1999/xhtml"&gt;
    &lt;head&gt;
        &lt;title&gt;Take A Chance On Me&lt;/title&gt;
    &lt;/head&gt;
    &lt;body&gt;
        &lt;div&gt;
  &lt;p&gt;
Take A Chance On Me
Monte Carlo Condensed Matter
A very brief guide to Monte Carlo simulation.
...
</str>
<lst name="mccm.pdf_metadata">
  <arr name="stream_source_info"><str>file</str></arr>
  <arr name="subject"><str>Monte Carlo Condensed Matter</str></arr>
  <arr name="Last-Modified"><str>Sun Mar 03 15:55:09 EST
             2002</str></arr>
...
  <arr name="creator"><str>PostScript PDriver module 4.49</str></arr>
  <arr name="title"><str>Take A Chance On Me</str></arr>
  <arr name="stream_content_type"><str>application/
             octet-stream</str></arr>
  <arr name="created"><str>Sun Mar 03 15:53:14 EST 2002</str></arr>
  <arr name="stream_size"><str>378454</str></arr>
  <arr name="stream_name"><str>mccm.pdf</str></arr>
</lst>
</response>
```

At the top in an XML node called `<str name="mccm.pdf"/>` is the content extracted from the PDF as an XHTML document. As it is XHTML wrapped in another separate XML document, the various `<and>` tags have been escaped: `<div>`. If you cut and paste the contents of `<str/>` node into a text editor and convert the `<` to `<` and `>` to `>`, then you can see the structure of the XHTML document that is indexed.

Below the contents of the PDF, you can also see a wide variety of PDF document-specific metadata fields, including subject, title, and creator, as well as metadata fields added by Solr Cell for all imported formats, including `stream_source_info`, `stream_content_type`, `stream_size`, and the already-seen `stream_name`.

So why would we want to see the XHTML structure of the content? The answer is in order to narrow down our results. We can use `XPath` queries through the `ext.xpath` parameter to select a subset of the data to be indexed. To make up an arbitrary example, let's say that after looking at `mccm.html` we know we only want the second paragraph of content to be indexed:

```
>> curl 'http://localhost:8983/solr/karaoke/update/extract?map.
content=text&map.div=divs_s&capture=div&captureAttr=true&xpath=\/\/xhtml:
p[1]'  -F "file=@mccm.pdf"
```

We now have only the second paragraph, which is the summary of what the document *Take a Chance on Me* is about.

Binary file size

Take a Chance on Me is a 372 KB file stored at `./examples/appendix/karaoke/mccm.pdf`, and it highlights one of the challenges of using Solr Cell. If you are indexing a thousand PDF documents that each average 372 KB, then you are shipping 372 megabytes over the wire, assuming the data is not already on Solr's file system. However, if you extract the contents of the PDF on the client side and only send that over the web, then what is sent to the Solr text field is just 5.1 KB. Look at `./examples/appendix/karaoke/mccm.txt` to see the actual text extracted from `mccm.pdf`. Generously assuming that the metadata adds an extra 1 KB of information, then you have a total content sent over the wire of 6.1 megabytes ((5.1 KB + 1.0 KB) * 1000).

Solr Cell offers a quick way to start indexing that vast amount of information stored in previously inaccessible binary formats without resorting to custom code per binary format. However, depending on the files, you may be needlessly transmitting a lot of data, only to extract a small portion of text. Moreover, you may find that the logic provided by Solr Cell for parsing and selecting just the data you want may not be rich enough. For these cases you may be better off building a dedicated client-side tool that does all of the parsing and munging you require.

Summary

At this point, you should have a schema that you believe will suit your needs, and you should know how to get your data into it. From Solr's native XML to CSV to databases to rich documents, Solr offers a variety of possibilities to ingest data into the index. Chapter 8 will discuss some additional choices for importing data. In the end, usually one or two mechanisms will be used. In addition, you can usually expect the need to write some code, perhaps just a simple bash or ant script to implement the automation of getting data from your source system into Solr.

Now that we've got data in Solr, we can finally get to querying it. The next chapter will describe Solr/Lucene's query syntax in detail, which includes phrase queries, range queries, wildcards, boosting, as well as the description of Solr's `DateMath` syntax. Finally, you'll learn the basics of scoring and how to debug them. The chapters after that will get to more interesting querying topics that of course depend on having data to search with.

4

Basic Searching

At this point, you have Solr running and some data indexed, and you're finally ready to put Solr to the test. Searching with Solr is arguably the most fun aspect of working with it, because it's quick and easy to do. While searching your data, you will learn more about its nature than before. It is also a source of interesting puzzles to solve when you troubleshoot why a search didn't find a document or conversely why it did, or similarly why a document wasn't scored sufficiently high.

In this chapter, you are going to learn about:

- The Full Interface for querying Solr
- Solr's query response XML
- Using query parameters to configure the search
- Solr/Lucene's query syntax
- The factors influencing scoring

Your first search, a walk-through

We've got a lot of data indexed, and now it's time to actually use Solr for what it is intended—searching (aka querying). When you hook up Solr to your application, you will use HTTP to interact with Solr, either by using an HTTP software library or indirectly through one of Solr's client APIs. However, as we demonstrate Solr's capabilities in this chapter, we'll use Solr's web-based admin interface. Surely you've noticed the search box on the first screen of Solr's admin interface. It's a bit too basic, so instead click on the **[FULL INTERFACE]** link to take you to a query form with more options.

The following screenshot is seen after clicking on the **[FULL INTERFACE]** link:

Solr Admin (musicbrainz)

192.168.0.10:8983
cwd=/SmileyDev/Projects-Mine/SolrBookServer/solr_example
SolrHome=mbartists/

Apache Solr

Solr/Lucene Statement	solr
Start Row	0
Maximum Rows Returned	10
Fields to Return	*,score
Query Type	standard
Output Type	standard
Debug: enable	*Note: you may need to "view source" in your browser to see explain() correctly indented.*
Debug: explain others	*Apply original query scoring to matches of this query to see how they compare.*
Enable Highlighting	
Fields to Highlight	
	Search

This form demonstrates the most common query options available for the built in Query Types. Please consult the Solr Wiki for additional Query Parameters.

Contrary to what the label **FULL INTERFACE** might suggest, this form only has a fraction of the options you might possibly specify to run a search. Let's jump ahead for a second, and do a quick search. In the **Solr/Lucene Statement** box, type ***:*** (an asterisk, colon, and then another asterisk). That is admittedly cryptic if you've never seen it before, but it basically means match anything in any field, which is to say, it matches all documents. Much more about the query syntax will be discussed soon enough. At this point, it is tempting to quickly hit *return* or *enter*, but that inserts a newline instead of submitting the form (this will hopefully be fixed in the future). Click on the **Search** button, and you'll get output like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">392</int>
```

```
    <lst name="params">
    <str name="explainOther"/>
    <str name="fl">*,score</str>
    <str name="indent">on</str>
    <str name="start">0</str>
    <str name="q">*:*</str>
    <str name="hl.fl"/>
    <str name="qt">standard</str>
    <str name="wt">standard</str>
    <str name="version">2.2</str>
    <str name="rows">10</str>
    </lst>
  </lst>
  <result name="response" numFound="1002272" start="0" maxScore="1.0">
    <doc>
      <float name="score">1.0</float>
      <str name="id">Release:449119</str>
      <str name="r_a_id">56063</str>
      <str name="r_a_name">The Spotnicks</str>
      <arr name="r_attributes"><int>0</int><int>1</int><int>100</int>
      </arr>
      <arr name="r_event_country"><str>JP</str></arr>
      <arr name="r_event_date"><date>1965-11-30T05:00:00Z</date></arr>
      <str name="r_lang">English</str>
      <str name="r_name">The Spotnicks in Tokyo</str>
      <int name="r_tracks">16</int>
      <str name="type">Release</str>
    </doc>
    <doc>
      <float name="score">1.0</float>
      <str name="id">Release:186779</str>
      <str name="r_a_id">56011</str>
      <str name="r_a_name">Metro Area</str>
      <arr name="r_attributes"><int>0</int><int>1</int><int>100</int>
      </arr>
      <arr name="r_event_country"><str>US</str></arr>
      <arr name="r_event_date"><date>2001-11-30T05:00:00Z</date></arr>
      <str name="r_name">Metro Area</str>
      <int name="r_tracks">11</int>
      <str name="type">Release</str>
    </doc>
  <!-- ** 7 other docs omitted for brevity ** -->
  </result>
  </response>
```

Browser note

Use Firefox for best results when searching Solr. Solr's search results return XML, and Firefox renders XML color coded and pretty-printed. For other browsers (notably Safari), you may find yourself having to use the **View Source** feature to interpret the results. Even in Firefox, however, there are cases where you will use **View Source** in order to look at the XML with the original indentation, which is relevant when diagnosing the scoring debug output.

Solr's generic XML structured data representation

Solr has its own generic XML representation of typed and named data structures. This XML is used for most of the `responseXML` and it is also used in parts of `solconfig.xml` too. The XML elements involved in this partial schema are:

- `lst`: A named list. Each of its child nodes should have a name attribute. This generic XML is often stored within an element not part of this schema, like `doc`, but is in effect equivalent to `lst`.
- `arr`: An array of values. Each of its child nodes are a member of this array.

The following elements represent simple values with the text of the element storing the value. The numeric ranges match that of the Java language. They will have a `name` attribute if they are underneath `lst` (or an equivalent element like `doc`), but not otherwise.

- `str`: A string of text
- `int`: An integer in the range -2^31 to 2^31-1
- `long`: An integer in the range -2^63 to 2^63-1
- `float`: A floating point number in the range 1.4e-45 to about 3.4e38
- `double`: A floating point number in the range 4.9e-324 to about 1.8e308
- `bool`: A boolean value represented as `true` or `false`
- `date`: A date in the ISO-8601 format like so: `1965-11-30T05:00:00Z`, which is always in the GMT time zone represented by `Z`

Solr's XML response format

The `<response/>` element wraps the entire response.

The first child element is `<lst name="responseHeader">`, which is intuitively the response header that captures some basic metadata about the response.

- `status`: Always `zero` unless something went very wrong.
- `QTime`: The number of milliseconds Solr takes to process the entire request on the server. Due to internal caching, you should see this number drop to a couple of milliseconds or so for subsequent requests of the same query. If subsequent identical searches are much faster, yet you see the same `QTime`, then your web browser (or intermediate HTTP Proxy) cached the response. Solr's HTTP caching configuration is discussed in Chapter 9.
- Other data may be present depending on query parameters.

The main body of the response is the search result listing enclosed by this: `<result name="response" numFound="1002272" start="0" maxScore="1.0">`, and it contains a `<doc>` child node for each returned document. Some of the fields are explained below:

- `numFound`: The total number of documents matched by the query. This is not impacted by the `rows` parameter and as such may be larger (but not smaller) than the number of child `<doc>` elements.
- `start`: The same as the `start` parameter, which is the offset of the returned results into the query's result set.
- `maxScore`: Of all documents matched by the query (`numFound`), this is the highest score. If you didn't explicitly ask for the score in the field list using the `fl` parameter, then this won't be here. Scoring is described later in this chapter.

The contents of the resultant element are a list of `doc` elements. Each of these elements represents a document in the index. The child elements of a `doc` element represent fields in the index and are named correspondingly. The types of these elements are in the generic data structure partial schema, which was described earlier. They are simple values if they are not multi-valued in the schema. For multi-valued values, the field would be represented by an ordered array of simple values.

There was no data following the results element in our demonstration query. However, there can be, depending on the query parameters using features such as faceting and highlighting. When those features are described, the corresponding XML will be explained.

Parsing the URL

The search form is a very simple thing, no more complicated than a basic one you might see in a tutorial if you are learning HTML for the first time. All that it does is submit the form using HTTP GET, essentially resulting in the browser loading a new URL with the form elements becoming part of the URL's query string. Take a good look at the URL in the browser page showing the XML response. Understanding the URL's structure is very important for grasping how search works:

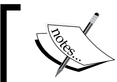
```
http://localhost:8983/solr/select?indent=on&version=2.2&q=*%3A*&start
=0&rows=10&fl=*%2Cscore&qt=standard&wt=standard&explainOther=&hl.fl=
```

- The `/solr/` is the web application context where Solr is installed on the Java servlet engine. If you have a dedicated server for Solr, then you might opt to install it at the root. This would make it just `/`. How to do this is out of scope of this book, but letting it remain at `/solr/` is fine.
- After the web application context is a reference to the Solr core (we don't have one for this configuration). We'll configure Solr Multicore in Chapter 7, at which point the URL to search Solr would look something like `/solr/corename/select?...`
- The `/select` in combination with the `qt=standard` parameter is a reference to the Solr `request handler`. More on this is covered later under the *Request Handler* section. As the standard request handler is the default handler, the `qt` parameter can be omitted in this example.
- Following the `?`, is a set of unordered URL parameters (aka query parameters in the context of searching). The format of this part of the URL is an `&` separated set of unordered `name=value` pairs. As the form doesn't have an option for all query parameters, you will manually modify the URL in your browser to add query parameters as needed.

Remember that the data in the URL must be URL-Encoded so that the URL complies with its specification. Therefore, the `%3A` in our example is interpreted by Solr as `:`, and `%2C` is interpreted as `,`. Although not in our example, the most common escaped character in URLs is a space, which is escaped as either `+` or `%20`. For more information on URL encoding see `http://en.wikipedia.org/wiki/Percent-encoding`.

Query parameters

There are a great number of query parameters for configuring Solr searches, especially when considering all of the components like faceting and highlighting. Only the core parameters are listed here, furthermore, in-depth explanations for some lie further in the chapter.



For the boolean parameters, a true value can be any one of `true`, `on`, or `yes`. False values can be any of `false`, `off`, and `no`.

Parameters affecting the query

The parameters affecting the query are as follows:

- `q`: The query string, aka the user query or just query for short. This typically originates directly from user input. The query syntax will be discussed shortly.
- `q.op`: By default, either `AND` or `OR` to signify if, all of the search terms or just one of the search terms respectively need to match. If this isn't present, then the default is specified near the bottom of the schema file (an admittedly strange place to put the default).
- `df`: The default field that will be searched by the user query. If this isn't specified, then the default is specified in the schema near the bottom in the `defaultSearchField` element. If that isn't specified, then an unqualified query clause will be an error.

Searching more than one field

In order to have Solr search more than one field, it is a common technique to combine multiple fields into one field (indexed, multi-valued, not stored) through the schema's `copyField` directive, and search that by default instead. Alternatively, you can use the `dismax` query type through `defType`, described in the next chapter, which features varying score boosts per field.

- `defType`: A reference to the query parser. The default is "lucene" with the syntax to be described shortly. Alternatively there is "dismax" which is described in the next chapter.
- `fq`: A filter query that limits the scope of the user query. Several of these can be specified, if desired. This is described later.
- `qt`: A reference to the query type, aka query handler. These are defined in `solrconfig.xml` and are described later.

Result paging

A query could match any number of the documents in the index, perhaps even all of them (such as in our first example of `*:*`). Solr doesn't generally return all the documents. Instead, you indicate to Solr with the `start` and `rows` parameters to return a contiguous series of them. The `start` and `rows` parameters are explained below:

- `start`: (default: `0`) This is the zero based index of the first document to be returned from the result set. In other words, this is the number of documents to skip from the beginning of the search results. If this number exceeds the result count, then it will simply return no documents, but it is not considered as an error.
- `rows`: (default: `10`) This is the number of documents to be returned in the response XML starting at index `start`. Fewer rows will be returned if there aren't enough matching documents. This number is basically the number of results displayed at a time on your search user interface.

It is not possible to ask Solr for all rows, nor would it be pragmatic for Solr to support that. Instead, ask for a very large number of rows, a number so big that you would consider there to be something wrong if this number were reached. Then check for this condition, and log it or throw an error. You might even want to prevent users (and web crawlers) from paging farther than 1000 or so documents into the results, because Solr doesn't scale well with such requests, especially under high load.

Output related parameters

The output related parameters are explained below:

- `fl`: This is the field list, separated by commas and/or spaces. These fields are to be returned in the response. Use `*` to refer to all of the fields but not the score. In order to get the score, you must specify the pseudo-field `score`.
- `sort`: A comma-separated field listing, with a directionality specifier (`asc` or `desc`) after each field. Example: `r_name asc, score desc`. The default is `score desc`. There is more to sorting than meets the eye, which is explained later in this chapter.

- `wt`: A reference to the writer type (aka query response writer) defined in `solrconfig.xml`. This is essentially the output format. Most output formats share a similar conceptual structure but they vary in syntax. The language-oriented formats are for scripting languages that have an `eval()` type method, which can conveniently turn a string into a data structure by interpreting the string as code. Here is a listing of the formats supported by Solr out-of-the-box:
    - `xml` (aliased to standard, the default): This is the XML format seen throughout most of the book.
    - `javabin`: A compact binary output used by SolrJ.
    - `json`: The JavaScript Object Notation format for JavaScript clients using `eval()`. `http://www.json.org/`
    - `python`: For Python clients using `eval()`.
    - `php`: For PHP clients using `eval()`. Prefer `phps` instead.
    - `phps`: PHP's serialization format for use with `unserialize()`. `http://www.hurring.com/scott/code/perl/serialize/`
    - `ruby`: For Ruby clients using `eval()`.
    - `xslt`: An extension mechanism using the eXtensible Stylesheet Transformation Language to output other formats. An XSLT file is placed in the `conf/xslt/` directory and is referenced through the `tr` request parameter. A great use of this technique is for exposing an **RSS** (**Really Simple Syndication**) or Atom feed. The Solr distribution includes examples of both.

A practical use of the XSLT option is to expose an RSS/Atom feed on your search results page. With very little work on your part, you can empower users to subscribe to a search to monitor for new data! Look at the Solr examples for a head start.

Custom output formats:

Usually you won't need a custom output format since you'll be writing the client and can use a Solr integration library like SolrJ or just talk to Solr directly with an existing response format. If you do need to support a special format, then you have three choices. The most flexible is to write the mediation code to talk to Solr that exposes the special format/protocol. The simplest if it will suffice is to use XSLT, assuming you know that technology. Finally, you could write your own query response writer.

- `version`: The requested version of the response XML's formatting. This is not particularly useful at the time of writing. However, if Solr's `responseXML` changes, then it will do so under a new version. By using this in the request (a good idea for your automated querying), you reduce the chances of your client breaking if Solr is updated.

Diagnostic query parameters

These diagnostic parameters are helpful during development with Solr. Obviously, you'll want to be sure NOT to use these, particularly `debugQuery`, in a production setting because of performance concerns. The use of `debugQuery` will be explained later in the chapter.

- `indent`: A boolean option, when enabled, will indent the output. It works for all of the response formats (example: XML, JSON, and so on)
- `debugQuery`: If `true`, then following the search results is `<lst name="debug">`, and it contains voluminous information about the parsed query string, how the scores were computed, and millisecond timings for all of the Solr components to perform their part of the processing such as faceting. You may need to use the `View Source` function of your browser to preserve the formatting used in the score computation section.
    - `explainOther`: If you want to determine why a particular document wasn't matched by the query, or the query matched many documents and you want to ensure that you see scoring diagnostics for a certain document, then you can put a query for this value, such as `id:"Release:12345"`, and `debugQuery's` output will be sure to include documents matching this query in its output.
- `echoHandler`: If `true`, then this emits the Java class name identifying the Solr query handler. Solr query handlers are explained later.
- `echoParams`: Controls if any query parameters are returned in the response header (as seen verbatim earlier). This is for debugging URL encoding issues or for checking which parameters are set in the request handler, but is not particularly useful. Specifying `none` disables this, which is appropriate for production real-world use. The standard request handler is configured for this to be `explicit` by default, which means to list those parameters explicitly mentioned in the request (for example the URL). Finally, you can use `all` to include those parameters configured in the request handler in addition to those in the URL.

Query syntax

Solr's query syntax is Lucene's syntax with a couple of additions that will be pointed out explicitly. What Solr/Lucene does is parse a query string using the rules outlined in this section to construct an internal query object tree. The existence of this feature (which is easy to take for granted) allows you or a user to express much more interesting queries than just AND-ing or OR-ing terms specified through `q.op`. The syntax that is discussed in this chapter can be thought of as the full Solr/Lucene syntax. There are no imposed limitations. If you do *not* want users to have this full expressive power (perhaps because they might unintentionally use this syntax and it either won't work or an error will occur), then you can choose an alternative with the `defType` query parameter. This defaults to `lucene`, but can be set to `dismax`, which is a reference to the `DisjunctionMax` parser. The parser and this mechanism in general will be discussed in the next chapter.

In the following examples:

1. `q.op` is set to `OR` (which is the default choice, if it isn't specified anywhere).
2. The default field has been set to `a_name` in the schema.
3. You may find it easier to scan the resulting XML if you set the field list to `a_name, score`.

Use debugQuery=on

To see a normalized string representation of the parsed query tree, enable query debugging. Then look for `parsedquery` in the debug output. See how it changes depending on the query.

Matching all the documents

Lucene doesn't natively have a query syntax to match all documents. Solr enhanced Lucene's query syntax to support it with the following syntax:

```
*:*
```

It isn't particularly common to use this, but it definitely has its uses.

Mandatory, prohibited, and optional clauses

Lucene has a somewhat unique way of combining multiple clauses in a query string. It is tempting to think of this as a mundane detail common to boolean operations in programming languages, but Lucene doesn't quite work that way.

A query expression is decomposed into a set of unordered clauses of three types:

- A clause can be **mandatory**: (for example, only artists containing the word `Smashing`)
  `+Smashing`
- A clause can be **prohibited**: (for example, all documents except those with `Smashing`)
  `-Smashing`
- A clause can be **optional**:
  `Smashing`

It's okay for spaces to come between + or - and the search word.

The term optional deserves further explanation. If the query expression contains at least one mandatory clause, then any optional clause is just that—optional. This notion may seem nonsensical, but it serves a useful function in scoring documents that match more of them higher. If the query expression does not contain any mandatory clauses, then *at least one* of the optional clauses must match. The next two examples illustrate optional clauses.

Here, `Pumpkins` is optional, and my favorite band will surely be at the top of the list, ahead of bands with names like `Smashing Atoms`:

```
+Smashing Pumpkins
```

Here, there are no mandatory clauses and so documents with `Smashing` or `Pumpkins` are matched, but not `Atoms`. Again, my favorite band is at the top because it matched both, though there are other bands containing one of those words too:

```
Smashing Pumpkins -Atoms
```

Boolean operators

The boolean operators `AND`, `OR`, and `NOT` can be used as an alternative syntax to arrive at the same set of mandatory, prohibited, and optional clauses that were mentioned previously. Use the `debugQuery` feature, and observe that the `parsedquery` string normalizes-away this syntax into the previous (clauses being optional by default such as `OR`).

Case matters! At least this means that it is harder to accidentally specify a boolean operator.

When the `AND` or `&&` operator is used between clauses, then both the left and right sides of the operand become mandatory, if not already marked as prohibited. So:

```
Smashing AND Pumpkins
```

is equivalent to:

```
+Smashing +Pumpkins
```

Similarly, if the `OR` or `||` operator is used between clauses, then both the left and right sides of the operand become optional, unless they are marked mandatory or prohibited. If the default operator is already `OR` then this syntax is redundant. If the default operator is `AND`, then this is the only way to mark a clause as optional.

To match artist names that contain `Smashing` **or** `Pumpkins` try:

```
Smashing || Pumpkins
```

The `NOT` operator is equivalent to the `-` syntax. So to find artists with `Smashing` but not `Atoms` in the name, you can do this:

```
Smashing NOT Atoms
```

We didn't need to specify a `+` on `Smashing`. This is because, as the only optional clause in the absence of mandatory clauses, it must match. Likewise, using an `AND` or `OR` would have no effect in this example.

It may be tempting to try to combine `AND` with `OR` such as:

~~`Smashing AND Pumpkins OR Green AND Day`~~

However, this doesn't work as you might expect. Remember that `AND` is equivalent to both sides of the operand being mandatory, and thus each of the four clauses becomes mandatory. Our data set returned no results for this query. In order to combine query clauses in some ways, you will need to use *sub-expressions*.

Sub-expressions (aka sub-queries)

You can use parenthesis to compose a query of smaller queries. The following example satisfies the intent of the previous example:

```
(Smashing AND Pumpkins) OR (Green AND Day)
```

Using what we know previously, this could also be written as:

```
(+Smashing +Pumpkins) (+Green +Day)
```

But this is not the same as:

```
+(Smashing Pumpkins) +(Green Day)
```

The sub-query above is interpreted as documents that must have a name with either `Smashing` or `Pumpkins` and either `Green` or `Day` in its name. So if there was a band named `Green Pumpkins`, then it would match. However, there isn't.

Limitations of prohibited clauses in sub-expressions

Lucene doesn't actually support a *pure negative query*, for example:

```
-Smashing -Pumpkins
```

Solr enhances Lucene to support this, but only at the top level query expression such as in the example above. Consider the following admittedly strange query:

```
Smashing (-Pumpkins)
```

This query attempts to ask the question: Which artist names contain either `Smashing` or do not contain `Pumpkins`? However, it doesn't work and only matches the first clause—(4 documents). The second clause should essentially match most documents resulting in a total for the query that is nearly every document. The artist named `Wild Pumpkins at Midnight` is the only one in my index that does not contain `Smashing` but does contain `Pumpkins`, and so this query should match every document *except* that one. To make this work, you have to take the sub-expression containing only negative clauses, and add the all-documents query clause: `*:*`, as shown below:

```
Smashing (-Pumpkins *:*)
```

Hopefully a future version of Solr will make this work-around unnecessary.

Field qualifier

To have a clause explicitly search a particular field, precede the relevant clause with the field's name, and then add a colon. Spaces may be used in-between, but that is generally not done.

```
a_member_name:Corgan
```

This matches bands containing a member with the name `Corgan`. To match, `Billy` and `Corgan`:

```
+a_member_name:Billy +a_member_name:Corgan
```

Or use this shortcut to match multiple words:

```
a_member_name:(+Billy +Corgan)
```

The content of the parenthesis is a sub-query, but with the default field being overridden to be `a_member_name`, instead of what the default field would be otherwise. By the way, we could have used `AND` instead of `+` of course. Moreover, in these examples, all of the searches were targeting the same field, but you can certainly match any combination of fields needed.

Phrase queries and term proximity

A clause may be a phrase query (a contiguous series of words to be matched in that order) instead of just one word at a time. In the previous examples, we've searched for text containing multiple words like `Billy` and `Corgan`, but let's say we wanted to match `Billy Corgan` (that is the two words adjacent to each other in that order). This further constrains the query. Double quotes are used to indicate a phrase query, as shown below:

```
"Billy Corgan"
```

Related to phrase queries is the notion of the term proximity, aka the **slop** factor or a **near** query. In our previous example, if we wanted to permit these words to be separated by no more than say three words in–between, then we could do this:

```
"Billy Corgan"~3
```

For the MusicBrainz data set, this is probably of little use. For larger text fields, this can be useful in improving search relevance. The `dismax` search handler, which is described in the next chapter, can automatically turn a user's query into a phrase query with a configured slop. However, before adding slop, you may want to gauge its impact on query performance.

Wildcard queries

A Lucene index fundamentally stores analyzed terms (words after lowercasing and other processing), and that is generally what you are searching for. However, if you really need to, you can search on partial words. But there are issues with this:

- No text analysis is performed on the search word. So if you want to find a word starting with `Sma`, then `Sma*` will find nothing but `sma*` will, assuming that typical text analysis like lowercasing is performed. Moreover, if the field that you want to use the wildcard query on is stemmed in the analysis, then `smashing*` would not find the original text `Smashing`, because the stemming process transforms this to `smash`. If you want to use wildcard queries, you may find yourself lowercasing the text before searching it to overcome that problem.

- Wildcard processing is much slower, especially if there is a leading wildcard, and it has hard-limits that are easy to reach if your data set is not very small. You should perform tests on your data set to see if this is going to be a problem or not. The reasons why this is slow are as follows:
    - Every term ever used in the field needs to be iterated over to see if it matches the wildcard pattern.
    - Every matched term is added to an internal query, which could grow to be large, but will fail if it attempts to grow larger than 1024 different terms.
- Leading wildcards are not enabled in Solr. If you are comfortable writing a little Java, then you can modify Solr's `QueryParser` or write your own and set `setAllowLeadingWildcard` to `true`.

> If you really need substring matches and on your data, then there is an advanced strategy discussed in the previous chapter involving what is known as N-Gram indexing.

To find artists containing words starting with `Smash`, you can do:

```
smash*
```

Or perhaps those starting with `sma` and ending with `ing`:

```
sma*ing
```

The asterisk matches any number of characters (perhaps none). You can also use `?` to force a match of any character at that position:

```
sma??*
```

That would match words that start with `sma` and that have at least two more characters but potentially more.

You can put a wildcard at the front, if you've enabled this with a bit of custom programming.

A nice thing about the wildcard matching is that the scoring is influenced by how close the indexed term is to the query pattern. So a word `Smash` might get a higher score than `Smashing` in the previous example. I say might because this is just one factor in the score.

Fuzzy queries

Fuzzy queries are useful when your search term needn't be an exact match, but the closer the better. The fewer the number of character insertions, deletions, or exchanges relative to the search term length, the better the score. The algorithm used is known as the Levenstein Distance algorithm. Fuzzy queries suffer from some of the same problems as the wildcard queries just described, but it is not as serious. For example:

```
Smashing~
```

Notice the tilde character at the end. Without this notation, simply `Smashing` would match only four documents because only that many artist names contain that word. `Smashing~` matched 578, words and it took my computer 706 milliseconds. You can modify the proximity threshold, which is a number between `0` and `1`, defaulting to `0.5`. For instance, changing the proximity to a more stringent `0.7`:

```
Smashing~0.7
```

Twenty-five matched documents resulted and it took 388 milliseconds. If you want to use fuzzy queries, then you should consider experimenting with different thresholds.

As with wildcard queries, fuzzy queries also influence the score so that closer matched terms generally score higher.

To illustrate how text analysis can still pose a problem, consider the search for:

```
SMASH~
```

There is an artist named `S.M.A.S.H.`, and our analysis configuration emits `smash` as a term. So `SMASH` would be a perfect match, but adding the tilde results in a search term in which every character is different due to the upper/lower case difference and so this search returns nothing. As with wildcard searches, if you intend on using fuzzy searches then you might want to consider lowercasing the query string.

Range queries

Lucene lets you query for numeric, date, and even text ranges. The following query matches all of the bands formed in the 1990s:

```
a_type:2 AND a_begin_date:[1990-01-01T00:00:00.000Z TO 1999-12-
31T24:59:99.999Z]
```

Observe that the date format is the full ISO-8601 date-time in GMT, which Solr mandates (the same format used by Solr to index dates and that which is emitted in search results). The fractional seconds part (milliseconds) is actually optional. The `[` and `]` brackets signify an inclusive range, and therefore it includes the dates on either end. To specify an exclusive range, use { and } but note that you can't mix and match, either the range is exclusive or inclusive.

Remember to use sortable numeric field types

In order to do numeric ranges, you must index the field with one of the sortable variations, such as `sint` or `sfloat`. The range query might return results, but it will most likely be incorrect.

For most numbers in the MusicBrainz schema, we have only identifiers, and so it made no sense to index them for sortability. There is more, but not much memory used internally for sortable fields. So, if there is a chance you might sort on it, then prefer the sortable variants. For our track count on the tracks data, we could do a query such as this to find all of the tracks that are longer than 5 minutes (300 seconds):

```
t_duration:[300000 TO *]
```

In this example, we can see Solr's support for *open-ended* range queries by using `*`. This feature is not in Lucene.

Although uncommon, you can also use range queries with text fields. For this to have any use, the field should have only one term indexed. You can control this either by using no analysis, or by using very little of it, perhaps with `keywordTokenizer`. You may want to do some experimentation. The following example finds all documents where somefield has a term starting with `B`. It also finds `C` by itself if it exists.

```
somefield:[B TO C]
```

Date math

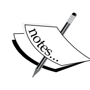
Solr extended Lucene with some date-time math that is especially useful in specifying date ranges. In addition, there is a way to specify the current date-time using `NOW`. The syntax offers addition, subtraction, and rounding at various levels of date granularity (years, seconds, and so on.) The operations can be chained together as needed, in which case they are executed from left to right. Spaces aren't allowed. For example:

```
r_event_date:[* TO NOW-2YEAR]
```

In the example above, we searched for documents where an album had a release date of before two years (two years before now), but not afterwards. `NOW` has millisecond precision. Let's say what we really wanted was precision to the day. By using / we can round down (it never rounds up):

```
r_event_date:[* TO NOW/DAY-2YEAR]
```

The units to choose from are: `YEAR`, `MONTH`, `DAY`, `DATE` (synonymous with `DAY`), `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`, and `MILLI` (synonymous with `MILLISECOND`). Furthermore, they can be pluralized by adding an `S` as in `YEARS`.



This so-called `DateMath` syntax is not just for querying dates, it is for supplying dates to be indexed by Solr too. When supplying dates to Solr for indexing, consider concatenating a rounding operation to a courser time granularity sufficient for your needs. Solr will evaluate the math and index the result. Full millisecond precision time take up more disk space and are slower to query than more course granularity times. Another index-time common usage is to timestamp added data. Using the `NOW` syntax as the default attribute of a timestamp field definition makes this easy.

Score boosting

You can easily modify the degree to which a clause in the query string contributes to the ultimate score by adding a multiplier. This is call **boosting**. A value between `0` and `1` reduces the score, and numbers greater than `1` increase it. Scoring details are described later in this chapter. In the following example, we search for artists (a band is a type of artist in MusicBrainz) that either have a member named `Billy`, or have a name containing the word `Smashing`.

```
a_member_name:Billy^2 OR Smashing
```

Here we search for artists named `Billy`, and either `Bob` or `Corgan`, but we're less interested in those that are also named `Corgan`:

```
+Billy Bob Corgan^0.7
```

Existence (and non-existence) queries

This is actually not a new syntax case, but an application of range queries. Suppose you wanted to match all of the documents that have a value in a field (whatever that value is, it doesn't matter). Here we find all of the documents that have `a_name`:

```
a_name:[* TO *]
```

As `a_name` is the default field, just `[* TO *]` will do.

This can be negated to find documents that *do not* have a value for `a_name`, as shown below:

```
-a_name:[* TO *]
```

Escaping special characters

The following characters are used by the query syntax, as described in this chapter:

`+ - && || ! ( ) { } [ ] ^ " ~ * ? : \`

In order to use any of these without their syntactical meaning, you need to escape them by a preceding \:

```
id:Artist\:11650
```

In some cases such as this one where the character is part of the text that is indexed, the double-quotes phrase query will also work, even though there is only one term:

```
id:"Artist:11650"
```

Filtering

Filtering in Solr is really quite simple. Let's say you are dispatching a user's query to Solr, but you want to limit the scope of that query further than what the query might be doing. As an example, let's say we wanted to make a search form for MusicBrainz that lets the user search for bands, not individual artists. Let's also say that the user's query string is `Green`. In the index, `a_type` is either `1` for an individual, `2` for a band, and `0` if unknown. Therefore, a clause that would find non-individuals would be this, combined with the user's query:

```
+Green +type:Artist -a_type:1
```

However, you should **not** use this approach.

Instead, use multiple `fq` query parameters, and leave the query string blank:

```
q=Green&fq=type%3AArtist&fq=-a_type%3A1
```

Remember that in the URL snippet above we needed to URL Encode special characters like the colons.

Filters:

- Improve performance, because each filter query is cached.
- Do not affect the scores of matched documents (nor would you want them to).

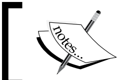
- Are easier to apply rather than modifying the user's query, which is error prone. Making a mistake could even expose data you are trying to hide (similar in spirit to SQL injection attacks).
- Clarify the logs, which show what the user queried for without it being confused with the filters.

In general, raw user query text doesn't wind up being part of a filter-query. Instead, the filters are usually known by your application in advance. Although it wouldn't necessarily be a problem for user query text to become a filter, there may be scalability issues if many unique filter queries end up being performed that don't get re-used and so consume needless memory.

Sorting

The sorting specification is specified with the `sort` query parameter. The default is to sort by score in a descending order. In order to sort in an ascending order, you would put this in the URL:

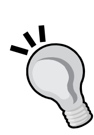
```
sort=score+asc
```



That was URL-encoded. Therefore, there is a + instead of a space. `%20` might have been used too.

In the following example, suppose we searched for artists that are not individuals (a previous example in the chapter), then we might want to ensure that those that are surely bands get top placement ahead of those that are unknown (2's then 0's). Secondly, we want the typical score descending search. This would simply be:

```
sort=a_type+desc,score+desc
```



Use the right field type/analysis!

Using the wrong field type or analysis configuration will not result in an error, just bad results! For sorting on numbers, you will want to use the sortable variants of the number types documented in the schema: `sint`, `slong`, `sfloat`, and `sdouble`. Dates are sortable, as are booleans. For sensible results with text, no tokenization should occur so that only one term gets indexed. Either don't do any text-analysis or use very little such as `KeywordTokenizer` with `LowerCaseFilterFactory`. You may need to copy the field to another, explicitly for sorting purposes.

Request handlers

Querying Solr, and most other interactions with Solr including indexing for that matter, is processed by what Solr calls a **request handler**. Request handlers are configured in the `solrconfig.xml` file and are clearly labeled as such. Most of them exist for special purposes like handling a CSV import, for example. Our searches in this chapter have been directed to a request handler labeled `standard` due to the presence of `qt=standard` in the URL. It happens to be the default that `/select` chooses, if not specified. Here is how this query handler is configured:

```
<requestHandler name="standard" class="solr.SearchHandler"
                default="true">
  <!-- default values for query parameters -->
    <lst name="defaults">
    <str name="echoParams">explicit</str>
    <!--
    <int name="rows">10</int>
    <str name="fl">*</str>
    <str name="version">2.1</str>
    -->
    </lst>
</requestHandler>
```

The request handlers that perform searches allow configuration of two things:

- Establishing default parameters and making some unchangeable
- Registering **Solr search components** such as faceting and highlighting

Create a request handler configuration for your application.

Instead of using the `standard` request handler for use by the application you are building, it is a good idea to create a request handler just for your application, perhaps even several to satisfy multiple search forms. In doing so, you can change various search aspects more easily in Solr through re-configuration, instead of having your application hard-wired more than it has to be. This centralizes the search configuration a bit more too.

By copying the standard configuration, removing the `default` boolean setting, and giving it a name such as `mb_artists`, we can now use this request handler with `qt=mb_artists` in the URL as shown below:

```
/solr/select?qt=mb_artists&q=Smashing&.....
```

An alternative to this is to precede the name with / in your configuration. Now this handler is invoked like this:

```
/solr/mb_artists&q=Smashing&.....
```

Let's now configure this request handler to filter searches to find only artists, without the querying application having to specify this. We'll also set few other options.

```
<requestHandler name="mb_artists" class="solr.SearchHandler">
    <lst name="defaults">
      <str name="echoParams">none</str>
      <int name="rows">20</int>
    </lst>
    <lst name="appends">
      <str name="fq">type:Artist</str>
    </lst>
    <lst name="invariants">
      <str name="facet">false</str>
    </lst>
</requestHandler>
```

Request handlers have several lists to configure. These use Solr's Generic XML data structure, which was described earlier.

- `defaults`: These simply establish default values for various request parameters. The client (aka the application or user) is free to change them.
- `appends`: For parameters that can be set multiple times (like `fq`), this section specifies values that will be set in addition to any that may be specified by the client.
- `invariants`: Sets defaults that cannot be overridden. This is useful for security purposes.
- `first-components`, `components`, `last-components`: These list the Solr search components to be registered for possible use with this request handler. By default, a set of search components are already registered to enable functionality such as querying and faceting. Setting `first-components` or `last-components` would prepend or append to this list respectively, whereas setting `components` would override the list completely. For more information about search components, read Chapter 6.

Scoring

Scoring in Lucene is an advanced subject, but in spite of this it is important to at least have a basic understanding of it. Instead of presenting the algorithm, comprehension of which is a bit of an advanced subject and not suitable for this chapter or even this book, we will discuss the factors influencing the score and where to look for diagnostic scoring information. If this overview is insufficient for your interest, then you can get the full details here at `http://lucene.apache.org/java/2_4_1/api/org/apache/lucene/search/Similarity.html`

Before some of these factors are described, keep in mind that no matter how complicated a query might be, they fundamentally boil down to a combination of term lookups of indexed terms, and this is on a field-basis (not entire documents).

An important thing to understand about scores is not to attribute much meaning to a score by itself; it's *almost* meaningless. The relative value to the max score is more relevant. A document scored as 0.25 might be a great match or not, there's no telling. But if you compare this score to another from the very same search and find it to be twice as large, then it is fair to say that the document matched the query twice as well. This being said, you will usually find that scores in the vicinity of 0.5 or better are decent matches. The factors influencing the score are as follows:

- **Term Frequency**—`tf`: The more times a term is found in a document's field, the higher the score it gets. This concept is most intuitive. Obviously, it doesn't matter how many times the term may appear in some other field, it's the searched field that is relevant (whether explicitly targeted or the default).
- **Inverse Document Frequency**—`idf`: The document frequency is the number of documents in which the term appears (on a per-field basis, of course). It is the *inverse* of the document frequency that is positively correlated with the score. In short, rare terms result in higher scores.
- **Co-ordination Factor**—`coord:` The greater the number of queried terms match, the greater the score—all things being equal otherwise. Any mandatory clauses must match and the prohibited ones must not match, leaving the relevance of this piece of the score to situations where there are multiple optional clauses.
- **Field Length**—`fieldNorm`: The shorter the matching field is (measured in number of indexed terms), the greater the matching document's score will be. For example, if there was a band named just `Smashing`, and another named `Smashing Pumpkins`, then this factor in the scoring would be higher for the first band upon a search for just `Smashing`, as it has one word, while the other has two. Remember that norms can be omitted from some fields in the schema configuration.

Query-time and index-time boosting

At index-time, you have the option to boost a particular document (entirely or just a field). This is internally stored as part of the `norms` number, which must be enabled for this to work. It's uncommon to perform index-time boosting.

At query-time, we have described earlier how to boost a particular clause of a query higher or lower if needed. Later the powerful **Disjunction-Max** (`dismax` for short) query will be demonstrated, which can apply searches to multiple fields with different boosting levels automatically.

Troubleshooting scoring

An invaluable tool in diagnosing scoring behavior is enabling query debugging with the `debugQuery` query parameter. There is no better way to describe it than with an example. Consider the fuzzy query:

```
a_name:Smashing~
```

We would intuitively expect that documents with fields containing `Smashing` would get the top scores, but that didn't happen. Execute the query mentioned above with `debugQuery=on`, and ensure that you're looking at the original indentation by using the **View Source** feature in your browser. Try right-clicking the XML to see the option.

The top score is `2.844`, and there were two documents matching, neither with `Smashing`. One had `Mashina`, and the other had `Smashin'`.

```
<doc>
  <float name="score">2.8440614</float>
  <str name="a_name">Mashina</str>
</doc>
<doc>
  <float name="score">2.8440614</float>
  <str name="a_name">Smashin'</str>
</doc>
<doc>
  <float name="score">2.72019</float>
  <str name="a_name">Smashing Atoms</str>
</doc>
```

Both the first two documents have words that differ from `smashing` by only two characters (remember the case difference). The third document finally matched `Smashing`. Its score was a little less, but not enough to overtake the top two. What's going on here? Let's look at the following debug output, showing the first and the third document. We'll skip the second, as it has the same score as the first:

```
<lst name="explain">
  <str name="Artist:227132">
2.8440614 = (MATCH) sum of:
  2.8440614 = (MATCH) weight(a_name:mashina^0.42857146 in 890035),
  product of:
    0.20135471 = queryWeight(a_name:mashina^0.42857146),
    product of:
      0.42857146 = boost
      14.124633 = idf(docFreq=1, numDocs=1002272)
      0.033262998 = queryNorm
    14.124633 = (MATCH) fieldWeight(a_name:mashina in 890035),
    product of:
      1.0 = tf(termFreq(a_name:mashina)=1)
      14.124633 = idf(docFreq=1, numDocs=1002272)
      1.0 = fieldNorm(field=a_name, doc=890035)
  </str>
<!-- skip 2nd doc ...-->
  <str name="Artist:93855">
2.72019 = (MATCH) sum of:
  2.72019 = (MATCH) weight(a_name:smashing^0.75 in 612886),
  product of:
    0.32951176 = queryWeight(a_name:smashing^0.75),
    product of:
      0.75 = boost
      13.208342 = idf(docFreq=4, numDocs=1002272)
      0.033262998 = queryNorm
  8.255214 = (MATCH) fieldWeight(a_name:smashing in 612886),
  product of:
    1.0 = tf(termFreq(a_name:smashing)=1)
    13.208342 = idf(docFreq=4, numDocs=1002272)
    0.625 = fieldNorm(field=a_name, doc=612886)
  </str>
```

What we see here is the mathematical breakdown of the various components of the score. We see that `mashina` (the term actually in the index) was given a query-time boost of `0.43`, whereas `smashing` was given a query-time boost of `0.75`. We expected this because the fuzzy matching was going to give higher weights to stronger matches, and it did. However, other factors pulled the final score in the other direction. Notice that the `fieldNorm` for `mashina` was `1.0` whereas `smashing` had a `fieldNorm` of `0.625`. This is because the document we wanted to score higher had a field with more indexed terms (`Smashing Atoms`) versus just the one that `Mashina` had. So arguably, `Mashina` is a closer match than `Smashing Atoms` to the fuzzy query `Smashing~`.

Summary

At this point, you've learned the basics of searching in Solr, from query parameters to interpreting the search results to nearly the full gamut of Solr's query syntax to the essential factors of scoring. We have spent a lot of time on the query syntax because you'll see the syntax pop-up in several places across Solr, not just the user's query. Such places include filtering the query, deleting by query, and query warming (discussed in later chapters). Even if you don't wish to expose the syntax to your users, you will probably be using it for various things. In the next chapter, you'll learn more about querying, notably advanced function queries, and the ever-useful `Dismax` query handler. You'll also learn about faceting—arguably the most important Solr search component beyond searching itself.

5
Enhanced Searching

So we've got the searching basics down, and we even know a thing or two about more advanced topics like scoring. In this chapter, we'll extend the searching topic into more advanced features of Solr's searching capabilities, such as:

- function queries
- the Dismax query handler
- faceting

Function queries

A function query allows you to introduce a *component* of the score that is computed, based on a mathematical expression of your choice involving indexed field value(s). This is not a replacement for Lucene's scoring algorithm, but it basically adds to the existing score.

A bad name for this feature

The name of this feature is poor as it does not reflect what it does. Perhaps **Scoring Function** or even **Score Query** might have been better. The reason for the name **Function Query** undoubtedly stems from the manner in which the feature is implemented. It is implemented as a Lucene Query type, albeit a very strange one that matches all of the documents but scores them differently.

There are two ways in which you can incorporate a function query into your searches in Solr:

- with the standard request handler using the `_val_` pseudo-field hack.
- with the dismax request handler using the `bf` parameter.

The latter is cleaner, and you'll probably use that approach. The dismax handler and this parameter is described a little later in this chapter. Given next is an example of the `_val_` trick with the standard request handler (the one we've been using so far):

```
t_name:Daydreaming && _val_:"t_trm_lookups"^0.01
```

- The first query clause is a search of tracks containing `Daydreaming`. The second clause is the function query clause. The actual function query is simply `t_trm_lookups` with a boost of `0.01`. This query and more complicated ones involving functions will be explained shortly.
- The use of `_val_` suggests that there is a field by this name, but there isn't. This is a hack so that the existing Lucene syntax can be used.
- The quotes around the function query are used to escape the function query from potential syntactical conflicts with Lucene's query syntax. These characters are listed in the last chapter. This example did not need to be escaped but it's a good habit to do so. Additionally, there cannot be any spaces within the function query.
- Lucene's syntax doesn't mandate specifying a boost, but with function queries you will most likely have one.
- We've made both the non-function query clause and the function query clause mandatory. Had we not done this, then every document would have matched the query, as a function query matches all documents.

An example: Scores influenced by a lookupcount

Let's consider an extremely simple case of a schema with a field containing a popularity number. In the MusicBrainz schema, there is actually something close—a TRM and PUID lookupcount. TRM and PUID are MusicBrainz's audio fingerprint technologies. These identifiers roughly correspond to a song, which in MusicBrainz appears as multiple tracks due to various releases that occur as singles, compilations, and so on. By the way, audio fingerprints aren't perfect, and so a very small percentage of TRM IDs and PUIDs refer to songs that are completely different. We're only using this to influence scoring so this isn't a problem for this example.

MusicBrainz records the number of times one of these IDs are looked up from its servers, which is a good measure of popularity. A track that contains a higher lookupcount should score higher than one with a smaller value, with all other factors being equal. This scheme could easily be aggregated to releases and artists, if desired. I've arranged for the sum of TRM and PUID lookupcounts being stored into our track data as `t_trm_lookups` with the following field specification in the schema:

```
<field name="t_trm_lookups" type="sint" />
```

About 25% of the tracks have a non-zero value.

When I search my MusicBrainz track index with the `Daydreaming` search example and with `debugQuery` set to `on`, I see that the top result has the following scoring explanation:

```
<str name="Track:10">
26.898853 = (MATCH) sum of:
   12.039736 = (MATCH) weight(t_name:daydreaming in 2412430),
   product of:
         0.99999964 = queryWeight(t_name:daydreaming), product of:
              12.039741 = idf(docFreq=111, numDocs=6977765)
              0.08305824 = queryNorm
         12.039741 = (MATCH) fieldWeight(t_name:daydreaming in
         2412430), product of:
             1.0 = tf(termFreq(t_name:daydreaming)=1)
             12.039741 = idf(docFreq=111, numDocs=6977765)
             1.0 = fieldNorm(field=t_name, doc=2412430)
   14.859118 = (MATCH) FunctionQuery(sint(t_trm_lookups)),
   product of:
         17890.0 = sint(t_trm_lookups)=17890
         0.01 = boost
         0.08305824 = queryNorm
</str>
```

The first half of this text shows that `daydreaming` accounted for a score of ~`12` (nearly half of the total score of `26`). The latter half shows that our function query added `14` to the score. The interesting part is highlighted as this is the result of the function query, except for the boost. As this function query is merely a field reference, its score is the value in this field for this document, which is `17890`.

> If you want to simply sort search results by a field value, then you would use the `sort` parameter described in the last chapter.

Field references

For fields used in a function query, keep the following points in mind:

- The field must be indexed.
- The field must not be multi-valued.
- No more than a single term/token may be indexed from text-analysis. This isn't a problem for numeric and date data types, but it is something to watch out for with text. Consider using the `KeywordTokenizer`, and beware of analysis steps that may introduce extra terms.
- Only number fields have their values referenced directly by the functions in the next sections. For date and text fields, the `ord()` function is implicitly used (explained shortly). If this poses a problem for how you wish to use dates, then you should index the date as a number.
- If there is no value for a field in the index, then zero is substituted for use in the function query.

Function reference

Instead of using a *naked* field reference, you will probably use a more elaborate mathematical expression composed of constants, field references, and functions (both basic mathematical ones and special Solr additions). Constants and field-references are used literally. An example involving a sampling of these is shown below:

```
div(log(t_trm_lookups),log(2))
```

This example and subsequent ones show just the function query itself, and not surrounding syntax for potential use in `_val_` or with the `bf` parameter, nor with boosts. In all cases, remember to omit spaces. One thing to observe from this example is that `div`, short for **divide**, is expressed as a function instead of using the common / operator. There are **no** operators in Solr function query expressions, just function calls and either constant or field reference literals.

Function argument limitations

For the functions listed as follows, any argument named `x`, `y`, or `z` can be any expression: constants, field references, or functions. Other arguments like `a`, or `min` *require* a constant. If you attempt to do otherwise, then you will get an unhelpful parsing error.

Mathematical primitives

The mathematical primitives are explained as follows:

- `sum(x,y,z,...)`: Sums, that is adds, all of the arguments. The argument count is variable. Note that if you need to subtract, then you'll have to use this with a negative argument.
- `product(x,y,z,...)`: Multiplies the arguments together. The argument count is variable.
- `div(x,y)`: Divides x by y as in the expression x/y.
- `pow(x,y)`: Raises x to the power y, as in the expression x^y.
- `abs(x)`: The absolute value of x, that is $|x|$. If x is negative, then it becomes positive.
- `log(x)`: The logarithm, base 10, of x. Remember that if you need another base such as 2, example, $log_2 x$, then this is equivalent to: *div(log(myfield),log(2))*.
- `sqrt(x)`: The square root of x, that is $\sqrt{x}$.

Miscellaneous math

- `map(x,min,max,target)`: If `x` is found to be between `min` and `max` inclusive, then the target is returned, otherwise `x` is returned. This is useful for dealing with default values or to limit `x`, to ensure that it isn't above or below some threshold.
- `max(x,c)`: Returns the greater value of `x` and `c`.
- `scale(x,minTarget,maxTarget)`: Returns `x` scaled to be between `minTarget` and `maxTarget`. For example, if the value of `x` is found to be one-third from the smallest and largest values of `x` across all documents, then `x` is returned as one-third of the distance between `minTarget` and `maxTarget`. The important limitations in Solr 1.3 are as follows:
    - `scale` will traverse the entire document set and evaluate the function to determine the smallest and largest values for each query invocation, and it is not cached. This makes it impractical for many uses as it is too slow.
    - `scale` will return `zero` if it is evaluated on a deleted document. Consider using the map function for `x` such as `scale(map(myfield,0,2),1,2)`.

- `linear(x,m,c)`: A macro for `sum(product(m,x),c)` for convenience and speed.
- `recip(x,m,a,c)`: A macro for `div(a,linear(x,m,c))` for convenience and speed.

ord and rord

Use of `ord()` and `rord()` are to be performed on fields that are indexed appropriately for sorting, even if the search results won't be sorted on this field. See Chapter 2 for information on field sorting requirements.

Date and text fields are implicitly referenced through `ord()`, if not explicitly referenced in `ord()` or `rord()`. You will see evidence of this in the debug output.

- `ord(fieldReference)`: Given a hypothetical ascending sorted array of all unique indexed values for `fieldReference`, this returns the array position, for example, the ordinal of a document's indexed value. `fieldReference` is of course a reference to a field. Unlike the other functions it cannot be any other kind of expression. The order of the values is in an ascending order and the first position is `1`. A non-existent value results in `0`.
- `rord(fieldReference)`: The same as `ord()`, but with the ordering reversed.

A definition of `ord` is not sufficient to fully convey its ramification. Suppose five documents are added to an index with the following values in a field `x`: `70`, `70`, `90`, `95`, `98`. Even though there are five documents, `ord(x)` is going to return values ranging from `1` to `4`, because there are only four distinct values; one of them, `70`, is repeated. There is another difference that is more subtle. The original values are not distributed in a linear fashion. They are more clumped together towards the higher values (do not consider duplicates). `ord` and `rord` in-effect linearizes the data so that the distribution of the original value is lost, assuming it was non-linear. If you are using dates and wish to avoid `ord`/`rord` because of this problem, then you can encode the date as a number in a numeric field, such as the number of days since an epoch.

To determine how high `ord`/`rord` can get, you can use Solr's web admin interface. Go to the **Schema Browser**. Click on an **indexed** field, and observe the **distinct** number.

An example with scale() and lookupcount

In our last example, the function query was simply a field's value, and we scaled it down with a boost so that it wouldn't overpower the other components of the score. It created a simple example. However, using a field value directly like this can be problematic. If the field values lie within a fixed range, then this approach would be fine, but it turns out that `t_trm_lookups` is as high as ~300,000, and it will increase steadily as time goes on. A further inspection of the data reveals, that only a handful of records exceed 90,000. We can use an even smaller fractional boost, but whatever we pick would need frequent adjustment as the data changes.

One option is to use the `scale` function such as this:

```
scale(t_trm_lookups,0,100)
```

I used a boost of `10` with this and was happy with the scores. The debug output for the function query was as follows:

```
6.069365 = scale(sint(t_trm_lookups)=17890,
          toMin=0.0,toMax=100.0,fromMin=0.0,fromMax=294759.0)
```

Here we see that the values in the index ranged from 0 to 294,759. Interesting output can be gleaned when enabling debugging.

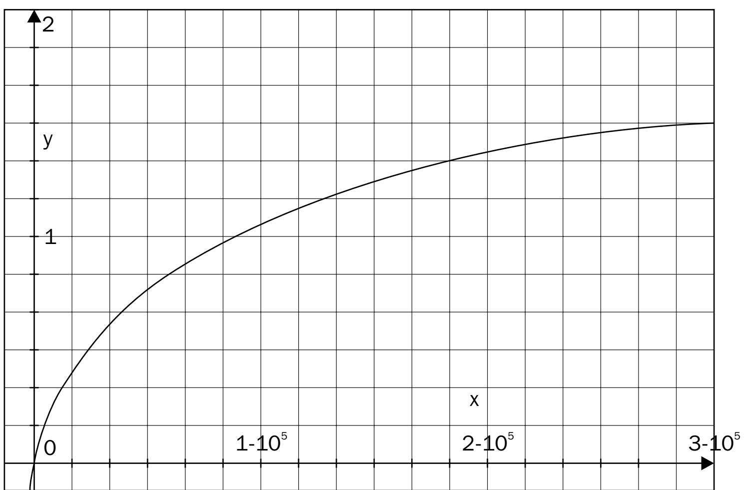
Due to problems with the scale function that were previously noted, it took about five seconds for that search. Perhaps a future version of Solr will address this. It's also very sensitive to the maximum value, which I've found to be a bit of a fluke with this data. A more accurate maximum value is 90,000, not 300,000. So read on for alternatives.

Using logarithms

Let's try an alternative approach that does not give the same scores but has desirable characteristics. If we use a logarithm, then we would have numbers that might not be capped at a value of our choosing, but it would at least have its growth stunted dramatically where we want. With a good deal of experimentation, I finally found the formula I was looking for:

$$\text{Log}_c\left((c-1)\cdot m\cdot x+1\right)$$

`m` is the inverse of the `t_trm_lookups` value where we want this function to evaluate to 1: 1/90,000. `c` is a number greater than one and is a value of your choosing that will alter how the curve below bends. I used `10`, which turned out to closely match the approach shown below:



Here is the formula as a function query:

```
log(linear(t_trm_lookups,0.0001,1))
```

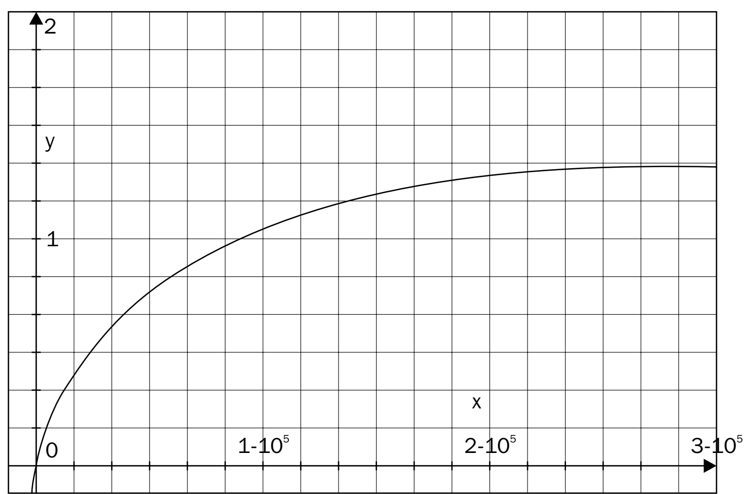
If I used a value for `c` other than `10`, I would need to divide this function by `log(c)`. I found a boost of 100 to work well. This search only took my machine 12 milliseconds. *Nice!* However, note that this curve keeps growing steadily as `t_trm_lookups` increases. One way to deal with this is by replacing `t_trm_lookups` in the function with `map(t_trm_lookups,100000,99999999,100000)`.

Using inverse reciprocals

Another way that is more elegant, is to use the reciprocal of a linear function instead of the logarithm. Here is an equation I devised for this:

$$\frac{-max^2 + max}{m \cdot x + max - 1} + max$$

Here, max is the value that this function approaches but never quite reaches. It should be greater than 1 and less than 2; 1.5 works well. You can experiment with this to see how it changes the bend in the curve below. m is the inverse of the maximum value that we can expect t_trm_lookups to reasonably be: 1/90,000. The function is plotted as shown below:



A simplified function query for this is as follows:

```
sum(recip(t_trm_lookups,0.0000111,-0.75,0.5),1.5)
```

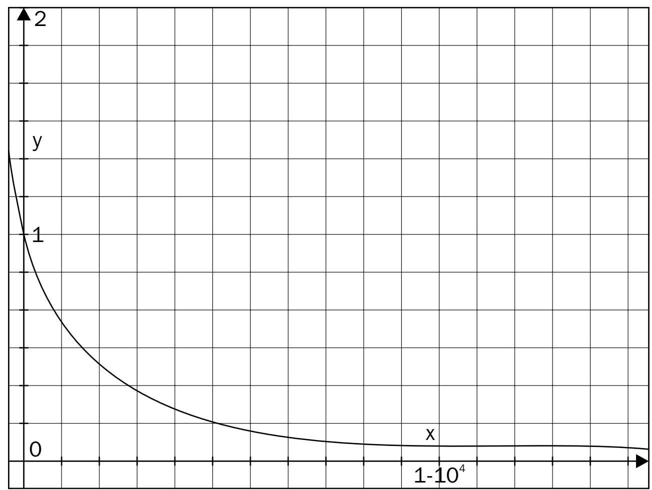
I used a boost of 100 and the results are as expected and are similar to the logarithm based function that was mentioned previously. However, if high values of t_trm_lookups were to some day wind up in the index, then this piece of the score computation would not exceed 1.5. In practice, the logarithmic approach would probably be fine too.

A problem with both approaches mentioned above is that the constants would ideally need modification from time to time to reflect the highest value in the index (less extreme outliers).

Using reciprocals and rord with dates

Using dates in scores presents some different issues. Suppose when we search for releases, we want to include a boost that is larger for more recent releases. At first glance, this problem may seem just like the previous one, because dates increase as the scores are expected to, but it is different in practice. Instead of the data ranging from zero to some value that changes occasionally, we now have data ranging from a non-zero value that might change rarely to a value that we always know but changes continuously (the current date). Moreover, ideally the score curve would bend on the oppose side of the linear function than the previous functions did. Instead, approach this from the other side, that is, by considering how much time there is between the current time and the document's date. So at `x=0` in the graph (`x` representing time), we want `1`, and we want it to slope downward towards `0`, but not below it. Here's an equation with this pattern:

$$\frac{c}{x + c}$$



Coming up with a value for `c` will take a little experimentation, but a decent starting point is 10 times the greatest value of `x`. Our MusicBrainz schema has `r_event_date`, which is a promising candidate for `x`. However, multi-value is not supported by function queries. I made a simple addition to the schema and index to record the earliest release event date: `r_event_date_earliest`. Here is a function query that should work well:

```
recip(map(rord(r_event_date_earliest),0,0,99000),1,95000,95000)
```

This example uses `rord`, and therefore the greatest indexed date will have the value `1`, the one before that `2`, and so on to about `9500`.

Since some of the releases have no value and `rord` returns `0`, we needed to map these to a number on the other side. I used 99,000. Alternatively, I might fix this by substituting an alternative estimate, based on other known dates at index-time so that it's never blank. This is probably a better approach.

> `~40` releases or so in MusicBrainz have release dates before my data snapshot was taken. Since this is a very small fraction, I'm ignoring this idiosyncrasy. For scoring purposes, at index-time I should pivot such dates in the opposite direction of the current date (at index-time) so that a month in the future becomes a month in the past.

There are `~9500` unique discrete dates indexed. If this number changes substantially, then the constants above would need to be updated too. One potential way to deal with that is to first store the dates with a lower precision (monthly) to linearize most of the data, and then cap the values beyond a threshold using the map function.
As a reminder, month rounding is easily accomplished at the time of indexing by concatenating `/MONTH` at the end of the date format. This is part of Solr's `date math` syntax that was mentioned in Chapter 4. By lowering the precision, we ensure that all/most months for quite some time have a release date occurring in it. This increases the accuracy of `rord(r_event_date_earliest)` being a months-in-the-past index for a substantial duration of time. If there is a decade of such data with few gaps, then the query function would be:

```
recip(map(map(rord(r_event_date_earliest),1201,999999,1201)
      ,0,0,2000),1,1200,1200)
```

> If you insist on using the exact difference between an indexed date and the current date (and thus not using `ord`/`rord`), then you're forced to store the dates as numbers (such as the number of days since an epoch). The reason is that there is no way to use date literals or a way to reference **now** since actual date values cannot be referenced numerically in a function query. This is a shortcoming of Solr.

Function query tips

Here are some tips on using function queries:

- Use a tool such as a graphing calculator or other software to plot the functions as you devise what your function will look like. If you are using Mac OS X as I am, then your computer already includes `Grapher`, which generated the charts in this chapter. It's a powerful tool. You might be inclined to use a spreadsheet like Excel, but that's really not the right tool. With luck, you may find some web sites that will suffice.
- Aim for getting the function to generate values between `0` to `~1`, or reversed if going the other direction. You'll then use a boost to scale it up. This approach makes your function queries more comparable by using a common baseline, as the values will fit within the same range.
- If your data changes in ways causing you to alter the constants in your function queries, then consider implementing a periodic automated test of your Solr data to ensure that the data fits within expected bounds. A **Continuous Integration (CI)** server might be configured to do this task. An approach is to run a search simply sorting by the data field in question to get the highest or lowest value.
- As you tweak the boost, you'll want to look at the proportion of the function query's score contribution (which includes the raw function query, the boost, and the queryNorm) relative to the total score. You'll most likely want this component of the score to be small so that the other factors of the score are more prominent. Even if you use a small boost, such that the function query's score seems negligible, then it still serves as a tie breaker when the other factors are equal.

Dismax Solr request handler

Solr's **Search Request Handler** is intuitively the request handler used for searching. An important early step involved in handling search requests is parsing the user's query string into a Lucene Query object. This step is handled by a Solr `query parser plugin`. The search handler allows choosing this plugin through the `defType` parameter, which defaults to lucene. Since we have not been setting the `defType` parameter (it's not used in the query form), we've been using this one so far. I'm now going to introduce you to the **dismax request handler**, which is the search request handler, configured with `defType` set to `dismax`. The dismax request handler is actually deprecated but it is commonly referred to this way when it would be more accurate to just say "dismax" or "dismax query parser". Similarly, it is common to refer to the "standard request handler" when it's the lucene query parser in particular that is being referenced. I admit that this book uses both interchangeably. There are other query parser plugins used for special purposes too, by the way.

The standard query parser we've been using so far for searching is fairly basic with no frills. A notable problem with parsing user queries directly with it is that the query must be well formed according to the syntax rules of the previous chapter.

> The dismax query parser has many more features and is intended to be the ideal choice for processing a query string from a user.

The dismax handler has the following features over the standard handler:

- Searches across multiple fields with different boosts through Lucene's `DisjunctionMaxQuery`.
- Limits the query syntax to a small subset and there is never a syntax error. This feature is not optional or configurable.
- Automatic phrase boosting of the entire search query.
- Convenient query boosting parameters, generally for use with function queries.
- Can specify the minimum number of words to match, depending on the number of words in a query string.

> **Use `debugQuery` to see the effects of this query handler**
> This dismax query handler essentially creates a new query, based on the user query and its configuration options. By enabling Solr's `debugQuery` option, which was described in Chapter 4, you can see what the resulting query is.

These features will subsequently be described in greater detail. But first, let's take a look at a search handler I've set up for searching artists. Solr configuration that is not related with the schema is located in `solrconfig.xml`.

```
<requestHandler name="mb_artists" class="solr.SearchHandler">
    <lst name="defaults">
        <str name="defType">dismax</str>
        <str name="qf">a_name a_alias^0.8 a_member_name^0.4</str>
        <str name="tie">0.1</str>
    </lst>
</requestHandler>
```

In Solr's full search interface screen, we can refer to this with a **Query Type** of `mb_artists`. This value aligns with the `qt` parameter, which you will observe in the URL when you submit the form. It wasn't necessary to set up such a request handler, because Solr is fully configurable from a URL, but it's a good practice and is convenient for Solr's search form.

Lucene's DisjunctionMaxQuery

The ability to search across multiple fields with different boosts in this query handler is a feature powered by Lucene's **DisjunctionMaxQuery** query type. In Solr you must use this query handler to use this feature, because it is not a capability that you can tap into with Solr's query syntax. Let's start with an example. If the query string is simply `rock`, then a dismax query handler might be configured to turn this into a `DisjunctionMaxQuery` similar to this:

```
fieldA:rock^2 OR fieldB:rock^1.2 OR fieldC:rock^0.5
```

Advanced topic warning

The following discussion is advanced, and you needn't understand it. Just know that a dismax query is ideal for searching multiple fields and to set the `tie` parameter to `0.1`, which is a reasonable value.

The boolean query mentioned above is not quite equivalent to what the dismax query actually does. The difference is in the scoring. A boolean query, such as this, will have a score based on the *sum* of each of the three clauses whereas a `DisjunctionMaxQuery` takes the *maximum* of each (this is a simplification). The dismax behavior should produce better scores for this use case, which is where you are looking in multiple fields for the same term, where some fields are deemed to be more significant than others. An example from the **Javadocs** of this feature explains that if a user searched for `albino elephant`, then dismax ensures that `albino` matching one field and `elephant` matching another gets a higher score than `albino` matching both fields but `elephant` neither.

Another wrinkle on this description of dismax scoring is the `tie` parameter, which is between zero (the default) and one. By raising this value above zero, the scoring begins to favor documents that matched multiple terms over those that were boosted higher. This can be moved to the extreme of one results in scoring that is closer to that of a boolean query, but it is not quite the same. In practice, a small value like `0.1` is effective.

Configuring queried fields and boosts

You use the qf parameter to tell the dismax query handler which fields you want to be searched and their corresponding boosts. As explained in the last chapter, the query parameters can be specified in the URL or in the search handler configuration in solrconfig.xml (you'll probably choose the latter). Here is the relevant configuration line from our dismax based handler configuration earlier:

```
<str name="qf">a_name a_alias^0.8 a_member_name^0.4</str>
```

This syntax is a space-separated list of field names that can have optional boosts applied to them using the same syntax for boosting that is used in the query syntax. This query handler is intended to find artists (which include bands) from a user's query. Such a query would ideally match the artist's name, but we'll also search aliases, and as a last resort find bands that the artist is a member of. Perhaps the user didn't recall the band name but knew the artist's name. This configuration would give them the band in the search results, most likely towards the end of the search results.

> Remember that the boosting does not strictly order the results in a cascading fashion. An exact match in a_alias that matched only part of a_name will probably appear on top. If in your application you are matching identifiers of some sort, then you may want to give a boost to that field that is very high, such as 1000, to virtually assure it will be on top.

Limited query syntax

The dismax query handler intentionally restricts the syntax permitted to terms, phrases, and uses + and - (but not AND, OR, &&, ||) to make a clause mandatory or prohibited. Anything else is escaped if needed to ensure that the underlying query is valid, but you will never get a syntax error. Therefore, attempting to use Solr's full syntax such as targeting a field, example: a_name:Corgan, will be escaped to a_name\:Corgan, and it wouldn't find anything because we have no such data. Imagine a large text field that has some URLs, and a user comes along and searches for one of these URLs with this query: http://lucene.apache.org. Such a query would certainly result in an error in the standard search handler, because there is no field in the schema named http. With the dismax handler it will be escaped, and it will find documents containing that URL.

The following query example uses all of the supported features:

```
"a phrase query" plus +mandatory without -prohibited
```

The limited syntax is a good feature for processing end user queries, because we want to search for what the user typed, and not return an error if the syntax is incorrect, which is almost certainly unintentional.

Don't want a limited syntax?

If you want to enable power-users to use boolean logic and other capabilities that Solr offers (covered in Chapter 4), then you can't use the dismax query handler. This is unfortunate, because the alternatives are not ideal. You basically have two choices: Use the standard query handler instead, and not be able to easily search multiple fields anymore, or use the other nice features of dismax. If you are comfortable with hacking Solr's Java code, then you can change `DisMaxQParserPlugin.java` to not limit the syntax. This is what I have done for an application. See SOLR-758.

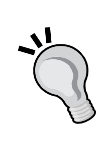
Boosting: Automatic phrase boosting

Suppose a user searches for `Billy Joel`. This is interpreted as two terms to search, and depending on how the search handler is configured, either both must be found in the document or just one. Perhaps for one of the matching documents, `Billy` is the sole name of a band, and it has a member named `Joel`. Great, Solr found this document and perhaps it is of interest to the user, after all, it contained both words the user typed. However, it's a fairly intuitive observation that a document field containing the entirety of what the user typed, `Billy Joel`, represents a closer match to what the user is looking for. Such a document would certainly be found by Solr too, without question, but it's hard to predict what the relative scoring might be. To improve the scoring, you might be tempted to automatically quote the user's query, but that would omit documents that don't have the adjacent words. What the dismax handler can do is add a phrased version of the user's query onto the original query with optional, that is, "should", semantics. So, in a nutshell, it turns this query:

```
Billy Joel
```

into

```
+(Billy Joel) "Billy Joel"
```



But remember, there is no query syntax to invoke Lucene's `DisjunctionMaxQuery`, which is targeting multiple fields, as discussed earlier. Putting the semantics of that aside, the syntax is accurate. It depicts that the original query is mandatory by using `+`, and it shows that we've added an optional phrase to search. It's optional because that is generally the default state in the absence of `+`, `-`, and other boolean operators that were discussed previously.

This works because a document containing the phrase Billy Joel not only matches that clause of the rewritten query, but it also matches Billy and Joel—three clauses in total. If in another document the phrase didn't match, but it had both words, then only two clauses would match. Lucene's scoring algorithm would give a higher coordination factor to the first document, and would score it higher, all other factors being equal—which they never are, but I digress.

Configuring automatic phrase boosting

Automatic phrase boosting is not enabled by default. In order to use this feature, you must use the pf parameter, which is an abbreviation of phrase fields. The syntax is identical to bf, and you might very well use the same value for both parameters. However, you have the option to change it. Here are some reasons to vary bf from pf:

- To use different boost factors so that you lower or raise the impact of phrase boosting. Some experimentation may guide you to make such adjustments.
- To omit fields that are always one term, such as an identifier, because there's no point in searching the field for phrases.
- To substitute a field for another that has the same data but analyzed differently. The main use case for this is **shingling**, which speeds up phrase queries on large data sets. Shingling is an advanced technique described in Chapter 9.
- To have some fields match only in their entirety.

For an example of the last bullet, imagine that our documents had a category field of some kind, perhaps a music genre. A sample genre is Hip Hop. The genre could be indexed so that it is one token, similar to that done for sorting. Again, this happens when no text analysis is performed or when such analysis is configured to have this effect such as by using KeywordTokenizerFactory. So if someone searches for Hip Hop then they get all hip-hop genre music, but if they just have hip or hop, then it won't match the genre. Here is a sample configuration in solrconfig.xml for our mb_artists handler with the fictitious genre addition:

```
<str name="pf">a_name a_alias^0.8 a_member_name^0.4 genre</str>
```

pf Tips

Start with the same value used as `bf`, and then potentially modify it. Remove fields that are always one word, such as an `ID`, because doing a phrase search on it would be pointless. Do not add fields that are not present in `bf`, because it will have no effect. The exception to this is if the field comes from the same source but is using different text analysis (such as a single token), and you want phrase searches on this data to only match entire field values.

Phrase slop configuration

In the previous chapter, we had mentioned the phrase **slop**, aka term proximity, by following a phrase with a tilde and a number, as shown below:

```
"Billy Joel"~1
```

But you can't use most of Solr's syntax with dismax such as this. Instead, there are two parameters to configure the slop: `qs` for any user-entered query phrases and `ps` for the phrase boosting mentioned previously. If slop is not specified, then there is no slop, which is equivalent to a value of zero. For more information about slop, see the corresponding discussion in the previous chapter. Here is a sample configuration of both slop settings:

```
<str name="qs">1</str>
<str name="ps">0</str>
```

Boosting: Boost queries

Continuing with the boosting theme is a simple way to affect the score of documents: boost queries. The dismax handler lets you specify multiple additional queries using `bq` parameter(s) which, like the automatic phrase boost, get added onto the user's query in a similar manner. Remember that a boosting query only serves to affect the scoring of documents that already matched the user's query in the `q` parameter. If a matched document also matches a `bq` query, then it will be scored higher. These queries are not limited by the dismax handler's syntax restrictions, as they are not intended to come from user input.

For this feature, we'll go into a more in-depth example. Perhaps we've decided that searches for current artists should get scored higher. With our data set, for the sake of this example, let's simply say that if `a_end_date` is blank, then such an artist is current, but not otherwise. In order to find documents with no `a_end_date`, you can try the following query in the web interface with the standard handler:

```
-a_end_date:[* TO *] AND *:*
```

Why the `AND *:*`

Remember from Chapter 4 that a pure negative query doesn't work correctly if it is not at the top level of the query that Lucene ultimately processes. Testing this query out in `q` with the standard handler will work without the `*:*` part, but once we use it in `bq`, then the `AND *:*` will be required for it to work.

If we put the previous query into the URL and add an initial arbitrary boost of two, then it looks like this after URL encoding:

```
bq=(-a_end_date%3A[*+TO+*]+AND+*%3A*)^2
```

Of course, URL encoding is only for the URL, and not for entry in the request handler configuration, where `bq` is probably most suitably configured.

Remember to specify a non-default boost

There is some code within dismax that supports legacy behavior of this feature. It kicks in when there is one boost query, and it has a boost of one, by default. This legacy behavior is not necessarily a problem, but it was for our query here, before I made the boost two. I noticed some strange results using `debugQuery` and looking at `parsedquery` in the output, which allowed me to see that my boost query wasn't incorporated into the final query in the way I expected. Looking at the source code showed the legacy logic and under what circumstances it took effect. It should be easy to avoid this problem, because you will want to tweak the boost value to your liking.

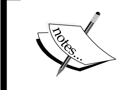
I experimented with a search for the band `Nirvana`. Nirvana, the well-known 90's alternative rock band, is no longer current, and it has an end date. But it appears that there are bands that are also named `Nirvana` in our MusicBrainz data set that don't have an end date. Here is a search for `Nirvana` with our `mb_artists` handler without specifying a boost query:

```
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">4</int>
  <lst name="params">
    <str name="qf">a_name a_alias^0.8 a_member_name^0.4</str>
    <str name="defType">dismax</str>
    <str name="tie">0.1</str>
    <str name="wt">standard</str>
    <str name="rows">10</str>
    <str name="start">0</str>
```

```
      <str name="explainOther"/>
      <str name="hl.fl"/>
      <str name="echoParams">all</str>
      <str name="indent">on</str>
      <str name="q">Nirvana</str>
      <str name="fl">id,a_name,a_end_date,score</str>
      <str name="qt">mb_artists</str>
      <str name="version">2.2</str>
    </lst>
  </lst>
  <result name="response" numFound="8" start="0" maxScore="13.412962">
    <doc>
      <float name="score">13.412962</float>
      <date name="a_end_date">1994-04-05T04:00:00Z</date>
      <str name="a_name">Nirvana</str>
      <str name="id">Artist:54</str>
    </doc>
    <doc>
      <float name="score">12.677703</float>
      <str name="a_name">Nirvana</str>
      <str name="id">Artist:236413</str>
    </doc>
    <doc>
      <float name="score">12.677703</float>
      <str name="a_name">Nirvana</str>
      <str name="id">Artist:303288</str>
    </doc>
    <doc>
      <float name="score">7.9235644</float>
      <str name="a_name">El Nirvana</str>
      <str name="id">Artist:407794</str>
    </doc>
    <doc>
      <float name="score">7.9235644</float>
      <str name="a_name">Nirvana 2002</str>
      <str name="id">Artist:512007</str>
    </doc>
    <doc>
      <float name="score">7.9235644</float>
      <str name="a_name">Nirvana Singh</str>
      <str name="id">Artist:520885</str>
    </doc>
    <doc>
      <float name="score">6.3388515</float>
```

```
      <str name="a_name">Nirvana Sitar &amp; String Group</str>
      <str name="id">Artist:132835</str>
    </doc>
    <doc>
      <float name="score">0.7352593</float>
      <str name="a_name">The String Quartet Tribute</str>
      <str name="id">Artist:186308</str>
    </doc>
  </result>
  </response>
```

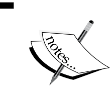
First in the results is Nirvana, `id # 54`. I know this because I also ran the query showing other fields and that one is definitely it. Our goal here is to add the boost query and to use a boost value that is sufficiently high so that Nirvana moves from the number one spot to number three, below the other two that have bands named the same but no end date. By using the boost query parameter indicated earlier and with a boost value of ten, I was able to do this. It takes some experimentation to find a good value. The scores for each document changed a bit. This happens when you fiddle with the scoring. The actual score values aren't relevant, though the relativity of each score to each other's score is.



This is a hypothetical scenario to illustrate the usage of this feature. Someone searching for Nirvana probably actually does want the band that came out on top without our boost query.

Boosting: Boost functions

Earlier in the chapter you learned about function queries. We used them with the standard request handler by using the `_val_` trick as part of the query. That method is a bit of a hack on the syntax, and it isn't a method that will work with the dismax handler because of self-imposed syntax restrictions. Instead, the dismax handler offers a convenient query parameter for direct entry of function queries: `bf`. As with `bq`, you can specify `bf` as many times as you wish. As with boost queries and automatic phrase boosting, these boost functions are incorporated into the final query in a similar manner.



For a thorough explanation of function queries, see the earlier section on this topic. The following example was taken from it but does not go into detail.

Consider the case where we'd like to boost searches for releases according to their release date. Releases released more recently get more of a boost than those released long ago. We'll use the `r_event_date_earliest` field, that needs to be indexed and not be multi-valued, which is indeed the case. A boosting function that satisfies this requirement would involve a parameter that looks like this, if specified in the request handler configuration:

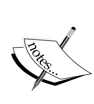
```
<str name="bf"> recip(map(rord(r_event_date_earliest),0,0,99000)
        ,1,95000,95000)^100 </str>
```

Notice that we didn't use quotes, which would be needed when using the `_val_` syntax. Remember to omit spaces too. If this were to be put in the URL for our experimentation, then it would need to be URL encoded. Only the commas need escaping to `%2C`:

```
bf=recip(map(rord(r_event_date_earliest)%2C0%2C0%2C99000)
        %2C1%2C95000%2C95000)^100
```

Min-should-match

With the standard handler, you have a choice of the default operator being `OR`, thereby requiring just one queried clause (that is word) to match, or choosing `AND` to make all queried clauses required. This of course only applies to clauses not otherwise explicitly marked required or prohibited in the query using `+` and `-`. But these are two extremes, and it would be useful to pick some middle ground. The dismax handler uses a strategy called **min-should-match**, a feature which describes *how many* clauses should match, depending on how many are there in the query—required and prohibited clauses are not included in the numbers. This allows you to quantify the number of clauses as either a percentage or a fixed number. The configuration of this setting is entirely contained within the `mm` query parameter using a concise syntax specification that I'll describe in a moment.



This feature is more useful if users use many words in their queries, at least three. This in turn suggests a text field that has some substantial text in it but that is not the case for our MusicBrainz data set. Nevertheless, we will put this feature to good use.

Basic rules

The following are the four basic `mm` specification formats expressed as examples:

3	3 clauses are required, the rest are optional.
-2	2 clauses are optional, the rest are required.
66%	66% of the clauses (rounded down) are required, the rest are optional.
-25%	25% of the clauses (rounded down) are optional, the rest are required.

Notice that `-` inverses the required/optional definition. It does not make any number negative from the standpoint of any definitions herein.

Note that 75% and -25% may seem the same but are not due to rounding. Given five queried clauses, the first requires three, whereas the second requires four. This shows that if you desire a round-up calculation, then you can invert the sign and subtract it from 100.

Two additional points about these rules are as follows:

- If the `mm` rule is a fixed number `n` but there are fewer queried clauses, then `n` is reduced to the queried clause count so that the rule will make sense. For example: if `mm` is `-5` and only two clauses are in the query, then all are optional. Sort of!
- Remember that in all circumstances across Lucene (and thus Solr), at least one clause in a query must match, even if every clause is optional. So in the example above and for `0` or `0%`, one clause must still match, assuming that there are no required clauses present in the query.

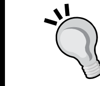
Multiple rules

In addition to the basic specification formats is the final format, which allows for one of the multiple basic formats to be chosen, depending on how many clauses are in the query. This format is composed of an ordered space-separated series of the following: `number<basicmm`—which can be read as "If the clause count is greater than `number`, then apply rule `basicmm`". Only the right-most rule that meets the clause count threshold is evaluated. As they are ordered in an ascending order, the chosen rule is the one that requires the greatest number of clauses. If none match because there are fewer clauses, then all clauses are required (that is a basic specification of 100%).

An example of the `mm` specification is given below:

```
2<75% 9<-3
```

This reads: If there are over nine clauses, then all but three are required (three are optional, and the rest are required). If there are over two clauses, then 75% are required (rounded down). Otherwise (one or two clauses) all clauses are required, which is the default rule.



I find it easier to interpret these rules if they are read right to left.

What to choose

A simple configuration for min-should-match is making all of the search terms optional. This is effectively equivalent to a default `OR` operator in the standard handler. This is configured as shown below:

```
0%
```

Conversely, the other extreme is requiring all of the terms, and this is equivalent to a default `AND` operator. This is configured as shown below:

```
100%
```

For MusicBrainz's dismax handlers, I do not expect users to be using many terms. However, for the most part, I expect them to be queried. If a user searches for three or more terms, then I'll let one be optional. Here is the `mm` spec:

```
2<-1
```

You may be inclined to require all of the search terms. Remember from the scoring discussion in Chapter 4 that the percentage of matching search terms is a factor in scoring. With this in mind, it is not necessarily a bad thing to let some of the search terms be optional if the user enters a few terms (or whatever number you choose). The user will get some results, which for many applications is better than returning none. However, this is only a suggestion.

A default search

There is one last feature of the dismax handler, and this is the following parameter:

- `q.alt`: This is the query that is performed if `q` is not specified. Unlike `q` it uses Solr's regular (full) syntax, not dismax's limited one.

This parameter is usually set to `*:*` to match all documents and is specified in the handler configuration in `solrconfig.xml`. You'll see with faceting in the next section, that there will not necessarily be a user query, and so you'll want to display facets over all of the data. Without `q.alt` there would be no way for your application to submit a query for all documents, as dismax's limited syntax does not permit `*:*` for the `q` parameter.

Faceting

Faceting, after searching, is arguably the second-most valuable feature in Solr. It is perhaps even the most fun you'll have, because you will learn more about your data than with any other feature. **Faceting** enhances search results with aggregated information over all of the documents found in the search to answer questions such as the ones mentioned below, given a search on MusicBrainz releases:

- How many are official, bootleg, or promotional?
- What were the top five most common countries in which the releases occurred?
- Over the past ten years, how many were released in each year?
- How many have names in these ranges: A-C, D-F, G-I, and so on?
- Given a track search, how many are < 2 minutes long, 2-3, 3-4, or more?

Moreover, in addition, it can power `term-suggest` aka `auto-complete` functionality, which enables your search application to suggest a completed word that the user is typing, which is based on the most commonly occurring words starting with what they have already typed. So if a user started typing `siamese dr`, then Solr might suggest that `dreams` is the most likely word, along with other alternatives.

Faceting, sometimes referred to as **faceted navigation**, is usually used to power user interfaces that display this summary information with clickable links that apply Solr filter queries to a subsequent search.

If we revisit the comparison of search technology to databases, then faceting is more or less analogous to SQL's `group by` feature on a column with `count(*)`. However, in Solr, facet processing is performed subsequent to an existing search as part of a single request-response with both the primary search results and the faceting results coming back together. In SQL, you would need to potentially perform a series of separate queries to get the same information.

A quick example: Faceting release types

Observe the following search results. `echoParams` is set to `explicit` (defined in `solrconfig.xml`) so that the search parameters are seen here. This example is using the standard handler (though perhaps dismax is more typical). The query parameter `q` is `*:*`, which matches all documents. In this case, the index I'm using only has releases. If there were non-releases in the index, then I would add a filter `fq=type%3ARelease` to the URL or put this in the handler configuration, as that is the data set we'll be using for most of this chapter. I wanted to keep this example brief so I set `rows` to `2`. Sometimes when using faceting, you only want the facet information and not the main search, so you would set `rows` to `0`, if that is the case.

It's important to understand that the faceting numbers are computed over the entire search result, which is all of the releases in this example, and not just the two rows being returned.

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">160</int>
  <lst name="params">
  <str name="wt">standard</str>
  <str name="rows">2</str>
  <str name="facet">true</str>
  <str name="q">*:*</str>
  <str name="fl">*,score</str>
  <str name="qt">standard</str>
  <str name="facet.field">r_official</str>
  <str name="f.r_official.facet.missing">true</str>
  <str name="f.r_official.facet.method">enum</str>
  <str name="indent">on</str>
  </lst>
</lst>
<result name="response" numFound="603090" start="0" maxScore="1.0">
  <doc>
    <float name="score">1.0</float>
    <str name="id">Release:136192</str>
    <str name="r_a_id">3143</str>
    <str name="r_a_name">Janis Joplin</str>
    <arr name="r_attributes"><int>0</int><int>9</int>
                             <int>100</int></arr>
    <str name="r_name">Texas International Pop Festival
           11-30-69</str>
    <int name="r_tracks">7</int>
```

```
      <str name="type">Release</str>
    </doc>
    <doc>
      <float name="score">1.0</float>
      <str name="id">Release:133202</str>
      <str name="r_a_id">6774</str>
      <str name="r_a_name">The Dubliners</str>
      <arr name="r_attributes"><int>0</int></arr>
      <str name="r_lang">English</str>
      <str name="r_name">40 Jahre</str>
      <int name="r_tracks">20</int>
      <str name="type">Release</str>
    </doc>
  </result>
    <lst name="facet_counts">
    <lst name="facet_queries"/>
    <lst name="facet_fields">
    <lst name="r_official">
      <int name="Official">519168</int>
      <int name="Bootleg">19559</int>
      <int name="Promotion">16562</int>
      <int name="Pseudo-Release">2819</int>
      <int>44982</int>
    </lst>
    </lst>
    <lst name="facet_dates"/>
    </lst>
  </response>
```

The facet related search parameters are highlighted at the top. The `facet.missing` parameter was set using the field-specific syntax, which will be explained shortly.

Notice that the facet results (highlighted) follow the main search result and are given a name `facet_counts`. In this example, we only faceted on one field, `r_official`, but you'll learn in a bit that you can facet on as many fields as you desire. The `name` attribute holds a **facet value**, which is simply an indexed term, and the integer following it is the number of documents in the search results containing that term, aka a **facet count**. The next section gives us an explanation of where `r_official` and `r_type` came from.

MusicBrainz schema changes

In order to get better self-explanatory faceting results out of the `r_attributes` field and to split its dual-meaning, I modified the schema and added some text analysis. `r_attributes` is an array of numeric constants, which signify various types of releases and it's *official-ness*, for lack of a better word. As it represents two different things, I created two new fields: `r_type` and `r_official` with `copyField` directives to copy `r_attributes` into them:

```
<field name="r_attributes" type="integer" multiValued="true"
           indexed="false" /><!-- ex: 0, 1, 100 -->
<field name="r_type" type="rType" multiValued="true"
           stored="false" /><!-- Album | Single | EP |... etc. -->
<field name="r_official" type="rOfficial" multiValued="true"
           stored="false" /><!-- Official | Bootleg | Promotional -->
```

And:

```
<copyField source="r_attributes" dest="r_type" />
<copyField source="r_attributes" dest="r_official" />
```

In order to map the constants to human-readable definitions, I created two field types: `rType` and `rOfficial` that use a regular expression to pull out the desired numbers and a synonym list to map from the constant to the human readable definition. Conveniently, the constants for `r_type` are in the range 1-11, whereas `r_official` are 100-103. I removed the constant `0`, as it seemed to be bogus.

```
<fieldType name="rType" class="solr.TextField" sortMissingLast="true"
                 omitNorms="true">
    <analyzer>
        <tokenizer class="solr.KeywordTokenizerFactory"/>
        <filter class="solr.PatternReplaceFilterFactory"
                       pattern="^(0|1\d\d)$" replacement=""
                       replace="first" />
        <filter class="solr.LengthFilterFactory" min="1" max="100" />
        <filter class="solr.SynonymFilterFactory"
                       synonyms="mb_attributes.txt"
                       ignoreCase="false" expand="false"/>
    </analyzer>
</fieldType>
```

The definition of the type `rOfficial` is the same as `rType`, except it has this regular expression: `^(0|\d\d?)$`.

The presence of LengthFilterFactory is to ensure that no zero-length (empty-string) terms get indexed. Otherwise, this would happen because the previous regular expression reduces text fitting unwanted patterns to empty strings.

The content of mb_attributes.txt is as follows:

```
# from: http://bugs.musicbrainz.org/browser/mb_server/trunk/
# cgi-bin/MusicBrainz/Server/Release.pm#L48
#note: non-album track seems bogus; almost everything has it
0=>Non-Album\ Track
1=>Album
2=>Single
3=>EP
4=>Compilation
5=>Soundtrack
6=>Spokenword
7=>Interview
8=>Audiobook
9=>Live
10=>Remix
11=>Other

100=>Official
101=>Promotion
102=>Bootleg
103=>Pseudo-Release
```

It does not matter if the user interface uses the name (for example: Official) or constant (for example: 100) when applying filter queries when implementing faceted navigation, as the text analysis will let the names through and will map the constants to the names. This is not necessarily true in a general case, but it is for the text analysis as I've configured it above.

The approach I took was relatively simple, but it is not the only way to do it. Alternatively, I might have split the attributes and/or mapped them as part of the import process. This would allow me to remove the multiValued setting in r_official. Moreover, it wasn't truly necessary to map the numbers to their names, as a user interface, which is going to present the data, could very well map it on the fly.

Field requirements

The principal requirement of a field that will be faceted on is that it must be indexed. In addition to all but the prefix faceting use case, you will also want to use text analysis that does not tokenize the text. For example, the value `Non-Album Track` is indexed the way it is in `r_type`. We need to be careful to escape the space where this appeared in `mb_attributes.txt`. Otherwise, faceting on this field would show tallies for `Non-Album` and `Track` separately. Depending on the type of faceting you want to do and other needs you have like sorting, you will often find it necessary to have a copy of a field just for faceting. Remember that with faceting, the facet values returned in search results are the actual terms indexed, and not the stored value, which isn't even used.

Types of faceting

Solr's faceting is broken down into three types. They are as follows:

- **field values** (text): This is the most fundamental and common type of faceting that works off of the indexed terms, which is the result of text-analysis on an indexed field. It needn't necessarily be text, but it is treated this way. Most faceting parameters are for configuring this type. The count for such faceting is grouped in the output under the name `facet_fields`.
- **dates**: This is for faceting on dates to count matching documents by equal date ranges. The facet counts are grouped in the output under `facet_dates`.
- **queries**: This works quite differently by counting the number of documents matching each specified query. This type is usually used for number ranges. The facet counts are grouped in the output under `facet_queries`.

In the rest of this chapter, we will describe how to do these different types of facets. But before that, there is one common parameter to enable faceting:

- `facet:` It defaults to blank. In order to enable faceting, you must set this to `true` or `on`. If this is not done, then the faceting parameters will be ignored.

In all of the examples here, we've obviously set `facet=true`.

Faceting text

The following request parameters are for typical text based facets. They need not literally be text but should not be indexed with one of the number or date field types.

- `facet.field`: You **must** set this parameter to a field name in order to text-facet on that field. Repeat this parameter for each field to be faceted on. Solr, in essence, iterates over all of the indexed terms for the field and tallies a count for the number of searched documents that have the term. Solr then puts this in the response. Lucene's index makes this much faster than you might think. See the previous *Field requirements* section.

> The remaining faceting parameters can be set on a per-field basis, otherwise they apply to all text faceted fields that don't have a field-specific setting. You will usually specify them per-field, especially if you are faceting on more than one field so that you don't get your faceting configuration mixed up. For brevity, many of these examples don't. For example: `f.r_type.facet.sort=lex` (`r_type` is a field name, `facet.sort` is a facet parameter).

- `facet.sort`: It is set to either `count` to sort the facet values by descending totals or to `lex` to sort alphabetically. If `facet.limit` is greater than zero (which is `true` by default), then Solr picks `count` as the default, otherwise `lex` is chosen.
- `facet.limit:` It defaults to `100`. It limits the number of facet values returned in the search results of a field. As these are usually going to be displayed to the user, it doesn't make sense to have a large number of these in the response. If you are confident that the indexed terms fit a very limited vocabulary, then you might choose to disable the limit with a value of `-1`, which will change the default sort of them to alphabetic.
- `facet.offset`: It defaults to `0`. It is the index into the facet value list from which the values are returned. This enables paging of facet values when used with `facet.limit`. If there are lots of values and if you want the user to scan through them, then you might page them as opposed to just showing them the most popular ones.
- `facet.mincount`: This defaults to `0`. It filters out facet values that have facet counts less than this. This is applied before `limit` and `offset` so that paging works as expected.

- `facet.missing`: It defaults to blank and is set to `true` or `on` for the facet value listing to include an unnamed count at the end, which is the number of searched documents that have no indexed terms. The first facet example demonstrates this.
- `facet.prefix`: It filters the facet values to those starting with this value. See a later section for an example.
- `facet.method`: Solr can be told to use either the `enum` or `fc` (field cache) algorithm to perform the faceting. The speed and memory usage of the query varies depending on your data. If you are faceting on a field that you know only has a small number of values (say less than 50), then it is advisable to explicitly set this to `enum`. When faceting on multiple fields, remember to set this for the specific fields desired and not universally for all facets. The request handler configuration is a good place to put this.

Alphabetic range bucketing (A-C, D-F, and so on)

Solr does not directly support alphabetic range bucketing (A-C, D-F, and so on). However, with a creative application of text analysis and a dedicated field, we can achieve this with little effort. Let's say we want to have these range buckets on the release names. We need to extract the first character of `r_name`, and store this into a field that will be used for this purpose. We'll call it `r_name_facetLetter`. Here is our field definition:

```
<field name="r_name_facetLetter" type="bucketFirstLetter"
             stored="false" />
```

And here is the `copyField`:

```
<copyField source="r_name" dest="r_name_facetLetter" />
```

The definition of the type `bucketFirstLetter` is the following:

```
<fieldType name="bucketFirstLetter" class="solr.TextField"
                sortMissingLast="true" omitNorms="true">
    <analyzer type="index">
        <tokenizer class="solr.PatternTokenizerFactory"
                   pattern="^([a-zA-Z]).*" group="1" />
        <filter class="solr.SynonymFilterFactory"
                synonyms="mb_letterBuckets.txt" ignoreCase="true"
                expand="false"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.KeywordTokenizerFactory"/>
    </analyzer>
</fieldType>
```

The `PatternTokenizerFactory`, as configured, plucks out the first character, and the `SynonymFilterFactory` maps each letter of the alphabet to a range like A-C. The mapping is in `conf/mb_letterBuckets.txt`. The field types used for faceting generally have a `KeywordTokenizerFactory` for the query analysis to satisfy a possible filter query on a given facet value returned from a previous faceted search. After validating these changes with Solr's analysis admin screen, we then re-index the data. For the facet query, we're going to advise Solr to use the `enum` method, because there aren't many facet values in total. Here's the URL to search Solr:

```
http://localhost:8983/solr/select?indent=on&q=*%3A*&qt=standard&wt=st
andard&facet=on&facet.field=r_name_facetLetter&facet.sort=lex&facet.
missing=on&facet.method=enum
```

The URL produced results containing the following facet data:

```
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
  <lst name="r_name_facetLetter">
    <int name="A-C">99005</int>
    <int name="D-F">68376</int>
    <int name="G-I">60569</int>
    <int name="J-L">49871</int>
    <int name="M-O">59006</int>
    <int name="P-R">47032</int>
    <int name="S-U">143376</int>
    <int name="V-Z">33233</int>
    <int>42622</int>
  </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
<lst name="facet_dates"/>
</lst>
```

Faceting dates

Solr has built-in support for faceting a date field by a range and divided interval. You can think of this as a convenient feature instead of being forced to use the more awkward facet queries described after this. Unfortunately, this feature does not extend to numeric types yet. I'll demonstrate a quick example against MusicBrainz release dates, and then describe the parameters and their options.

```
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">145</int>
```

```
    <lst name="params">
      <str name="facet.date">r_event_date_earliest</str>
      <str name="facet.date.end">NOW/YEAR</str>
      <str name="facet.date.gap">+1YEAR</str>
      <str name="facet.date.other">all</str>
      <str name="rows">0</str>
      <str name="facet">on</str>
      <str name="indent">on</str>
      <str name="echoParams">explicit</str>
      <str name="q">smashing</str>
      <str name="qt">mb_releases</str>
      <str name="f.r_event_date_earliest.facet.date.start">
                 NOW/YEAR-5YEARS</str>
    </lst>
  </lst>
  <result name="response" numFound="248" start="0"/>
  <lst name="facet_counts">
    <lst name="facet_queries"/>
    <lst name="facet_fields"/>
    <lst name="facet_dates">
    <lst name="r_event_date_earliest">
      <int name="2004-01-01T00:00:00Z">1</int>
      <int name="2005-01-01T00:00:00Z">1</int>
      <int name="2006-01-01T00:00:00Z">3</int>
      <int name="2007-01-01T00:00:00Z">11</int>
      <int name="2008-01-01T00:00:00Z">0</int>
      <str name="gap">+1YEAR</str>
      <date name="end">2009-01-01T00:00:00Z</date>
      <int name="before">95</int>
      <int name="after">0</int>
      <int name="between">16</int>
    </lst>
    </lst>
  </lst>
  </response>
```

This example demonstrates a few things, not only date faceting:

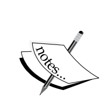
- `qt=mb_releases` is a dismax query type handler and ensures that we're looking at releases.
- `q=smashing` indicates that we're faceting on a search instead of all the documents, granted we kept the rows at zero, which is unrealistic but not pertinent.

- The facet start date was specified using the field specific syntax. It is just a demonstration. We'd probably do this with every parameter.
- The `<date name="end">` part below the facet counts indicates the upper bound of the last date facet count. It may or may not be the same as `facet.date.end` (see `facet.date.hardend` explained in the next section).
- The `before`, `after`, and `between` counts are for specifying `facet.date.other`.

Date facet parameters

All of the date faceting parameters start with `facet.date`. As with most other faceting parameters, they can be made field specific in the same way. The parameters are explained as follows:

- `facet.date`: You must set this parameter to your date field's name to date-facet on that field. Repeat this parameter for each date field to be faceted on.



> The remainder of these date faceting parameters can be specified on a per-field basis in the same fashion that the non-date parameters can. For example, `f.r_event_date_earliest.facet.date.start`.

- `facet.date.start`: Mandatory, this is a date to specify the start of the range to facet on. The syntax is the same as used elsewhere in Solr, which is described in Chapter 4 under the *Date Math* section. Using `NOW` with some Solr date math is quite effective as in this example: `NOW/YEAR-5YEARS`, which is interpreted as five years ago, starting at the beginning of the year.
- `facet.date.end`: Mandatory, this is a date to specify the end of the range exclusively. It has the same syntax as `facet.date.start`. Note that the actual end of the range may be different (see `facet.date.hardend`).
- `facet.date.gap`: Mandatory, this specifies the time interval to divide the range. It uses a subset of Solr's `Date Math` syntax, as it's a time duration and not a particular time. It should always start with a `+`. Examples: `+1YEAR` or `+1MINUTE+30SECONDS`. Note that after URL encoding, `+` becomes `%3B`.
- `facet.date.hardend`: It defaults to `false`. This parameter instructs Solr on what to do when `facet.date.gap` does not divide evenly into the facet date range (start–>end). If this is `true`, then the last date span will have a smaller duration than the others. Moreover, you will observe that the end date value in the facet results is the same as `facet.date.end`. Otherwise, by default, the end is essentially increased sufficiently so that the date spans are all equal.

- `facet.date.other`: It defaults to `none`. This parameter adds more faceting counts depending on its value. It can be specified multiple times. See the example using this at the start of this section.
    - `before`: count of documents before the faceted range
    - `after`: count of documents following the faceted range
    - `between`: documents within the faceted range (somewhat redundant)
    - `none`: (disabled) the default
    - `all`: shortcut for all three (`before`, `between`, and `after`)

Faceting on arbitrary queries

This is the final type of facet, and it offers a lot of flexibility. Instead of choosing a field to facet on its values (whether text based or date), we specify some number of Solr queries that each itself becomes a facet. For each facet query specified, the number of search results matching the query is counted, and this number is returned in the results. As with all other faceting, the set of documents that are faceted is the search result, which is `q` less any filtered with `fq`.

There is only one parameter for configuring facet queries:

- `facet.query`: A Solr query to be evaluated over the search results. The number of matching documents is returned as an entry in the results next to this query. Specify this multiple times to have Solr evaluate multiple facet queries.

As facet queries are the only way to facet for numeric ranges, we'll use that as an example. In our MusicBrainz tracks index, there is a field named `t_duration`, which is how long the song is in seconds. In the search below, we've used `echoParams` for making the search parameters clear.

```
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">106</int>
  <lst name="params">
    <str name="indent">on</str>
    <str name="rows">0</str>
    <str name="q">t_name:Geek</str>
  <arr name="facet.query">
    <str>t_duration:[* TO 119]</str>
    <str>t_duration:[120 TO 179]</str>
```

```
      <str>t_duration:[180 TO 239]</str>
      <str>t_duration:[240 TO *]</str>
    </arr>
      <str name="facet">true</str>
    </lst>
  </lst>
  <result name="response" numFound="200" start="0"/>
  <lst name="facet_counts">
      <lst name="facet_queries">
      <int name="t_duration:[* TO 119]">55</int>
      <int name="t_duration:[120 TO 179]">36</int>
      <int name="t_duration:[180 TO 239]">64</int>
      <int name="t_duration:[240 TO *]">45</int>
    </lst>
    <lst name="facet_fields"/>
    <lst name="facet_dates"/>
  </lst>
  </response>
```

In this example, the `facet.query` parameter was specified four times to divide a range of numbers into four buckets: less than 2 minutes, 2 to < 3 minutes, 3 to < 4 minutes and > 4 minutes. These numbers add up to 200, which is the total number of documents. Note that the queries need not be disjointed, but they were in this example. It's certainly possible to query for dates using various range durations and to reference other fields in the facet queries too, whatever Solr query suits your needs.

Excluding filters

Consider a scenario where you are implementing faceted navigation and you want to let the user pick several values of a field to filter on instead of just one. Typically, when an individual facet value is chosen, this becomes a filter that would cause any other value in that field to have a zero facet count, if it would even show up at all. In this scenario, we'd like to exclude this filter for this facet. I'll demonstrate this with a `before` and `after` clause.

Here is a search for releases containing `smashing`, faceting on `r_type`. We'll leave `rows` at `0` for brevity, but observe the `numFound` value nonetheless. At this point, the user has not chosen a filter (therefore no `fq`).

```
http://localhost:8983/solr/select?indent=on&qt=mb_releases&rows=0&q=s
mashing&facet=on&facet.field=r_type&facet.mincount=1&facet.sort=lex
```

And the output of the previous URL is:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">24</int>
</lst>
<result name="response" numFound="248" start="0"/>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
  <lst name="r_type">
    <int name="Album">29</int>
    <int name="Compilation">41</int>
    <int name="EP">7</int>
    <int name="Interview">3</int>
    <int name="Live">95</int>
    <int name="Other">19</int>
    <int name="Remix">1</int>
    <int name="Single">45</int>
    <int name="Soundtrack">1</int>
  </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
</response>
```

Now the user chooses the `Album` facet value that interests him/her. This adds a filter query. As a result, now the URL is as before but has `&fq=r_type%3AAlbum` at the end and has this output:

```
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">17</int>
</lst>
<result name="response" numFound="29" start="0"/>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
  <lst name="r_type">
    <int name="Album">29</int>
  </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
</response>
```

Notice that the other r_type facet counts are gone because of the filter, yet we want these so that we can give the user a choice for expanding the filter. The reduced numFound of 29 is good though, because at this moment the user did indeed filter on a value so far.

The solution: Local Params

Solr can solve this problem with some additional metadata on both the filter query and the facet field reference using a new and obscure Solr feature called **Local Params**. Local Params are name-value parameters inserted at the start of a query and in some other places like facet field references. The previous example would change as follows:

- fq would now be {!tag=foo}r_type:Album
- facet.field would now be {!ex=foo}r_type

Remember to URL Encode this added syntax when used in the URL. The only problem character is =, which becomes %3D.

Explanation:

- tag is a local parameter to arbitrarily label a parameter.
- The name foo was an arbitrarily chosen tag name, it truly doesn't matter what it's named. If multiple fields and filter queries are to be tagged correspondingly, then you could use the field name as the tag name to differentiate them consistently.
- ex is a local parameter on a facet field that refers to tagged filter queries to be excluded in the facet count. Multiple tags can be referenced by commas separating them. For example: {!ex=t1,t2,t3}r_type.

The new complete URL is:

```
http://localhost:8983/solr/select?indent=on&qt=mb_releases&ro
ws=0&q=smashing&facet=on&facet.field={!ex%3Dfoo}r_type&facet.
mincount=1&facet.sort=lex&fq={!tag%3Dfoo}r_type%3AAlbum.
```

And here is the output. The facet counts are back, but numFound remains at the filtered 29:

```
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">4</int>
```

```
</lst>
<result name="response" numFound="29" start="0"/>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
  <lst name="r_type">
    <int name="Album">29</int>
    <int name="Compilation">41</int>
    <int name="EP">7</int>
    <int name="Interview">3</int>
    <int name="Live">95</int>
    <int name="Other">19</int>
    <int name="Remix">1</int>
    <int name="Single">45</int>
    <int name="Soundtrack">1</int>
  </lst>
  </lst>
  <lst name="facet_dates"/>
</lst>
</response>
```

At this point, if the user chooses additional values from this facet, then the filter query can be modified to allow for more possibilities, such as: `fq={!tag%3Dfoo}r_type%3AAlbum+r_type%3AOther`, which filters for releases that are either of type `Album` or `Other`.

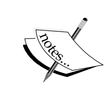
Facet prefixing (term suggest)

When one thinks of faceting, one doesn't think of term-suggest, aka auto-complete. Within Solr, however, the faceting technology is suited for this purpose too.

For this example, we have a text box containing:

```
smashing pu
```

All of the words in the user's text box except the last one become the main query for the term-suggest. We may want to make it a phrase query. For our example, this is just `smashing`. If there isn't anything, then we'd want to ensure that the search handler used would query for all documents. The faceted field is `r_name`, and we want to sort by occurrence. We also want there to be at least one occurrence, and we probably don't want more than ten values. We don't need the actual search results either. This leaves the `facet.prefix` faceting parameter to make this work. This parameter filters the facet values to those starting with this value.



> Remember that facet values are the final result of text analysis, and therefore are probably lowercased for fields you might want to do term completion on. You'll need to pre-process the prefix value similarly, or else nothing will be found.

We're obviously going to set this to `pu`, the last word that the user has partially typed. Here is a URL for such a search:

```
http://localhost:8983/solr/select?q=smashing&qt=mb_releases&wt=jso
n&indent=on&facet=on&rows=0&facet.limit=10&facet.mincount=1&facet.
field=r_name&facet.prefix=pu
```

In this example, we're going to use the JSON output format. Here is the result:

```
{
  "responseHeader":{
  "status":0,
  "QTime":5},
  "response":{"numFound":248,"start":0,"docs":[]
},
  "facet_counts":{
  "facet_queries":{},
  "facet_fields":{
    "r_name":[
     "pumpkins",10,
     "pumpkin",2,
     "pure",2,
     "pumpehuset",1,
     "punk",1]},
  "facet_dates":{}}}
```

This is exactly the information needed to fill up a pop-up box of choices that the user can conveniently choose.

However, there are some issues to be aware of with this feature:

- You may want to retain the case information of what the user is typing so that it can then be re-applied to the Solr results. Remember that `facet.prefix` will probably need to be lowercased depending on text analysis.
- If stemming text analysis is performed on the field at the time of indexing, then the user is going to get junk. Either don't do stemming or use an additional field for suitable text analysis of this feature.

- If you would like to do term-completion of multiple fields, then you'll be disappointed that you can't do that directly. The easiest way is to combine several fields at index-time. Alternatively, a query searching multiple fields with faceting configured for multiple fields can be done. It would be up to you to merge the faceting results based on ordered counts.

Summary

In this chapter, we've covered some topics regarding searching beyond the basics. Function queries are an advanced technique that enables you to influence the scoring based on numbers or dates in your documents. With the provided guidance on which mathematical formulas to use, you need not be a math whiz to leverage this feature. The dismax handler will almost certainly be the primary Solr search handler you use for all of the extra bells and whistles over the standard handler—most importantly, the ability to search across multiple fields and with varying boosts. Finally, beyond searching is faceting. It is possibly the most valuable and popular search component.

In the next chapter, we'll cover **Solr Search Components**. You've actually been using them already because performing a query, enabling debug output, and faceting are each actually implemented as search components. But there's also search result highlighting, spelling correction, suggesting similar documents, collapsing/rolling up search results, editorially elevating or evicting results, and more!

6
Search Components

One of Solr's primary extension mechanisms is Solr Search Components. You've actually been using several of them already: `QueryComponent` performs the actual searches (notably the `q` parameter), `DebugComponent` outputs the invaluable query debugging information when setting `debugQuery`, and finally `FacetComponent` performs the faceting we used in Chapter 5. However, there are many more that do all sorts of useful things that can really enhance your search experience:

- **Highlighting**: for returning highlighted text snippets of matching text in the original data
- **Query Elevation**: for (manually) modifying search results for certain queries
- **Spell Checking**: for recommending alternative searches in query results
- **More-Like-This**: for finding documents similar to other documents
- **Stats**: for mathematical statistics of indexed numbers
- **Field Collapsing**: for rolling-up/aggregating records that have a common field value

About components

In `solrconfig.xml`, there are multiple `<requestHandler/>` elements defined. Solr dispatches a request to the appropriate one, based on a handler's `url` parameter, if present, or the `qt` (query type) URL parameter, which names a specific handler. Any request handlers with `class="solr.SearchRequestHandler"` are intuitively related to searching. The Java code implementing `org.apache.solr.SearchRequestHandler` doesn't actually do any searching. Instead, it maintains a list of `SearchComponents` that are invoked in series for a search request. The search components used and their order are configured in `solrconfig.xml`.

What follows is our request handler for releases but modified to explicitly configure the components:

```
<requestHandler name="mb_releases" class="solr.SearchHandler">
    <!-- default values for query parameters -->
    <lst name="defaults">
        <str name="defType">dismax</str>
        <str name="qf">r_name r_a_name^0.2</str>
        <str name="pf">r_name r_a_name^0.2</str>
        <str name="qs">1</str>
        <str name="ps">0</str>
        <str name="tie">0.1</str>
        <str name="q.alt">*:*</str>
        <str name="fq">type:Release</str>
    </lst>
    <!-- note: these components are the default ones -->
    <arr name="components">
        <str>query</str>
        <str>facet</str>
        <str>mlt</str>
        <str>highlight</str>
        <str>stats</str>
        <str>debug</str>
    </arr>
<!-- INSTEAD, "first-components" and/or "last-components"
    may be specified. -->
</requestHandler>
```

The named search components in the above XML comment are the default ones that are automatically registered if you do not specify the `components` section. This named list is also known as the **standard component list**. To specify additional components, you can either re-specify `components` with changes, or you can add it to the `first-components` or `last-components` lists, which are prepended and appended respectively to the standard component list.

Many components depend on other components being executed first, especially the query component, so you will usually add components to `last-components`.

Search components must be registered with Solr to be activated so that they can then be referred to in a components list. All of the standard components are pre-registered. Here's an example of how a search component named `elevator` is registered in `solrconfig.xml`:

```
<searchComponent name="elevator" class="solr.QueryElevationComponent" >
    <str name="queryFieldType">string</str>
    <str name="config-file">elevate.xml</str>
</searchComponent>
```

The rest of this chapter describes the different search component implementations that come with Solr, and one external one. The functionality in `QueryComponent`, `FacetComponent`, and `DebugComponent` has been described in previous chapters, and so will be omitted.

The highlighting component

You are probably most familiar with search highlighting when you use an Internet search engine like Google. Most search results come back with a snippet of text from the site containing the word(s) you search for, highlighted. Solr can do the same thing. In the following screenshot we see Google highlighing a search including **Solr** and **search** (in bold):

Introduction to The **Solr** Enterprise **Search** Server
Solr is a standalone enterprise **search** server with a web-services like API. You put documents in it (called "indexing") via XML over HTTP. **...**
lucene.apache.org/**solr**/features.html - 13k - Cached - Similar pages -

A non-obvious way to make use of the highlighting feature is to not actually do any highlighting. Instead Solr's highlighter can be used to inform the user which fields in the document satisfied their search, not to actually highlight the matched values. In this scenario, there would be a search that searches across many fields or a catch-all field and then `hl.fl` (the highlighted field list) is set to `*`. Of course a mix of approaches could be used depending on the fields.

A highlighting example

Admittedly the MusicBrainz data set does not make an *ideal* example to show off highlighting because there's no meaty text, but it can certainly be useful nonetheless.

The following is a sample use of highlighting on a search for `Corgan` in the artist MusicBrainz data set. Recall that the `mb_artists` request handler is configured to match against the artist name, alias, and members fields.

```
http://localhost:8983/solr/select?indent=on&q=corgan&rows=3&qt=mb_
artists&hl=true
```

And here is the output of the above URL:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">89</int>
</lst>
<result name="response" numFound="5" start="0">
    <doc>
        <date name="a_begin_date">1967-03-17T05:00:00Z</date>
        <str name="a_name">Billy Corgan</str>
        <date name="a_release_date_latest">
                    2005-06-21T04:00:00Z</date>
        <str name="a_type">1</str>
        <str name="id">Artist:102693</str>
        <str name="type">Artist</str>
    </doc>
    <doc>
        <str name="a_name">Billy Corgan &amp; Mike Garson</str>
        <str name="a_type">2</str>
        <str name="id">Artist:84909</str>
        <str name="type">Artist</str>
    </doc>
    <doc>
        <arr name="a_member_id"><str>102693</str></arr>
        <arr name="a_member_name"><str>Billy Corgan</str></arr>
        <str name="a_name">Starchildren</str>
        <str name="id">Artist:35656</str>
        <str name="type">Artist</str>
    </doc>
</result>
<lst name="highlighting">
    <lst name="Artist:102693">
        <arr name="a_name">
        <str>Billy &lt;em&gt;Corgan&lt;/em&gt;</str>
        </arr>
    </lst>
```

```
        <lst name="Artist:84909">
            <arr name="a_name">
            <str>Billy &lt;em&gt;Corgan&lt;/em&gt; &amp; Mike Garson</str>
            </arr>
        </lst>
        <lst name="Artist:35656">
            <arr name="a_member_name">
            <str>Billy &lt;em&gt;Corgan&lt;/em&gt;</str>
            </arr>
        </lst>
    </lst>
    </response>
```

What should be noted in this example is the manner in which the highlighting results are in the response data. Also note that not all of the result highlighting was against the same field.

It is possible to enable highlighting and discover that some of the results are not highlighted. Sometimes this can be due to complex text analysis, and more likely it could simply be that there is a mismatch between the fields queried and those highlighted.

Highlighting configuration

There are numerous configuration parameters for the highlighter component. Understand that like faceting, nearly all of these parameters can be overridden on a per-field basis. The syntax looks like `f.fieldName.paramName=value` for example: `f.allText.snippets=0`

Even though there are so many options, don't get overwhelmed. Like most things in Solr, the defaults are quite reasonable. The only parameter required is `hl`, which enables highlighting. You'll probably set `hl.fl`, and enable `hl.usePhraseHighlighter` and `hl.highlightMultiTerm` and a couple others that suit your fancy.

The following are the parameters observed by the highlighter search component:

- `hl`: Set to `true` to enable search highlighting. Without this, the other parameters are ignored, and highlighting is effectively disabled.
- `hl.fl`: A comma or space separated list of fields that will be highlighted. It is important for a field to be marked as stored in the schema in order to highlight on it. This requirement should be intuitive, as the surrounding text in highlighting has to come from somewhere. If this parameter is omitted, then it defaults to the default field(s) that are queried: `df` parameter for the standard handler or `qf` parameter for dismax. You may use an asterisk wildcard to conveniently highlight on all of the text fields, such as `*` or `r_*`. If you use a wildcard, then consider enabling the `hl.requiredFieldMatch` option.
- `hl.requireFieldMatch`: If set to `true`, then a field will not be highlighted for a result unless the query also matched against that field. This is `false` by default, meaning that it's possible to query one field and highlight another and get highlights back as long as the terms searched for are found within the highlighted field. If you use a wildcard in `hl.fl`, then you will probably enable this. However, if you query against an all-text catch-all field (probably using copy-field directives), then leave this as `false` so that the search results can indicate from which field the queried text was found in.
- `hl.usePhraseHighlighter`: If the query contained a phrase (it was quoted), then this will ensure that only the phrase is highlighted and not the words out of context of the queried phrase. So, if "a b c" is the query with quotes, then the "b" in the stored text "x b z" will not be highlighted if this option is enabled. This is strangely `false` by default, perhaps for legacy reasons, but you should probably always enable this.
- `hl.highlightMultiTerm`: If any wildcard or fuzzy queries are used, then this will ensure that the highlighting matches such terms correctly. This defaults to `false` and it requires `hl.usePhraseHighlighter`. You should probably always enable this.
- `hl.snippets`: This is the maximum number of highlighted snippets (aka fragments) that will be generated per field. It defaults to `1`, which you will probably not change. By setting this to `0` for a particular field (example: `f.allText.hl.snippets=0`), you can effectively disable highlighting for that field. You might do that if you used a wildcard for `hl.fl` and want to make an exception.
- `hl.fragsize`: The maximum number of characters returned in each snippet (aka fragment), measured in characters. The default is `100`. If `0` is specified, then the field is not fragmented and whole field values are returned. Obviously be wary of doing this for large text fields.

- `hl.mergeContiguous`: If set to `true`, then overlapping snippets are merged. The merged fragment size is not limited by `hl.fragsize`. The default is `false`, but you will probably set this to `true` when `hl.snippets` is greater than zero and `fragsize` is non-zero.
- `hl.maxAnalyzedChars`: The maximum number of characters in a field that will be sought for highlighting. If you want to disable the limit, then set this to `-1`. The default is `51200` characters.
- `hl.alternateField`: If a snippet couldn't be generated (no terms matched) for a field, then this parameter refers to a field that will be returned as the snippet. You might use some sort of summary field for a document or potentially the searched field itself. There is none by default.
- `hl.maxAlternateFieldLength`: The maximum number of characters to return for `hl.alternateField`. It's `0` by default, which means unlimited. Set this to something reasonably close to `hl.snippets * hl.fragsize` to maintain consistent sizing in the results.
- `hl.formatter`: This is an extension point to supply alternate formatting algorithms. The default is `simple`, which is presently the only supplied option. In the unlikely event that it does not suffice, then you can see how `org.apache.solr.highlight.HtmlFormatter.java` works and how it's registered in `solrconfig.xml` within the highlighting element.
    - `hl.simple.pre` and `hl.simple.post`: This is the text that will be inserted immediately before and after matched terms in the snippet in order to demarcate them from the surrounding text. The default is `<em>` and `</em>` (HTML emphasis tags). Note that the circumstantial presence of whatever values are chosen in the original text, such as HTML with pre-existing emphasis tags, are not escaped, and in rare circumstances may lead to a false highlight.
- `hl.fragmenter`: This is an extension point in Solr to specify the fragmenting algorithm. `gap` is the default typical choice based on a fragment size. `regex` is an alternative in which the highlighted fragment boundaries can be defined with a regular expression. It's an advanced choice that is atypical. In order to see how default settings and registration of fragmenters (and formatters) are configured, look in `solrconfig.xml` for the `highlighting` element.

The various options available for the `regex` fragmenter are as follows:

- `hl.regex.pattern`: This is a regular expression matching a block of text that will serve as the snippet/fragment to subsequently be highlighted. The default is `[-\w ,/\n\"']{20,200}`, which roughly looks for sentences. If you are using the `regex` fragmenter, then you will most likely tune a regular expression to your needs. The regular expression language definition used by Java and thus Solr is here at `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`.
- `hl.regex.slop`: This is the factor to which `hl.fragsize` can vary to accommodate the regular expression. The default is `0.6`, which means that fragment sizes may vary between `40` and `160` if `hl.fragsize` is `100`.
- `hl.increment`: Sets the minimum lucene position increment gap from one term to the next to trigger a new fragment. There is no `regex` in the parameter name but it is indeed only for the `regex` fragmenter. It defaults to `50`, which is fine.
- `hl.regex.maxAnalyzedChars`: For performance reasons, this puts a limit on the number of leading characters of the field that are fragmented based on the regular expression. After this limit is exceeded, the remaining characters up to `hl.maxAnalyzedChars` are fragmented in a fashion consistent with the `gap` fragmenter, which is faster. The default is `10000` characters.

The parts of the highlighting configuration that have configurable implementations (the fragmenter and formatter) have their defaults configured in `solrconfig.xml` within the `<highlighting/>` element if you wish to change them there. They can, of course, be configured in the URL too.

Query elevation

At times, it may be desired to make editorial (that is manual) modifications to the search results of particular user queries. This might be done as a solution to a popular user query that doesn't score an expected document sufficiently high (if it even matched at all). The query might have found nothing at all, perhaps due to a common misspelling. The opposite may also be true: the top result for a popular user query might yield a document that technically matched according to your search configuration, but certainly isn't what you were looking for. Another usage scenario is implementing a system akin to paid keywords for certain documents to be on top for certain user queries.

This feature isn't a general approach to fix queries not yielding effective search results; it is a band-aid for that problem. If a query isn't returning an expected document scored sufficiently high enough (if at all), then use Solr's query debugging to observe the score computation. You may end up troubleshooting text analysis issues too if a search query doesn't match an expected document—perhaps by adding a synonym. The end result may be tuning the boosts or applying function queries to incorporate other relevant fields into the scoring. When you are satisfied with the scoring and just need to make an occasional editorial decision, then this component is for you.

Configuration

This search component is not in the standard component list and so it must be registered with a handler in `solrconfig.xml`. Here we'll add it to the `mb_artists` request handler definition (just for this example, anyway).

```
<requestHandler name="mb_artists" class="solr.SearchHandler">
    <lst name="defaults">
        <!-- (omitted) -->
    </lst>
    <arr name="last-components">
        <str>elevateArtists</str>
    </arr>
</requestHandler>

<searchComponent name="elevateArtists"
                        class="solr.QueryElevationComponent" >
    <str name="queryFieldType">text</str>
    <str name="config-file">elevateArtists.xml</str>
    <str name="forceElevation">false</str>
</searchComponent>
```

This excerpt also reveals the registration of the search component using the same name as that referenced in `last-components`. A name was chosen reflecting the fact that this elevation configuration is only for artists. There are three named configuration parameters for a query elevation component, and they are explained as follows:

- `config-file`: This is a reference to the configuration file containing the editorial adjustments. As most other configuration files, it can be located in Solr's `conf` directory.

config-file can also be placed within the data directory (usually a sibling to conf) where it will be reloaded, when Solr's internal IndexReaders get reloaded which occurs for commits of new data, Solr core reloads, and some other events. This presents an interesting option if the elevation choices need to be loaded more often.

- queryFieldType: This is a reference to a field type in schema.xml. It is used to normalize both a query (the q parameter) and the query text attribute found in the configuration file for comparison purposes. A field type might be crafted just for this purpose, but it should suffice to simply choose one that at least performs lowercasing. By default, there is no normalization.
- forceElevation: The query elevation component fools Solr into thinking the specified documents matched the user's query and scored the highest. However, by default, it will not violate the desired sort as specified by the sort parameter. In order to force the elevated documents to the top no matter what sort is, set this parameter to true.

Let's take a peek at elevateArtists.xml:

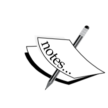
```
<elevate>
    <query text="corgan">
        <doc id="Artist:11650" /><!-- the Smashing Pumpkins -->
        <doc id="Artist:510" /><!-- Green Day -->
        <doc id="Artist:35656" exclude="true" /><!-- Starchildren -->
    </query>
  <!-- others queries... -->
</elevate>
```

In this elevation file, we've specified that when a user searches for corgan, the Smashing Pumpkins then Green Day should appear in the top two positions in the search results (assuming typical sort of score descending) and that the artist Starchildren is to be excluded. Note that query elevation kicks in when the configured query text matches the user's query exactly, while taking into consideration configured text analysis. Thus a search for billy corgan would not be affected by this configuration.

This component is quite simple with unsurprising results, so an example of this in action is not given. The only thing notable about the results when searching for corgan with the configuration mentioned above is that the top two results, the Smashing Pumpkins and Green Day, have scores of 1.72 and 0.0 respectively, yet the maxScore value in the result element is 11.3. Normally a default sort results in the first document having the same score as the maximum score, but in this case that happens at the third position, as the first two were inserted by this component. Moreover, normally a result document has a score greater than 0, but in this case one was inserted by this component that never matched the user's query.

Spell checking

One of the better ways to enhance the search experience is offering spelling corrections. This is sometimes presented at the top of search results with such text as "Did you mean ...". Solr supports this with the Solr spellcheck search component.



A related technique is to use fuzzy queries (that is the tilde syntax). However, fuzzy queries don't tell you what alternative spellings were used, are not as scalable for large indexes, and might require more programming than using this search component.

For spelling corrections to work, Solr must clearly have a corpus of words (for example, a dictionary) to suggest alternatives to those in the user's query. This component and the author use the term **dictionary** loosely as the collection of correctly known spelled words, and not their definitions. Solr can be configured in either of the following two ways:

- **A text file of words**: For a freely available English word list, check out **SCOWL** at `http://wordlist.sourceforge.net`.
- **Indexed content**: This is generally preferred, principally because your data contains proper nouns and other words not in a dictionary.

Before reading on about configuring spell checking in `solrconfig.xml`, you may want to jump ahead and take a quick peek at an example towards the end of this section, and then come back.

Schema configuration

If your dictionary is going to be based on indexed content as is recommended, then a field should be set aside exclusively for this purpose. This is so that it can be analyzed specially and so that other fields can be copied into it since the index-based dictionary uses just one field. Most Solr setups would have one field; our MusicBrainz searches, on the other hand, are segmented by the data type (example: artists, releases, tracks), and so one for each would be best. For the purposes of demonstrating this feature, we will only do it for artists.

In `schema.xml`, we need to define a field type for the indexed words and for querying them. We're going to use two different strategies: the conventional one named `textSpell`, and an alternative one named `textSpellPhrase` that will be described in a bit.

```
<!--
SpellCheck analysis config based off of http://wiki.apache.org/solr/
SpellCheckingAnalysis
-->
<fieldType name="textSpell" class="solr.TextField"
     positionIncrementGap="100" stored="false" multiValued="true">
    <analyzer type="index">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.SynonymFilterFactory"
            synonyms="synonyms.txt" ignoreCase="true" expand="true"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true"
                words="stopwords.txt"/>
        <filter class="solr.StandardFilterFactory"/>
        <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true"
                words="stopwords.txt"/>
        <filter class="solr.StandardFilterFactory"/>
        <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
    </analyzer>
</fieldType>

<!-- only useful for names & titles -->
<fieldType name="textSpellPhrase" class="solr.TextField"
    positionIncrementGap="100" stored="false" multiValued="true">
    <analyzer>
        <tokenizer class="solr.KeywordTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
</fieldType>
```

With text analysis, there is no one-size-fits-all.

The field type named `textSpell` is what the author recommends for most scenarios. However, an alternative approach only applicable to short strings like titles and names as we have with MusicBrainz is to treat the whole name as a single token. Optionally, a `regexp` can be used to take out unusual characters and to remove a few common stop-words like `the`.

A field type for spellchecking is not marked as stored because the `spellcheck` component only uses the indexed terms. The important thing is to ensure that the text analysis doesn't do stemming as the corrections presented would suggest the stems, which would look very odd to the user. It's also hard to imagine a use-case that doesn't apply lowercasing.

Now we need to create a field for this data:

```
<field name="a_spell" type="textSpell" />
<field name="a_spellPhrase" type="textSpellPhrase" />
```

And we need to get data into it with some `copyField` directives:

```
<copyField source="a_name" dest="a_spell" />
<copyField source="a_alias" dest="a_spell" />
<!-- (ditto for a_spellPhrase) -->
```

Arguably, `a_member_name` may be an additional choice to copy as well, as the dismax search we configured (seen below) searches it too, albeit at a reduced score. This as well as many decisions with search configuration can be subjective.

Configuration in solrconfig.xml

To use any search component, it needs to be in the components list of a request handler. The `spellcheck` component is not in the standard list so it needs to be added.

```
<requestHandler name="mb_artists" class="solr.SearchHandler">
    <!-- default values for query parameters -->
    <lst name="defaults">
        <str name="defType">dismax</str>
        <str name="qf">a_name a_alias^0.8 a_member_name^0.4</str>
        <!-- etc. -->
    </lst>
    <arr name="last-components">
        <str>spellcheck</str>
    </arr>
</requestHandler>
```

This component should already be defined in `solrconfig.xml`. Within the spellchecker search component, there are one or more XML blocks named `spellchecker` so that different dictionaries and other options can be configured. These might also be loosely referred to as the dictionaries, because the parameter that refers to this choice is named that way (more on that later). We have four spellcheckers as follows:

- `a_spell`: An index-based spellchecker that is a typical recommended configuration.
- `jarowinkler`: This uses the same built dictionary, `spellcheckIndexDir`, as `a_spell`, but contains an alternative configuration setting for experimentation.
- `a_spellPhrase`: Uses a less common approach, wherein the entire field is treated as a single word for correction.
- `file`: A sample configuration where the input dictionary comes from a file (not included).

A complete MusicBrainz implementation would have a different spellchecker for each MB data type, with all of them configured similarly.

Following the excerpt below is a description of all options available:

```
<!-- The spell check component can return a list of alternative
  spelling suggestions.  -->
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
    <str name="queryAnalyzerFieldType">textSpell</str><!-- ‘q' only -->

    <lst name="spellchecker">
        <str name="name">a_spell</str>
        <str name="field">a_spell</str>
        <str name="buildOnOptimize">true</str>
        <str name="spellcheckIndexDir">./spellchecker_a_spell</str>
    </lst>
    <lst name="spellchecker">
        <!-- Use previous spellchecker index with different
         distance measure -->
        <str name="name">jarowinkler</str>
        <str name="field">a_spell</str>
        <str name="distanceMeasure">
         org.apache.lucene.search.spell.JaroWinklerDistance</str>
        <str name="spellcheckIndexDir">./spellchecker_a_spell</str>
    </lst>
    <lst name="spellchecker">
        <str name="name">a_spellPhrase</str>
        <str name="field">a_spellPhrase</str>
```

```
        <str name="buildOnOptimize">true</str>
        <str name="spellcheckIndexDir">
                   ./spellchecker_a_spellPhrase</str>
    </lst>
    <!-- just an example -->
    <lst name="spellchecker">
        <str name="name">file</str>
        <str name="classname">solr.FileBasedSpellChecker</str>
        <str name="sourceLocation">spellings.txt</str>
        <str name="characterEncoding">UTF-8</str>
        <str name="spellcheckIndexDir">./spellcheckerFile</str>
    </lst>
</searchComponent>
```

Configuring spellcheckers (dictionaries)

The double layer of spellchecker configuration is perhaps a little confusing. The outer one just names the search component, it's mostly just a container. The inner ones are distinct configurations to choose at search time.

The following options are common to both index and file based spellcheckers:

- `name`: The name of the spellcheck configuration. It defaults to `default`. Be sure not to have more than one configuration with the same name.
- `classname`: The Java class name implementing the spellchecker. It defaults to the index implementation, `solr.IndexBasedSpellChecker`. The leading `solr.` is a shortcut abbreviation in Solr's configuration file for those internally defined by Solr. The other known implementation is `solr.FileBasedSpellChecker`.
- `spellcheckIndexDir`: This is a reference to the directory location where the spellchecker's internal dictionary is **built**, not its source. It is relative to Solr's data directory, which in turn defaults to being within the Solr home directory. This is actually optional, which results in an in-memory dictionary.

> In our spellchecker named `jarowinkler`, we're actually referring to another spellchecker's index so that we can try other configuration options without having to duplicate the data or building time. If this is done, be sure to use the `spellcheck.reload` command for this dictionary if it changes (as described later).
>
> For a high load Solr server, an in-memory index is appealing. Until SOLR-780 is addressed (slated for Solr 1.5), you'll have to take care to tell Solr to build the dictionary whenever the Solr core gets loaded. This happens at startup or if you tell Solr to reload a core.

- `buildOnCommit` and `buildOnOptimize`: These boolean options (defaulting to `false`) enable the spellchecker's internal index to be built automatically when either Solr performs a commit or optimize. However, enabling `buildOnOptimize` is recommended.
- `accuracy`: Sets the minimum spelling correction accuracy to act as a threshold. It falls between `0` and `1` with a default of `0.5`.
- `distanceMeasure`: This is a Java class name implementing the algorithm to gauge similarity between a possible misspelling and a candidate correction. It defaults to `org.apache.lucene.search.spell.LevensteinDistance` (which is the same algorithm used in fuzzy query matching). Alternatively, `org.apache.lucene.search.spell.JaroWinklerDistance` works quite well.
- `fieldType`: This is a reference to a field type in `schema.xml` for performing text-analysis on words to be spellchecked by the `spellcheck.q` parameter (not `q`). If this isn't specified, then it defaults to the field type of the field parameter (used by only the index-based spellchecker) and if not specified, then defaults to a simple whitespace delimiter, which most likely would be a misconfiguration. When using the file-based spellchecker with `spellcheck.q`, be sure to specify this.

IndexBasedSpellChecker options

The `IndexBasedSpellChecker` gets the dictionary from the indexed content of a field in a Lucene/Solr index, and the options are explained as follows:

- `sourceLocation`: If specified, then it refers to an external Lucene/Solr index path. This is an unusual choice but shows that the source dictionary does not need to come from Solr's main index; it could be from another location, perhaps from another Solr core. This is an advanced option. If you are doing this, then you'll probably also need to use the `spellcheck.reload` command mentioned later.

Warning: This option name is actually common to both types of spellcheckers but is defined differently.

- `field`: It is mandatory and refers to the name of the field within the index that contains the dictionary. Furthermore, it must be indexed as the data is taken from there, and not from the stored content, which is ignored. Generally, this field exists expressly for spell correction purposes and other fields are copied into it.

- `thresholdTokenFrequency`: Specifies a document frequency threshold, which will exclude words that don't occur often. This is expressed as a fraction in the range `0-1`, defaulting to `0`, which effectively disables the threshold, letting all words through.

If there is a lot of data and lots of common words (as opposed to proper nouns), then this threshold should be effective. If testing shows spelling candidates including strange fluke words found in the index, then introduce a threshold that is high enough to weed out such outliers. The threshold will probably be less than `0.01` (1 percent of documents).

FileBasedSpellChecker options

The `FileBasedSpellChecker` gets the dictionary from a plain text file.

- `sourceLocation`: Is mandatory and references a plain text file with each word on its own line. Note that an option by the same name but different meaning exists for the index-based spellchecker.
- `characterEncoding`: This is optional but should be set. It is the character encoding (example US-ASCII or UTF-8 or ISO-8601 or ...) of `sourceLocation`, defaulting to that of your operating system, which is probably not suitable.

We've not yet discussed the parameters to a search with the spellchecker component enabled. But at this point of the configuration discussion, understand that you have a choice of just letting the user query, `q` get processed or you can use `spellcheck.q`.

Processing of the q parameter

When a user query (`q` parameter) is processed by the `spellcheck` component to look for spelling errors, Solr needs to determine what words are to be examined. This is a two step process. The first step is to pull out the queried words from the query string, ignoring any Solr/Lucene syntax such as `AND`. The next step is to process the words with an analyzer so that, among other things, lowercasing is performed. The analyzer chosen is through a field type specified directly within the search component configuration with `queryAnalyzerFieldType`. It really should be specified, but it's actually optional. If left unspecified, there would be no text-analysis, which would in all likelihood be a misconfiguration.

This whole algorithm is implemented by a spellcheck **query converter**. The default query converter, which is known as `SpellingQueryConverter`, is probably fine. However, if you experience issues, then consult Solr's source and Wiki to see how to write your own and configure it.

Processing of the spellcheck.q parameter

If the `spellcheck.q` parameter is given (which really isn't a query per se), then the string is processed with the text analysis referenced by the `fieldType` option of the spellchecker being used. If a file-based spellchecker is being used, then you should set this explicitly. Index-based spellcheckers will sensibly use the field type of the referenced indexed spelling field. The dichotomy of the ways in which the analyzer is configured between both `q` and `spellcheck.q`, arguably needs improvement.

Building the dictionary from its source

Each spellchecker requires it to be **built**, which is the process in which the dictionary is read and is built into the `spellcheckIndexDir`. If it isn't built, then no corrections will be offered, and you'll probably be very confused. You'll be even more confused troubleshooting the results if it was built once but is far out of date and needs to be built again.

The strategy that generally works with the least hassle is to enable `buildOnOptimize` and to issue an optimize command when the entire data set is committed. Furthermore, ensure that `spellcheckIndexDir` is set so the results of the built spellcheck index is persisted between restarts.

Generally, building is required if it has never been built before, and it should be built periodically when the dictionary changes. It need not necessarily be built for every change, but it obviously won't benefit from any such modifications.

In order to perform a build of a spellchecker, simply enable the component (`spellcheck=true`), add a special parameter called `spellcheck.build`, and set it to `true`:

```
http://localhost:8983/solr/select?&qt=mb_artists&rows=0&spellcheck=
true&spellcheck.build=true&spellcheck.dictionary=jarowinkler
```

The other spellcheck parameters will be explained shortly. It is important to note that only one spellchecker (example: dictionary) was built. To build more than one, separate requests must be issued. Anecdotally, the time it took to build this dictionary of nearly 400K documents, each of which were very small, was 25 seconds on a mediocre machine.

There is an additional related option similar to `spellcheck.build` called `spellcheck.reload`. This doesn't rebuild the index, but it basically re-establishes connections with the index (both `sourceLocation` for index-based spellcheckers and `spellcheckIndexDir` for all types). If you've decided to have some external process building the dictionary or simply another spellchecker as we've done, then Solr needs to know how to reload it to see the changes—a quick operation.

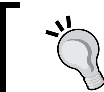
Issuing spellcheck requests

At this point, we've covered how to configure the spellchecker and dictionaries but not how to issue requests to actually use it. Let's do it! Fortunately there aren't many search parameters governing this end of the component. The parameters are explained as follows:

- `spellcheck`: This boolean must be set to `true` to enable the component in order to see suggested spelling corrections.
- `spellcheck.dictionary`: It references which dictionary (spellchecker) to use, configured in `solrconfig.xml`. It defaults to `default`.
- `spellcheck.q` or `q`: The string containing words to be processed by this component can be specified as the `spellcheck.q` parameter, and if not present, then the `q` parameter. Please look for the information presented earlier on how these are processed. But the gist of it is this: assuming you're handling user queries to Solr that might contain some query syntax, then the default `q` is right, as Solr will then know to filter out possible uses of Lucene/Solr's syntax (example: `AND`,`OR`, fieldname:word, and so on.). If not, then `spellcheck.q` is preferred, as it won't go through that unnecessary processing. It also allows its parsing to be different on a spellchecker-by-spellchecker basis, which we'll leverage in our example.
- `spellcheck.count`: In order to see more candidate corrections other than the top candidate, increase this beyond the default of `1`. Corrections are ordered by those closest to the original, as determined by the `distanceMeasure` algorithm.

Although counter-intuitive, raising this number can actually affect the result ordering. The results get better! The internal algorithm sees ~`10` times as many as this number and then it orders them by closest match. So raising this number to, let's say, `10` to balance performance with quality can positively increase the quality of the results, even though the Solr client may only be interested in the top result.

- `spellcheck.onlyMorePopular`: If this boolean option is enabled, then it will additionally offer spelling suggestions against words that were found in the index, so long as the suggestions occurred more often. If `extendedResults` is also enabled, then looking for `origFreq` being greater than `0` will indicate when this happens. This is disabled by default.



> This parameter is named poorly and gives results that are unlikely to be useful. Beware of using it!

- `spellcheck.extendedResults`: This boolean option adds frequency information (both for the original word, and for the suggestions). It defaults to `false` and is helpful when debugging.
- `spellcheck.collate`: This adds a revised query string to the output that alters the original query (from `spellcheck.q` or `q`) to use the top recommendation for each word offered. It defaults to `false` but should be enabled in most circumstances, as the user interface will most likely want to present a convenient link to use the spelling suggestions. It's smart enough to leave other query syntax in place if it is there.

Example usage for a mispelled query

We'll try out both the conventional spellcheck configuration `a_spell` and the whole name `a_spellPhrase` one. `jarowinkler` works identically, but simply has slightly different results (better or worse is subjective), so we won't bother. In all cases, we're going to enable `spellcheck.collate`, `spellchecheck.extendedResults`, and use `spellcheck.count` of `3`. I've disabled showing the query results with `rows=0`, because the actual query results aren't the point of these examples. In these examples, it is imagined that the user is searching for the band `Smashing Pumpkins` but with misspellings.

Here is a search on `a_spell` for `Smashinh Pumpkins`:

```
<?xml version="1.0"?>
<response>
    <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">124</int>
        <lst name="params">
            <str name="spellcheck">true</str>
            <str name="rows">0</str>
            <str name="indent">on</str>
            <str name="echoParams">explicit</str>
```

```
            <str name="spellcheck.collate">true</str>
            <str name="spellcheck.extendedResults">true</str>
            <str name="q">Smashinh Pumpkins</str>
            <str name="spellcheck.count">3</str>
            <str name="qt">mb_artists</str>
            <str name="version">2.2</str>
            <str name="spellcheck.dictionary">a_spell</str>
        </lst>
    </lst>
    <result name="response" numFound="0" start="0"/>
    <lst name="spellcheck">
        <lst name="suggestions">
            <lst name="smashinh">
                <int name="numFound">3</int>
                <int name="startOffset">0</int>
                <int name="endOffset">8</int>
                <int name="origFreq">0</int>
                <lst name="suggestion">
                    <int name="frequency">4</int>
                    <str name="word">smashing</str>
                </lst>
                <lst name="suggestion">
                    <int name="frequency">1</int>
                    <str name="word">smashin</str>
                </lst>
                <lst name="suggestion">
                    <int name="frequency">1</int>
                    <str name="word">slashing</str>
                </lst>
            </lst>
            <bool name="correctlySpelled">false</bool>
            <str name="collation">smashing Pumpkins</str>
        </lst>
    </lst>
</response>
```

Few of the output notes to take into consideration are as follows:

- The order of the suggestions for words are always ordered by so-called edit-distance (closest match), which is not displayed and isn't available. It may seem here that it is ordered by frequency, which is a coincidence.
- `startOffset` and `endOffset` are the index into the query of the spellchecked word. This information can be used by the client to display the query differently.
- `numFound` is always the number of suggestions returned, not the total number available if `spellcheck.count` were raised.
- `correctlySpelled` is intuitively `true` or `false`, depending on whether all of the query's words were found in the dictionary or not.

There are two very important caveats to understand about the `spellcheck` component, as we've configured it in a typical way (with `a_spell`):

Just because a spelling suggestion is offered does not mean that using it will yield matching documents. It just means that the individual corrected word is indeed in the index, and that's all. When there are multiple words in the query that are both mandatory (as is typical) then each word, after correction, might individually be in some document but not the same document. Thus the collation query will not necessarily work (`numFound` would be `0`). Our first example figured out exactly what the user wanted.

A query, which is comprised of multiple words that are all correctly spelled, yet whose query returns no results, will not result in spelling corrections. This should be obvious but it is worth stating.

It is plausible to imagine a future enhancement to the `spellcheck` component that would only offer corrections which would yield non-zero results. For performance reasons, such a feature would only be activated when one word is misspelled. In the meantime, it is possible for a client to do this with the results, but it would be much slower.

An alternative approach

Both problems mentioned in bold in the previous section can be fixed with an alternative strategy in which the entire field contents comprise the dictionary term like a phrase instead of each word individually. Scalability and practicality limit this approach to short name-like fields as we have in MusicBrainz. Therefore, this approach is preferred for a real-world use for MusicBrainz.

The size on the disk of `data/spellchecker_a_spell` is ~19MB compared to ~61MB for `data/spellchecker_a_spellPhrase`, which is even larger than the main index itself (60MB). This is not necessarily a problem, but is something to be aware of.

In order to use this technique, we must set `spellcheck.q` because the text analysis configuration mentioned earlier allows us the same that is used for the indexed field `a_spellPhrase`, which is non-tokenized.

Our search here is for `Smacking Pumpkin`. Boy, the user messed up here but it's close enough. A query like this using the previous approach with `a_spell` would fail.

```
<response>
    <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">547</int>
        <lst name="params">
            <str name="spellcheck">true</str>
            <str name="rows">0</str>
            <str name="spellcheck.collate">true</str>
            <str name="q">Smacking Pumpkin</str>
            <str name="qt">mb_artists</str>
            <str name="version">2.2</str>
            <str name="spellcheck.dictionary">a_spellPhrase</str>
            <str name="spellcheck.q">Smacking Pumpkin</str>
            <str name="indent">on</str>
            <str name="echoParams">explicit</str>
            <str name="spellcheck.extendedResults">true</str>
            <str name="spellcheck.count">3</str>
        </lst>
    </lst>
    <result name="response" numFound="0" start="0"/>
    <lst name="spellcheck">
        <lst name="suggestions">
           <lst name="smacking pumpkin">
                <int name="numFound">3</int>
                <int name="startOffset">0</int>
                <int name="endOffset">16</int>
                <int name="origFreq">0</int>
                <lst name="suggestion">
                    <int name="frequency">1</int>
                    <str name="word">smahing pumpkins</str>
                </lst>
```

```
                    <lst name="suggestion">
                        <int name="frequency">1</int>
                        <str name="word">smashing pumpkins</str>
                    </lst>
                    <lst name="suggestion">
                        <int name="frequency">1</int>
                        <str name="word">smashing pumkins</str>
                    </lst>
                </lst>
                <bool name="correctlySpelled">false</bool>
                <str name="collation">smahing pumpkins</str>
            </lst>
        </lst>
    </response>
```

All three corrections given are for the same document in the index, the Smashing Pumpkins. The first and third suggestions given happen to be stored in SP's `a_alias` field, which are not just aliases in the English sense of the word but within MusicBrainz are misspellings too. In light of this, `a_alias` shouldn't be added to the spelling index so that misspellings are not suggested.

The more-like-this search component

Have you ever searched for something and found a link that wasn't quite what you were looking for but was reasonably close? If you were using an Internet search engine like Google, then you may have tried the "more like this…" (or similar words like "find similar...") link that came with the search result to have the search engine show you pages similar to that one. This feature is also accessible from their web browser toolbar, which when invoked uses your current page as the input for the similarity search. Solr supports **more-like-this (MLT)** too.

The MLT capability in Solr can be used in the following three ways:

- **As a Solr component**: The MLT search component is infrequently used because it performs MLT analysis on each result returned in a search.
- **As a dedicated request handler**: It is more common for applications to request MLT to be performed on just one document when a user requests it, which occurs at some point after a search. The key input to this option is a reference to the indexed document that you want similar results for.

- **As a request handler with an external input document**: What if you want similarity results based on something that isn't in the index? A final option that Solr supports is returning MLT results based on text data sent to the MLT handler (through HTTP POST). For example, if you were to send a text file to the handler, then Solr's MLT handler would return the documents in the index that are most similar to it. This is atypical but an interesting option nonetheless.

The essences of the internal workings of MLT operate like this:

1. Gather all of the terms with frequency information from the input document:
    - If the input document is a reference to a document within the index, then loop over the fields listed in `mlt.fl`, and then the term information needed is readily there for the taking, if the field has the `termVectors` enabled. Otherwise get the stored text, and re-analyze it to derive the terms.
    - If the input document is posted as text to the handler, then analyze it to derive the terms. The analysis used is that configured for the first field listed in `mlt.fl`.
2. Filter the terms based on configured thresholds. What remains are only the "interesting terms".
3. Construct a query with these interesting terms across all of the fields listed in `mlt.fl`.

Configuration parameters

In the configuration options below, the input document is either each search result returned if MLT is used as a component, or it is the first document returned from a query to the MLT request handler, or it is the bulk text sent to the request handler. It simply depends on how you use it.

Parameters specific to the MLT search component

Using the MLT search component adorns an existing search with MLT results for each document returned.

- `mlt`: You must set this to `true` to enable MLT when using it as a search component. It defaults to `false`.
- `mlt.count`: The number of MLT results to be returned for each document returned in the main query. It defaults to `5`.

Parameters specific to the MLT request handler

Using the MLT request handler is more like a regular search except that the results are documents similar to the input document. Additionally, any filters (the `fq` parameter) that are specified are also in effect.

- `q`, `start`, `rows`: The MLT request handler uses the same standard parameters for the query start offset, and row count as used for querying. But in this case, `start` and `rows` is for paging into the MLT results instead of the results of the query. The query is typically one that simply references one document such as `id:12345` (if your unique field looks like this). `start` defaults to `0` and `rows` to `10`.
- `mlt.match.offset`: This parameter is the offset into the results of `q` for picking which document is the input document. It defaults to `0` so that the first result from `q` is chosen. As `q` will typically search for one document, this is rarely modified.
- `mlt.match.include`: The input document is normally included in the response if it is in the index (see the match element in the output of the example) because this parameter defaults to `true`. Set this to `false` to exclude this if that information isn't needed.
- `mlt.interestingTerms`: If this is set to `list` or `details`, then the so-called "interesting terms" that the MLT uses for the similarity query are returned with the results in an `interestingTerms` element. If you enable `mlt.boost`, then specifying `details` will additionally return the query boost value used for each term. `none` or `blank`, the default, disables this. Aside from diagnostic purposes, it might be useful to display these in the user interface, either listed out or in a tag cloud.

Use `mlt.interestingTerms` while experimenting with the results to get an insight into what the MLT query is actually doing.

- `facet`, ...: The MLT request handler supports faceting the MLT results. See the previous chapter on how to use faceting.

Additionally, remember to configure the MLT request handler in `solrconfig.xml`. An example of this is shown later in the chapter.

Common MLT parameters

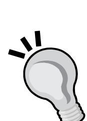
These parameters are common to both the search component and request handler MLT. Some of the thresholds here are for tuning which terms are "interesting" by MLT. In general, expanding thresholds (that is, lowering minimums and increasing maximums) will yield more useful MLT results at the expense of performance. The parameters are explained as follows:

- `mlt.fl`: A comma or space separated list of fields to consider in MLT. The "interesting terms" are searched within these fields only.

> These field(s) must be indexed. Furthermore, assuming the input document is in the index instead of supplied externally (as is typical), then each field should ideally have `termVectors` set to `true` in the schema (best for query performance although index size is a little larger). If that isn't done, then the field must be stored so that MLT can re-analyze the text at runtime to derive the term vector information. It isn't necessary to use the same strategy for each field.

- `mlt.qf`: Different field boosts can optionally be specified with this parameter. This uses the same syntax as the `qf` parameter used by the dismax handler (for example: `field1^2.0 field2^0.5`). The fields referenced should also be listed in `mlt.fl`. If there is a title/label field, then this field should probably be boosted higher.
- `mlt.mintf`: The minimum number of times (frequency) a term must be used within a document (across those fields in `mlt.fl` anyway) for it to be an "interesting term". The default is `2`. For small documents, such as in the case of our MusicBrainz data set, try lowering this to one.
- `mlt.mindf`: The minimum number of documents that a term must be used in for it to be an "interesting term". It defaults to `5`, which is fairly reasonable. For very small indexes, as little as `2` is plausible, and maybe larger for large multi-million document indexes with common words.
- `mlt.minwl`: The minimum number of characters in an "interesting term". It defaults to `0`, effectively disabling the threshold. Consider raising this to two or three.
- `mlt.maxwl`: The maximum number of characters in an "interesting term". It defaults to `0` and disables the threshold. Some really long terms might be flukes in input data and are out of your control, but most likely this threshold can be skipped.

- `mlt.maxqt`: The maximum number of "interesting terms" that will be used in an MLT query. It is limited to `25` by default, which is plenty.
- `mlt.maxntp`: Fields without `termVectors` enabled take longer for MLT to analyze. This parameter sets a threshold to limit the number of terms to consider in a given field to further limit the performance impact. It defaults to `5000`.
- `mlt.boost`: This boolean toggles whether or not to boost the "interesting terms" used in the MLT query differently, depending on how interesting the MLT module deems them to be. It defaults to `false`, but try setting it to `true` and evaluating the results.



Usage advice

For ideal query performance, ensure that `termVectors` is enabled for the field(s) used (those referenced in `mlt.fl`). In order to further increase performance, use fewer fields, perhaps just one dedicated for use with MLT. Using the `copyField` directive in the schema makes this easy. The disadvantage is that the source fields cannot be boosted differently with `mlt.qf`. However, you might have two fields for MLT as a compromise. Use a typical full complement of analysis (Solr filters) including lowercasing, synonyms, using a stop list (such as `StopFilterFactory`), and stemming in order to normalize the terms as much as possible. The field needn't be stored if its data is copied from some other field that is stored. During an experimentation period, look for "interesting terms" that are not so interesting for inclusion in the stop list. Lastly, some of the configuration thresholds, which scope the "interesting terms", can be adjusted based on experimentation.

MLT results example

Firstly, an important disclaimer on this example is in order. **The MusicBrainz data set is not conducive to applying the MLT feature, because it doesn't have any descriptive text.** If there were perhaps an artist description and/or widespread use of user-supplied tags, then there might be sufficient information to make MLT useful. However, to provide an example of the input and output of MLT, we will use MLT with MusicBrainz anyway.

If you're using the request handler method (the recommended approach), which is what we'll be using in this example, then it needs to be configured in `sorlconfig.xml`. The important bit is the reference to the class, the rest of it is our prerogative.

```
<requestHandler name="mlt_tracks" class="solr.MoreLikeThisHandler">
    <lst name="defaults">
        <str name="mlt.fl">t_name</str>
```

```
        <str name="mlt.mintf">1</str>
        <str name="mlt.mindf">2</str>
        <str name="mlt.boost">true</str>
    </lst>
</requestHandler>
```

This configuration shows that we're basing the MLT on just track names. Let's now try a query for tracks similar to the song "The End is the Beginning is the End" by The Smashing Pumpkins. The query was performed with `echoParams` to clearly show the options used:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">2</int>
    <lst name="params">
        <str name="mlt.mintf">1</str>
        <str name="mlt.mindf">2</str>
        <str name="mlt.boost">true</str>
        <str name="mlt.fl">t_name</str>
        <str name="rows">5</str>
        <str name="mlt.interestingTerms">details</str>
        <str name="indent">on</str>
        <str name="echoParams">all</str>
        <str name="fl">t_a_name,t_name,score</str>
        <str name="q">id:"Track:1810669"</str>
        <str name="qt">mlt_tracks</str>
    </lst>
</lst>
<result name="match" numFound="1" start="0" maxScore="16.06509">
    <doc>
        <float name="score">16.06509</float>
        <str name="t_a_name">The Smashing Pumpkins</str>
        <str name="t_name">The End Is the Beginning Is the End</str>
    </doc>
</result>
<result name="response" numFound="853390" start="0"
              maxScore="6.352738">
    <doc>
        <float name="score">6.352738</float>
        <str name="t_a_name">In Grey</str>
        <str name="t_name">End Is the Beginning</str>
    </doc>
```

```
        <doc>
            <float name="score">5.6811075</float>
            <str name="t_a_name">Royal Anguish</str>
            <str name="t_name">The End Is the Beginning</str>
        </doc>
        <doc>
            <float name="score">5.6811075</float>
            <str name="t_a_name">Mangala Vallis</str>
            <str name="t_name">Is the End the Beginning</str>
        </doc>
        <doc>
            <float name="score">5.6811075</float>
            <str name="t_a_name">Ape Face</str>
            <str name="t_name">The End Is the Beginning</str>
        </doc>
        <doc>
            <float name="score">5.052292</float>
            <str name="t_a_name">The Smashing Pumpkins</str>
            <str name="t_name">The End Is the Beginning Is the End</str>
        </doc>
    </result>
    <lst name="interestingTerms">
        <float name="t_name:end">1.0</float>
        <float name="t_name:is">0.7420872</float>
        <float name="t_name:the">0.6686879</float>
        <float name="t_name:beginning">0.6207893</float>
    </lst>
    </response>
```

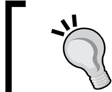
The result element named `match` is there due to `mlt.match.include` defaulting to `true`. The result element named `response` has the main MLT search results. The fact that so many documents were found is not material to any MLT response; all it takes is one interesting term in common. Perhaps the most objective number of interest to judge the quality of the results is the top scoring hit's score (`6.35`). The "interesting terms" were deliberately requested so that we can get an insight on the basis of the similarity. The fact that `is` and `the` were included shows that we don't have a stop list for this field—an obvious thing we'd need to fix. Nearly any stop list is going to have such words.

For further diagnostic information on the score computation, set `debugQuery` to `true`. This is a highly advanced method but exposes information invaluable to understand the scores. Doing so in our example shows that the main reason the top hit was on top was not only because it contained all of the interesting terms as did the others in the top 5, but also because it is the shortest in length (a high `fieldNorm`). The #5 result had "Beginning" twice, which resulted in a high term frequency (`termFreq`), but it wasn't enough to bring it to the top.

Stats component

This component computes some mathematical statistics of specified numeric fields in the index. The main requirement is that the field be indexed. The following statistics are computed over the non-null values (`missing` is an obvious exception):

- `min`: The smallest value.
- `max`: The largest value.
- `sum`: The sum.
- `count`: The quantity of non-null values accumulated in these statistics.
- `missing`: The quantity of records skipped due to missing values.
- `sumOfSquares`: The sum of the square of each value. This is probably the least useful and is used internally to compute `stddev` efficiently.
- `mean`: The average value.
- `stddev`: The standard deviation of the values.



As of this writing, the stats component does not support multi-valued fields. There is a patch added to SOLR-680 for this.

Configuring the stats component

This component performs a simple task and so as expected, it is also simple to configure.

- `stats`: Set this to `true` in order to enable the component. It defaults to `false`.
- `stats.field`: Set this to the name of the field in order to perform statistics on. It is required. This parameter can be set multiple times in order to perform statistics on more than one field.

- `stats.facet`: Optionally, set this to the name of the field in which you want to facet the statistics over. Instead of the results having just one set of stats (assuming one `stats.field`), there will be a set for each facet value found in this specified field, and those statistics will be based on that corresponding subset of data. This parameter can be specified multiple times to compute the statistics over multiple field's values. As explained in the previous chapter, the field used should be analyzed appropriately (that is, it is not tokenized).

Statistics on track durations

Let's look at some statistics for the duration of tracks in MusicBrainz at:

```
http://localhost:8983/solr/select/?rows=0&indent=on&qt=
mb_tracks&stats=true&stats.field=t_duration
```

And here are the results.

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">5202</int>
</lst>
<result name="response" numFound="6977765" start="0"/>
<lst name="stats">
    <lst name="stats_fields">
        <lst name="t_duration">
          <double name="min">0.0</double>
          <double name="max">36059.0</double>
          <double name="sum">1.543289275E9</double>
          <long name="count">6977765</long>
          <long name="missing">0</long>
          <double name="sumOfSquares">5.21546498201E11</double>
          <double name="mean">221.1724348699046</double>
          <double name="stddev">160.70724790290328</double>
        </lst>
    </lst>
</lst>
</response>
```

This query shows that on an average, a song is `221` seconds (or 3 minutes 41 seconds) in length. An example using `stats.facet` would produce a much longer result, which won't be given here in order to leave space for more interesting components. However, there is an example at `http://wiki.apache.org/solr/StatsComponent`.

Field collapsing

If you apply the patch attached to issue SOLR-236, then Solr supports field collapsing (that is result roll-up/aggregation). It is similar to an SQL `group by` query. In short, this search component will filter out documents from the results where a preceding document exists in the result that has the same value in a chosen field.

SOLR-236 is slated for Solr 1.5, but it's been incubating for years and has received the most number of user votes in JIRA.

For an example of this feature, consider attempting to provide a search for tracks where the tracks collapse to the artist. If a search matches multiple tracks produced by the same artist, then only the highest scoring track will be returned for that artist. That particular document in the results can be said to have rolled-up or collapsed those that were removed.

An excerpt of a search for `Cherub Rock` using the `mb_tracks` request handler collapsing on `t_a_id` (a track's artist) is as follows:

```
<response>
<lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">14</int>
    <lst name="params">
        <str name="collapse.field">t_a_id</str>
        <str name="rows">5</str>
        <str name="indent">on</str>
        <str name="echoParams">explicit</str>
        <str name="q">Cherub Rock</str>
        <str name="fl">score,id,t_a_id,t_a_name,t_name,t_r_name</str>
        <str name="qt">mb_tracks</str>
    </lst>
</lst>
<lst name="collapse_counts">
    <str name="field">t_a_id</str>
    <lst name="doc">
        <int name="Track:414903">68</int>
        <int name="Track:5358835">1</int>
    </lst>
    <lst name="count">
        <int name="11650">68</int>
        <int name="175552">1</int>
```

```
        </lst>
        <str name="debug">HashDocSet(18) Time(ms): 0/0/0/0</str>
    </lst>
    <result name="response" numFound="18" start="0" maxScore="15.212023">
        <!-- omitted result docs for brevity -->
    </result>
    </response>
```

The number of results went from 87 (which was observed from a separate query without the collapsing) down to 18. The `collapse_counts` section at the top of the results summarizes any collapsing that occurs for those documents that were returned (rows=5) but not for the remainder. Under the named `doc` section it shows the IDs of documents in the results and the number of results that were collapsed. Under the `count` section, it shows the collapsed field values—artist IDs in our case. This information could be used in a search interface to inform the user that there were other tracks for the artist.

Configuring field collapsing

Due to the fact that this component extends the built-in query component, it can be registered as a replacement for it, even if a search does not need this added capability. Put the following line by the other search components in `solrconfig.xml`:

```
<searchComponent name="query"
      class="org.apache.solr.handler.component.CollapseComponent"/>
```

Alternatively, you could name it something else like `collapse`, and then each query handler that uses it would have to have its standard component list defined (by specifying the components list) to use this component in place of the query component.

The following are a list of the query parameters to configure this component (as of this writing):

- `collapse.field`: The name of the field to collapse on and is required for this capability. The field requirements are the same as sorting—if text, it must not tokenize to multiple terms. Note that collapsing on multiple fields is not supported, but you can work around it by combining fields in the index.
- `collapse.type`: Either `normal` (the default) or `adjacent`. `normal` collapsing will filter out any following documents that share the same collapsing field value, whereas `adjacent` will only process those that are adjacent.
- `collapse.facet`: Either `after` (the default) or `before`. This controls whether faceting should be performed afterwards (and thus be on the collapsed results) or beforehand.

- `collapse.threshold`: By default, this is set to `1`, which means that only one document with the collapsed field value may be in the results—typical usage. By setting this to, say, `3` in our example, there would be no more than three tracks in the results by the Smashing Pumpkins. Any other track that would normally be in the results collapses to the third one.

> A possible use of this option is a search spanning multiple types of documents (example: Artists, Tracks, and so on), where you want no more than X (say 5) of a given type in the results. The client might then group them together by type in the interface. With faceting on the type and performing faceting before collapsing, the interface could tell the user the total of each type beyond those on the screen.

- `collapse.maxdocs`: This component will, by default, iterate over the entire search results, and not just those returned, in order to perform the collapsing. If many matched, then such queries might be slow. By setting this value to, say `200`, it will stop at that point and not do more collapsing. This is a trade-off to gain performance at the expense of an inaccurate total result count.
- `collapse.info.doc` and `collapse.info.count`: These are two booleans defaulting to `true`, which control whether to put the collapsing information in the results.

It bears repeating that this capability is not officially in Solr yet, and so the parameters and output, as described here, may change. But one would expect it to basically work the same way. The public documentation for this feature is at Solr's Wiki: `http://wiki.apache.org/solr/FieldCollapsing`. However, as of this writing, it is out of date and has errors. For the definitive list of parameters, examine `CollapseParams.java` in the patch, as that is the file that defines and documents each of them.

Other components

There are some other Solr search components too. What follows is a basic summary of a few of them.

Terms component

This component is used to expose raw indexed term information, including term frequency, for an indexed field. It has a lot of options for paging into this voluminous data and filtering out terms by term frequency. A possible use of this component is for implementing search auto-suggest. Recall that the faceting component described in the last chapter can be used for this too. The faceting component does a better job of implementing auto-suggest because it scopes the results to the user query and filter queries and is most likely the desired effect, while the `TermsComponent` does not. However, on the other hand, it is very fast as it is a more low-level capability than the facet component.

`http://wiki.apache.org/solr/TermsComponent`

termVector component

This component is used to expose the raw **term vector** information for fields that have this option enabled in the schema—`termVectors` set to `true`. It is `false` by default. The term vector is per field and per document. It lists each indexed term in order with the offsets into the original text, term frequency, and document frequency.

`http://wiki.apache.org/solr/TermVectorComponent`

LocalSolr component

LocalSolr is a third party search component. What it does is give Solr native abilities to query by vicinity of a latitude and longitude given a radial distance. Naturally, the documents in your schema need to have a latitude and longitude pair of fields. The query requires a pair of these to specify the center point of the query plus a radial distance. Results can be sorted by distance from the center. It's pretty straightforward to use. Note that it is not necessary to have this component do a location-based search in Solr. Given indexed location data, you can perform a query searching for a document with latitudes and longitudes in a particular numerical range to search in a box. This might be good enough, and it will be faster.

`http://www.gissearch.com/geo_search_intro`

Summary

Consider what you've seen with Solr search components: highlighting search results, editorially modifying query results for particular user queries, suggesting search spelling corrections, suggesting documents "more like this", calculating mathematical statistics of indexed numbers, collapsing/rolling-up search results. By now it should be clear why the text search capability of your database is inadequate for all but basic needs. Even Lucene-based solutions don't necessarily have the extensive feature-set that you've seen here. You may have once thought that searching was a relatively basic thing, but Solr search components really demonstrate how much more there is to it.

The chapters thus far have aimed to show you the majority of the features in Solr and to serve as a reference guide for them. The remaining chapters don't follow this pattern. In the next chapter, you're going to learn about various deployment concerns, such as logging, testing, security, and backups.

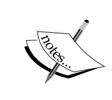
7
Deployment

Now that you have identified the data you want to search, defined the Solr schema properly, and done the tweaks to the default configuration you need, you're ready to deploy your new Solr based search to a production environment. While deployment may seem simple after all of the effort you've gone through, it brings its own set of challenges. In this chapter, we'll look at the following issues that come up when going from "Solr runs on my desktop" to "Solr is ready for the enterprise".

- Implementation methodology
- Install Solr into a Servlet container
- Logging
- A SearchHandler per search interface
- Solr cores
- JMX
- Securing Solr

Implementation methodology

There are a number of questions that you need to ask yourself in order to inform the development of a smooth deployment strategy for Solr. The deployment process should ideally be fully scripted and integrated into the existing **Configuration Management** (**CM**) process of your application.



Configuration Management is the task of tracking and controlling changes in the software. CM attempts to make the changes knowable that occur in software as it evolves to mitigate mistakes caused due to those changes.

Questions to ask

The list of questions to be asked is as follows:

- Is my deployment platform the same as my development and test environments? If I develop on Windows but deploy on Linux have I, for example, dealt with differences in file path delimiters?
- Do I have an existing build tool such as Ant with which to integrate the deployment process into?
- How will I get the initial data into Solr? Is there a nightly process in the application that will perform this step? Can I trigger the load process from the deploy script?
- Have I changed the source code for Solr? Do I need to version it in my own source control repository?
- Do I have full access to populate data in the production environment, or do I have to coordinate with System Administrators who are responsible for controlling access to production?
- Do I need to define acceptance tests for proving Solr is returning the appropriate results for a specific search?
- What are the defined performance-targets that Solr needs to meet?
- Have I projected the request rate to be served by Solr?
- Do I need multiple Solr servers to meet the projected load? If so, then what approach am I to use? Replication? Distributed Search? We cover this in-depth in Chapter 9.
- Will I need multiple indexes in a Multi Core configuration to support the dataset?
- Into what kind of Servlet container will Solr be deployed?
- What is my monitoring strategy? What level of logging detail do I need?
- Do I need to store data directories separately from application code directories?
- What is my backup strategy for my indexes, if any?
- Are any scripted administration tasks required (index optimizations, old snapshot removal, deletion of stale data, and so on)?

Installing into a Servlet container

Solr is deployed as a simple **WAR** (**Web application archive**) file that packages up servlets, JSP pages, code libraries, and all of the other bits that are required to run Solr. Therefore, Solr can be deployed into any Java EE Servlet Container that meets the Servlet 2.4 specifications, such as Apache Tomcat, Websphere, JRun, and GlassFish, as well as Jetty, which ships with Solr to run the example app.

Differences between Servlet containers

The key thing to resolve when working with Solr and the various Servlet containers is that, technically you are supposed to compile a single WAR file and deploy that into the Servlet container. It is the container's responsibility to figure out how to unpack the components that make up the WAR file and deploy them properly. For example, with Jetty you place the WAR file in the `/webapps` directory, but when you start Jetty, it unpacks the WAR file in the `/work` directory as a subdirectory, with a somewhat cryptic name that looks something like `Jetty_0_0_0_0_8983_solr.war__solr__k1kf17`. In contrast, with Apache Tomcat, you place the `solr.war` file into the `/webapp` directory. When you either start up Tomcat, or Tomcat notices the new `.war` file, it unpacks it into the `/webapp` directory. Therefore, you will have the original `/webapp/solr.war` and the newly unpacked (exploded) `/webapp/solr` version. The Servlet specification carefully defines what makes up a WAR file. However, it does not define exactly how to unpack and deploy the WAR files, so your specific steps will depend on the Servlet container you are using.

If you are not strongly predisposed to choosing a particular Servlet container, then consider Jetty, which is a remarkably lightweight, stable, and fast Servlet container. While written by the Jetty project, they have provided a reasonably unbiased summary of the differences in the projects here at `http://www.webtide.com/choose/jetty.jsp`.

Defining solr.home property

Probably, the biggest thing that trips up folks deploying into different containers is specifying the `solr.home` property. Solr stores all of its configuration information outside of the deployed `webapp`, separating the **data** part from the **code** part for running Solr. In the example app, while Solr is deployed and running from a subdirectory in `/work`, the `solr.home` directory is pointing to the top level `/solr` directory, where all of the data and configuration information is kept. You can think of `solr.home` as being analogous to where the data and configuration is stored for a relational database like MySQL. You don't package your MySQL database as part of the WAR file, and nor do you package your Lucene indexes.

By default, Solr expects the `solr.home` directory to be a subdirectory called `/solr` in the current working directory. With both Jetty and Tomcat you can override that by passing in a JVM argument that is somewhat confusingly namespaced under the solr namespace as `solr.solr.home`:

```
-Dsolr.solr.home=/Users/epugh/solrbook/solr
```

Alternatively, you may find it easier to specify the `solr.home` property by appending it to the `JAVA_OPTS` system variable. On Unix systems you would do:

```
export JAVA_OPTS=\"$JAVA_OPTS -Dsolr.solr.home=/Users/epugh/
       solrbook/solr"
```

Or lastly, you may choose to use JNDI with Tomcat to specify the `solr.home` property as well as where the `solr.war` file is located. **JNDI** (**Java Naming and Directory Interface**) is a very powerful, if somewhat difficult, to use directory service that allows Java clients such as Tomcat to look up data and objects by name.

By configuring the stanza appropriately, I was able to load up the `solr.war` and `/solr` directories from the example app shipped with Jetty under Tomcat. The following stanza went in the `/apache-tomcat-6-0.18/conf/Catalina/localhost` directory that I downloaded from `http://tomcat.apache.org`, in a file called `solr.xml`:

```
<Context docBase="/Users/epugh/solr_src/example/webapps/solr.war"
debug="0" crossContext="true" >
<Environment name="solr/home" type="java.lang.String"
         value="/Users/epugh/solr_src/example/solr" override="true" />
</Context>
```

I had to create the `./Catalina/localhost` subdirectories manually.

Note the somewhat confusing JNDI name for `solr.home` is `solr/home`. This is because JNDI is a tree structure, with the `home` variable being specified as a node of the Solr branch of the tree. By specifying multiple different context stanzas, you can deploy multiple separate Solrs in a single Tomcat instance.

Logging

Solr's logging facility provides a wealth of information, from basic performance statistics, to what queries are being run, to any exceptions encountered by Solr. The log files should be one of the first places to look when you want to investigate any issues with your Solr deployment. There are two types of logs:

- the HTTP server request style logs, which record the individual web requests coming into Solr
- the application logging that uses SLF4J, which uses the built-in Java JDK logging facility to log the internal operations of Solr

HTTP server request access logs

The HTTP server request logs record the requests that come in and are defined by the Servlet container in which Solr is deployed. For example, the default configuration for managing the server logs in Jetty is defined in `jetty.xml`:

```
<Ref id="RequestLog">
<Set name="requestLog">
<New id="RequestLogImpl" class="org.mortbay.jetty.NCSARequestLog">
<Arg><SystemProperty name="jetty.logs"
     default="./logs"/>/yyyy_mm_dd.request.log</Arg>
<Set name="retainDays">90</Set>
<Set name="append">true</Set>
<Set name="extended">false</Set>
<Set name="LogTimeZone">GMT</Set>
</New>
</Set>
</Ref>
```

The log directory is created in the subdirectory of the Jetty directory. If you have multiple drives and want to store your data separately from your application directory, then you can specify a different directory. Depending on how much traffic you get, you can adjust the number of days to preserve the log files. I recommend you keep the log files for as long as possible by archiving them. The search request data in these files can be very valuable for tuning Solr. By using web analytics tools such as a venerable commercial package WebTrends or the open source AWStats package to parse your request logs, you can quickly visualize how often different queries are run, and what search terms are frequently being used. This leads to a better understanding of what your users are searching for, versus what you expected them to search for initially.

Tailing the HTTP logs is one of the best ways to keep an eye on a deployed Solr. You'll see each request as it comes in and can gain a feel for what types of transactions are being performed, whether it is frequent indexing of new data, or different types of searches being performed. The request time data will let you quickly see performance issues. Here is a sample of some requests being logged. You can see the first request is a POST to the `/solr/update` URL from a browser running locally (127.0.0.1) with the date. The request was successful, with a 200 HTTP status code being recorded. The POST took 149 milliseconds. The second line shows a request for the admin page being made, which also was successful and took a slow 3816 milliseconds, primarily because in Jetty, the JSP page is compiled the first time it is requested. The last line shows a search for `dell` being made to the `/solr/select` URL. You can see that up to 10 results were requested and that it was successfully executed in 378 milliseconds. On a faster machine with more memory and a properly 'warmed' Solr cache, you can expect a few 10s of millisecond result time. Unfortunately you don't get to see the number of results returned, as this log only records the request.

```
127.0.0.1 - - [25/02/2009:22:57:14 +0000] "POST /solr/update HTTP/1.1"
200 149
127.0.0.1 - - [25/02/2009:22:57:33 +0000] "GET /solr/admin/ HTTP/1.1"
200 3816
127.0.0.1 - - [25/02/2009:22:57:33 +0000] "GET /solr/admin/
              solr-admin.css
          HTTP/1.1" 200 3846
127.0.0.1 - - [25/02/2009:22:57:33 +0000] "GET /solr/admin/favicon.ico
          HTTP/1.1" 200 1146
127.0.0.1 - - [25/02/2009:22:57:33 +0000] "GET /solr/admin/
              solr_small.png
          HTTP/1.1" 200 7926
127.0.0.1 - - [25/02/2009:22:57:33 +0000] "GET /solr/admin/favicon.ico
          HTTP/1.1" 200 1146
127.0.0.1 - - [25/02/2009:22:57:36 +0000] "GET /solr/select/
?q=dell%0D%0A&version=2.2&start=0&rows=10&indent=on
          HTTP/1.1" 200 378
```

While you may not see things quite the same way Neo did in the Matrix, you will get a good gut feeling about how Solr is performing!

AWStats is quite a full-featured open source request log file analyzer under the GPL license. While it doesn't have the GUI interface that WebTrends has, it performs pretty much the same set of analytics. AWStats is available from `http://awstats.sourceforge.net/`.

Solr application logging

Logging events is a crucial part of any enterprise system, and Solr uses Java's built-in logging (JDK [1.4] logging or JUL) classes provided by the `java.util.logging` package. However, this choice of a specific logging package has been seen as a limitation by those who prefer other logging packages, such as Log4j. Solr 1.4 resolves this by using the **Simple Logging Facade for Java** (**SLF4J**) package, which logs to another target logging package selected at runtime instead of at compile time. The default distribution of Solr continues to target the built-in JDK logging, but now alternative packages are easily supported.

Configuring logging output

By default, Solr's JDK logging configuration sends its logging messages to the standard error stream:

```
2009-02-26 13:00:51.415::INFO: Logging to STDERR via org.mortbay.log.
StdErrLog
```

Obviously, in a production environment, Solr will be running as a service, which won't be continuously monitoring the standard error stream. You will want the messages to be recorded to a log file instead. In order to set up basic logging to a file, create a `logging.properties` file at the root of Solr with the following contents:

```
# Default global logging level:
.level = INFO

# Write to a file:
handlers = java.util.logging.ConsoleHandler, java.util.logging.
FileHandler

# Write log messages in human readable format:
java.util.logging.FileHandler.formatter = java.util.logging.
SimpleFormatter
java.util.logging.ConsoleHandler.formatter = java.util.logging.
SimpleFormatter

# Log to the logs subdirectory, with log files named solrxxx.log
java.util.logging.FileHandler.pattern = ./logs/solr_log-%g.log
java.util.logging.FileHandler.append = true
java.util.logging.FileHandler.count = 10
java.util.logging.FileHandler.limit = 10000000 #Roughly 10MB
```

When you start Solr, you need to pass the following code snippet in the location of the `logging.properties` file:

```
>>java -Djava.util.logging.config.file=logging.properties -jar
  start.jar
```

By specifying two log handlers, you can send output to the console as well as log files. The FileHandler logging is configured to create up to 10 separate logs, each with 10 MB of information. The log files are appended, so that you can restart Solr and not lose previous logging information. Note, if you are running Solr under some sort of services tool, it is probably going to redirect the STERR output from the ConsoleHandler to a log file as well. In that case, you will want to remove the `java.util.ConsoleHandler` from the list of handlers. Another option is to reduce how much is considered as output by specifying `java.util.logging.ConsoleHandler.level = WARNING`.

Logging to Log4j

Most Java developers prefer Log4j over JDK logging. You might choose to configure Solr to use it instead, for any number of reasons:

- You're using a Servlet container that itself uses Log4j, such as JBoss. This would result in a more simplified and integrated approach.
- You wish to take advantage of the numerous Log4j **appenders** available, which can log to just about anything, including Windows Event Logs, SNMP (email), syslog, and so on.
- To use a Log4j compatible logging viewer such as:
    - **Chainsaw**—`http://logging.apache.org/chainsaw/`
    - **Vigilog**—`http://vigilog.sourceforge.net/`
- **Familiarity**—Log4j has been around since 1999 and is very popular.

The latest supported Log4j JAR file is in the 1.2 series and can be downloaded here at `http://logging.apache.org/log4j/1.2/`. Avoid **1.3** and **3.0**, which are defunct.

Alternatively, you might prefer to use Log4j's unofficial successor **Logback** at `http://logback.qos.ch/`, which improves upon Log4j in various ways, notably configuration options and speed. It was developed by the same person, Ceki Gülcü.

In order to change Solr to use Log4j, just remove the `slf4j-jdk14-1.5.5.jar` from the `webapp/WEB-INF/lib` directory and replace it with `slf4j-log4j12-1.5.5.jar`. Of course, you must also place Log4j's JAR file in that directory. You can find the various SLF4J distributions at `http://www.slf4j.org/dist/`. Make sure that you download the distribution that matches the version of SLF4J packaged with Solr or upgrade Solr's versions. Otherwise you may end up with JAR compatibility issues. As one poster to the `solr-dev` mailing list memorably called it: *JARmageddon*.

For information on configuring Log4j, log in to the web site at `http://logging.apache.org/log4j/`.

Jetty startup integration

Regardless of which logging solution you go with, you don't want to make the startup arguments for Solr more complex. You can leverage Jetty's configuration to specify these system properties during startup. Edit `jetty.xml` and add the following stanza to the outermost `<Configure id="Server" class="org.mortbay.jetty.Server"/>` element:

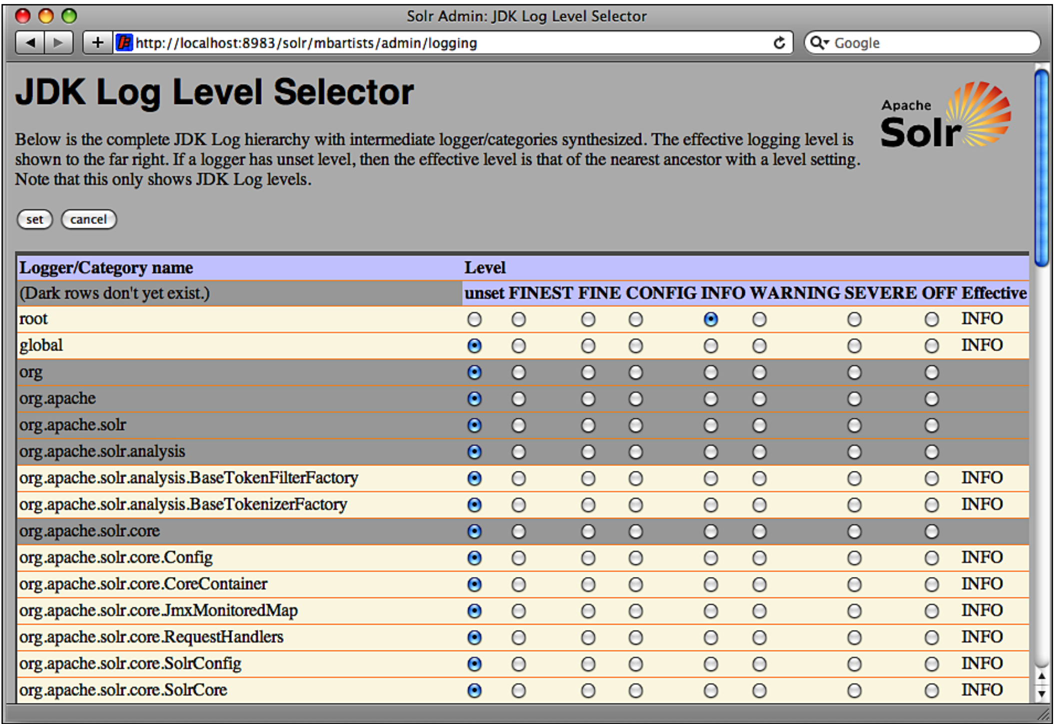
```
<Call class="java.lang.System" name="setProperty">
  <Arg>log4j.properties</Arg>
  <Arg>file:/Users/epugh/log4j.properties</Arg>
</Call>
```

Managing log levels at runtime

One of the challenges with most logging solutions is that you need to log enough details to troubleshoot issues, but not so much that your log files become ridiculously big and you can't winnow through all of the information to find what you are looking for. **Splunk** is a commercial product for managing log files and making actionable decisions on the information stored within.

There is more information at `http://www.splunk.com/`.

Sometimes you need more information then you are typically logging to debug a specific issue, so Solr provides an admin interface at `http://localhost:8983/solr/admin/logging` to change the logging verbosity of the components in Solr. Unfortunately, it only works with JDK logging.



While you can't change the overall setup of your logging strategy, such as the appenders or file rollover strategies at runtime, you can change the level of detail to log without restarting Solr. If you change a component like `org.apache.solr.core.SolrCore` to a fine grain of logging, then make a search request to see more detailed information. One thing to remember is that these customizations are NOT persisted through restarts of Solr. If you find that you are reapplying log configuration changes after every restart, then you should change your default logging setup to specify custom logging detail levels.

A SearchHandler per search interface

The two-fold questions to answer early on when configuring Solr is as follows:

- Are you providing generic search services that may be consumed by a variety of end user clients?
- Are you providing search to specific end user applications?

If you are providing generic search functionality to an unknown set of clients, then you may have just a single `requestHandler` handling search requests at `/solr/select`, which provides full access to the index. However, it is more likely that Solr is powering interfaces for one or more applications that are known to make certain kinds of searches. For example, say you have an e-commerce site that supports searching for products. In that case, you may want to only display products that are available for purchase. A specifically named `requestHandler` that always returns the stock products (using `appends`, as `fq` can be specified multiple times) and limits the rows to 50 (using `invariants`) would be appropriate:

```
<requestHandler name="/products" class="solr.SearchHandler" >
<lst name="invariants">
  <int name="rows">50</int>
</lst>
<lst name="appends">
  <str name="fq">inStock:true</str>
</lst>
</requestHandler>
```

However, the administrators of the same site would want to be able to find all products, regardless of if they are in stock or not. They would be using a different search interface and so you would provide a different request handler that returns all of the information available about your products:

```
<requestHandler name="/allproducts" class="solr.SearchHandler" />
```

Later on, if either your public site needs change, or if the internal searching site changes, you can easily modify the appropriate request handler without impacting other applications interacting with Solr.

> You can always add new request handlers to meet new needs by requiring the `qt` request parameter to be passed in the query like this: `/solr/select?qt=allproducts`. However, this doesn't look quite as clean as having specific URLs like `/solr/allproducts`. Fully named `requestHandler` can also have access to them controlled by use of Servlet security (see the *Security* section later in this chapter).

Solr cores

Recall from Chapter 2 that you can either put different types of data in the same index or use separate indexes. Up to this point, the only way you would know how to use separate indexes is to actually run multiple instances of Solr. However, adding another complete instance of Solr for each type of data you want to index is more heavyweight than needed. Introduced in Solr 1.3 are Solr cores, the solution to managing multiple indexes within a single Solr instance. As a result of hot core reloading and swapping, it also makes administering a single core/index easier. Each Solr core consists of its own configuration files and index of data. Performing searches and indexing in a multicore setup is almost the same as using Solr without cores. You just add the name of the core to the individual URLs. Instead of doing a search through the URL:

```
http://localhost:8983/solr/select?q=dave%20matthews
```

in a multicore environment, you would access a core named *mbartists* through:

```
http://localhost:8983/solr/mbartists/select?q=dave%20matthews
```

Other than the introduction of the core name in the URL, you still perform all of your management tasks, searches, and updates in the same way as you always did in a single core setup.

Configuring solr.xml

When Solr starts up, it checks for the presence of a `solr.xml` file in the `solr.home` directory. If one exists, then it loads up all the cores defined in `solr.xml`. We've used multiple cores in the sample Solr setup shipped with this book to manage the various indexes used in the examples. You can see the multicore configuration at `./examples/cores/solr.xml`:

```
<solr persistent="false" sharedLib="lib">
  <cores adminPath="/admin/cores">
      <core name="mbtracks" instanceDir="mbtracks"
            dataDir="../../cores_data/mbtracks" />
      <core name="mbartists" instanceDir="mbartists"
            dataDir="../../cores_data/mbartists" />
      <core name="mbreleases" instanceDir="mbreleases"
            dataDir="../../cores_data/mbreleases" />
      <core name="crawler" instanceDir="crawler"
            dataDir="../../cores_data/crawler" />
      <core name="karaoke" instanceDir="karaoke"
            dataDir="../../cores_data/karaoke" />
  </cores>
</solr>
```

Some of the key configuration values are:

- `persistent="false"` specifies that any changes we make at runtime to the cores, like copying them, are not persisted. If you want to persist between restarting the changes to the cores, then set `persistent="true"`. You would definitely do this if your indexing strategy called for indexing into a virgin core then swapping with the live core when done.
- `sharedLib="lib"` specifies the path to the `lib` directory containing shared JAR files for all the cores. If you have a core with its own specific JAR files, then you would place them in the `core/lib` directory. For example, the karaoke core uses Solr Cell (see Chapter 3) for indexing rich content, so the JARs for parsing and extracting data from rich documents are located in `./examples/cores/karaoke/lib/`.

Managing cores

While there isn't a nice GUI for managing Solr cores the way there is for some other options, the URLs you use to issue commands to Solr Cores are very straightforward, and they can easily be integrated into other management applications. If you specify `persistance="true"` in `solr.xml`, then these changes will be preserved through a reboot by updating `solr.xml` to reflect the changes. We'll cover a couple of the common commands using the example Solr setup in `./examples`. The individual URLs listed below are stored in plain text files in `./examples/7/` to make it easier to follow along in your own browser:

- `STATUS`: Getting the status of the current cores is done through `http://localhost:8983/solr/admin/cores?action=STATUS`.
  You can select the status of a specific core, such as `mbartists` through `http://localhost:8983/solr/admin/cores?action=STATUS&core=mbartists`. The status command provides a nice summary of the various cores, and it is an easy way to monitor statistics showing the growth of your various cores.
- `CREATE`: You can generate a new core called `karaoke_test` based on the karaoke core, on the fly, using the `CREATE` command through `http://localhost:8983/solr/admin/cores?action=CREATE&name=karaoke_test&instanceDir=./examples/cores/karaoke_test&config=./cores/karaoke/conf/solrconfig.xml&schema=./cores/karaoke/conf/schema.xml&dataDir=./examples/cores_data/karaoke_test`. If you create a new core that has the same name as an old core, then the existing core serves up requests until the new one is generated, and then the new one takes over.

- RENAME: Renaming a core can be useful to when you have fixed names of cores in your client, and you want to make a core fit that name. To rename the mbartists core to the more explicit core name music_brainz_artists, use the URL http://localhost:8983/solr/admin/cores?action=RENAME&core=mbartists&other=music_brainz_artists. This naming change only happens in memory, as it doesn't update the filesystem paths for the index and configuration directories and doesn't make much sense unless you are persisting the name change to solr.xml.
- SWAP: Swapping two cores is one of the key benefits of using Solr cores. Swapping allows you to have an offline "on deck" core that is fully populated with updated data. In a single atomic operation, you can swap out the current live core that is servicing requests with your freshly populated "on deck" core. As it's an atomic operation, your clients won't see any delay, and there isn't any chance of mixed data being sent to the client. As an example, we can swap the mbtracks core with the mbreleases core through http://localhost:8983/solr/admin/cores?action=SWAP&core=mbreleases&other=mbtracks. You can verify the swap occurred by going to the mbtracks Admin page and verifying that Solr Home is displayed as cores/mbreleases/.
- RELOAD: As you make minor changes to a core's configuration through solrconfig.xml or schema.xml, you don't want to be stopping and starting Solr constantly. In an environment with even a couple of cores, it can take some tens of seconds to restart all the cores and can go up to a couple of minutes. By using the reload command, you can trigger just a reload of a specific core without impacting the others. A good use of this is to configure a Solr core to be optimized for bulk indexing data. Once data is fully indexed, change the configuration to optimize for searching performance and just reload the core! A simple example for mbartists is http://localhost:8983/solr/admin/cores?action=RELOAD&core=mbartists.
- MERGE: The merge command is new to Solr 1.4 and allows you to merge one or more indexes into yet another core. This can be very useful if you've split data across multiple cores and now want to bring them together without re-indexing the data all over again. You need to issue commits to the individual indexes that are sources for data. After merging, issue another commit to make the searchers aware of the new data. The full set of commands using curl is listed in ./7/MERGE_COMMAND.txt.

Why use multicore

Solr's support of multiple cores in a single instance does more than enabling serving multiple indexes of data in a single Solr instance. Multiple cores also addresses some key needs for maintaining Solr in a production environment:

- **Rebuilding an index:** While Solr has a lot of features to handle doing sparse updates to an index with minimal impact on performance, occasionally you need to bulk update significant amounts of your data. This invariable leads to performance issues, as your searchers are constantly being reopened. By supporting the ability to populate a separate index in a bulk fashion, you can optimize the offline index for updating content. Once the offline index has been fully populated, you can use the `SWAP` command to take the offline index and make it the live index.
- **Testing configuration changes**: Configuration changes can have very differing impact depending on the type of data you have. If your production Solr has massive amounts of data, moving that to a test or development environment may not be possible. By using the `CREATE` and the `MERGE` commands, you can make a copy of a core and test it in relative isolation from the core being used by your end users. Use the `RELOAD` command to restart your test core to validate your changes. Once you are happy with your changes, you can either `SWAP` the cores or just reapply your changes to your live core and `RELOAD` it.
- **Merging separate indexes together:** You may find that over time you have more separate indexes than you need, and you want to merge them together. You can use the `mergeindex` command to merge two cores together into a third core. However, note that you need to do a commit on both cores and ensure that no new data is indexed while the merge is happening.
- **Renaming cores at runtime**: You can build multiple versions of the same basic core and control which one is accessed by your clients by using the `RENAME` command to rename a core to match the URL the clients are connecting to.

Why using multiple cores isn't the default approach?

Multi core support was first added in Solr 1.3 and has matured further in Solr 1.4. We strongly encourage you to start out with the multiple core approach, even if your `solr.xml` currently has a single core configured in it! While slightly more complex than just having a single index, doing this allows you to take advantage of all the administrative goodness for cores. Perhaps one day some of the commands like `RELOAD` and `STATUS` might eventually be supported in a single core, or perhaps the single core configuration might become deprecated. Multiple cores will be the key to Solr's future support for massively distributed indexes and/or huge numbers of individualized indexes. Therefore, you can expect to see it continue to evolve.

You can learn more about Solr core related features at `http://wiki.apache.org/solr/CoreAdmin`.

JMX

Java Management Extensions (**JMX**) is a Java standard API for monitoring and managing applications and network services. Originally meant to help with server administration, it was added to J2SE version 5 and is becoming more widely supported. JMX enabled applications and services expose information and available operations for resources such as **MBeans** (**Managed Bean**). MBeans can be managed remotely by a wide variety of management consoles such as the JConsole GUI that comes with Java and the web based JMX Console provided by JBoss. While fairly bare bones, the JMX Console is easy to use and provides significant management control over JBoss. More information on the JMX Console is available at `http://www.jboss.org/community/docs/DOC-10941`.

As of version 1.4, Solr exposes information about its components through MBeans. However, actual management operations, such as re-indexing information, are not exposed through JMX. You can still leverage JMX to monitor the status of Solr, and in large enterprise environments the JMX standard simplifies integrating monitoring tasks into existing monitoring platforms. A great use for JMX is to find out how many documents you've indexed in Solr.

Starting Solr with JMX

In `solrconfig.xml`, the stanza `<jmx/>` needs to be uncommented to enable JMX support. In order to actually start up with JMX, you need to provide some extra parameters to support remote connections, including the port to be connect to:

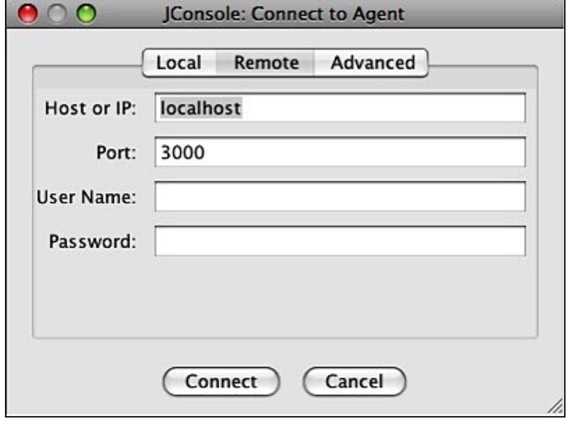
```
>>java -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.
port=3000 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.
management.jmxremote.authenticate=false -jar start.jar
```

However, this configuration is totally insecure. In a production environment, you would want to require usernames and passwords for access. For more information, please refer to the JMX documentation at `http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html#remote`.

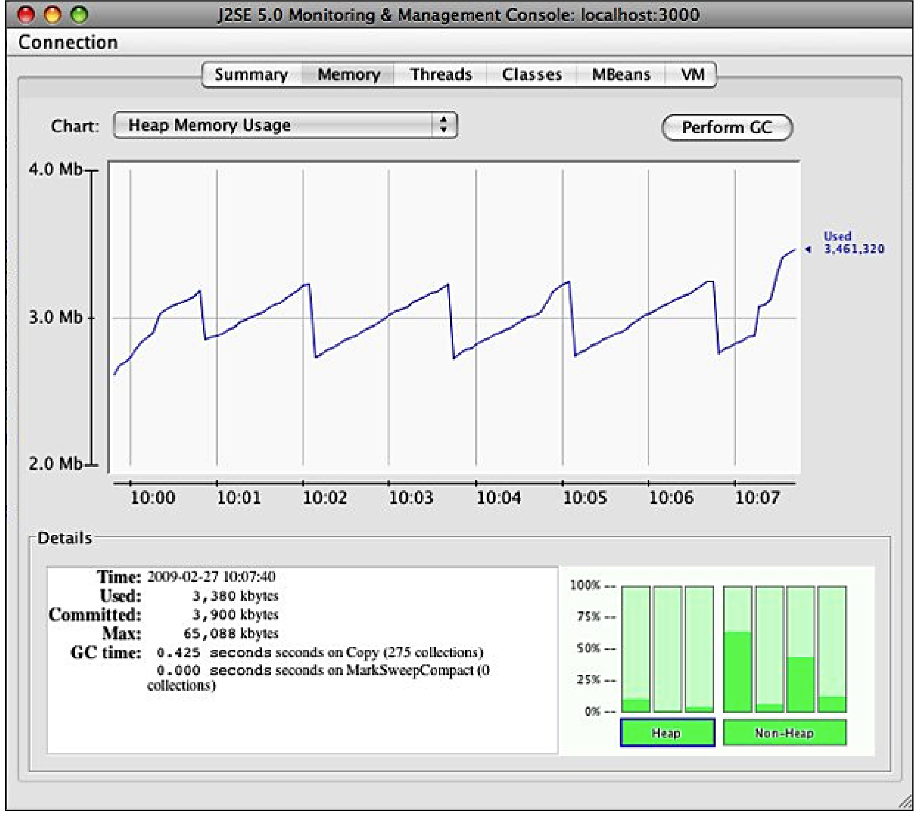
J2SE ships with JConsole, a GUI client for connecting to JMX servers. In order to start it, run the following command:

```
>> [JDK_HOME]/bin/jconsole
```

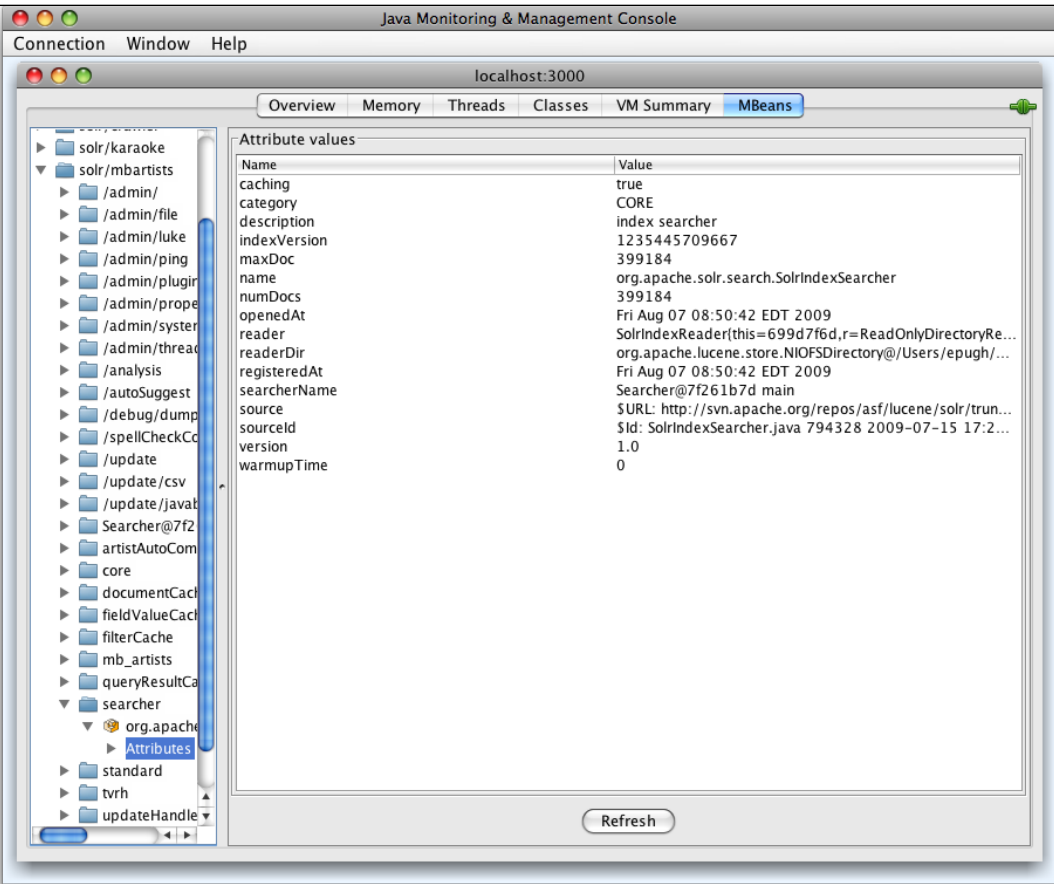
In order to connect to Solr, choose the **Remote** tab, and enter **localhost** for the **Host or IP** and **3000** for the **Port**. As we have started without requiring authentication, you do not need to enter a username and password.



For Solr, the key tabs to use in JConsole are **Memory** and **MBeans**. **Memory** provides a visual charting of the consumption of memory and can help you monitor low memory situations and when to start optimizing your indexes (as discussed in Chapter 9).



You can also monitor the various components of Solr by choosing the **MBeans** tab. In order to find out how many documents you've indexed, you would look at the **SolrIndexSearch** Mbean. Select **solr** from the tree listing on the left, and drill down to the **searcher** folder and select the **org.apache.solr.search.SolrIndexSearcher** component. You can see in the screenshot below that there are currently 15 documents indexed and the most ever was 25 documents. While you can pull this type of information out of the admin statistics web page, the JMX standard provides a much simpler method that can be easily integrated into other tools.



In order to save yourself typing in the extra startup parameters, see the previous *Jetty Startup Integration* section for how to add these JMX startup parameters like `-Dcom.sun.management.jmxremote` to your Jetty configuration.

Take a walk on the wild side! Use JRuby to extract JMX information

While JConsole is useful as a GUI application, it is hard to integrate into a larger environment. However, by leveraging the standard nature of JMX, we can easily script access to Solr components to use in our own monitoring systems. This makes it easy to expose extra information to our users such as '15 documents are available for searching'. There are a number of scripting packages for Java that you might look at, including Jython, Groovy, and BeanShell. However, we are going to look at using JRuby to access information.

JRuby is an implementation of the Ruby language running on the Java Virtual Machine that blends the library support of Java with the simplicity of the Ruby language in a winning combination. More information is available at `http://jruby.codehaus.org/`. JRuby is very simple to install on Unix using your operating system's package manager. Installation directions are available at `http://wiki.jruby.org/wiki/Getting_Started`.

Once you have JRuby installed, you need to install the `jmx4r` gem that provides the simple interface to JMX. The Ruby standard is to package functionality in **gems**, which are similar to traditional Java JAR files.

```
>>jruby -S gem install jmx4r
```

Assuming you have started Solr with JMX enabled on port 3000, you are now ready to interactively query Solr for status through JMX using the **JRuby Interactive Browser** (**JIRB**) tool. JIRB allows you to type in a Ruby code and interactively manipulate your environment.

Start JIRB from the command line by running the following command:

```
>> jirb
```

And enter the following commands at the interpreter prompts:

```
require 'rubygems'
require 'jmx4r'
JMX::MBean.establish_connection :port => 3000
```

You now have an interactive connection to the running Solr through JMX. In order to find out how many queries have been issued, you just request the searcher MBean by name `solr:type=searcher,id=org.apache.solr.search.SolrIndexSearcher`:

```
searcher = JMX::MBean.find_by_name
      "solr:type=searcher,id=org.apache.solr.search.SolrIndexSearcher"
```

You may need to use JConsole to figure out the name of the MBean that you want. Simply select the **Info** tab for a specific MBean, and use the MBean name attribute. Once you have the MBean, you can view all available attributes in a hash data structure by typing the following snippet of code:

```
irb(main):013:0> searcher.attributes
=> {"source_id"=>"sourceId", "category"=>"category", "description"=>"d
escription", "source"=>"source", "name"=>"name", "version"=>"version",
"searcher_name"=>"searcherName", "caching"=>"caching", "num_
docs"=>"numDocs", "max_doc"=>"maxDoc", "reader"=>"reader",
"reader_dir"=>"readerDir", "index_version"=>"indexVersion",
"opened_at"=>"openedAt", "registered_at"=>"registeredAt", "warmup_
time"=>"warmupTime"}
```

The attribute `searcher.num_docs` will return the current number of indexed documents in Solr.

Returning to our previous example of finding out how many documents are in the index, you just need to issue the following:

```
>> jirb
require 'rubygems'
require 'jmx4r'
JMX::MBean.find_by_name
    ("solr:type=searcher,id=org.apache.solr.search.
SolrIndexSearcher").num_docs
=> "15"
```

You can now integrate this Ruby script into some sort of regular process that saves the number of documents in your database, so you can display that information to your users.

You also can now start getting information about other parts of the system, like how many search queries have been issued per second, and how long are they averaging, by looking at the search handler MBean:

```
search_handler = JMX::MBean.find_by_name
    "solr:type=standard,id=org.apache.solr.handler.component.
     SearchHandler"
search_handler.avg_requests_per_second
=> .0043345
search_handler.avg_time_per_request
=> 45.0
```

In order to see all the available Solr Mbean's and their JMX names, just issue:

```
puts JMX::MBean.find_all_by_name("solr:*").map{ |mbean| mbean.object_
name}
```

Ruby is a wonderful language for scripting utility tasks that interact with Solr and other systems.

Jmx4r is hosted at `http://github.com/jmesnil/jmx4r/` and has a comprehensive suite of tests and example code. It's a good library to look at for tips on using JRuby.

Securing Solr

Solr, by default, comes completely open. Anyone can make search requests, anyone can upload documents, anyone can access the administration interface, and anyone can delete data. Solr is built like this because it is designed to fit into any environment, and if it is shipped with significant security functionality built-in, then it wouldn't be as wonderfully flexible as it is. Having said this, it isn't difficult to lock down Solr to use in any kind of environment. We can do this by making use of the standard practices, which you would apply to any kind of web application.

Limiting server access

The single biggest thing you can do to secure Solr is to lock down who has access to the server. Using standard firewall techniques, you can control what IP addresses are allowed to connect to the Solr through the 8983 port. Unless you have very unusual needs, you won't expose Solr to the Internet directly; instead users will access Solr through some sort of web application, that in turn forwards requests to Solr, collects the results, and displays them to your users. By limiting the IP addresses that can connect to Solr to just those belonging to your web farm, you've ensured that random Internet users and internal users don't mess with Solr.

If you lock down access through IP addresses, then don't forget that if you have external processes uploading content, you need to make sure those IP addresses are added.

Using IP addresses to control access is like using a sledge hammer and doesn't help if someone is connecting to Solr from one of the valid IP addresses. Fortunately, Solr is just a WAR file deployed in a Servlet container, so you can use all of the capabilities of Servlet containers to control access. In order to limit access to `/solr/update*` and `/solr/admin/*` in Jetty by requiring BASIC authentication from your users, you merely edit the `web.xml` for Solr by adding the following stanza at the bottom:

```
<security-constraint>
<web-resource-collection>
<web-resource-name>Solr Admin</web-resource-name>
<url-pattern>/admin/*</url-pattern>
</web-resource-collection>
<auth-constraint>
<role-name>admin</role-name>
</auth-constraint>
</security-constraint>

<security-constraint>
<web-resource-collection>
<web-resource-name>Solr Update</web-resource-name>
<url-pattern>/update*</url-pattern>
</web-resource-collection>
<auth-constraint>
<role-name>admin</role-name>
<role-name>content_updater</role-name>
</auth-constraint>
</security-constraint>

<login-config>
<auth-method>BASIC</auth-method>
<realm-name>Test Realm</realm-name>
</login-config>
```

This specifies that access to the `/update*` URLs is limited to anyone in the roles of `admin` or `content_updater`, although only admin users can access the `/admin/*` URLs. The `realm-name` is what ties the security constraints to the users configured in Jetty.

Customizing web.xml in Jetty

Sometimes cracking open a WAR file just to customize the `web.xml` can be a pain. But if you are a Jetty user, then you can put the changes into the `./etc/webdefault.xml` file and Jetty will apply the changes to any WAR file deployed. This is a nice trick if you have just a single `webapp` in the Jetty container. See `./examples/solr/etc/webdefault.xml` and `./examples/solr/etc/jetty.xml` for an example.

Edit the `jetty.xml` file and uncomment the `<Set name="UserRealms"/>` stanza so that it looks like the following:

```
<Set name="UserRealms">
<Array type="org.mortbay.jetty.security.UserRealm">
<Item>
<New class="org.mortbay.jetty.security.HashUserRealm">
<Set name="name">Solr Realm</Set>
<Set name="config"><SystemProperty name="jetty.home" default="."/>
     /etc/realm.properties</Set>
</New>
</Item>
</Array>
</Set>
```

The `./etc/realm.properties` file contains a list of users with their password and roles to which they belong. We've specified that the user named `administrator` has the roles of `content_updater` and `admin`, and therefore can access any `/update` and `/admin` URLs. However, the user `eric` can only access the `/update` URLs:

```
administrator: $ecretpa$$word,content_updater,admin
eric: mypa$$word, content_updater
guest: guest,read-only
```

Adding authentication introduces an extra roadblock for automated scripts that need to interact with Solr to upload information. However, if you use BASIC authentication, then you can easily pass the username and password as part of the URL request. The only downside is that the password is being transmitted in cleartext, and you should wrap the entire request in SSL for maximum security:

```
http://USERNAME:PASSWORD@localhost:8080/solr/update
```

Normally you wouldn't want to store passwords in plain text on the server in a file such as `realm.properties`, because they can be obfuscated or encrypted. More information is available at `http://docs.codehaus.org/display/JETTY/Realms`.

Securing public searches: Although you can get all access to Solr through a controlled web application, you may want to expose Solr publicly, albeit in a limited way. One scenario for this is exposing a search in an RSS/Atom feed made possible with Solr's XSLT support (see Chapter 4 for more on XSLT). Another is using AJAX from the end user's browser to perform search. Due to AJAX restrictions, that scenario would require the web application and Solr to be on the same server.

If using JavaScript to access Solr interests you, then check out SolrJS which is a JavaScript API to Solr. It is currently in transition to becoming a `contrib` module for eventual inclusion with Solr. In the mean time, more information can be found at `http://wiki.apache.org/solr/SolrJS`, including links to a demonstration site featuring faceted search. SolrJS is a great example of how easy it is to integrate Solr in new and interesting ways.

There may be other scenarios where firewall rules and/or passwords might still be used to expose parts of Solr, such as for modifying the index, but some search requests must be exposed to the public. In this case, you need to configure the exposed request handlers with `invariants` and/or `appends` clauses as applicable. For a limited example of this, see the *A SearchHandler per search interface* section earlier in this chapter.

If there are certain records needed to be excluded, then you'll need to specify an appropriate `fq` (filter query). If there are certain fields on documents that need to be kept private, then this can be problematic to completely secure, especially if you are working with sensitive data. It's simple enough to specify `fl` (field list) through `invariants`, but there are a good number of other parameters that might expose the data (for example, highlighting, maybe faceting) in ways you didn't realize. Therefore, if you are working with sensitive data and you have private fields, then exposing Solr in this way is not recommended.

Controlling JMX access

If you have started Solr with JMX enabled, then you should also have a JMX username and password configured. While today the JMX interface only exposes summary information about the Solr components and memory consumption, in future versions actual management options like triggering optimizing indexes will most likely be exposed through JMX. So, putting JMX access under lock and key is a good idea.

Securing index data

One of the weaknesses of Solr due to the lack of a built-in security model is that there aren't well defined approaches for controlling which users can manipulate the indexes by adding, updating, and deleting documents, and who can search what documents. Nevertheless, there are some approaches for controlling access to documents being searched.

Controlling document access

You can start off with some of the ideas talked about in the *A SearchHandler per search interface* section to control search access to your index. However, if you need to control access to documents within your index and must control it based on the user accessing the content, then one approach is to leverage the faceted search capabilities of Solr. You may want to look back at Chapter 5 to refresh your memory on faceting. For example, you may have a variety of documents that have differing visibility depending on if someone is a member of the public or an internal publicist. The public can only see a subset of the data, but a publicist can see more information, including information that isn't ready for public viewing. When indexing documents, you should store in a separate `multiValued` field the roles that a user must belong to in order to gain access to the document:

```
<field name="roles" type="text" indexed="true" stored="true"
multiValued="true" />
```

A document that was for everyone would be indexed with the role values Public and Publicist. Another document that was for internal use would just have the Publicist role. Then, at query time, you could append extra request parameters to limit what is returned depending on the roles that someone belonged to by treating the roles as a facet:

```
/solr/select/?q=music&start=0&facet=on&facet.field=roles&fq=role%3Apu
blic
```

In the example above, we are querying for `music` that is accessible by anyone with the role `public`. Obviously, this requires significant logic to be implemented on the client side interfacing with Solr, and is not as robust a solution as we may wish.

Other things to look at

Remote streaming is the ability to give Solr the URL to a remote resource or local file and have Solr download the contents as a stream of data. This can be very useful when indexing large documents as it reduces the amount of data that your updating process needs to move around. However, it means that if you have the `/debug/dump` request handler enabled, then the contents of any file can be exposed. Here is an example of displaying to anyone my `~/.ssh/authorized_keys` file:

```
http://localhost:8983/solr/debug/dump?stream.file=/Users/epugh/.ssh/
authorized_keys
```

If you have this turned on, then make sure that you are monitoring the log files, and also the access to Solr is tightly controlled. The example application has this function turned on by default.

In addition, in a production environment, you want to comment out the `/debug/dump` request handler, unless you are actively debugging an issue.

Summary

We briefly covered a wide variety of the issues that surround taking a Solr configuration that works in a development environment and getting it ready for the rigors of a production environment. Solr's modular nature and stripped down focus on search allows it to be compatible with a broad variety of deployment platforms. Solr offers a wealth of monitoring options, from log files, to HTTP request logs, to JMX options. Nonetheless, for a really robust solution, you must define what the key metrics are, and then implement automated solutions for tracking them.

Now that we have set up our Solr server, we need to take advantage of it to build better applications. In the next chapter, we'll look at how to easily integrate Solr search through various client libraries.

8
Integrating Solr

As the saying goes, if a tree falls in the woods and no one hears it, did it make a sound? Therefore, if you have a wonderful search engine, but your users can't access it, do you really have a wonderful search engine? Fortunately, Solr is very easy to integrate into a wide variety of client environments by dint of its very modern, easy-to-use RESTful interface. In this chapter, we will:

- Look at accessing Solr results through various language based clients, including Java, Ruby, PHP, and JavaScript.
- Learn how you can meet the needs of new applications by leveraging Solr's support for JSON.
- Briefly cover building your own Google-like search engine by crawling the `MusicBrainz.org` site with the Heritrix web crawler and indexing with Solr.
- Talk about the advantages of replacing the direct use of the Lucene library with an Embedded Solr.

We will look through a couple of examples using our MusicBrainz dataset. You can download the full sample code for these integrations from the Packt Publishing web site. This includes a prebuilt Solr instance and scripts to load `mbtracks` with seven million records and `mbartists` with 400,000 records. When you have downloaded the zipped file, you should follow the setup instructions in the `README.txt` file.

Structure of included examples

We have tried (where appropriate) to include a wide variety of sample integrations that you can run as you work through this chapter. These various examples stored in `./examples/8/` are as self-contained as we could make them, and you shouldn't run into any problems making them work. Check the support section of the book web site for any errata.

Inventory of examples

This is a quick summary of the various examples of using Solr, available in `./examples/8/`.

- **crawler** is an example of doing web crawling and integrating Solr through the Java SolrJ client.
- **jquery_autocomplete** is an example of using the jQuery Autocomplete library to populate suggestions based on Solr searches.
- **solrjs** is an example of building a fully featured Solr Search UI using just JavaScript.
- **php** is a barebones example of PHP integration with Solr.
- **solr-php-client** is a richer example of integrating Solr results into a PHP based application.
- **myfaves** is a Ruby on Rails application using `acts_as_solr` to search for music artists.
- **blacklightopac** is a demonstration of a full featured faceted search UI in Ruby on Rails.

SolrJ: Simple Java interface

SolrJ is the simple Java interface to Solr that insulates you from the dirty details of parsing and sending messages back and forth between your application and Solr. Instead, you get to work in the familiar world of objects like **SolrQuery**, **QueryResponse**, and **SolrDocument**. SolrJ is a core part of the Solr project, and therefore is updated as soon as new features are added to Solr.

One of the interesting aspects of SolrJ is that because Solr and SolrJ are both written in Java, you can directly access a Solr core without using HTTP through the `EmbeddedSolrServer` class. While this does speed up indexing by removing the cost of transporting data over the wire, it does tie you to running your client on the same local box as Solr so that it can access the Solr configuration files and Lucene indexes directly. It's also simpler to just use Solr's **remote streaming** feature (the `stream.file` parameter) for this purpose. We'll take a look at an example of using the SolrJ interface through both the `EmbeddedSolrServer` class and the more typical `CommonsHttpSolrServer` in a small application that scrapes information from `MusicBrainz.org` using a Java based web crawler called **Heritrix**.

What about Nutch?

Nutch is a Lucene sub-project, focused on building Internet scale web searches similar to Google with specifics such as a web crawler, link graphing database, and parsers for HTML and other common formats found on the Internet.

Nutch has gone through varying levels of activity and community involvement and recently reached version 1.0. While many folks look at Solr and Nutch as competitors, the projects actually target different problems. Nutch is focused on the web search space, while Solr is a generic search tool with features like spellcheck and faceting included. Nutch natively understands web search concepts such as the value of links towards calculating a page rank score, and how to factor in what an HTML `<title/>` tag is, when building the scoring model to return results. While you could build a web search tool with Solr, you would have to teach it those concepts. This isn't to say that closer integration isn't possible. In Nutch 1.x, support has been added to the crawler to index data into Solr instead of Lucene directly. However, that support is still very new.

In research done a few years ago by Oregon State University, Nutch did very well in comparison to Google (`http://nutch.sourceforge.net/twiki/Main/Evaluations/OSU_Queries.pdf`), the biggest challenges reported revolved around the topic of crawling web sites. Most of the complaints were that Nutch would sometimes end up in infinite loops or miss certain pages because of redirecting while crawling. The consensus is that Nutch is much stronger in providing good search results once the web pages have been downloaded and indexed into Nutch. To address this hole is the **NutchWAX** (**Nutch + Web Archive eXtensions**) project. NutchWAX allows you to index archived web content in the standard ARC format used by the InternetArchive. ARC is a very compact format for storing web pages in an archived format produced by Heritrix. It has been used to build full text Nutch indexes with over 500 million web pages in them by running on top of the **Hadoop** distributed computing framework. The InternetArchive has published some good practices for indexing with NutchWAX and Hadoop at `http://archive-access.sourceforge.net/projects/nutch/best-practices.pdf`. Both Nutch and Heritrix are constantly evolving projects, so keep an eye on both to see if one suits your needs better than the other.

Using Heritrix to download artist pages

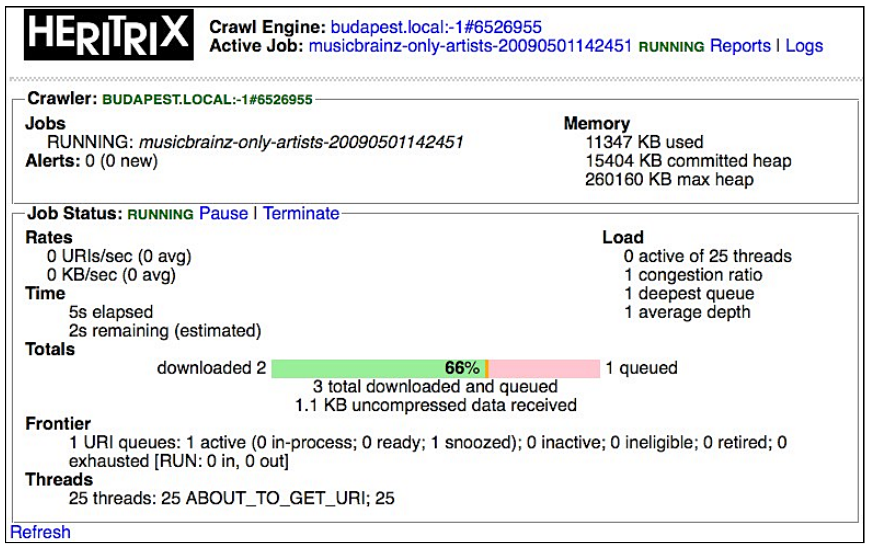
Heritrix is an extremely full featured and extensible web crawler used by the **InternetArchive** for archiving the contents of the Internet. The InternetArchive is a non-profit organization established to preserve web sites by taking regular snapshots of them. You may be more familiar with the site under the name *The Wayback Machine*. By looking back at the original indexed version of the Solr homepage taken on January 19th, 2007 at `http://web.archive.org/web/*/http://lucene.apache.org/solr`, we learn that Solr had just graduated from the Apache Incubator program!

Going into the full details of using Heritrix is outside the scope of this book. However, you can play with a version configured for crawling only artist pages on `MusicBrainz.org` in `./examples/8/crawler/heritrix-2.0.2/`. Start Heritrix by running:

```
>> ./bin/heritrix -a password
```

and then browsing to the web interface at `http://localhost:8080/` and logging in using the admin password you specified through the `-a` parameter. You will see a web console with a single **Engine** configured. Click on it to see the profiles configured for it. You should see a **musicbrainz-only-artists** profile; click on **Copy**, and choose the default option of generating a new ready-to-run job.

You will now have a new job configured with a name similar to **musicbrainz-only-artists-20090501142451**. Click on **Launch** to start the crawler covering the `MusicBrainz.org` site. You will see the console interface of the crawler and can monitor the progress of downloading content:



The crawler can take a very long time to run, as it is designed to not overload the sites being crawled and we only have 25 threads configured. `MusicBrainz.org` has roughly 119,000 distinct pages, and running Heritrix for 10 hours only downloaded 6600 pages. The pages being crawled are stored in the compact text format called an ARC file that contains multiple web resources individually compressed using `.gzip`. There are various options for checkpointing the resulting ARC files, as they are generated so that you can start using them while it continues to crawl. Learn more about checkpointing and more advanced features at Heritrix's site at `http://crawler.archive.org/`. The resulting ARC file is stored in the `./crawler/heritrix-2.0.2/jobs/[THE_NAME_OF_YOUR_JOB]/arcs/` directory. For the rest of this example, we will work with the already generated ARC files in the `./crawler/heritrix-2.0.2/jobs/completed-musicbrainz-only-artists-20090707185058/arcs` directory that contains 1532 downloaded pages.

You can view the basic meta information of the resources in an ARC file such as the timestamp, MIME type, and URL by using the arcreader command line client:

```
>> ./bin/arcreader jobs/completed-music-brainz-only-artists-
20090707185058/arcs/IAH-20090707185106--00000-budapest.local.arc.gz

20090430202747 72.29.166.157 http://musicbrainz.org/show/artist/
?artistid=217990 text/html - - 3700547 28627 IAH-20090707185106 -00000-
budapest.local

20090430202755 72.29.166.157 http://musicbrainz.org/browseartists.
html?index=P&offset=50 text/html - - 3707421 35317 IAH-20090707185106 -
00000-budapest.local
```

Indexing HTML in Solr

Solr does provide some basic support for working with HTML documents that can make indexing simpler. For example, if you look at `./examples/cores/crawler/conf/schema.xml`, you can see that the schema has been optimized for storing HTML documents. There are two new field types defined: `html-text` and `html-shingle`. Both field types leverage the `HTMLStripStandardTokenizerFactory` tokenizer to strip out the various HTML related tags and just preserve the textual content of the web page during indexing for searching against. However, `html-shingle` is designed specifically for multiword phrase searches by using a technique called **shingling** that results in faster phrase queries at the expense of more disk use and indexing time. The `html-text` field is indexed in a more straightforward manner. We delve more into shingling in general in Chapter 9.

The fields we are storing are:

```
<fields>
    <field name="url" type="string" />
    <field name="mimeType" type="string" />
    <field name="host" type="string" />
    <field name="path" type="string" />
    <field name="docText" type="html-text"/>
    <field name="docTextShingle" type="html-shingle" stored="false" />
</fields>

<copyField source="docText" dest="docTextShingle" />

<uniqueKey>url</uniqueKey>
<defaultSearchField>docText</defaultSearchField>
```

with the `url` being the unique key and the `docText` being the default field for searches, `Host` and `path` fields give us something with which to facet our results.

There is a very simple Java application in `./examples/8/crawler/SolrJforMusicBrainz` that deletes any existing records, parses the individual records from the ARC files, and inserts them into Solr if they have the MIME type of `text/html`. SolrJforMusicBrainz is built using the **Maven** project management/build tool. Maven is an Apache project at `http://maven.apache.org/` that brings a specific approach to structuring Java projects and introduced the concept of public repositories for storing JAR dependencies. Maven has seen significant adoption across the Java world.

Solr currently uses Ant for builds and publishes the resulting artifacts for Maven users. However, a recurring thread on *solr-dev* is to move to a Maven based build.

In order to compile the application yourself, assuming you have installed Maven 2, execute the following command from the `./SolrJforMusicBrainz` directory:

```
>> mvn package
```

You will download all of the JAR dependencies for both Solr and Heritrix, resulting in a roughly 12 megabyte executable JAR file at `./SolrJforMusicBrainz/target/SolrJForMusicBrainz-1.0-SNAPSHOT.jar`.

In order to index the web pages stored in the ARC format, execute the JAR file, passing in the directory in which the ARC files are located, whether you are using a remote or local Solr connection, and the specific connection information. In order to connect to your already running Solr, run:

```
>> java -jar target/SolrJForMusicBrainz-1.0-SNAPSHOT.jar ../heritrix-
2.0.2/jobs/completed-musicbrainz-only-artists-20090707185058/arcs/ REMOTE
http://localhost:8983/solr/crawler
```

You should see a long list of URLs being indexed, along with how many milliseconds it took to process all of the documents:

```
http://musicbrainz.org/show/artist/?artistid=317388
http://musicbrainz.org/show/artist/?artistid=593877
http://musicbrainz.org/show/artist/?artistid=419076
Execution time was 12454 ms for 210 documents
```

In order to index the ARC files into an embedded Solr, run:

```
>> java -jar target/SolrJForMusicBrainz-1.0-SNAPSHOT.jar ../heritrix-
2.0.2/jobs/completed-musicbrainz-only-artists-20090707185058/arcs/
EMBEDDED ../../../cores
```

You will see similar output as before, but interleaved with all of the logging generated by the Embedded Solr instance as well:

```
http://musicbrainz.org/show/artist/?artistid=334589
May 4, 2009 9:06:45 PM org.apache.solr.update.processor.
LogUpdateProcessor finish
INFO: {add=[http://musicbrainz.org/show/artist/?artistid=334589]} 0 17
May 4, 2009 9:06:45 PM org.apache.solr.core.SolrCore execute
INFO: [crawler] webapp=null path=/update params={} status=0 QTime=17
```

The interesting thing about the embedded method for connecting to Solr is that you can start it up while another Solr server instance is already running. Normally when you accidentally start a second instance of Solr running, you get an exception like:

```
java.net.BindException: Address already in use
```

However, because the Embedded Solr instance is directly starting up Solr in a separate JVM and not opening a connection to a server port, it won't have a conflict with an already running Solr instance use of a port. You can actually start up an Embedded Solr and write documents to it, while the main instance of Solr is running. This can lead to somewhat indeterministic results, as you don't have a single Solr acting as the traffic cop in coordinating reading and writing to the underlying Lucene indexes. Moreover, because of the extensive caching that is performed at the multiple layers of Solr, you may not realize that the data has changed.

Run `mvn eclipse:eclipse` to generate the `.project` and `.classpath` files required to import the SolrJForMusicBrainz project into Eclipse as an Eclipse project. You will need to configure a `M2_REPO` classpath variable in Eclipse pointing at your local repository. Learn more at `http://maven.apache.org/plugins/maven-eclipse-plugin/`.

If you don't use Eclipse, then please refer to the Maven web site's documentation for IDE integration for your preferred IDE.

SolrJ client API

SolrJ has a very straightforward object model for representing interaction with Solr. You can play with the basic methods for interacting with Solr by running the `BrainzSolrClient` class in your IDE. `BrainzSolrClient` merely provides some settings to pass into an instance of `Indexer`, the main class that parses ARC records and indexes them into Solr. Regardless of whether you choose the remote or the embedded approach for interacting with Solr, you use the same interface defined in *SolrServer*.

Starting a connection to a remote Solr is very simple, with the only parameter being the URL to the Solr instance. Note the inclusion of the *crawler* core in the URL:

```
public SolrServer startRemoteSolr() throws MalformedURLException,
        SolrServerException {
    CommonsHttpSolrServer solr = new
        CommonsHttpSolrServer("http://localhost:8983/solr/crawler");
    solr.setRequestWriter(new BinaryRequestWriter());
    return solr;
}
```

New to Solr 1.4 is the ability to specify requests and responses in a Java binary format called `javabin` that is much smaller and faster than XML, as there isn't any XML parsing, and the data being transmitted over the wire is much smaller. The amount of XML required for submitting a request can be quite large if you have numerous fields. Setting the request writer to use the BinaryRequestWriter turns this on. By default, the SolrJ client performs updates using the binary `javabin` format.

Starting up an Embedded Solr is a bit more complex, as you are starting Solr instead of running it in a separate servlet container:

```
public SolrServer startEmbeddedSolr() throws IOException,
        ParserConfigurationException, SAXException,
        SolrServerException {
```

```
        File root = new File("../../../cores");
        container = new CoreContainer();
        SolrConfig config = new SolrConfig(root + "/crawler",
                    "solrconfig.xml",null);
        CoreDescriptor descriptor = new CoreDescriptor(container,
                        "crawler",root + "/solr");
        SolrCore core = new SolrCore("crawler", root +
                 "/../cores_data/crawler", config,    null, descriptor);
        container.register(core, false);
        EmbeddedSolrServer solr = new EmbeddedSolrServer(container,
                            "crawler");
        return solr;
    }
```

You need to specify a bit more when firing up the Embedded Solr, as you are not starting up all of the cores, only a single specific one. Therefore, the `SolrConfig` and `CoreDescriptor` classes wrap the information about `solrconfig.xml` and your specific named core. Both of these are used to define the SolrCore, which is then registered in a CoreContainer. Both `EmbeddedSolrServer` and `CommonsHtttpSolrServer` implement the same *SolrServer* interface, so you can change the connectivity method you choose at runtime.

Performing a query is very straightforward:

```
SolrQuery solrQuery = new SolrQuery("Smashing Pumpkins");
QueryResponse response = solr.query(solrQuery);
```

You can customize the query, for instance, by adding faceting to find out the most popular hosts and paths indexed by the crawler:

```
SolrQuery solrQuery = new SolrQuery("*:*");
solrQuery.setRows(0);
solrQuery.setFacet(true);
solrQuery.addFacetField("host");
solrQuery.addFacetField("path");
solrQuery.setFacetLimit(10);
solrQuery.setFacetMinCount(2);
QueryResponse response = solr.query(solrQuery);
```

The query result in XML would be something similar to:

```
<result name="response" numFound="1446" start="0"/>
<lst name="facet_fields"
  <lst name="host">
    <int name="musicbrainz.org">1432</int>
    <int name="blog.musicbrainz.org">3</int>
```

```
    <int name="stats.musicbrainz.org">3</int>
    <int name="musicbrainz.uservoice.com">2</int>
    <int name="www.musicbrainz.org">2</int>
  </lst>
  <lst name="path">
    <int name="/showartist.html">473</int>
    <int name="/browseartists.html">381</int>
    <int name="/show/artist/">209</int>
    <int name="/show/user/">65</int>
    <int name="/mod/search/pre/editor-open.html">64</int>
    <int name="/browselabels.html">29</int>
  </lst>
</lst>
```

Any type of query that you would want to do with Solr is available through the SolrJ client. SolrJ also makes deleting documents simple by providing two easy methods: `deleteByQuery()` and `deleteById()`. `deleteById()` takes in the defined `uniqueKey` field (in this case, the URL). Removing the **Contact Us** page is as simple as running:

```
solr.deleteById("http://musicbrainz.org/doc/ContactUs")
```

As part of the indexing process, we want to clear out the existing index. So we use the `deleteByQuery`, and specify the entire index. Obviously this can be very dangerous, and if you have a really large Solr index it will take a while to actually commit that change to the filesystem:

```
solr.deleteByQuery( "*:*" ); // delete everything!
solr.commit();
```

Of course, none of this matters if you can't index documents. In the example below, you can see the heart of the loop for parsing through the list of ARC files and extracting and indexing the information. As previously mentioned, we only want HTML pages to be indexed, so we check for a MIME type of `text/html`. Every 100 documents that are added to Solr causes a commit to be issued to Solr. At the end of the loop, a single optimize request is issued:

```
File arcFiles[] = arcDir.listFiles(new ArcFilenameFilter());
int hits = 1;
for (File arcFile : arcFiles) {
    System.out.println("Reading " + arcFile.getName());
    ArchiveReader r = ArchiveReaderFactory.get(arcFile);
    r.setDigest(true);
    for (ArchiveRecord rec : r) {
        if (rec != null) {
            ArchiveRecordHeader meta = rec.getHeader();
            if (meta.getMimetype().trim().startsWith("text/html")) {
```

```
                ByteArrayOutputStream baos = new
                    ByteArrayOutputStream();
                rec.dump(baos)
                if (indexIntoSolr) {
                    SolrInputDocument doc = new SolrInputDocument();
                    doc.addField("url", meta.getUrl(), 1.0f);
                    doc.addField("mimeType", meta.getMimetype(),
                        1.0f);
                    doc.addField("docText", baos.toString());
 // should parse out HTML body and specify character encoding
                    URL url = new URL(meta.getUrl());
                    doc.addField("host", url.getHost());
                    doc.addField("path", url.getPath());
                    solr.add(doc);
                }
                hits++;
            }
        }
        rec.close();
    }
}
solr.commit();
solr.optimize();
```

Optimizing the performance of indexing content is often a matter of trial and error. As we are streaming the HTML for each ARC record into the `docText` field data from the filesystem, we ask ourselves: Is there a better way to convert that output stream to a string rather than using a `ByteArrayOutputStream`? We could also potentially optimize sending information to Solr by building a `Collection` of `SolrInputDocuments`, and then adding them all at once:

```
Collection<SolrInputDocument> docs = new
           ArrayList<SolrInputDocument>();
// Loop through Archive Records and add documents
// docs.add( doc );
server.add( docs );
```

Of course, this requires more memory on the indexer side and means that each add operation may take longer. Another option would be to use the `StreamingUpdateSolrServer` where there is no performance penalty to calling `add()` for each document, unlike the `CommonsHttpSolrServer` interface. You can see the performance gain by running the following command:

```
>> java -jar target/SolrJForMusicBrainz-1.0-SNAPSHOT.jar ../heritrix-2.0.2/jobs/completed-musicbrainz-only-artists-20090707185058/arcs/ STREAMING http://localhost:8983/solr/crawler
```

You should see a roughly 40 percent gain in performance by streaming the documents using three threads processing a queue of 20 documents.

Indexing POJOs

POJOs (**Plain Old Java Objects**) typically follows the JavaBean pattern of having a set of properties that have getter and setter methods for them. Moreover, in many use cases, you are looking to index information that is exposed as Java objects, such as a Product versus documents like the ARC records in the previous example. Often these objects are backed by a relational database of some type, and you manage them through object relational mapping tools such as Hibernate, JPA or JDO. Working with objects can provide much richer types of manipulations than working with documents and allows you to leverage the power of strong typing to validate your code.

Annotations were introduced in JDK 1.5 to provide a richer means of supplying extra information to tools and libraries beyond what is in the Java code itself. For example, the classic `JavaDoc` tag `@throws SolrServerException` on a method such as `startEmbeddedSolr()` can be thought of as a type of annotation that has meaning to the JavaDoc tools. However, unlike `JavaDoc` tags, annotations can be read from source files, class files, and reflectively at runtime. Solr leverages annotations to markup a POJO with information that SolrJ needs to index it.

`./SolrJForMusicBrainz/src/main/java/solrbook/RecordItem.java` is an example of a JavaBean that imports the Solr Field class and allows each property to be annotated. In the example below, `RecordItem` has the properties `id` and `html` mapped to the Solr fields `url` and `docText`:

```
import org.apache.solr.client.solrj.beans.Field;

public class RecordItem {

    @Field("url")
    String id;

    @Field
    String mimeType;

    @Field("docText")
    String html;

    @Field
    String host;

    @Field
    String path;
```

Indexing the `RecordItem` POJOs is very similar to using the `SolrDocument` directly:

```
RecordItem item = new RecordItem();

item.setId(meta.getUrl());
item.setMimeType(meta.getMimetype());
```

```
item.setHtml(baos.toString());
URL url = new URL(meta.getUrl());
item.setHost(url.getHost());
item.setPath(url.getPath());
solr.addBean(item);
```

You can also index a collection of beans through `solr.addBeans(collection)`. Performing a query that returns results as POJOs is very similar to returning normal results. You build your `SolrQuery` object the exact same way as you normally would, and perform a search returning a `QueryResponse` object. However, instead of calling `getResults()` and parsing a `SolrDocumentList` object, you would ask for the results as POJOs:

```
public List<RecordItem> performBeanSearch(String query) throws
        SolrServerException {
    SolrQuery solrQuery = new SolrQuery(query);
    QueryResponse response = solr.query(solrQuery);
    List<RecordItem> beans = response.getBeans(RecordItem.class);
    System.out.println("Search for '" + query + "': found " +
           beans.size() + " beans.");
    return beans;
}
>> Perform Search for '*:*': found 10 beans.
```

You can then go and process the search results, for example rendering them in HTML with JSP.

When should I use Embedded Solr

There has been extensive discussion on the Solr mailing lists on whether removing the HTTP layer and using a local Embedded Solr is really faster than using the `CommonsHttpSolrServer`. Originally, the conversion of Java `SolrDocument` objects into XML documents and sending them over the wire to the Solr server was considered fairly slow, and therefore Embedded Solr offered big performance advantages. However, as of Solr 1.4, a binary format is used to transfer messages, which is more compact and requires less processing than XML. In order to use the SolrJ client with pre 1.4 Solr servers, you must explicitly specify that you wish to use the XML response writer through `solr.setParser(new XMLResponseParser())`. The common thinking is that storing a document in Solr is typically a much smaller portion of the time spent on indexing compared to the actual parsing of the original source document to extract its fields. Additionally, by putting both your data importing process and your Solr process on the same computer, you are limiting yourself to only the CPUs available on that computer. If your importing process requires significant processing, then by using the HTTP interface you can have multiple processes spread out on multiple computers munging your source data.

There are a couple of use cases where using Embedded Solr is really attractive:

- Streaming locally available content directly into Solr indexes
- Rich client applications
- Upgrading from an existing Lucene search solution to a Solr based search

In-Process streaming

If you expect to stream large amounts of content from a single filesystem, which is mounted on the same server as Solr in a fairly un-manipulated manner as quickly as possible, then Embedded Solr can be very useful. This is especially if you don't want to go through the hassle of firing up a separate process or have concerns about having a servlet container, such as Jetty, running.

Consider writing a custom DIH DataSource instead.

Instead of using SolrJ for fast importing, consider using Solr's **DataImportHandler (DIH)** framework. Like Embedded Solr, it will result in an in-process import. Look at the `org.apache.solr.handler.dataimport.DataSource` interface and existing implementations like `JdbcDataSource`. Using DIH gives you supporting infrastructure like starting and stopping imports, a debugging interface, chained transformations, and the ability to integrate with data available from other DIH data-sources (such as inlining reference data from an XML file).

A good example of an open source project that took the approach of using Embedded Solr is Solrmarc. **Solrmarc** (hosted at `http://code.google.com/p/solrmarc/`) is a project to parse MARC records, a standardized machine format for storing bibliographic information.

What is interesting about Solrmarc is that it heavily uses meta programming methods to avoid binding to a specific version of the Solr libraries, allowing it to work with multiple versions of Solr. So, for example, creating a Commit command looks like:

```
Class<?> commitUpdateCommandClass =
  Class.forName("org.apache.solr.update.CommitUpdateCommand");
commitUpdateCommand = commitUpdateCommandClass
  .getConstructor(boolean.class).newInstance(false);
```

instead of

```
CommitUpdateCommand  commitUpdateCommand = new
  CommitUpdateCommand();
```

Solrmarc uses the Embedded Solr approach to locally index content. After it is optimized, the index is moved to a Solr server that is dedicated to serving search queries.

Rich clients

In my mind, the most compelling reason for using the Embedded Solr approach is when you have a rich client application developed using technologies such as **Swing** or **JavaFX** and are running in a much more constrained client environment. Adding search functionality using the Lucene libraries directly is a more complicated lower-level API and it doesn't have any of the value-add that Solr offers (for example, faceting). By using Embedded Solr you can leverage the much higher-level API of Solr, and you don't need to worry about the environment your client application exists in blocking access to ports or exposing the contents of a search index through HTTP. It also means that you don't need to manage spawning another Java process to run a Servlet container, leading to fewer dependencies. Additionally, you still get to leverage skills in working with the typically server based Solr on a client application. A win-win situation for most Java developers!

Upgrading from legacy Lucene

Probably a more common use case is when you have an existing Java-based web application that was architected prior to Solr becoming the well known and stable product that it is today. Many web applications leverage Lucene as the search engine with a custom layer to make it work with a specific Java web framework such as Struts. As these applications become older, and Solr has progressed, revamping them to keep up with the features that Solr offers has become more difficult. However, these applications have many ties into their homemade Lucene based search engines. Performing the incremental step of migrating from directly interfacing with Lucene to directly interfacing with Solr through Embedded Solr can reduce risk. Risk is minimized by limiting the impact of the change to the rest of the web application by isolating change to the specific set of Java classes that previously interfaced directly with Lucene. Moreover, this does not require a separate Solr server process to be deployed. A future incremental step would be to leverage the scalability aspects of Solr by moving away from the Embedded Solr to interfacing with a separate Solr server.

Using JavaScript to integrate Solr

During the Web 1.0 epoch, JavaScript was primarily used to provide basic client-side interactivity such as a roll-over effect for buttons in the browser on what were essentially static pages generated wholly by the server. However, in today's Web 2.0 environment, the rise of AJAX usage has led to JavaScript being used to build much richer web applications that blur the line between client-side and server-side functionality. Solr's support for the **JavaScript Object Notation** format (**JSON**) for transferring search results between the server and the web browser client makes it simple to consume Solr information by modern Web 2.0 applications. JSON is a human-readable format for representing JavaScript objects, which is rapidly becoming a defacto standard for transmitting language independent data with parsers available to many languages, including Java, C#, Ruby, and Python, as well as being syntactically valid JavaScript code! The `eval()` function will return a valid JavaScript object that you can then manipulate:

```
var json_text = ["Smashing Pumpkins","Dave Matthews Band","The
    Cure"];
var bands = eval('(' + json_text + ')');
alert("Band Count: " + bands.length()); // alert "Band Count: 3"
```

While JSON is very simple to use in concept, it does come with its own set of complexities related to security and browser compatibility. To learn more about the JSON format, the various client libraries that are available, and how it is and is not like XML, visit the homepage at `http://www.json.org`.

As you may recall from Chapter 3, you change the format of the response from Solr from the default XML to JSON by specifying the JSON writer type as a parameter in the URL: `wt=json`. The results are returned in a fairly compact, single long string of JSON text:


```
{"responseHeader":{"status":0,"QTime":0,"params":{"q":"hills ro
lling","wt":"json"}},"response":{"numFound":44,"start":0,"docs
":[{"a_name":"Hills Rolling","a_release_date_latest":"2006-11-
30T05:00:00Z","a_type":"2","id":"Artist:510031","type":"Artist"}]}}
```

If you add the indent=on parameter to the URL, then you will get some pretty printed output that is more legible:

```
{
  "responseHeader":{
  "status":0,
  "QTime":1,
  "params":{
    "q":"hills rolling",
    "wt":"json",
    "indent":"on"}},
  "response":{"numFound":44,"start":0,"docs":[
    {
    "a_name":"Hills Rolling",
    "a_release_date_latest":"2006-11-30T05:00:00Z",
    "a_type":"2",
    "id":"Artist:510031",
    "type":"Artist"}
    ]
}}
```

You may find that you run into difficulties while parsing JSON in various client libraries, as some are more strict in the format than others. Solr does output very clean JSON, such as quoting all keys and using double quotes and offers some formatting options for customizing handling of lists of data. If you run into difficulties, a very useful web site for validating your JSON formatting is http://www.jsonlint.com/. Paste in a long string of JSON and the site will validate the code and highlight any issues in the formatting. This can be invaluable for finding a trailing comma, for example.

Wait, what about security?

You may recall from Chapter 7 that one of the best ways to secure Solr is to limit what IP addresses can access your Solr install through firewall rules. Obviously, if users on the Internet are accessing Solr through JavaScript, then you can't do this. However, if you look back at Chapter 7, there is information on how to expose a read-only request handler that can be safely exposed to the Internet without exposing the complete admin interface.

Building a Solr powered artists autocomplete widget with jQuery and JSONP

Recently it has become de rigueur for any self-respecting Web 2.0 site to provide suggestions when users type information into a search box. Even Google has joined this trend:

Building a Web 2.0 style autocomplete text box that returns results from Solr is very simple by leveraging the JSON output format and the very popular jQuery JavaScript library's **Autocomplete** widget.

jQuery is a fast and concise JavaScript library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. It has gone through explosive usage growth in 2008 and is one of the most popular Ajax frameworks. jQuery provides low level utility functions but also completes JavaScript UI widgets such as the Autocomplete widget. The community is rapidly evolving, so stay tuned to the `jQuery.com` blog at `http://blog.jquery.com/`. You can learn more about jQuery at `http://www.jquery.com/`.

The jQuery Autocomplete widget can use both local and remote datasets. Therefore, it can be set up to display suggestions to the user based on results from Solr. A working example is available in the `/examples/8/jquery_autocomplete/index.html` file that demonstrates suggesting an artist as you type in his or her name. You can see a live demo of Autocomplete online at `http://view.jquery.com/trunk/plugins/autocomplete/demo/` and read the documentation at `http://docs.jquery.com/Plugins/Autocomplete`.

There are three major sections to the page:

- the JavaScript script import statements at the top
- jQuery JavaScript that actually handles the events around the text being input
- a very basic HTML for the form at the bottom

We start with a very simple HTML form that has a single text input box with the `id="artist"`:

```
<div id="content">

    <form autocomplete="off">
        <p>
            <label>Artist Name:</label>
            <input type="text" id="artist" size="30"/>
            Press "F2" key to see logging of events.
        </p>

        <input type="submit" value="Submit" />
    </form>

</div>
```

We then add a function that runs, after the page has loaded, to turn our basic text field into a text field with suggestions:

```
$(function() {

    function formatForDisplay(doc) {
        return doc.a_name;
    }

    $("#artist").autocomplete(
   'http://localhost:8983/solr/mbartists/select/?wt=json&json.wrf=?', {
        dataType: "jsonp",
        width: 300,
        extraParams: {rows: 10, fq: "type:Artist", qt:
              "artistAutoComplete"},
        minChars: 3,
```

```
            parse: function(data) {
                log.debug("resulting documents count:" +
                    data.response.docs.size);
                return $.map(data.response.docs, function(document) {
                log.debug("doc:" + doc.id);
                    return {
                        data: doc,
                        value: doc.id.toString(),
                        result: doc.a_name
                    }
                });
            },
            formatItem: function(doc) {
                return formatForDisplay(doc);
            }
        }).result(function(e, doc) {
            $("#content").append("<p>selected " + formatForDisplay(doc)
                          + "(" + doc.id + ")" + "</p>");
            log.debug("Selected Artist ID:" + doc.id);
        });
    });
```

The `$("#artist").autocomplete()` function takes in the URL of our data source, in our case Solr, and an array of options and custom functions and ties it to the text field. The `dataType: "jsonp"` option that we supply informs Autocomplete that we want to retrieve our data using **JSONP**. JSONP stands for **JSON with Padding**, which is not a very obvious name. It means that when you call the server for JSON data, you are specifying a JavaScript callback function that gets evaluated by the browser to actually do something with your JSON objects. This allows you to work around the web browser cross-domain scripting issues of running Solr on a different URL and/or port from the originating web page. jQuery takes care of all of the low level plumbing to create the callback function, which is supplied to Solr through the `json.wrf=?` URL parameter.

Notice the `extraParams` data structure:

```
width: 400,
extraParams: {rows: 10, fq: "type:Artist"},
minChars: 3,
```

These items are tacked onto the URL, which is passed to Solr. Unfortunately, Autocomplete uses the URL parameter `limit` with the value specified for the `max` option to control the number of results to be returned, which doesn't work for Solr. We work around this by specifying the `rows` parameter as an `extraParams` entry.

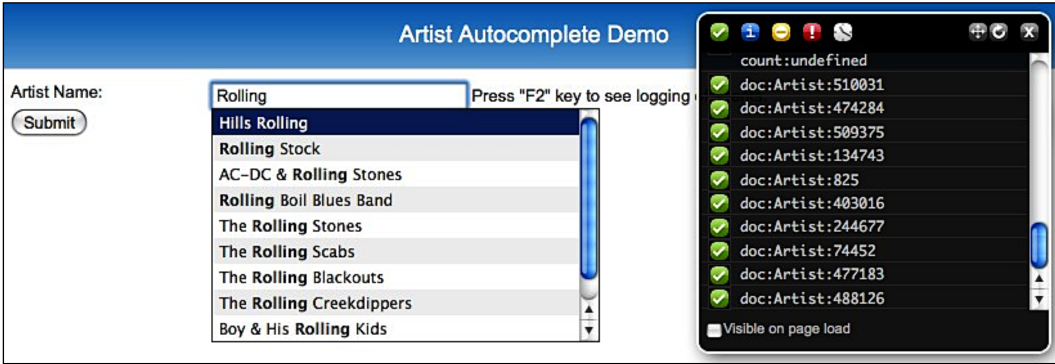
Following the best practices, we have created a specific request handler called `artistAutoComplete`, which is a dismax handler to search over all of the fields in which an artists name might show up: `a_name`, `a_alias`, `and a_member_name`. The handler is specified by appending `qt=artistAutoComplete` to the URL through `extraParams` as well.

The `parse:` parameter defines a function that is called to handle the JSON result data from Solr. It consists of a `map()` function that takes the response and calls another anonymous function. This function deals with each document and builds the internal data structure that Autocomplete needs to handle the searching and filtering in order to match what the user has typed.

Once the user has selected a suggestion, the `result()` function is called, and the selected JSON document is available to be used to show the appropriate user feedback on the suggestion being selected. In our case, it is a message appended to the `<div id="content">` div.

By default, Autocomplete uses the parameter `q` to send what the user has entered into the text field to the server, which matches up perfectly with what Solr expects. Therefore, we don't see it but call it out as an explicit parameter.

You may have noticed the logging statements in the JavaScript. The example leverages the very nice **Blackbird** JavaScript logging utility. Blackbird is an open source JavaScript library that bills itself as saying *goodbye to alert()* dialogs and is available from `http://www.gscottolson.com/blackbirdjs/`. By pressing *F2*, you will see a console that displays some information about the processing being done by the Autocomplete widget. You should now have a nice Solr powered text autocomplete field so that when you enter `Rolling`, you get a list of all of the artists including the Stones.



One thing that we haven't covered is the pretty common use case for an Autocomplete widget that populates a text field with data that links back to a specific row in a table in a database. For example, in order to store a list of **My Favorite Artists**, I would want the Autocomplete widget to simplify the process of looking up the artists but would need to store the list of favorite artists in a relational database. You can still leverage Solr's superior search ability, but tie the resulting list of artists to the original database record through a primary key ID, which is indexed as part of the Solr document. If you try to lookup the primary key of an artist through the artist's name, then you may run into problems, such as having multiple artists with the same name or unusual characters that don't translate cleanly from Solr to the web interface to your database record. Typically in this use case, you would add the `mustMatch: true` option to the `autocomplete()` function to ensure that freeform text that doesn't result in a match is ignored. You can add a hidden field to store the primary key of the artist and use that in your server-side processing versus the name in text box. Add an `onChange` event handler to blank out the `artist_id` hidden field if any changes occur so that the `artist` and `artist_id` always matchup:

```
<input type="hidden" id="artist_id"/>
<input type="text" id="artist" size="30"/>
```

The `parse()` function is modified to clear out the `artist_id` field whenever new text is entered into the `autocomplete` field. This ensures that the `artist_id` and `artist` fields do not become out of sync:

```
parse: function(data) {
log.debug("resulting documents count:" + data.response.docs.size);
$("#artist_id").get(0).value = ""; // clear out hidden field
    return $.map(data.response.docs, function(doc) {
```

The `result()` function call is updated to populate the hidden `artist_id` field when an artist is picked:

```
result(function(e, doc) {
    $("#content").append("<p>selected " + formatForDisplay(doc) +
                "(" + doc.id + ")" + "</p>");
    $("#artist_id").get(0).value = doc.id;
    log.debug("Selected Artist ID:" + doc.id);
});
```

Look at /examples/8/jquery_autocomplete/index_with_id.html for a complete example. Change the field artist_id from input type="hidden" to type="text" so that you can see the ID changing more easily as you select different artists.

> Keen readers may have noticed that, albeit similar, the example in this section and what Google is doing are fundamentally different. Google is doing a term suggest type of autocomplete, where as we are doing a search result autocomplete. The difference is that Google (and Solr can do this with a creative use of faceting, see Chapter 5) returns individual search words for the response, whereas search result autocomplete returns particular documents. Both are useful, and it depends on what you want to do. For the MusicBrainz data, the search result autocomplete makes the most sense. In order to do what Google does, you could do autocompletion based on matching existing facets groupings. You can expect Solr to become smarter about the terms indexed, which would support term suggest autocompletion better.

SolrJS: JavaScript interface to Solr

As previously mentioned in Chapter 7, SolrJS is also built on the jQuery library and provides a full featured Solr search interface with the usual goodies such as supporting facets and providing autocompletion of suggestions for queries. SolrJS adds some interesting visualizations of result data, including widgets for displaying tag clouds of facets, plotting country code-based data on a map of the world, or filtering results by date fields. When it comes to integrating Solr into your web application, if you are comfortable with the jQuery library and JavaScript, then this can be a very effective way to add a really nice Ajax view of your search results without changing the underlying web application. If you're working with an older web framework that is brittle and hard to change, such as IBM's Lotus Notes and Domino framework, then this keeps the integration from touching the actual business objects, and keeps the modifications in the HTML and JavaScript layer.

The SolrJS project homepage is at http://solrjs.solrstuff.org/ and has a great demo of displaying Reuters business news wire results from 1987. SolrJS is currently migrating to the main Apache Solr project, so check the Wiki page at http://wiki.apache.org/solr/SolrJS for updates.

A slightly tweaked copy of the homepage is stored in `/examples/8/solrjs/reuters.html`. So let's go ahead and look at the relevant portions of the HTML that drive SolrJS. You may see some patterns that look familiar to the previous Autocomplete example, because SolrJS uses a slightly older version of jQuery and integrates with Solr the same way using JSON.

SolrJS has a concept of widgets that provides rich UI functionality. It comes with widgets that do autocomplete, tag cloud, facet view, country code, and calendar based date ranges, as well as a results widget. They all inherit from an `AbstractClientSideWidget` and follow pretty much the same pattern. You configure them by passing in a set of options, such as what fields to read data in for autocompletion, or what fields to display results in.

```
new $sj.solrjs.AutocompleteWidget({id:"search", target:"#search",
  fulltextFieldName:"allText", fieldNames:["topics", "organisations",
  "exchanges"]});
new $sj.solrjs.TagcloudWidget({id:"topics", target:"#topics",
  fieldName:"topics", size:50});
```

A central **SolrJS Manager** object coordinates all of the event handling between the various widgets, allowing them to update their display appropriately as selections are made. Widgets are added to the `solrjsManager` object through `addWidget()` method:

```
solrjsManager.addWidget(resultWidget);
```

A custom UI is quickly built by creating your own result widget based on the `ExtensibleResultWidget` and customizing the `renderResult()` method.

Working with SolrJS and creating new widgets for your specific display purposes comes easily to anyone who comes from an object-oriented background. The various widgets that come with SolrJS serve more as a foundation and source of ideas rather than as a finished set of widgets. You'll find yourself customizing them extensively to meet your specific display needs.

Accessing Solr from PHP applications

There are a number of ways to access Solr from PHP based applications, and none of them seem to have taken hold of the market as the best approach. So keep an eye on the Wiki page at `http://wiki.apache.org/solr/SolPHP` for new developments. While you can tie into Solr using the standard XML interface for handling results (and that is what the listed standalone `SolrUpdate.php` and `SolrQuery.php` classes do), you can also directly consume results by using one of the two PHP writer types: `php` and `phps`. In order to access either of the writer types, you need to uncomment them in `solrconfig.xml`:

```
<queryResponseWriter name="php"
  class="org.apache.solr.request.PHPResponseWriter"/>
<queryResponseWriter name="phps"
  class="org.apache.solr.request.PHPSerializedResponseWriter"/>
```

Adding the URL parameter `wt=php` produces simple PHP output in a typical array data structure:

```
array(
  'responseHeader'=>array(
  'status'=>0,
  'QTime'=>0,
  'params'=>array(
    'wt'=>'php',
    'indent'=>'on',
    'rows'=>'1',
    'start'=>'0',
    'q'=>'Pete Moutso')),
```

```
'response'=>array('numFound'=>523,'start'=>0,'docs'=>array(
  array(
  'a_name'=>'Pete Moutso',
  'a_type'=>'1',
  'id'=>'Artist:371203',
  'type'=>'Artist'))
))
```

The same response using the Serialized PHP output specified by `wt=phps` URL parameter is a much less human-readable format but much more compact to transfer over the wire:


```
a:2:{s:14:"responseHeader";a:3:{s:6:"status";i:0;s:5:"QTime";i:1;s:6:"
params";a:5:{s:2:"wt";s:4:"phps";s:6:"indent";s:2:"on";s:4:"rows";s:1:
"1";s:5:"start";s:1:"0";s:1:"q";s:11:"Pete Moutso";}}s:8:"response";a:
3:{s:8:"numFound";i:523;s:5:"start";i:0;s:4:"docs";a:1:{i:0;a:4:{s:6:"
a_name";s:11:"Pete Moutso";s:6:"a_type";s:1:"1";s:2:"id";s:13:"Artist:
371203";s:4:"type";s:6:"Artist";}}}}
```


solr-php-client

Showing a lot of progress towards becoming the dominant solution for PHP integration is the `solr-php-client`, a project on Google Code: `http://code.google.com/p/solr-php-client/`. Interestingly enough, this project leverages the JSON writer type to communicate with Solr instead of the PHP writer type, showing the prevalence of JSON for facilitating inter-application communication in a language agnostic manner. The developers chose JSON over XML because they found that JSON parsed much quicker than XML in most PHP environments. Moreover, using the native PHP format requires using the `eval()` function, which has a performance penalty and opens the door for code injection attacks.

`solr-php-client` can both create documents in Solr as well as perform queries for data. In `/examples/8/solr-php-client/demo.php`, there is a demo of creating a new artist document in Solr for the singer Susan Boyle, and then performing some queries. Susan Boyle was a contestant on the TV show *Britain's Got Talent* and may be a major artist in the future. You can learn more about her from her Wikipedia entry at `http://en.wikipedia.org/wiki/Susan_Boyle`.

Installing the demo in your specific local environment is left as an exercise for the reader. On a Macintosh, you would place the `solr-php-client` directory in `/Library/WebServer/Documents/`.

An array data structure of key value pairs that match your schema can be easily created and then used to create an array of `Apache_Solr_Document` objects to be sent to Solr. Notice that we are using the artist ID value `-1`. Solr doesn't care what the ID field contains, just that it is present. Using `-1` ensures that we can find Susan Boyle by ID later!

```
$artists = array(
 'suan_boyle' => array(
  'id' => 'Artist:-1',
  'type' => 'Artist',
  'a_name' => 'Susan Boyle',
  'a_type' => 'person',
  'a_member_name' => array('Susan Boyle')
 )
);
```

The value for `a_member_name` is an array, because `a_member_name` is a multi-valued property.

Sending the documents to Solr and triggering the commit and optimize operations is as simple as:

```
$solr->addDocuments( $documents );
$solr->commit();
$solr->optimize();
```

If you are not running Solr on the default port, then you will need to tweak the `Apache_Solr_Service` configuration:

```
$solr = new Apache_Solr_Service( 'localhost', '8983',
  '/solr/mbartists' );
```

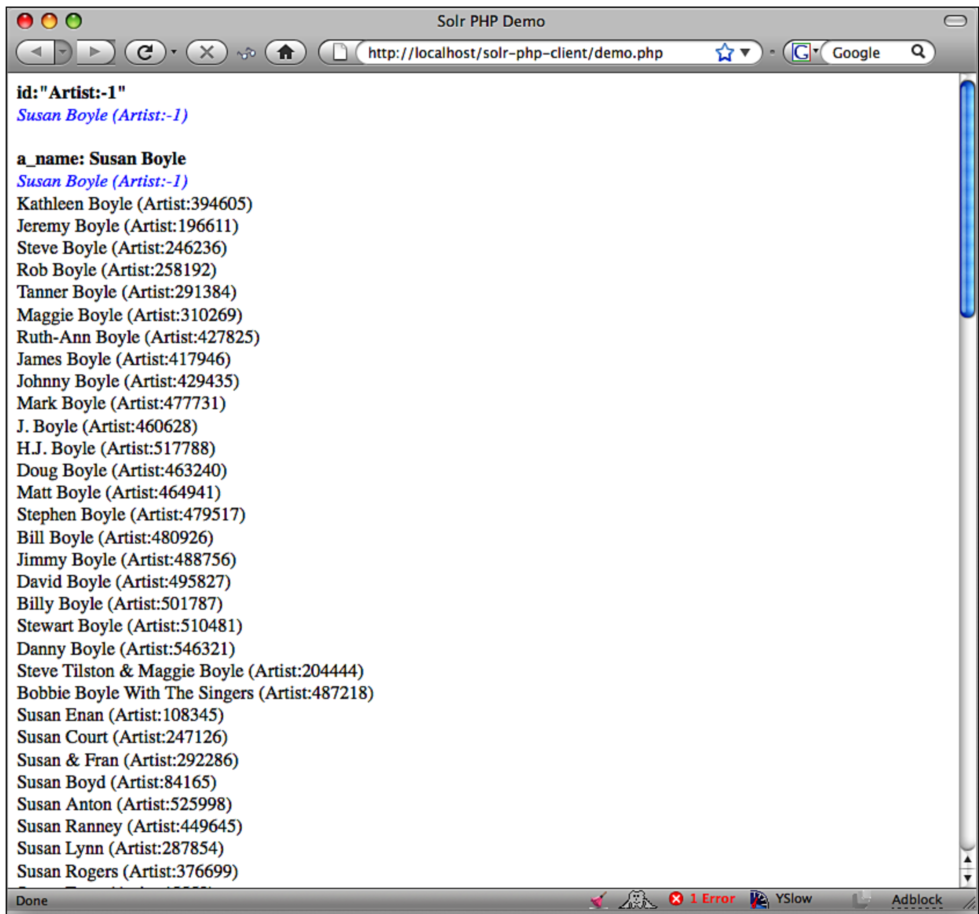
Queries can be issued using one line of code. The variables `$query`, `$offset`, and `$limit` contain what you would expect them to.

```
$response = $solr->search( $query, $offset, $limit );
```

Displaying the results is very straightforward as well. Here we are looking for the artist `Susan Boyle` based on her ID of `-1` to highlight the result using a `blue` font:

```
foreach ( $response->response->docs as $doc ) {
  $output = "$doc->a_name ($doc->id) <br />";
  // highlight Susan Boyle if we find her.
  if ($doc->id == 'Artist:-1') {
  $output = "<em><font color=blue>" . $output . "</font></em>";
 }
  echo $output;
}
```

Successfully running the demo creates Susan Boyle and issues a number of queries, producing a page similar to the one below. Notice that if you know the ID of the artist, it's almost like using Solr as a relational database to select a single specific row of data. Instead of `select * from artist where id=-1` we did `q=id:"Artist:-1"`, but the result is the same!



Drupal options

Drupal is a very successful open source **Content Management System** (**CMS**) that has been used for building everything from the `Recovery.gov` site to political campaigns to university web sites. Drupal, written in PHP, is notable for its rich wealth of modules that provide integration with many different systems, and now Solr! Drupal's built-in search has always been considered adequate, but not great. So Solr, now being an option for Drupal developers, is going to be very popular.

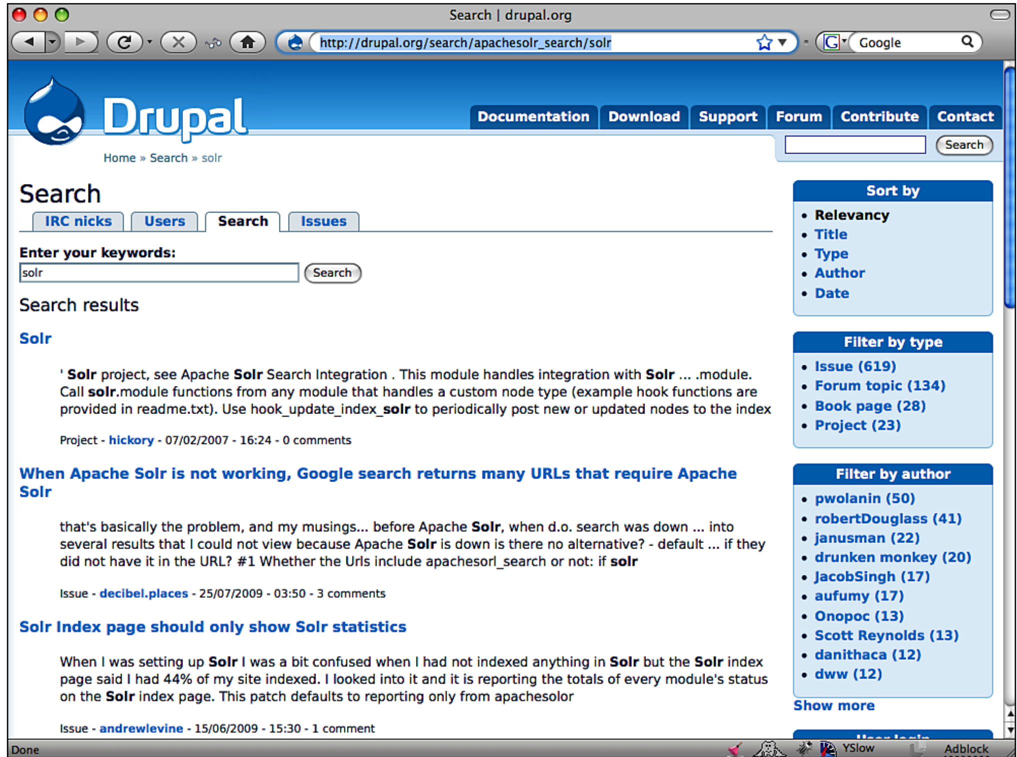
Apache Solr Search integration module

The Apache Solr Search integration module, hosted at `http://drupal.org/project/apachesolr`, builds on top of the core search services provided by Drupal, but provides extra features such as faceted search and better performance by offloading servicing search requests to another server. The module seems to have had significant adoption and is the basis for some other Drupal modules.

Incidentally, it uses the source code of the `solr-php-client` internally with one of the installation steps for checking out revision 6 of the `solr-php-client`. The Drupal project is scrupulous about maintaining only GPL licensed code in their source control repository. Therefore, you need to manually install the BSD licensed `solr-php-client`:

```
>>svn checkout -r6 http://solr-php-client.googlecode.com/svn/trunk/
  SolrPhpClient
```

In order to see the Apache Solr module in action, just visit the `Drupal.org` and perform a search to see the faceted results. In the screenshot below, you can see that they have facets by **Author** and **Type**, as well as sorting by **Relevancy**, **Title**, **Type**, **Author**, and **Date**.



Hosted Solr by Acquia

Acquia is a company providing commercially supported Drupal distributions that contain some proprietary modules to make managing Drupal easier. As of early 2009, they have a hosted search system in beta, which is based on Lucene and Solr for Drupal sites. Acquia's adoption of Solr as a better solution for Drupal then Drupal's own search shows the rapid maturing of the Solr community and platform.

Acquia maintains "in the cloud" (Amazon EC2), a large infrastructure of Solr servers saving individual Drupal administrators from the overhead of maintaining their own Solr server. A module provided by Acquia is installed into your Drupal and monitors for content changes. Every five or 10 minutes, the module sends content that either hasn't been indexed, or needs to be re-indexed, up to the indexing servers in the Acquia network. When a user performs a search on the site, the query is sent up to the Acquia network, where the search is performed, and then Drupal is just responsible for displaying the results. Acquia's hosted search option supports all of the usual Solr goodies including faceting. Drupal has always been very database intensive, with only moderately complex pages performing 300 individual SQL queries to render. Moving the load of performing searches off one's Drupal server into the cloud drastically reduces the load of indexing and performing searches on Drupal.

Acquia has developed some slick integration beyond the standard Solr features based on their tight integration into the Drupal framework, which include:

- The **Content Construction Kit** (**CCK**) allows you to define custom fields for your nodes through a web browser. For example, you can add a select field onto a blog node such as oranges/apples/peaches. Solr understands those CCK data model mappings and actually provides a facet of oranges/apples/peaches for it.
- Turn on a single module and instantly receive content recommendations giving you **more like this** functionality based on results provided by Solr. Any Drupal content can have recommendations links displayed with it.
- Multi-site search: A strength of Drupal is the support of running multiple sites on a single codebase, such as `drupal.org`, `groups.drupal.org`, and `api.drupal.org`. Currently, part of the Apache Solr module is the ability to track where a document came from when indexed, and as a result, add the various sites as new filters in the search interface.

I think that Acquia's hosted search product is a very promising idea, and I can see hosted Solr search becoming a very common integration approach for many sites that don't wish to manage their own Java infrastructure or need to customize the behavior of Solr drastically. Acquia is currently evaluating many other enhancements to their service that take advantage of the strengths of the Drupal platform and the tight level of integration they are able to perform. So expect to see more announcements. You can learn more about what is happening here at `http://acquia.com/products-services/acquia-search`.

Ruby on Rails integrations

There has been a lot of churn in the Ruby on Rails world for adding Solr support, with a number of competing libraries and approaches attempting to add Solr support in the most Rails-native way. Rails brought to the forefront the idea of **Convention over Configuration**. In most traditional web development software, from ColdFusion, to Java EE, to .NET, the framework developers went with the approach that their framework should solve any type of problem and work with any kind of data model. This led to these frameworks requiring massive amounts of configuration, typically by hand. It wasn't unusual to see that adding a column to a user record would require modifying the database, a data access object, a business object, and the web tier. Four changes in four different files to add a new field! While there were many attempts to streamline this, from using annotations to tooling like IDE's and Xdoclet, all of them were band-aids over the fundamental problem of too much configurability. The Rails sweet spot for development is exposing an SQL database to the web. Add a column to the database and it is now part of your object relational model with no additional coding. The various libraries for integrating Solr in Ruby on Rails applications attempt to follow this idea of Convention over Configuration in how they interact with Solr. However, often there are a lot of mysterious rules (conventions!) to learn, such as prefixing String schema fields with `_s` when developing the Solr schema.

The classic plugin for Rails is `acts_as_solr` that allows Rails `ActiveRecord` objects to be transparently stored in a Solr index. Other popular options include **Solr Flare** and **rsolr**. An interesting project is **Blacklight**, a tool oriented towards libraries putting their catalogs online. While it attempts to meet the needs of a specific market, it also contains many examples of great Ruby techniques to leverage in your own projects.

Similar to the PHP integrations discussed previously, you will need to turn on the Ruby writer type in `solrconfig.xml`:

```
<queryResponseWriter name="ruby"
  class="org.apache.solr.request.RubyResponseWriter"/>
```

The Ruby hash structure looks very similar to the JSON data structure with some tweaks to fit Ruby, such as translating nulls to nils, using single quotes for escaping content, and the Ruby `=>` operator to separate key-value pairs in maps. Adding a `wt=ruby` parameter to a standard search request returns results in a Ruby hash structure like this:

```
{
  'responseHeader'=>{
  'status'=>0,
  'QTime'=>1,
  'params'=>{
    'wt'=>'ruby',
    'indent'=>'on',
    'rows'=>'1',
    'start'=>'0',
    'q'=>'Pete Moutso'}},
  'response'=>{'numFound'=>523,'start'=>0,'docs'=>[
    {
    'a_name'=>'Pete Moutso',
    'a_type'=>'1',
    'id'=>'Artist:371203',
    'type'=>'Artist'}]
}}
```

acts_as_solr

A very common naming pattern for plugins in Rails that manipulate the database backed object model is to name them `acts_as_X`. For example, the very popular `acts_as_list` plugin for Rails allows you to add list semantics, like first, last, move_next to an unordered collection of items. In the same manner, `acts_as_solr` takes `ActiveRecord` model objects and transparently indexes them in Solr. This allows you to do fuzzy queries that are backed by Solr searches, but still work with your normal `ActiveRecord` objects. Let's go ahead and build a small Rails application that we'll call **MyFaves** that both allows you to store your favorite MusicBrainz artists in a relational model and allows you to search for them using Solr.

acts_as_solr comes bundled with a full copy of Solr 1.3 as part of the plugin, which you can easily start by running rake solr:start. Typically, you are starting with a relational database already stuffed with content that you want to make searchable. However, in our case we already have a fully populated index available in /examples, and we are actually going to take the basic artist information out of the mbartists index of Solr and populate our local myfaves database with it. We'll then fire up the version of Solr shipped with acts_as_solr, and see how acts_as_solr manages the lifecycle of ActiveRecord objects to keep Solr's indexed content in sync with the content stored in the relational database. Don't worry, we'll take it step by step! The completed application is in /examples/8/myfaves for you to refer to.

Setting up MyFaves project

We'll start with the standard plumbing to get a Rails application set up with our basic data model:

```
>>rails myfaves
>>cd myfaves
>>./script/generate scaffold artist name:string group_type:string
  release_date:datetime image_url:string
>>rake db:migrate
```

This generates a basic application backed by an **SQLite** database. Now we need to install the acts_as_solr plugin.

acts_as_solr has gone through a number of revisions, from the original code base done by Erik Hatcher and posted to the *solr-user* mailing list in August of 2006, which was then extended by Thiago Jackiw and hosted on Rubyforge. Today the best version of acts_as_solr is hosted on GitHub by Mathias Meyer at http://github.com/mattmatt/acts_as_solr/tree/master. The constant migration from one site to another leading to multiple possible 'best' versions of a plugin is unfortunately a very common problem with Rails plugins and projects, though most are settling on either RubyForge.org or GitHub.com.

In order to install the plugin, run:

```
>>script/plugin install git://github.com/mattmatt/acts_as_solr.git
```

We'll also be working with roughly 399,000 artists, so obviously we'll need some page pagination to manage that list, otherwise pulling up the artists /index listing page will timeout:

```
>>script/plugin install git://github.com/mislav/will_paginate.git
```

Edit the `./app/controllers/artists_controller.rb` file, and replace in the `index` method the call to `@artists = Artist.find(:all)` with:

```
@artists = Artist.paginate :page => params[:page], :order =>
  'created_at DESC'
```

Also add to `./app/views/artists/index.html.erb` a call to the view helper to generate the page links:

```
<%= will_paginate @artists %>
```

Start the application using `./script/server`, and visit the page `http://localhost:3000/artists/`. You should see an empty listing page for all of the artists. Now that we know the basics are working, let's go ahead and actually leverage Solr.

Populating MyFaves relational database from Solr

Step one will be to import data into our relational database from the `mbartists` Solr index. Add the following code to `./app/models/artist.rb`:

```
class Artist < ActiveRecord::Base
 acts_as_solr :fields => [:name, :group_type, :release_date]
end
```

The `:fields` array of hashes maps the attributes of the Artist `ActiveRecord` object to the artist fields in Solr's `schema.xml`. Because acts_as_solr is designed to store data in Solr that is mastered in your data model, it needs a way of distinguishing among various types of data model objects. For example, if we wanted to store information about our User model object in Solr in addition to the Artist object then we need to provide a type_field to separate the Solr documents for the artist with the primary key of 5 from the user with the primary key of 5. Fortunately the mbartists schema has a field named type that stores the value Artist, which maps directly to our ActiveRecord class name of Artist and we are able to use that instead of the default acts_as_solr type field in Solr named type_s.

There is a simple script called populate.rb at the root of /examples/8/myfaves that you can run that will copy the artist data from the existing Solr mbartists index into the MyFaves database:

```
>>ruby populate.rb
```

populate.rb is a great example of the types of scripts you may need to develop to transfer data into and out of Solr. Most scripts typically work with some sort of batch size of records that are pulled from one system and then inserted into Solr. The larger the batch size, the more efficient the pulling and processing of data typically is at the cost of more memory being consumed, and the slower the commit and optimize operations are. When you run the populate.rb script, play with the batch size parameter to get a sense of resource consumption in your environment. Try a batch size of 10 versus 10000 to see the changes. The parameters for populate.rb are available at the top of the script:

```
MBARTISTS_SOLR_URL = 'http://localhost:8983/solr/mbartists'
BATCH_SIZE = 1500
MAX_RECORDS = 100000  # the maximum number of records to load,
    or nil for all
```

There are roughly 399,000 artists in the mbartists index, so if you are impatient, then you can set MAX_RECORDS to a more reasonable number.

The process for connecting to Solr is very simple with a hash of parameters that are passed as part of the GET request. We use the magic query value of *:* to find all of the artists in the index and then iterate through the results using the start parameter:

```
connection = Solr::Connection.new(MBARTISTS_SOLR_URL)
 solr_data = connection.send(Solr::Request::Standard.new({
  :query => '*:*',
  :rows=> BATCH_SIZE,
  :start => offset,
  :field_list =>['*','score']
 }))
```

In order to create our new Artist model objects, we just iterate through the results of solr_data. If solr_data is nil, then we exit out of the script knowing that we've run out of results. However, we do have to do some parsing translation in order to preserve our unique identifiers between Solr and the database. In our MusicBrainz Solr schema, the ID field functions as the primary key and looks like Artist:11650 for The Smashing Pumpkins. In the database, in order to sync the two, we need to insert the Artist with the ID of 11650. We wrap the insert statement a.save! in a begin/rescue/end structure so that if we've already inserted an artist with a primary key, then the script continues. This just allows us to run the populate script multiple times:

```
solr_data.hits.each do |doc|
  id = doc["id"]
  id = id[7..(id.length)]
  a = Artist.new(:name => doc["a_name"], :group_type => a["a_type"],
      :release_date => doc["a_release_date_latest"])
```

```
    a.id = id
    begin
      a.save!
      rescue ActiveRecord::StatementInvalid => ar_si
      raise ar_si unless ar_si.to_s.include?("PRIMARY KEY must be
            unique") #sink duplicates
    end
  end
```

Now that we've transferred the data out of our `mbartists` index and used `acts_as_solr` according to the various conventions that it expects, we'll change from using the mbartists Solr instance to the version of Solr shipped with `acts_as_solr`.

Solr related configuration information is available in `./myfaves/config/solr.xml`. Ensure that the default development URL doesn't conflict with any existing Solr's you may be running:

```
development:
  url: http://127.0.0.1:8982/solr
```

Start the included Solr by running `rake solr:start`. When it starts up, it will report the process ID for Solr running in the background. If you need to stop the process, then run the corresponding rake task: `rake solr:stop`. The empty new Solr indexes are stored in `./myfaves/solr/development`.

Build Solr indexes from relational database

Now we are ready to trigger a full index of the data in the relational database into Solr. `acts_as_solr` provides a very convenient rake task for this with a variety of parameters that you can learn about by running `rake -D solr:reindex`. We'll specify to work with a batch size of `1500` artists at a time:

```
>>rake solr:start
>>% rake solr:reindex BATCH=1500
(in /examples/8/myfaves)
Clearing index for Artist...
Rebuilding index for Artist...
Optimizing...
```

This drastic simplification of configuration in the Artist model object is because we are using a Solr schema that is designed to leverage the **Convention over Configuration** ideas of Rails. Some of the conventions that are established by `acts_as_solr` and met by Solr are:

- Primary key field for model object in Solr is always called `pk_i`.
- Type field that stores the disambiguating class name of the model object is called `type_s`.

- Heavy use of the dynamic field support in Solr. The data type of ActiveRecord model objects is based on the database column type. Therefore, when `acts_as_solr` indexes a model object, it sends a document to Solr with the various suffixes to leverage the dynamic column creation. In `/examples/8/myfaves/vendor/plugins/acts_as_solr/solr/solr/conf/schema.xml`, the only fields defined outside of the management fields are dynamic fields:

```
<dynamicField name="*_t" type="text" indexed="true"
    stored="false"/>
```

- The default search field is called `text`. And all of the fields ending in `_t` are copied into the `text` search field.
- Fields to facet on are named `_facet` and copied into the text search field as well.

The document that gets sent to Solr for our Artist records creates the dynamic fields `name_t`, `group_type_s` and `release_date_d`, for a text, string, and date field respectively. You can see the list of dynamic fields generated through the schema browser at `http://localhost:8982/solr/admin/schema.jsp`.

Now we are ready to perform some searches. `acts_as_solr` adds some new methods such as `find_by_solr()` that lets us find ActiveRecord model objects by sending a query to Solr. Here we find the group `Smash Mouth` by searching for matches to the word `smashing`:

```
% ./script/console
Loading development environment (Rails 2.3.2)
>> artists = Artist.find_by_solr("smashing")
=> #<ActsAsSolr::SearchResults:0x224889c @solr_data={:total=>9,
      :docs=>[#<Artist id: 364, name: "Smash Mouth"...
>> artists.docs.first
=> #<Artist id: 364, name: "Smash Mouth", group_type: 1,
      release_date: "2006-09-19 04:00:00", created_at: "2009-04-17
      18:02:37", updated_at: "2009-04-17 18:02:37">
```

Let's also verify that `acts_as_solr` is managing the full lifecycle of our objects. Assuming Susan Boyle isn't yet entered as an artist, let's go ahead and create her:

```
>> Artist.find_by_solr("Susan Boyle")
=> #<ActsAsSolr::SearchResults:0x26ee298 @solr_data={:total=>0,
    :docs=>[]}>
>> susan = Artist.create(:name => "Susan Boyle", :group_type => 1,
    :release_date => Date.new)
=> #<Artist id: 548200, name: "Susan Boyle", group_type: 1,
    release_date: "-4712-01-01 05:00:00", created_at: "2009-04-21
    13:11:09", updated_at: "2009-04-21 13:11:09">
```

Check the log output from your Solr running on port 8982, and you should also have seen an update query triggered by the insert of the new Susan Boyle record:

```
INFO: [] webapp=/solr path=/update params={} status=0 QTime=24
```

Now, if we delete Susan's record from our database:

```
>> susan.destroy
=> #<Artist id: 548200, name: "Susan Boyle", group_type: 1,
    release_date: "-4712-01-01 05:00:00", created_at: "2009-04-21
    13:11:09", updated_at: "2009-04-21 13:11:09">
```

Then there should be another corresponding update issued to Solr to remove the document:

```
INFO: [] webapp=/solr path=/update params={} status=0 QTime=57
```

You can verify this by doing a search for Susan Boyle directly, which should return no rows at `http://localhost:8982/solr/select/?q=Susan+Boyle`.

Complete MyFaves web site

Now, let's go ahead and put in the rest of the logic for using our Solr-ized model objects to simplify finding our favorite artists. We'll store the list of favorite artists in the browser's session space for convenience. If you are following along with your own generated version of MyFaves application, then the remaining files you'll want to copy over from `/examples/8/myfaves` are as follows:

- `./app/controller/myfaves_controller.rb` contains the controller logic for picking your favorite artists.
- `./app/views/myfaves/` contains the display files for picking and showing the artists.
- `./app/views/layouts/myfaves.html.erb` is the layout for the MyFaves views. We use the Autocomplete widget again, so this layout embeds the appropriate JavaScript and CSS files.
- `./public/javascripts/blackbirdjs/` contains everything required to use the Blackbird logging library.
- `./public/stylesheets/jquery.autocomplete.css` and `./public/stylesheets/indicator.gif` are stored locally in order to fix pathing issues with the `indicator.gif` showing up when the autocompletion search is running.

The only other edits you should need to make are:

- Edit `./config/routes.rb` by adding `map.resources :myfaves` and `map.root :controller => "myfaves"`.
- Delete `./public/index.html` to use the new root route.
- Copy the index method out of `./app/controllers/artists_controllers.rb`, because we want the index method to respond with both HTML and JSON response types.
- Run `rake db:sessions:create` to generate a sessions table, then `rake db:migrate` to update the database with the new sessions table. Edit `./config/environment.rb` and add `config.action_controller.session_store = :active_record_store`. As we are storing Artist model objects in our session, we need to store them in the database versus in a cookie for space reasons.

You should now be able to run `./script/server` and browse to `http://localhost:3000/myfaves`. You will be prompted to enter an artist's name to search for. If you don't receive any results, then make sure you have started Solr using `rake solr:start`. Also, if you have only loaded a subset of the full 399,000 artists, then your choices may be limited. You can load all of the artists through the `populate.rb` script and then run `rake solr:reindex`, it will take a long time. Something good to do just before you head out for lunch or home for the evening!

If you look at `./app/views/myfaves/index.rhtml`, then you can see the jQuery autocomplete call is a bit different:

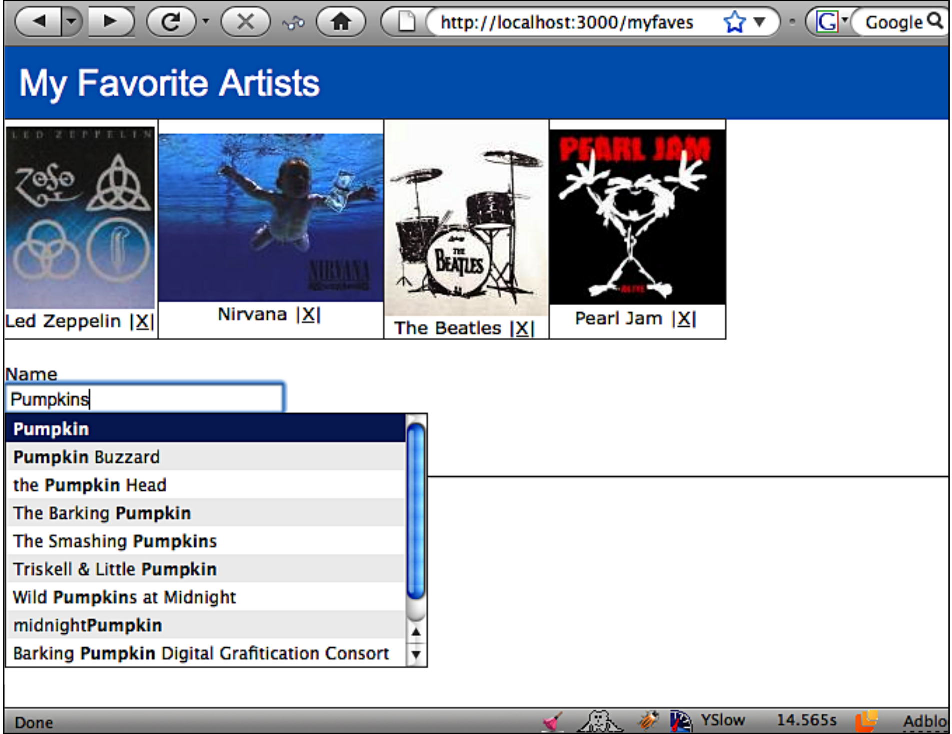
```
$("#artist_name").autocomplete( '/artists.json?callback=?', {
```

The URL we are hitting is `/artists.json`, with the `.json` suffix telling Rails that we want JSON data back instead of normal HTML. If we ended the URL with `.xml`, then we would have received XML formatted data about the artists. We provide a slightly different parameter to Rails to specify the JSONP callback to use. Unlike the previous example, where we used `json.wrf`, which is Solr's parameter name for the callback method to call, we use the more standard parameter name `callback`. We changed the `ArtistController` `index` method to handle the `autocomplete` widgets data needs through JSONP. If there is a `q` parameter, then we know the request was from the `autocomplete` widget, and we ask Solr for the `@artists` to respond with. Later on, we render `@artists` into JSON objects, returning only the `name` and `id` attributes to keep the payload small. We also specify that the JSONP callback method is what was passed when using the `callback` parameter:

```
def index
  if params[:q]
   @artists = Artist.find_by_solr(params[:q], :limit =>
```

```
            params[:limit]).docs
      else
       @artists = Artist.paginate :page => params[:page], :order =>
            'created_at DESC'
      end
      respond_to do |format|
       format.html # index.html.erb
       format.xml { render :xml => @artists }
       format.json { render :json => @artists.to_json(:only => [:name,
            :id]), :callback => params[:callback] }
      end
    end
```

At the end of all of this, you should have a nice interface for quickly picking artists:



When you are selecting `acts_as_solr` as your integration method, you are implicitly agreeing to the various conventions established for indexing data into Solr. `acts_as_solr` is a wonderful solution if you are indexing just a few unrelated models and don't have multiple data sources feeding your Solr indexes. While `acts_as_solr` has evolved to support more complex solutions (for example, by adding faceting support or the ability to perform more complex mappings with custom logic), it has its limits.

If you have a very complex data model with lots of inter-relationships that do not more or less map one-to-one with what you'd expect from search results, then you may find yourself running into edge cases that `acts_as_solr` doesn't support cleanly—especially if you are doing searches against specific fields in Solr versus the default `text` field. However, if your requirement is to quickly get your **ActiveRecord** model objects searchable, then `acts_as_solr`
can't be beat!

Blacklight OPAC

Blacklight is an open source **Online Public Access Catalog** (**OPAC**) that demonstrates the power of a highly configurable Ruby on Rails frontend paired with Solr. OPACs are the modern web enabled version of the classic card catalog that allow libraries to easily put their collections online. Blacklight supports parsing of various standard library catalog storage formats including MARC records and TEI XML format. Blacklight 2.0 was released in March of 2009 as a **Rails Engine** plugin. Rails Engine plugins allow users to integrate the rich functionality of the plugin, while keeping the plugin related code and assets, such as JavaScript, CSS and images, separate from the hosting application, thus facilitating upgrades to the Blacklight Engine. You may find that Blacklight provides an excellent starting point for your own Solr/Ruby on Rails development.

Let's go ahead and index information from `MusicBrainz.org` into Blacklight, just to see how easy it is. Please refer to the sample application in `/examples/8/blacklightopac/blacklight/`. Blacklight project is releasing frequent updates, so you should refer to the main web site at `http://www.blacklightopac.org/`.

Almost all of the dependencies are included in the `blacklight` sample application. You will need to install a couple of gems:

```
>>sudo gem install curb
>>sudo gem install bcrypt-ruby
```

Indexing MusicBrainz data

Blacklight builds on top of the `rsolr` library for communicating back and forth with the Solr server and adds some concepts around mapping data into Solr. Unlike `acts_as_solr`, Blacklight doesn't require the source data to be in a database. Instead you build a custom **Mapper** to fetch the data for Blacklight.

Blacklight requires some synchronization between the Solr and Ruby on Rails sides to make things work. Blacklight expects a search handler called **search** to be configured, while specifying which schema fields are facets and which are just straight fields of data to be returned. We are going to index various artists and their music releases from the `MusicBrainz.org` site, while creating facets for the **languages**, **scripts**, and **types** of releases. For example, The Dave Matthews Band's album *Under the Table and Dreaming* is in English, using the standard Latin script for the album notes, and is an Album. We are going to be indexing many artists from non-Western countries. Fortunately, Solr and Blacklight support alternative character sets such as Cyrillic, Kanji, and Chinese characters. You can see that we are still using conventions for how we name the schema fields, with `_t` signifying text, and `_facet` signifying a field for faceting on:

```
<requestHandler name="search" class="solr.SearchHandler" >
  <lst name="defaults">
   <str name="fl">id, format_code_t, language_facet, script_facet,
        type_facet, releases_t, title_t, score</str>
   <str name="facet">on</str>
   <str name="facet.mincount">1</str>
   <str name="facet.limit">10</str>
   <str name="facet.field">language_facet</str>
   <str name="facet.field">script_facet</str>
   <str name="facet.field">type_facet</str>
  </lst>
 </requestHandler>
```

We also need to tell Blacklight through the `./config/solr.xml` which facets and fields to display in the UI. We are using the field `title_t` to store the artist's name:

```
facet_fields:
- type_facet
- language_facet
- script_facet
index_view_fields:
- title_t
- language_facet
- script_facet
- type_facet
- releases_t
```

One of the nice features about Blacklight is that it provides an architectural pattern for mapping information from any data source into Solr that you can mimic for your own use. We added `./lib/tasks/brainz.rake` to give us the ability to load the information from MusicBrainz by running a simple Rake task: `rake app:index:brainz`. The Rake task is defined in `./lib/tasks/brainz.rake`. The core of the task instantiates a **BrainzMapper** class (that we developed) that provides a collection of documents related to Artists and their music Releases for Solr to index. In order to reduce memory usage, we index artists alphabetically, while committing the results to Solr periodically:

```
solr = Blacklight.solr
mapper = BrainzMapper.new
('A' .. 'Z').each do |char|
 mapper.from_brainz("#{char}*") do |doc,index|
  puts "#{index} -- adding doc w/id : #{doc[:id]} to Solr"
  solr.add(doc)
 end
 puts "Sending commit to Solr..."
 solr.commit
end
puts "Complete."
```

The real magic of the Blacklight mapper pattern is in the BrainzMapper class in `./lib/brainz_mapper.rb`. While the class may look a little hairy, it is actually quite simple. The pattern is defined by the base class **BlockMapper**. BlockMapper expects us to define a series of `map` methods for each field that we want to store in Solr. For example, to store the artist's name in the previously mentioned `title_t` field, we define it this way:

```
map :title_t do |rec,index|
 rec[:artist].name
end
```

This says that to map the `:title_t` field, we are handed our record object and the index of that record in our overall collection of records to be stored in Solr. In our case, we have populated the record object as a hash with two keys, `:artist` and `:releases`, whose values are an artist and their releases. In the `:title_t` mapping case, we ask the record hash for the artist object and call the `.name()` method.

How about a slightly more complex example, mapping all of the releases for an artist:

```
map :releases_t do |rec, index|
 rec[:releases].collect {|release|release.entity.title}.compact.uniq
end
```

In this case, when we map the `releases_t` field, we obtain the releases object, which is an array of `MusicBrainz::Model::Release` objects. From each one we get the title of the release. The resulting array is compacted to remove any nil objects, and then only unique release titles are returned, as sometimes we have multiple releases listed with the same name. Blacklight properly handles storing a single value or an array of values in the `releases_t` field, as any field ending in `_t` is specified as `multiValued="true"` in `schema.xml`.

Very similar logic is used for mapping our facets as well. In this case, we are using the `MusicBrainz::Utils.get_language_name` method to translate from three letter language codes like "ENG" to "English" in order to have a prettier display in our facets:

```
map :language_facet do |rec,index|
rec[:releases].collect {|release| MusicBrainz::Utils.get_language_
name(release.entity.text_language)}.
           compact.uniq
end
```

Okay, we've seen the mapping logic, but where does the data come from? How are we populating the individual record hash object with `:artist` and `:releases` values? Web services to the rescue! MusicBrainz has an XML based web service that follows the **REST** design pattern that you can learn more about at `http://musicbrainz.org/doc/XMLWebService`. Even by using the web service directly, you still need to parse and manipulate XML documents. Fortunately, there is the very nice **rbrainz** Ruby gem available from `http://rbrainz.rubyforge.org/` that abstracts away all of the plumbing for communicating with MusicBrainz through XML. Instead, we work with higher level abstractions like **Query** and **Artist** objects. In the query below, we are asking for all of the artists similar to Dave Matthews Band, returning records 50 through 100.

```
require 'rbrainz'
query = MusicBrainz::Webservice::Query.new
results = query.get_artists({:name => 'Dave Matthews Band', :limit =>
        50, :offset => 50})
```

MusicBrainz uses Lucene for its search engine, and it permits you to use Lucene's syntax (see Chapter 4 of this book) in your queries. So, to find every band except the Dave Matthews Band we would execute:

```
results = query.get_artists({:name => 'Dave Matthews NOT Band'})
```

The method `create_records_from_music_brainz(query_string)` in `./lib/brainz_mapper.rb` returns a collection of record hashes containing artist and release data downloaded from MusicBrainz through `rbrainz`.

In order to run Blacklight, first start the included Solr in `./examples/8/blacklightopac/blacklight/jetty` through

```
>>java -jar start.jar
```

Then, run the indexing process in `./examples/8/blacklightopac/blacklight/rails` which downloads artists alphabetically from A to Z:

```
>>rake app:index:brainz
```

Indexing is very slow due to all of the HTTP requests being made to MusicBrainz web site. Artists are downloaded in batches of 100, with up to 1000 artists per letter, and then each artist requires a separate HTTP request to find their music releases. So indexing a thousand artists for the letter *P* requires roughly 1010 HTTP queries ((1000 / 100) + 1000). Additionally, you'll notice that the query parameter using just a single alphabetical character, such as `D*`, leads to somewhat odd matches. Records are only indexed into Solr once all of the artist/release data for a letter is downloaded, so you need to wait for a complete letter to finish. However, soon you will have thousands of artists and their releases in Solr that you can browse through.

Customizing display

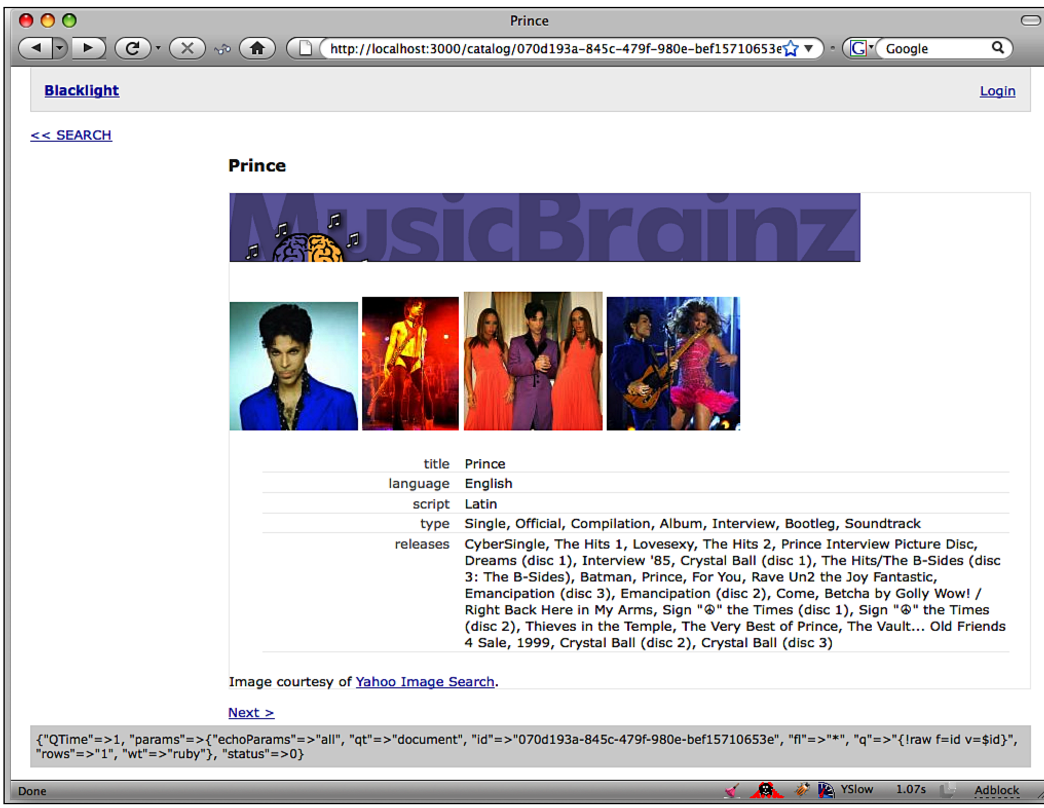
The user interface for Blacklight is fairly clean but pretty bland and displays every type of information the same way. However, based on the `format_code_t` field, you can easily customize the display. If you are indexing records with different types, such as Artists, Record Labels, and so on, then you can have a different display by populating `format_code_t` differently. We've chosen to just index Artists in this example, and defined `:format_code_t` to be *brainz*. As every record indexed uses the same value, we populate the `shared_field_data` parameter when calling the `from_brainz` method of the mapper:

```
mapper.from_brainz("#{char}*", {:format_code_t => 'brainz'})
       do |doc,index|

def from_brainz(query_string, shared_field_data={}, &blk)

  shared_field_data.each_pair do |k,v|
   # map each item in the hash to a solr field
   map k.to_sym, v
  end
```

Any values put into the `shared_field_data` hash will be set on every field. A common use case for the `shared_field_data` hash is to set an `:indexed_by_s` property that specifies the name of the user who invoked the indexing process.



There are two ways of customizing the display of fields. One of them is the above mentioned `./config/solr.xml` that allows us to filter the list of fields to display on the index page and the details page. However, that is a one-size-fits-all solution and still doesn't let you tweak the actual user interface depending on the data to display. There is another option that leverages the dynamic pathing of Rails to specify that view files should first be loaded from `./app/views`, and if not found, then load them from the Blacklight plugin. For example, we created a custom **partial**, which is to be rendered for the detailed view of an artist that incorporates the MusicBrainz logo and some photos of the artist. By placing the partial in `./app/views/catalog/_show_partials/_brainz.html.erb`, the name of the partial is mapped directly to the `format_code_t` value of *brainz*. So, if you indexed multiple entities, then `./app/views/catalog/_show_partials/_artists.html.erb` and `./app/views/catalog/_show_partials/_releases.html.erb` map onto `format_code_t` of `artists` and `releases` respectively. Sometimes, you don't want to override

Blacklight's UI. For example, we don't have a custom display partial when displaying listings for a search. Blacklight checks for the existence of `./app/views/catalog/_index_partials/_brainz.html.erb`. If it doesn't find that file, then it defaults to the `_default.html.erb` partial stored in `./vendor/plugins/blacklight/app/views/catalog/_index_partials/_default.html.erb`. This makes it very easy to override the default behaviors of Blacklight without requiring changes to the underlying plugin. This facilitates the upgrade of the plugin, as new Blacklight versions are released.

solr-ruby versus rsolr

For a lower-level client interface to Solr from Ruby environments, there are two libraries duking it out to be the client of choice. In one corner you have **solr-ruby**, which is the client library officially supported by the Apache Solr project. `solr-ruby` is fairly widely used, including providing the API to Solr used by the `acts_as_solr` Rails plugin we looked at previously. The new kid on the block is **rsolr**, which is a re-imagining of what a proper **DSL** (**Domain Specific Language**) would look like for interacting with Solr. `rsolr` is used by Blacklight OPAC as its interface to Solr. Both of these solutions are solid. However, `rsolr` is currently gaining more attention, has better documentation, and nice features such as a direct Embedded Solr connection through JRuby. `rsolr` also has support for using either `curb` (Ruby bindings to `curl`, a very fast HTTP library) or the standard `Net::HTTP` library for the HTTP transport layer.

In order to perform a select using `solr-ruby`, you would issue:

```
response = solr.query('washington', {
  :start =>0,
  :rows=>10
 })
```

In order to perform a select using `rsolr`, you would issue:

```
response = solr.select({
  :q=>'washington',
  :start=>0,
  :rows=>10
 })
```

So you can see that doing a basic search is pretty much the same in either library. Differences do crop up more as you dig into the details on parsing and indexing records. Both libraries are evolving, with neither having a dominant position at this point. You can learn more about `solr-ruby` on the Solr Wiki at `http://wiki.apache.org/solr/solr-ruby` and learn more about `rsolr` at `http://github.com/mwmitchell/rsolr/tree`.

Summary

As you've seen, Solr offers a plethora of integration options, from its ability to customize its output using the various writer types to the large number of different clients that provide powerful frontends for both indexing content as well as providing a jump start in developing the search frontend user interface. The simplicity of using HTTP GET to request actions to be performed by Solr and responding with simple documents makes it very straightforward to integrate Solr based search into your applications regardless of what your preferred development environment is.

In the next chapter, we are going to look at how to scale Solr to meet growing demand by covering approaches for scaling an individual Solr server as well as scaling out by leveraging multiple Solr servers working cooperatively.

9
Scaling Solr

You've deployed Solr, and the world is beating a path to your door, drastically increasing the average queries per minute, and meanwhile you've indexed tenfold the amount of information you originally expected. You'll discover that Solr is beginning to respond to queries slower and slower and indexing new content is taking longer and longer. When that happens, it's time to start looking at what configuration changes you can make to Solr to support more load. We'll look at a series of changes/optimizations that you can make, going from simplest changes giving most bang for your buck to more complex changes that require more analysis and system changes.

In this chapter, we will cover the following topics:

- Tuning complex systems
- Optimizing a single Solr server (Scale High)
- Moving to multiple Solr servers (Scale Wide)
- Combining replication and sharding (Scale Deep)

Tuning complex systems

Tuning any complex system, whether it's a database, a message queuing system, or the deep dark internals of an operating system, is something of a black art. Researchers and vendors have spent decades figuring out how to measure the performance of systems and coming up with approaches for maximizing the performance of those systems. For some systems that have been around for decades, such as databases, you can just search online for "Tuning Tips for X Database" and find explicit rules that suggest what you need to do to gain performance. However, even with those well researched systems, it still can be a matter of trial and error.

In order to measure the impact of your changes, you should look at a couple of metrics and optimize for these three parameters:

- **Transactions Per Second (TPS)**: In the Solr world, how many search queries and document updates are you able to perform per second? You can get a sense of that by using the **Statistics** page at `http://localhost:8983/solr/mbtracks/admin/stats.jsp` and looking at the `avgTimePerRequest` and `avgRequestsPerSecond` parameters for your request handlers.
- **CPU usage**: To quickly gain a sense of CPU usage on Windows, use **PerfMon**, and on Unix based systems, use **top**. PerfMon allows you to quickly graph the CPU utilization of your various processes. `top` is a command line tool that displays the current system load of the various processes running. It is very powerful, and you may want to run `man top` to learn the various options available for customizing the display. Unfortunately, top doesn't do graphing, so you'll need to do that on your own.
- **Memory usage**: When tuning for memory management, you are looking to ensure that the amount of memory allocated to Solr doesn't constantly grow. While it's okay for the memory consumption to go up a bit, letting it grow unconstrained eventually means you will receive out-of-memory errors! Balance increases in memory consumption with significant increases in TPS. You can use PerfMon, top, and JConsole to keep an eye on memory usage.

In order to get a sense of what the **Steady State** is for your application, you can gather the statistics by downloading the values exposed through JMX over a period of time. Look back at the JMX section of Chapter 7. You can also develop a script that loads a set of documents and then issues queries to define your Steady State. Developing a script that accurately mirrors the real world interactions with Solr can be challenging. However, it gives you something that can be run over and over quickly that allows more of an apple-to-apple comparison of the impact of your changes.

Solr's architecture has benefited from its heritage, as the search engine developed in-house from 2004 to 2006 that powers `CNET.com`, a site that is ranked 77th in traffic by `Alexa.com` today. Solr, right out of the box, is already very performant, with extensive effort spent by the community to ensure that there are no bottlenecks, and leveraging best practices for using Lucene for search. But the tuning of Solr hasn't matured to where there are hard and fast rules for optimizing it that you should follow by rote step to increase scalability. Many of the options enhance performance for certain operations, but hinder performance for other operations. However, the three system changes to perform in increasing complexity are:

- **Scale High**: Optimize a single instance of Solr. Look at things such as caching and memory configuration. Run Solr on a dedicated server with very fast CPUs and hard drives. In the scale high approach, you are trying to maximize what you can get from a single server.
- **Scale Wide**: Look at moving to multiple Solr servers. If your queries run quickly with an acceptable `avgTimePerRequest`, then replicate your complete index across multiple Solr servers in a master/slave configuration. If your queries are running slowly, then use sharding to split the load of processing a single query across multiple sharded Solr servers. Both these are approaches that can be considered as scaling wider.
- **Scale Deep**: If you need both sharding for query performance and multiple replicated indexes to support the query load, then move to each shard being a master server with multiple slave servers. This is the scale deep approach and is the most complex architecture to implement.

There is some great research being performed on measuring the limits of scaling Solr. This is being done by a consortium of libraries called **HathiTrust**. You can follow their research (and others working in this space) by following links from `http://wiki.apache.org/solr/SolrPerformanceData`.

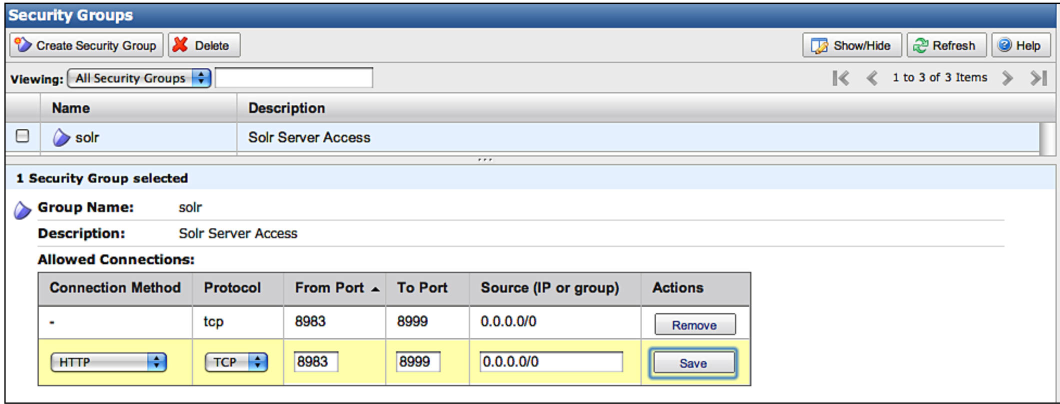
Using Amazon EC2 to practice tuning

Amazon EC2 is a cloud computing service provided by Amazon that provides a great environment to practice tuning techniques. We have published an image of a Linux server called **solrbook-packtpub** that you can use to try out some of the techniques that we'll talk about in this chapter. You can spin up as many Solr instances as you want, spread across multiple servers, and run the testing scripts as well. Amazon currently charges a mere 10 cents US for an hour of server usage. So by using the same server image that we used you should receive similar results to what we've received. For a couple of dollars, using EC2 will let you have as close to apple-to-apples comparison with our results as possible! We've put together a set of Ruby scripts in `./examples/9/amazon` that exercise the Solr server, and you can use them against your own local copy of Solr or run them against the Amazon image.

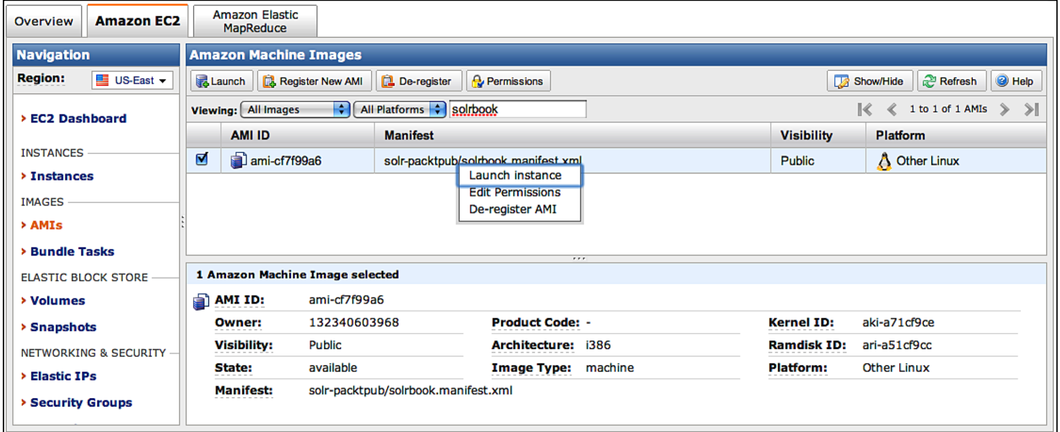
These scripts that we use are obviously only one approach to testing an application and don't reflect real world usage that a site might undergo. Instead, they attempt to make it easy to play around with various parameters and get a better sense of what scaling high, wide, and deep will result in. If you were doing real world testing, then you might want to record/playback actual search requests.

Firing up Solr on Amazon EC2

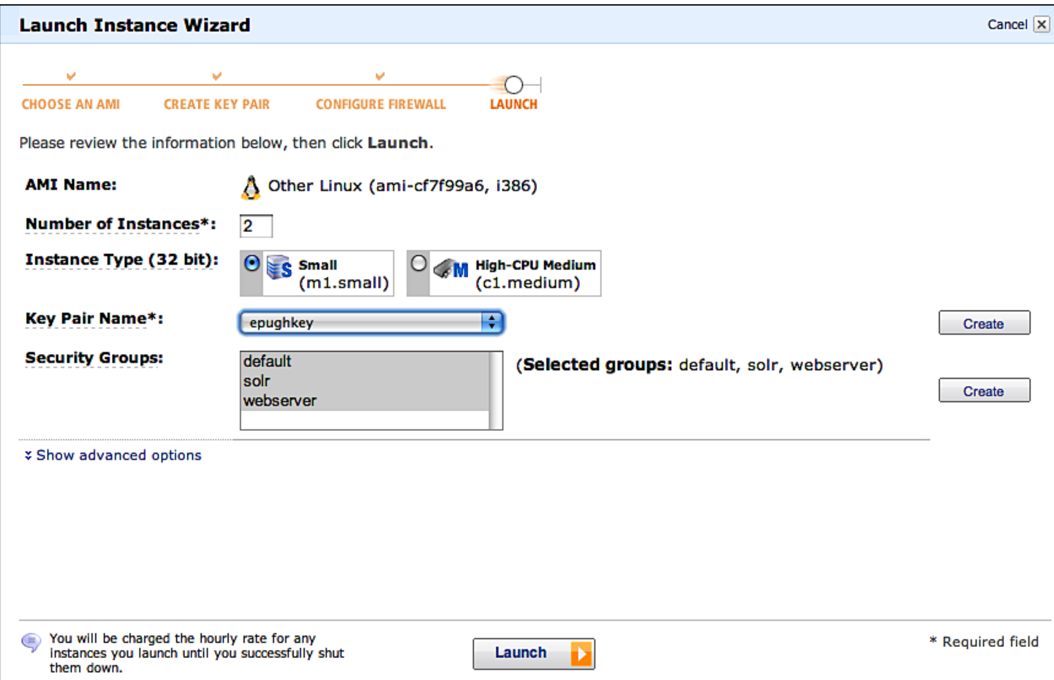
In order to start using Solr on Amazon EC2, we assume that you have set up an account with **Amazon Web Services**, available from `http://aws.amazon.com/`. Firstly, we use the AWS Management Console web app at `https://console.aws.amazon.com` to create a **Security Group** for Solr that opens up ports, which the Solr master and slaves will communicate over. Just click on **Security Groups** and create a new Solr security group with ports 8983 through 8999 open:



Then click on **AMIs** and search for *solr-packtpub/solrbook* to find the **Amazon Machine Instance** (**AMI**) prepared for this book. Just right-click on it and choose **Launch instance**:



You will then be prompted to fill in some information about the instances you want to launch, similar to the screenshot below:



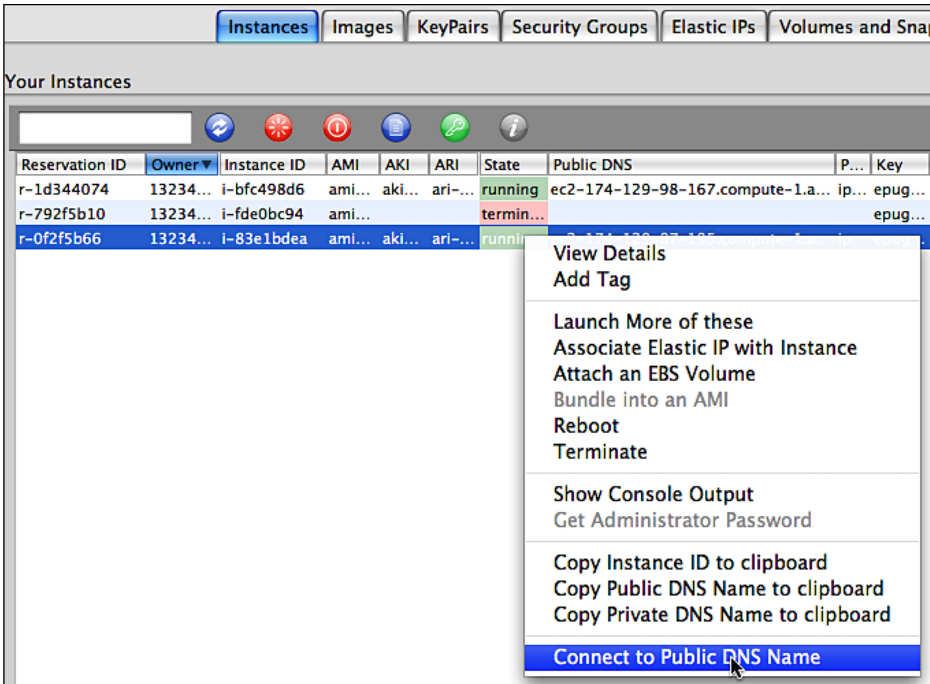
In a couple of minutes, you'll have a Linux server configured with all of the apps and code required for the scaling demos running in the cloud! In order to connect through SSH to the new instances, click on the **Connect** button for specifics.

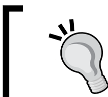
In the *solrbook* image, a copy of the Solr trunk branch has been checked out in `~/asf_solr_src/` for you to use, and the example app already compiled. The Ruby scripts for testing are stored in `~/examples/9/amazon/`.

In order to start up an instance SSH into your instance, go to `~/examples/`. Run `java -jar start.jar`. You can browse the running Solr by connecting to the **Public DNS** URL listed in the Console webapp:

```
http://ec2-174-129-98-167.compute-1.amazonaws.com:8983/solr
```

Amazon provides a Firefox plugin called **Elasticfox** that duplicates much of what the Management Console does, but has additional functionality such as pop-up console log viewing and simplifying SSH'ing into your instances by remembering what private keys go with which instances:





Don't forget to terminate your EC2 instances when you are done! You are charged hourly regardless of whether they are actively used or not.

Optimizing a single Solr server (Scale High)

There are a large number of different options that Solr gives you for enhancing performance of a specific Solr instance, and for most of these options, deciding to modify them depends on the specific performance result you are trying to tune for. This section is structured from most generally useful to more specific optimizations.

JVM configuration

Solr runs inside a **Java Virtual Machine** (**JVM**), an environment that abstracts your Java based application from the underlying operating system. There are many different parameters that you can tune the JVM for. However, most of them are "black magic", and changing them from the defaults can quickly cause problems if you don't know what you're doing. Additionally, the folks who write JVMs spend a lot of time coming up with sophisticated algorithms that mean the JVM will usually tune itself better than you can. However, there is a fairly simple configuration change that most Java server applications benefit from (not just Solr), which is to increase the amount of minimum and maximum memory allocated to the JVM and specify that you are running a server application, so the JVM can optimize its optimization strategy for a long running process:

```
java -Xms512M -Xmx1024M -server -jar start.jar
```

You want to make sure that you don't specify a higher `Xmx` value than the actual memory you have and leave plenty of memory left over for the operating system as well as any other processes that may be running on the server simultaneously. However, if you have a 4 GB index and can allocate 6 GB of RAM (and specify large caches, see the *Solr caching* section explained later in the chapter), then you will gain more query performance.

If you can, use the latest released Java version. At this time, that is a 1.6 based release. Java VMs have gotten faster with each release. You might even investigate a third party VM like JRockit if you are willing to pay for one: `http://www.oracle.com/technology/products/jrockit/`.

HTTP caching

Solr has great support for using HTTP caching headers to enable down-stream HTTP software to cache results. Web browsers, intermediate proxy servers, and web servers can decide if they need to re-query for updated results by using various rules. For example, often applications allow a user to take a query and make it an **Alert** that will email them results if there is a match. This leads to the same search running over and over, even if the results are almost always the same. Placing an intermediate caching server, such as Squid, in front of Solr should reduce the load on Solr and potentially reduce Solr's internal "query cache" requirements, thus freeing up more RAM. When a request uses certain caching headers, Solr can then indicate whether the content has changed by either sending back an HTTP 200 status code if it has, or a **304 Not Modified** code when the content hasn't changed since the last time the request asked for it.

In order to specify that you want Solr to do HTTP caching, you need to configure the `<httpCaching/>` stanza in `solrconfig.xml`. By default, Solr is configured to never return 304 codes to instead always return a 200 response (a normal non-cached response) with the full body of the results. Change `httpCaching` to:

```
<httpCaching lastModifiedFrom="openTime"
             etagSeed="Solr" never304="false">
   <cacheControl>max-age=43200, must-revalidate</cacheControl>
</httpCaching>
```

We have specified that sending back 304 messages is okay and specified in the `cacheControl` that the max time to store responses is 43200 seconds, which is half a day. We've also specified through `must-revalidate` that any *shared cache*, such as a Squid proxy, needs to check back with Solr to see if anything has changed, even if the `max-age` hasn't expired, which acts as an extra check.

By running `curl` with the mbartists core, we can see additional cache related information in the header, as well as the full XML response from Solr (not listed):

```
>> curl -v http://localhost:8983/solr/mbartists/select/
?q=Smashing+Pumpkins
< HTTP/1.1 200 OK
< Cache-Control: max-age=43200
< Expires: Thu, 11 Jun 2009 15:02:00 GMT
< Last-Modified: Thu, 11 Jun 2009 02:55:39 GMT
< ETag: "YWFkZWIyNjVmODgwMDAwMFNvbHI="
< Content-Type: text/xml; charset=utf-8
< Content-Length: 1488
< Server: Jetty(6.1.3)
```

So let's look at what we get back if we take advantage of the **Last-Modified** header information by specifying that we have downloaded the content after the last modified time:

```
>>curl -v -z "Thu, 11 Jun 2009 02:55:40 GMT" http://localhost:8983/
solr/mbartists/select/?q=Smashing+Pumpkins
* About to connect() to localhost port 8983 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 8983 (#0)
> GET /solr/mbartists/select/?q=Smashing+Pumpkins HTTP/1.1
> User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3
OpenSSL/0.9.7l zlib/1.2.3
> Host: localhost:8983
> Accept: */*
> If-Modified-Since: Thu, 11 Jun 2009 02:55:40 GMT
>
< HTTP/1.1 304 Not Modified
```

```
< Cache-Control: max-age=43200
< Expires: Thu, 11 Jun 2009 15:13:43 GMT
< Last-Modified: Thu, 11 Jun 2009 02:55:39 GMT
< ETag: "YWFkZWIyNjVmODgwMDAwMFNvbHI="
< Server: Jetty(6.1.3)
```

Specifying an **If-Modified-Since** time just one second after the **Last-Modified** time means that Solr gives us back a **304 Not Modified** code and doesn't have to return all of the XML results over the wire, which is much faster and puts less load on the server.

Entity tags are a newer method for uniquely identifying responses that are more robust and flexible than using last modified date. An **ETag** is a string that identifies a specific version of a component. In the case of Solr, they are generated by combining the current version of the index with the ETag seed value specified in `httpCaching` configuration. Every time the index is modified, the current ETag value will change. If we add the fake artist "Eric's Band" to mbartists, and then run our previous query, we'll see that the ETag has changed because the version of the Solr index has changed:

```
>> curl http://localhost:8983/solr/mbartists/update -H "Content-
Type: text/xml" --data-binary '<add><doc><field name="a_name">Eric's
Band</field><field name="id" boost="2.0">Fake:99999</field><field
name="type">Artist</field></doc></add>'

>> curl http://localhost:8983/solr/mbartists/update/ -H "Content-Type:
text/xml" --data-binary '<commit waitFlush="false"/>'

>>curl -v -z "Thu, 11 Jun 2009 03:55:40 GMT" http://localhost:8983/
solr/mbartists/select/?q=Smashing+Pumpkins
>
< HTTP/1.1 304 Not Modified
< Cache-Control: max-age=43200
< Expires: Thu, 11 Jun 2009 15:35:58 GMT
< Last-Modified: Thu, 11 Jun 2009 03:26:33 GMT
< ETag: "NmFkZWIyNjVmODgwMDAwMFNvbHI="
< Server: Jetty(6.1.3)
```

Squid is one of the most popular caching proxy servers available, and the setup and configuration of it as a proxy server is beyond the scope of this book. You can learn more about Squid from its site `http://www.squid-cache.org/`.

Remember, the fastest query response possible from Solr's perspective is the query that it doesn't have to make!

Solr caching

Caching is a key part of what makes Solr fast and scalable, and the proper configuration of caches is a common topic on the `solr-user` mailing list! Solr uses multiple **Least Recently Used** in-memory caches. The caches are associated with individual **Index Searchers**, which represent a snapshot view of the data. Following a commit, new index searchers are opened and then **auto-warmed. Auto-warming** is where the cached values of the former searcher are copied over to the new searcher. Following auto-warming, predefined searches are run as configured in `solrconfig.xml`. Put some representative queries in the `newSearcher` and `firstSearcher` listeners, particularly for queries that need sorting on fields. Finally, the new searcher can serve new incoming requests.

New to Solr 1.4 is the **FastLRUCache,** which is faster than **LRUCache** because it doesn't require a separate thread for managing the removal of unused items. The obverse of this is that storing data is a bit slower because the calling thread is responsible for making sure that the cache hasn't grown too large.

There are a number of different caches configured in `solrconfig.xml`:

- **filterCache**: This stores unordered lists of documents that match a query. This is primarily used for storing filter queries (the `fq` parameter) for re-use, but it's also used in faceting under certain circumstances. It is arguably the most important cache. The filter cache can optionally be used for queries (the `q` parameter) that are not score-sorted if `useFilterForSortedQuery` is enabled in `solrconfig.xml`. However, unless testing reveals performance gains, it is best left disabled—the default setting.
- **queryResultCache**: This stores ordered lists of documents. The order is defined by any sorting parameters passed. This cache should be large enough to store the results of the most common searches, which you can identify by looking at your server logs. This cache doesn't use much memory, as only the ID of the documents is stored in the cache. The `queryResultWindowSize` setting allows you to preload document IDs into the cache if you expect users to request documents that bound the ordered list. So, if a user asks for products 20 through 29, then there is a good chance they will next look for 30 through 39. If the `queryResultWindowSize` is `50`, then the documents bounding the initial request from 0 to 50 will be returned and cached. When the user asks for 30 through 39, they will have their data cached and won't have to access the Lucene indexes!

- **documentCache**: This caches field values that have been defined in `schema.xml` as being stored, so that Solr doesn't have to go back to the filesystem to retrieve the contents when needed. Fields are stored by default. The documented wisdom on sizing this cache is to be larger than the max results * max concurrent queries being executed by Solr to prevent documents from being re-fetched during a query. As this cache contains the fields being stored, it can grow large very quickly.

These caches are all configured the same way:

- **class:** Defines whether to use LRUCache or FastLRUCache. The current wisdom is that for caches that don't have a high hit ratio, and therefore have more churn, you should use LRUCache. If you have a high hit ratio, then the benefits of FastLRUCache kick in.
- **size:** Defines the maximum items that the cache can support and is mostly dependent on how much RAM is available to the JVM.
- **autowarmCount:** Specifies how many items should be copied from an old search to a new one during the auto-warming process. Set the number too high and you slow down commits; set it too low and the new searches following those commits won't gain the benefits of the previously cached data. Look at the `warmupTime` statistic for your searches to balance these needs. There are some other options too, such as `initialSize`, `acceptableSize`, `minSize`, `showItems`, and `cleanupThread` specific to `FastLRUCache`, but specifying these are uncommon. There is a wealth of specific information available on the Wiki at `http://wiki.apache.org/solr/SolrCaching` that covers this constantly evolving topic.

Tuning caches

Using the statistics admin page, you can get a sense of how large you need to make your caches. If the hit ratio for your caches is low, then it may be that they aren't caching enough to be useful. However, if you find that the caches have a significant number of evictions, then that implies they are filling up too quickly and need to be made larger. Caches can be increased in size as long as Solr has sufficient RAM to operate in.

If your hit ratio for a cache is very low, then you should evaluate reducing its size, perhaps turning it off altogether by commenting out the cache configuration sections in `solrconfig.xml`. This will reduce memory needs that aren't being used effectively and may also help improve performance by removing the overhead of managing the caches.

Schema design considerations

Good schema design is probably one of the most important things you can do to enhance the scalability of Solr. You should refer to Chapter 2 for a refresher on many of the design questions that are inextricably tied to scalability. The biggest schema issue to look at for maximizing scalability is: Are you storing the minimum information you need to meet the needs of your users? There are a number of attributes in your schema field definitions which inform us about what is being indexed:

- `omitNorms`: Norms yield better scores of analyzed text content. They are also where index-time boosts are stored. If you have a field such as an ID or a string value that isn't a text field or needs boosting, then by skipping the norms, you can reduce the RAM required to perform a search as well as the time it takes to merge segments and optimize the index size.
- `omitTermFreqAndPositions`: This new Solr 1.4 feature allows you to skip indexing term related data such as the frequency and payload. If your schema version is at least 1.2, then this will be set appropriately already. The version is specified in `schema.xml` in the XML stanza:

  ```
  <schema name="example" version="1.2">
  ```
- `indexed`: You may find that you have fields of data that you don't ever search against. By specifying that they are NOT indexed, you can reduce the number of fields that require indexing.
- `stored`: Storing field values in the index simplifies and speeds search results, because results need not be cross-referenced and retrieved from original sources. It is also required for features such as highlighting. But storing field values will obviously increase the index size and indexing time. A quick thing to check is to do a simple search with `fl=*` as a parameter; the fields in the result are the stored fields. Some fields will probably not need to be stored. It is likely that your index has some data repeated but indexed differently for specialized indexing purposes like faceting or sorting—only one of those, if any, needs to be stored.

Another thing to look at is: If you need to store lots of data in a document, are you appropriately loading it? For example, if you have very large text fields being stored in the index, then it may make sense to compress those fields using the `compressed` field option to reduce the index size. If you don't always read all the fields, then enabling lazy field loading in `solrconfig.xml` via `<enableLazyFieldLoading>true</enableLazyFieldLoading>` can be very helpful.

If you are using compression, then you probably should be using the lazy loading feature of fields and reducing the cost of uncompressing the content to only when you need it. Otherwise, you are probably better off with a larger index size and faster querying.

Lastly, reduce to the minimum the amount of text analysis you perform in `schema.xml`. Text analysis can be very expensive in CPU time and can balloon up the size of your index.

Indexing strategies

Indexing documents into Solr is often a major bottleneck, either because your content is constantly changing or because you need to load a large volume of data initially. However, one way to speed up indexing is to index documents in batches. Solr supports sending multiple documents in a single add operation, and this can lead to a drastic speedup in performance.

However, as the size of your individual documents increase, performance may start to decrease. A reasonable rule of thumb is doing document add operations in batches of 10 for large documents, and 100 for small documents.

If we look at some basic indexing, the script `simple_test.rb` has a constant BATCH_SIZE that specifies the number of documents added at tone time. In the following examples, I ran the script and Solr on the same Amazon EC2 in order to avoid network overhead skewing the repeatability of the results. With one thread that adds documents one at a time, it took almost an hour to load 347,240 releases:

```
>> time ruby simple_test.rb http://localhost:8983/solr/mbreleases
        ../mb_releases.csv
real    59m4.350s
user    14m51.680s
sys     2m6.300s
```

But if we change the number of documents to be added from 1 to be in batches of 100 by changing BATCH_SIZE and running the same test, then we get a time of 22 minutes (almost three times faster):

```
>> time ruby simple_test.rb http://localhost:8983/solr/mbreleases
        ../mb_releases.csv
real    22m18.082s
user    7m21.576s
sys     0m33.934s
```

This shows that sending the documents in batches certainly helps the performance. However, if we can work with multiple threads and submit documents, then we should see a real increase. The `threaded_test.rb` script supports passing in the number of separate processes that are submitting documents (`4` in this example):

```
>> time ruby threaded_test.rb 4 http://localhost:8983/solr/mbreleases
        ../mb_releases.csv
```

You may want to have commits happen more frequently than at the very end of the indexing process. Uncomment the line by removing the leading `#` to have commits happen every 200 documents during the indexing process:

```
rsolr.commit if row_counter % 200 == 0  # uncomment this to see
      impact of frequent commits!  W
```

However, you may find that you are starting to see this error message instead of completing the indexing:

```
<h2>HTTP ERROR: 503</h2><pre>Error opening new searcher. exceeded
limit of maxWarmingSearchers=2, try again later.</pre>
```

`threaded_test.rb` is committing every 200 documents, and with 4 threads submitting 10 documents at a time, commits are happening very frequently. Every time a commit happens, a new searcher is created, which invokes the searcher warmup process where the cache is populated, which can take a while. While you can bump the `maxWarmingSearchers` by changing the value in `solrconfig.xml`, you are likely to still hit the new limit because each additional warming searcher slows things down for the rest. In order to deal with this, reduce how often commits are happening. You can reduce the amount of time auto-warming takes by reducing the `autowarmCount` and removing the `newSearch` query. Of course, this will lead to slower queries as well! Commits are only required to make the changes to the index visible to users. If you are bulk loading data, then you don't need real-time display of the changes.

StreamingUpdateSolrServer

If you are comfortable with Java and looking to optimize bulk adding of documents, then look at the `StreamingUpdateSolrServer` that is new to Solr 1.4 and is part of the SolrJ client. `StreamingUpdateSolrServer` extends `CommonsHttpSolrServer` by streaming document additions to Solr, which makes it pointless for your code to do batching. Additions are performed in an asynchronous manner, and a configurable number of threads can send data to Solr simultaneously. `StreamingUpdateSolrServer` is meant only for bulk adding/ updating documents and doesn't provide immediate error messaging feedback the way `CommonsHttpSolrServer` does. Look back at Chapter 8's crawler example to learn more.

Disable unique document checking

By default, when indexing content, Solr checks the uniqueness of the primary keys being indexed so that you don't end up with multiple documents sharing the same primary key. If you bulk load data into an index that you know does not already contain the documents being added, then you can disable this check. For XML documents being posted, add the parameter `allowDups=true` to the URL. For CSV documents being uploaded, there is a similar option `overwrite` that can be set to `false`.

Commit/optimize factors

There are some other factors that can impact how often you want commit and optimize operations to occur. If you are using Solr's support for scaling wide through replication of indexes, either through the legacy Unix scripts invoked by the post commit/post optimize hooks or the newer pure Java replication, then each time a commit or optimize happens you are triggering the transfer of updated indexes to all of the slave servers. If transfers occur frequently, then you can find yourself needlessly using up network bandwidth to move huge numbers of index files.

A similar issue is that if you are using the hooks to trigger backups and are frequently doing commits, then you may find that you are needlessly using up CPU and disk space by generating backups.

> Think about if you can have two strategies for indexing your content. One that is used during bulk loads that focuses on minimizing commits/optimizes and indexes your data as quickly as possible, and then a second strategy used during day-to-day routine operations that potentially indexes documents more slowly, but commits and optimizes more frequently to reduce the impact on any search activity being performed.

Another setting that causes a fair amount of debate is the `mergeFactor` setting, which controls how many segments Lucene should build before merging them together on disk. The rule of thumb is that the more static your content is, the lower the merge factor you want. If your content is changing frequently, or if you have a lot of content to index, then a higher merge factor is better. So, if you are doing sporadic index updates, then a merge factor of 2 is great, because you will have fewer segments which lead to faster searching. However, if you expect to have large indexes (> 10 GB), then having a higher merge factor like 25 will help with the indexing time.

Enhancing faceting performance

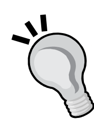
There are a few things to look at when ensuring that faceting performs well. First of all, faceting and filtering (the `fq` parameter) go hand-in-hand, thus monitoring the filter cache to ensure that it is adequately sized. The filter cache is used for faceting itself as well. In particular, any `facet.query` or `facet.date` based facets will store an entry for each facet count returned. You should ensure that the resulting facets are as reusable as possible from query-to-query. For example, it's probably not a good idea to have direct user input to be involved in either a `facet.query` or in `fq` because of the variability. As for dates, try to use fixed intervals that don't change often or round `NOW` relative dates to a chunkier interval (for example, `NOW/DAY` instead of just `NOW`). For text faceting (example `facet.field`), the filter-cache is basically not used unless you explicitly set `facet.method` to `enum`, which is something you should do when the total distinct values in the field are somewhat small, say less than 50. Finally, you should add representative faceting queries to `firstSearcher` in `solrconfig.xml`. So that when Solr executes its first user query, the relevant caches are warmed up.

Using term vectors

A **term vector** is a list of terms resulting from the text analysis of a field's value. It optionally contains the term frequency, document frequency, and numerical offset into the text. In Solr 1.4, it is now possible to tell Lucene that a field should store these for efficient retrieval. Without them, the same information can be derived at runtime but that's slower. While disabled by default, enabling term vectors for a field in `schema.xml` enhances:

- MoreLikeThis queries, assuming that the field is referenced in `mlt.fl` and the input document is a reference to an existing document (that is not externally posted)
- Highlighting search results

Enabling term vectors for a field does increase the index size and indexing time, and isn't required for either MoreLikeThis or highlighting search results. Typically, if you are using these features, then the enhanced results gained are worth the longer indexing time and greater index size.



Term vectors are very exciting when you look at clustering documents together. Clustering allows you to identify documents that are most similar to other documents. Currently, you can use facets to browse related documents, but they are tied together explicitly by the facet. Clustering allows you to link together documents by their contents. Think of it as dynamically generated facets.

Currently, there is ongoing work in the `contrib/cluster` source tree on integrating the Carrot2 clustering platform. Learn more about this evolving capability at `http://wiki.apache.org/solr/ClusteringComponent`.

Improving phrase search performance

For large indexes exceeding perhaps a million documents, phrase searches can be slow. What slows down phrase searches are the presence of terms in the phrase that show up in a lot of documents. In order to ameliorate this problem, the particularly common and uninteresting words like "the" can be filtered out through a stop filter. But this thwarts searches for a phrase like "to be or not to be" and prevents disambiguation in other cases where these words, despite being common, are significant. Besides, as the size of the index grows, this is just a band-aid for performance as there are plenty of other words that shouldn't be considered for filtering out yet are reasonably common.

The solution: Shingling

Shingling is a clever solution to this problem, which reduces the frequency of terms by indexing consecutive words together instead of each word individually. It is similar to the n-gram family of analyzers described in Chapter 2 in order to do substring searching, but operates on terms instead of characters. Consider the text "The quick brown fox jumped over the lazy dog". Depending on the shingling configuration, this could yield these indexed terms: "the quick", "quick brown", "brown fox", "fox jumped", "jumped over", "over the", "the lazy", "lazy dog".

In our MusicBrainz data set, there are nearly seven million tracks, and that is a lot! These track names are ripe for shingling. Here is a field type `shingle`, a field using this type, and a `copyField` directive to feed the track name into this field:

```
<fieldType name="shingle" class="solr.TextField"
      positionIncrementGap="100" stored="false" multiValued="true">
    <analyzer type="index">
        <tokenizer class="solr.StandardTokenizerFactory"/>
```

```
        <!-- potentially word delimiter, synonym filter, stop words,
         NOT stemming -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.ShingleFilterFactory" maxShingleSize="2"
                outputUnigrams="false"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <!-- potentially word delimiter, synonym filter, stop words,
         NOT stemming -->
        <filter class="solr.LowerCaseFilterFactory"/>
        <!-- outputUnigramIfNoNgram only honored if SOLR-744 applied.
         Not critical; just means single-words not looked up. -->
        <filter class="solr.ShingleFilterFactory" maxShingleSize="2"
         outputUnigrams="false"/>
    </analyzer>
</fieldType>

<field name="t_shingle" type="shingle" stored="false" />

<copyField source="t_name" dest="t_shingle" />
```

Shingling is implemented by `ShingleFilterFactory` and is performed in a similar manner at both index-time and query-time. Every combination of consecutive terms of one term in length up to the configured `maxShingleSize` (defaulting to `2`) is emitted. `outputUnigrams` controls whether or not each original term (a single word) passes through and is indexed on its own as well. When `false`, this effectively sets a minimum shingle size of `2`.

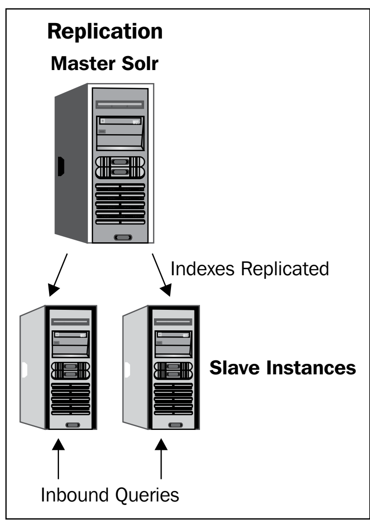
For the best performance, a shingled *query* needs to emit few terms for it to work. As such, `outputUnigrams` should be `false` on the query side, because multi-term queries would result in not just the shingles but each term passing through as well. Admittedly, this means that a search against this field with a single word will fail. However, a shingled field is best used solely for phrase queries alongside non-phrase variations. The dismax handler can be configured this way by using the `pf` parameter to specify `t_shingle`, and `qf` to specify `t_name`. A single word query *would not need to match* `t_shingle` because it would be found in `t_name`.

In order to fix `ShingleFilterFactory` for finding single word queries, it is necessary to apply patch SOLR-744, which gives an additional boolean option `outputUnigramIfNoNgram`. You would set that to `true` at query-time only, and set `outputUnigrams` to `true` at index-time only.

Evaluating the performance improvement of this addition proved to be tricky because of Solr's extensive caching. By configuring Solr for nearly non-existent caching, some rough (non-scientific) testing showed that a search for *Hand in my Pocket* against the shingled field versus the non-shingled field was two to three times faster.

Moving to multiple Solr servers (Scale Wide)

Once you've optimized Solr running on a single server, and reached the point of diminishing returns for optimizing further, the next step is to split the querying load over multiple slave instances of Solr. The ability to scale wide is a hallmark of modern scalable Internet systems, and Solr 1.4 shares that ability.



Script versus Java replication

Prior to Solr 1.4, replication was performed by using some Unix shell scripts that transferred data between servers through **rsync**, scheduled using **cron**. This replication was based on the fact that by using rsync, you could replicate only Lucene segments that had been updated from the master to the slave servers. The script-based solution has worked well for many deployments, but suffers from being relatively complex, requiring external shell scripts, cron jobs, and rsync daemons in order to be setup. You can get a sense of the complexity by looking at the Wiki page `http://wiki.apache.org/solr/CollectionDistribution` and looking at the various rsync and snapshot related scripts in `./examples/cores/crawler/bin` directory.

Introduced in Solr 1.4 is an all-Java-based replication strategy that has an advantage of not requiring complex external shell scripts and is faster. Configuration is done through the already familiar `solrconfig.xml`, and the configuration files such as `solrconfig.xml` can now be replicated, allowing specific configurations for master and slave Solr servers. Replication can now work across both Unix and Windows environments, and is integrated into the existing Admin interface for Solr. The admin interface now controls replication—for example, to force the start of replication or aborting a stalled replication. The simplifying concept change between the script approach and the Java approach was to remove the need to move snapshot files around by exposing metadata about the index through a REST API supplied by the **ReplicationHandler** in Solr. As the Java approach is the way forward for Solr's replication needs, we are going to focus on it.

Starting multiple Solr servers

We'll test running multiple separate Solr servers by firing up multiple copies of the `solr-packtpub/solrbook` image on Amazon EC2. The images contain both the server-side Solr code as well as the client-side Ruby scripts. Each distinct Solr server runs on its own virtualized server with its own IP address. This lets you experiment with multiple Solr's running on completely different servers. Note: If you are sharing the same `solrconfig.xml` for both master and slave servers, then you also need to configure at startup what role a server is playing.

- `-Dslave=disabled` specifies that a Solr server is running as a **master** server. The master server is responsible for pushing out indexes to all of the slave servers. You will store documents in the master server, and perform queries against the pool of slave servers.
- `-Dmaster=disabled` specifies that a Solr server is running as a **slave** server. Slave servers either periodically poll the master server for updated indexes, or you can manually trigger updates by calling a URL or using the Admin interface. A pool of slave servers, managed by a load balancer of some type, performs searches.

If you don't have access to multiple servers for testing Solr or want to use the EC2 service, then you can still follow along by running multiple Solr servers on the same server, say maybe on your local computer. Then you can use the same configuration directory and just specify separate data directories and ports.

- `-Djetty.port=8984` will start up Solr on port 8984 instead of the usual port 8983. You'll need to do this if you have multiple Servlet engines on the same physical server.

- `-Dsolr.data.dir=./solr/data8984` specifies a different data directory from the default one, configured in `solrconfig.xml`. You wouldn't want two Solr servers on the same physical server attempting to share the same data directory! I like to put the port number in the directory name to help distinguish between running Solr servers, assuming different servlet engines are used.

Configuring replication

Configuring replication is very easy. We have already configured the replication handler for the `mbreleases` core through the following stanza in `./examples/cores/mbreleases/solrconfig.xml`:

```
<requestHandler name="/replication" class="solr.ReplicationHandler" >
  <lst name="${master:master}">
    <str name="replicateAfter">startup</str>
    <str name="replicateAfter">commit</str>
    <str name="confFiles">stopwords.txt</str>
  </lst>
  <lst name="${slave:slave}">
   <str name="masterUrl">http://localhost:8983/solr/replication</str>
   <str name="pollInterval">00:00:60</str>
  </lst>
</requestHandler>
```

Notice the use of `${}` values for doing configuration of `solrconfig.xml` at runtime. This allows us to configure a single request handler for replication, and pass `-Dmaster=disabled` and `-Dslave=disabled` to control which list of parameters are used. The master server has been set to trigger replication on startup of Solr and when commits are performed. Configuration files can also be replicated to the slave servers through the list of `confFiles`. Replicating configuration files is useful when you modify them during runtime and don't want to go through a full redeployment process of Solr. Just update the configuration file on the master Solr, and they will be pushed down to the slave servers on the next pull. The slave servers are smart enough to pick up the fact that a configuration file was updated and reload the core. Java based replication is still very new, so check for updated information on setting up replication on Wiki at `http://wiki.apache.org/solr/SolrReplication`.

Distributing searches across slaves

Assuming you are working with the Amazon EC2 instance, go ahead and fire up three separate EC2 instances. Two of the servers will serve up results for search queries, while one server will function as the master copy of the index. Make sure to keep track of the various IP addresses!

Indexing into the master server

You can log onto the master server by using SSH with two separate terminal sessions. In one session, start up the server while specifying that `-Dslave=disabled`:

```
>> cd ~/examples
>> java -Dslave=disabled -Xms512M -Xmx1024M -Dfile.encoding=UTF8
   -Dsolr.solr.home=cores -Djetty.home=solr -Djetty.logs=solr/logs
   -jar solr/start.jar
```

In the other terminal session, we're going to take a CSV file of the MusicBrainz album release data to use as our sample data. The CSV file is stored in a ZIP format in `./examples/9/mb_releases.csv.zip`. Unzip the file so you have the full 69 megabyte dataset with over 600 thousand releases running:

```
>> unzip mb_releases.csv.zip
```

You can index the CSV data file through curl from either your desktop or locally on the Amazon EC2 instance. By doing it locally, we avoid the cost of transferring the 69 megabytes over the Internet:

```
>> curl http://localhost:8983/solr/mbreleases/update/csv -F f.r_
attributes.split=true -F f.r_event_country.split=true -F f.r_event_
date.split=true -F f.r_attributes.separator=' ' -F f.r_event_country.
separator=' ' -F f.r_event_date.separator=' ' -F commit=true -F stream.
file=/root/examples/9/mb_releases.csv
```

You can monitor the progress of streaming the release data by using the statistics page at `http://[MASTER URL]:8983/solr/mbreleases/admin/stats.jsp#update` and looking at the `docPending` value. Refresh the page, and it will count up to the total 603,090 documents!

Configuring slaves

Once the indexing is done, and it can take a while to complete, check the number of documents indexed; it should be 603,090. Now you are ready to push the indexes to the slaves. Log into each slave server through SSH, and edit the `./examples/cores/mbreleases/conf/solrconfig.xml` file to update the `masterUrl` parameter in the replication request handler to point to the IP address of the master Solr server:

```
<lst name="${slave:slave}">
   <str name="masterUrl">http://ec2-67-202-19-216
      .compute-1.amazonaws.com:8983/solr/mbreleases/replication</str>
   <str name="pollInterval">00:00:60</str>
</lst>
```

Then start each one by specifying that it is a slave server by passing `-Dmaster=disabled`:

```
>> cd ~/examples
>> java -Dmaster=disabled -Xms512M -Xmx1024M -Dfile.encoding=UTF8 -Dsolr.solr.home=cores -Djetty.home=solr -Djetty.logs=solr/logs -jar solr/start.jar
```

If you are running multiple Solr's on your local server instead, don't forget to distinguish between Solr slaves by passing in a separate port and data directory, by adding `-Djetty.port=8984 -Dsolr.data.dir=./solr/data8984`.

You can trigger a replication by using the Replication admin page for each slave. The page will reload showing you how much of the data has been replicated from your master server to the slave server. In the following screenshot, you can see that **71** of **128** megabytes of data have been replicated:

Current Replication Status	Start Time: Thu Jun 18 16:48:52 EDT 2009
	Files Downloaded: 32 / 35
	Downloaded: 71.92 MB / 128.94 MB [55.0%]
	Downloading File: _0.fdt, Downloaded: 24 MB / 80.99 MB [29.0%]
	Time Elapsed: 14s, Estimated Time Remaining: 11s, Speed: 5.14 MB/s
Controls	Disable Poll
	Replicate Now Abort

Typically, you would want to use a proper DNS name for the `masterUrl`, such as `master.solrsearch.mycompany.com`, so you don't have to edit each slave server. Alternatively, you can specify the `masterUrl` as part of the URL and manually trigger an update:

```
>> http://[SLAVE_URL]:8983/solr//mbreleases/replication?
   command=fetchindex&masterUrl=[MASTER_URL]
```

Distributing search queries across slaves

We now have three Solr's running, one master and two slaves in separate SSH sessions. We don't have a single URL that we can provide to clients, which leverages the pool of slave Solr servers. We are going to use **HAProxy**, a simple and powerful HTTP proxy server to do a round robin load balancing between our two slave servers running on the master server. This allows us to have a single IP address, and have requests redirected to one of the pool of servers, without requiring configuration changes on the client side. Going into the full configuration of HAProxy is out of the scope of this book; for more information visit HAProxy's homepage at `http://haproxy.1wt.eu/`.

On the master Solr server, edit the `/etc/haproxy/haproxy.cfg` file, and put your slave server URL's in the section that looks like:

```
listen  solr-balancer 0.0.0.0:80
        balance roundrobin
        option forwardfor
        server slave1 ec2-174-129-87-5.compute-1.amazonaws.com:8983
               weight 1 maxconn 512 check
        server slave2 ec2-67-202-15-128.compute-1.amazonaws.com:8983
               weight 1 maxconn 512 check
```

The `solr-balancer` process will listen to port 80, and then redirect requests to each of the slave servers, equally weighted between them. If you fire up some small and medium capacity EC2 instances, then you would want to weigh the faster servers higher to get more requests. If you add the master server to the list of servers, then you might want to weigh it low. Start up HAProxy by running

```
>> service haproxy start
```

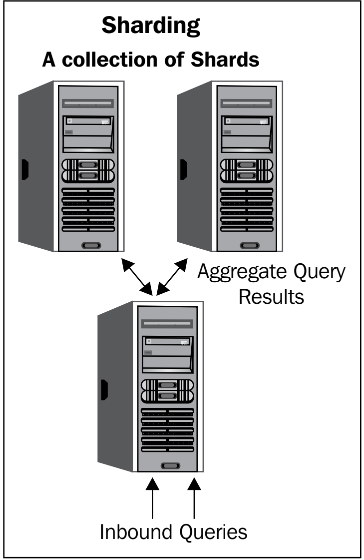
You should now be able to hit port 80 of the IP address of the master Solr, `http://ec2-174-129-93-109.compute-1.amazonaws.com`, and be transparently forwarded to one of the slave servers. Go ahead and issue some queries and you will see them logged by whichever slave server you are directed to. If you then stop Solr on one slave server and do another search request, you will be transparently forwarded to the other slave server!

If you aren't using the *solrbook* AMI image, then you can look at `haproxy.cfg` in `./examples/9/amazon/`.

There is a SolrJ client side interface that does load balancing as well. `LBHttpSolrServer` requires the client to know the addresses of all of the slave servers and isn't as robust as a proxy, though it does simplify the architecture. More information is on the Wiki at `http://wiki.apache.org/solr/LBHttpSolrServer`.

Sharding indexes

Sharding is the process of breaking a single logical index in a horizontal fashion across records versus breaking it up vertically by entities. It is a common database scaling strategy when you have too much data for a single database. In Solr terms, sharding is breaking up a single Solr core across multiple Solr servers versus breaking up a single Solr core over multiple cores through a multi core setup. Solr has the ability to take a single query and break it up to run over multiple Solr shards, and then aggregate the results together into a single result set. You should use sharding if your queries take too long to execute on a single server that isn't otherwise heavily taxed, by combining the power of multiple servers to work together to perform a single query. You typically only need sharding when you have millions of records of data to be searched.



If running a single query is fast enough, and if you are just looking for capacity increase to handle more users, then use the whole index replication approach instead!

Sharding isn't a completely transparent operation the way that replicating whole indexes is. The key constraint is when indexing the documents, you need to decide which Solr shard gets which documents. Solr doesn't have any logic for distributing indexed data over shards. Then when querying for data, you supply a `shards` parameter that lists which Solr shards to aggregate results from. This means a lot of knowledge of the structure of the Solr architecture is required on the client side. Lastly, every document needs a unique key (ID), because you are breaking up the index based on rows, and these rows are distinguished from each other by their document ID.

Assigning documents to shards

There are a number of approaches you can take for splitting your documents across servers. Assuming your servers share the same hardware characteristics, such as if you are sharding across multiple EC2 servers, then you want to break your data up more or less equally across the servers. We could distribute our `mbreleases` data based on the release names. All release names that start between *A* and *M* would go to one shard, the remaining *N* through *Z* would be sent to the other shard. However, the chance of an even distribution of release names isn't very likely! A better approach to evenly distribute documents is to perform a hash on the unique ID and take the mod of that value to determine which shard it should be distributed to:

```
SHARDS = ['http://ec2-174-129-178-110
           .compute-1.amazonaws.com:8983/solr/mbreleases',
          'http://ec2-75-101-213-59
           .compute-1.amazonaws.com:8983/solr/mbreleases']
unique_id = document[:id]
if unique_id.hash % SHARDS.size == local_thread_id
  # index to shard
end
```

As long as the number of shards doesn't change, every time you index the same document, it will end up on the same shard! With reasonably balanced documents, the individual shards calculation of what documents are relevant should be good enough. If you have many more documents on one server versus another, then the one with fewer documents will seem as relevant as the one with many documents, as relevancy is calculated on a per-server basis.

You can test out the script `shard_indexer.rb` in `./examples/9/amazon/` to index the `mb_releases.csv` across as many shards as you want by using the hashing strategy. Just add each shard URL to the SHARDS array defined at the top of `shard_indexer.rb`:

```
>> ruby shard_indexer.rb ../mbreleases.csv
```

You might want to change this algorithm if you have a pool of servers supporting your shards that are of varying capacities and if relevance isn't a key issue for you. For your higher capacity servers, you might want to direct more documents to be indexed on those shards. You can do this by using the existing logic, and then by just listing your higher capacity servers in the SHARDS array multiple times.

Searching across shards

The ability to search across shards is built into the query request handlers. You do not need to do any special configuration to activate it. In order to search across two shards, you would issue a search request to Solr, and specify in a `shards` URL parameter a comma delimited list of all of the shards to distribute the search across as well as the standard query parameters:

```
>> http://[SHARD_1]:8983/solr/select?shards=ec2-174-129-178-110.
compute-1.amazonaws.com:8983/solr/mbreleases,ec2-75-101-213-59.compute-
1.amazonaws.com:8983/solr/mbreleases&indent=true&q=r_a_name:Joplin
```

You can issue the search request to any Solr instance, and the server will in turn delegate the same request to each of the Solr servers identified in the `shards` parameter. The server will aggregate the results and return the standard response format:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">697</int>
    <lst name="params">
      <str name="indent">true</str>
      <str name="q">r_a_name:Joplin</str>
      <str name="shards">
        ec2-174-129-178-110.compute-1.amazonaws.com
        :8983/solr/mbreleases,ec2-75-101-213-59.compute-
        1.amazonaws.com:8983/solr/mbreleases
      </str>
    </lst>
  </lst>
  <result name="response" numFound="15" start="0"/>
</response>
```

The URLs listed in the shards parameter do not include the transport protocol, just the plain URL with the port and path attached. You will get no results if you specify `http://` in the shard URLs. You can pass as many shards as you want up to the length a GET URI is allowed, which is at least 4000 characters.

You can verify that the results are distributed and then combined by issuing the same search for `r_a_name:Joplin` to each individual shard and then adding up the `numFound` values.

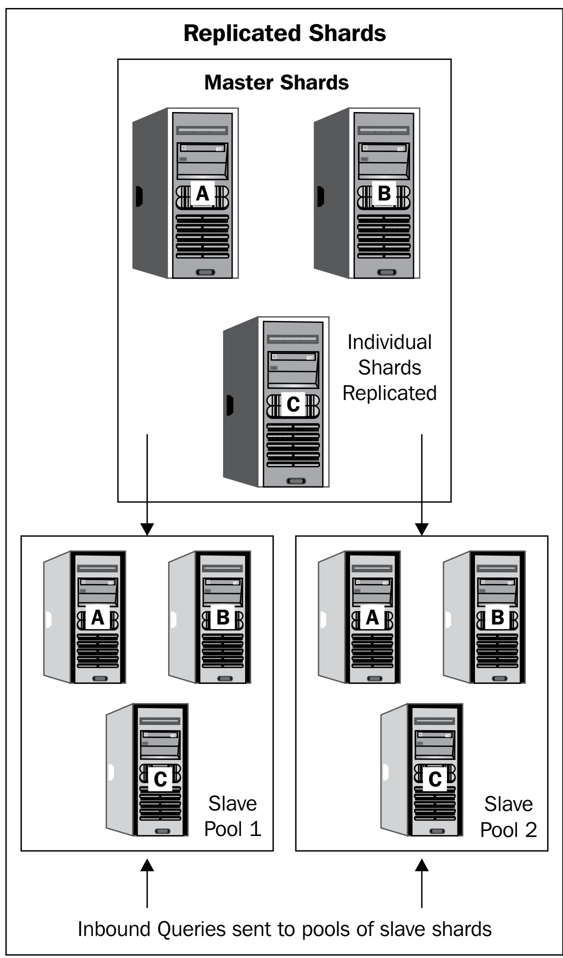
There are a few key points to keep in mind when using shards to support distributed search:

- Sharding is only supported by certain components such as Query, Faceting, Highlighting, Stats, and Debug.
- Each document must have a unique ID. This is how Solr figures out how to merge the documents back together.
- If multiple shards return documents with the same ID, then the first document is selected and the rest are discarded. This can happen if you have issues in cleanly distributing your documents over your shards.

Combining replication and sharding (Scale Deep)

Once you've scaled wide by either replicating indexes across multiple servers or sharding a single index, and then discover that you still have performance issues it's time to combine both approaches to provide a deep structure of Solr servers to meet your demands. This is conceptually quite simple, and getting it set up to test is fairly straight forward. The challenge typically is keeping all of the moving pieces up-to-date, and making sure that you are keeping your search indexes up-to-date. These operational challenges require a mature set of processes and sophisticated monitoring tools to ensure that all shards and slaves are update to date and are operational.

In order to tie the two approaches together, you continue to use sharding to spread out the load across multiple servers. Without sharding, it doesn't matter how large your pool of slave servers is because you need more CPU power than what just one slave server has to handle an individual query. Once you have sharded across the spectrum of shard servers, you treat each one as a Master Shard server, configured in the same way as we did in the previous replication section. This develops a tree of a master shard server with a pool of slave servers. Then, to issue a query, you have multiple small pools of one slave server per shard that you issue queries against. You can even have dedicated Solr, which don't have their own indexes, to be responsible for delegating out the queries to the individual shard servers and then aggregate the results before returning them to the end user.



Data updates are handled by updating the top Master Shard servers and then replicated down to the individual slaves, grouped together into small groups of distributed sharded servers.

Obviously, this is a fairly complex setup and requires a fairly sophisticated load balancer to frontend this whole collection, but it does allow Solr to handle extremely large data sets.

Where next for Solr scaling?

There has been a fair amount of discussion on Solr mailing lists about setting up distributed Solr on a robust foundation that adapts to changing environment. There has been some investigation regarding using Apache Hadoop, a platform for building reliable, distributing computing as a foundation for Solr that would provide a robust fault-tolerant filesystem. Another interesting sub project of Hadoop is ZooKeeper, which aims to be a service for centralizing the management required by distributed applications. There has been some development work on integrating ZooKeeper as the management interface for Solr. Keep an eye on the Hadoop homepage for more information about these efforts at `http://hadoop.apache.org/` and Zookeeper at `http://hadoop.apache.org/zookeeper/`.

Summary

Solr offers many knobs and levers for increasing performance. From turning the simpler knobs for enhancing the performance of a single server, to pulling the big levers of scaling wide through replication and sharding, performance and scalability with appropriate hardware are issues that can be solved fairly easily. Moreover, for those projects where truly massive search infrastructure is required, the ability to shard over multiple servers and then delegate to multiple slaves provides an almost linear scalability capacity.

Index

Symbols

$("#artist").autocomplete() function 242
*** fallback 46**
-Djetty.port=8984 290
-Dmaster=disabled 290
-Dslave=disabled 290
-Dsolr.data.dir=./solr/data8984 291
<dataSource/> element 77
<response /> element 93
<types/> tag 40
@throws SolrServerException 234
[FULL INTERFACE] link 89
val pseudo-field hack 117, 118

A

a_name field + a_ngram field, n-gramming costs 61
a_name field, n-gramming costs 61
a_spell, spellchecker 172
a_spellPhrase, spellchecker 172
abs(x), mathematical primitives 121
accuracy, spellchecker option 174
acts_as_solr, Ruby On Rails integrations
:fields array 256
about 255, 256
MyFaves, project setting up 255, 256
MyFaves relational database, popularity from Solr 256-258
MyFaves web site, completing 260-263
Solr indexes, building from relational database 258-260
allowDups 69
alphabetic range bucketing (A-C, D-F, and so on), faceting 148, 149
Amazon EC2
about 273
Solr, using on 274-276
Amazon Machine Instance. *See* **AMI**
AMI 274
analyzer chains
CharFilterFactory 49
index type 49
query type 49
tokenizer 50
types 49
analyzers
miscellaneous 62, 63
AND *:*need for 135
AND operator 100
AND operator, combining with OR operator 101
AND or && operator 101
Apache ant
about 13
URL 11
Apache Lucene. *See* **Lucene**
Apache Tomcat 199
appends 111
arr, XML element 92
artist_startDate field 33
artistAutoComplete 243
auto-complete. *See* **term-suggest**
Auto-warming 280
automatic phrase boosting
about 132, 133
configuring 133
phrase slop, configuring 134
AWStats 202

B

batchSize 78
bf parameter 117
Blacklight Online Public Access Catalog. *See* **Blacklight OPAC, Ruby On Rails integrations**
Blacklight OPAC, Ruby On Rails integrations
  about 263
  data, indexing 263-267
Boolean operators
  AND 100
  AND operator, combining with OR operator 101
  AND or && operator 101
  NOT 100
  NOT operator 101
  OR 100
  OR or || operator 101
bool element 92
boost functions
  boosting 137, 138
  r_event_date_earliest field 138
boosting 70, 107
boost queries
  boosting 134-137
bq parameter(s) 134
bucketFirstLetter 148
buildOnCommit 174
buildOnCommit, spellchecker option 174
buildOnOptimize, spellchecker option 174

C

caches
  tuning 281
CapitalizationFilterFactory filter 63
CCK 252
Chainsaw
  URL 204
characterEncoding, FileBasedSpellChecker option 175
CharFilterFactory 62
CI 128
classname 173
CM 197
CMS 250
Co-ordination Factor. *See* **coord**
collapse.facet, field collapsing 192
collapse.field, field collapsing 192
collapse.info.doc, field collapsing 193
collapse.maxdocs, field collapsing 193
collapse.threshold, field collapsing 193
collapse.type, field collapsing 192
combined index 32
CommonsHttpSolrServer 235
complex systems, tuning
  about 271
  CPU usage 272
  memory usage 272
  scale deep 273
  scale high 273
  scale wide 273
  system changes 272
components
  about 111, 159
  solrconfig.xml 159
compressed, field option 41
configuration files, Solr
  <requestHandler> tag 25
  solrconfig.xml file 25
  standard request handler 26
Configuration Management. *See* **CM**
ConsoleHandler 204
Content Construction Kit 252
Content Management System. *See* **CMS**
Continuous Integration. *See* **CI**
coord 112
copyField directive
  about 46
  uses 46
CoreDescriptor classes 231
core, managing 209, 210
count, Stats component 189
CPU usage 272
cron 289
CSV, sending to Solr
  about 72
  configuration options 73, 74
curl
  using, to interact with Solr 66, 68

D

data, indexing
stream.body parameter 67
stream.file parameter 67
stream.url parameter 67
through HTTP POST 67
ways 67
database
and Lucene search index, differences 9, 10
DataImportHandler. *See* **DIH**
dataSource attribute 78
date element 93
date facet, parameters
facet.date 151
facet.date.end 151
facet.date.gap 151
facet.date.hardend 151
facet.date.other 152
facet.date.start 151
dates, Faceting 146
debugQuery, diagnostic parameter
about 98
explainOther 98
defaults 111
defaultSearchField, schema.xml settings 47
defType, query parameter 95
defType parameter 128
deleteById() 232
deleteByQuery() 232
denormalizing
one to many associated data 36, 37
one to one associated data 36
deployment process, Solr 197, 198
df, query parameter 95
diagnostic query parameters
debugQuery 98
echoHandler 98
echoParams 98
indent 98
dictionary
about 169
building, from source 176, 177
DIH
about 74, 236
capabilities 74
dataSource attribute 78
development console 76, 77
documents, entities 78
entity 78
getting started 75
mb-dih-artists-jdbc.xml file 75, 76
query attribute 78
reference document, URL 74
Solr, registering with 75
solrconfig.xml 75
DIH, development console
DataSources, JdbcDataSource type 77, 78
DIH control form 77
documents, entities 79
fields 79
importing with 80
DIH, transformers
dateTimeFormat attributes 79
splitBy attributes 79
template attributes 79
DIH fields
column attribute 79
name attribute 79
directory structure, Solr
build 13
client 13
dist 13
example 14
example/etc 14
example/multicore 14
example/solr 14
example/webapps 14
lib 14
site 14
src 14
src/java 14
src/scripts 14
src/solrj 14
src/test 14
src/webapp 14
Disjunction-Max. *See* **dismax**
DisjunctionMaxQuery
about 130
boosts, configuring 131
queried fields, configuring 131
dismax 113

dismax handler. *See* **Dismax Solr request handler**
dismax query handler 131
dismax request handler 128
Dismax Solr request handler
about 128
automatic phrase boosting 132, 133
boost functions, boosting 137, 138
boost queries, boosting 134-137
debugQuery option used 129
default search 140, 141
DisjunctionMaxQuery 130
features, over standard handler 129
limited query syntax 131
min-should-match 138
mm query parameter 138
phrase slop, configuring 134
distanceMeasure, spellchecker option 174
distributed search 32
div(x,y), mathematical primitives 121
doc element 93
docText field data 233
document
deleting 70
documentCache 281
Domain Specific Language. *See* **DSL**
double element 92
DoubleMetaphone, phonetic encoding algorithms 58
DoubleMetaphoneFilterFactory analysis filter, options
inject 59
maxCodeLength 59
Drupal, options
Apache Solr Search integration module 251
Solr, hosted by Acquia 252
DSL 269
dynamic fields
* fallback 46
about 45

E

echoHandler, diagnostic parameter 98
echoParams 152
echoParams, diagnostic parameter 98
EdgeNGram analyzer 61
EdgeNGramFilterFactory 61
EdgeNGramTokenizerFactory 61
Elasticfox 276
Embedded-Solr 65
embedded Solr
legacy Lucene, upgrading from 237
using for rich clients 237
using in in-process streaming 236, 237
EmbeddedSolrServer class 224
encoder attribute 59
EnglishPorterFilter Factory, stemming 54
Entity tags 279
ETag 279
ETL 78
eval() function 238
existence (and non-existence) queries 107
explicit mapping 56
Extract Transform and Load. *See* **ETL**
extraParams entry 242

F

facet 146
facet.date 151, 286
examples 151
facet.date.end 151
facet.date.gap 151
facet.date.hardend 151
facet.date.other 152
facet.date.start 151
facet.field 147
facet.limit 147
facet.method 148
facet.mincount 147
facet.missing 148
facet.missing parameter 143
facet.offset 147
facet.prefix 148, 156
facet.query 286
facet.query parameter 152, 153
facet.sort 147
facet_counts 143
faceted navigation 7, 141, 145, 153
faceted search 149, 220, 221

faceting
about 141
alphabetic range bucketing (A-C, D-F, and so on) 148, 149
date facet parameters 151, 152
dates 146, 149, 150
example 142, 143
facet.field 147
facet.limit 147
facet.method 148
facet.mincount 147
facet.missing 148
facet.missing parameter 143
facet.offset 147
facet.prefix 148
facet.sort 147
facet_counts 143
facet prefixing (term suggest) 156-158
field, requisites 146
field values (text) 146
filters, excluding 153-155
Local Params 155
on arbitrary parameters 152, 153
queries 146
release types, exampleexample 142, 143
schema changes, MusicBrainz example 144, 145
text 147
types 146
faceting, dates
about 149
examples 150
Facet prefixing 156
Familiarity
URL 204
FastLRUCache 280
fetchSize 78
field, attributes
default (optional) 42
name 42
required (optional) 42
type 42
field, IndexBasedSpellChecker option 174
field collapsing, search components
about 191, 192
collapse.facet 192
collapse.field 192
collapse.info.count 193
collapse.info.doc 193
collapse.maxdocs 193
collapse.threshold 193
collapse.type 192
configuring 192, 193
SOLR-236 191
field definitons, schema.xml file
attributes 42
copyField, using 46
copyField directive, using 46
default (optional) 42
dynamic fields 45
name 42
required (optional) 42
schema.xml, settings 47
sorting 44
sorting, limitations 44, 45
type 42
field length. *See* **fieldNorm**
field list. *See* **fl**
fieldNorm 112
field options, schema.xml file
compresses 41
indexed 41
multiValued 41
omitNorms (advanced) 41
positionIncrementGap (advanced) 42
sortMissingFirst 41
sortMissingLast 41
stored 41
termVectors (advanced) 41
field qualifier 102, 103
field references, function queries 120
fieldType, spellchecker option 174
field types, schema.xml file
<fields/> tag 40
<types/> tag 40
class attribute 40
field values (text), Faceting 146
file, spellchecker 172
FileBasedSpellChecker options
characterEncoding 175
sourceLocation 175
FileHandler logging 204
filterCache 280
filter element 50

filtering 108, 109
filters, Faceting
excluding 153, 155
first-components 111
fl 220
fl, output related parameter 96
float element 92
fq, query parameter 95
function argument
limitations 120
function queries
val pseudo-field hack 117
about 117
bf parameter 117
Daydreaming search example 119
example 118
field references 120
function references 120
incorporating, to searches 117
t_trm_lookups 118
function query, tips 128
function references
mathematical primitives 121
function references, function queries 120

G

g, query parameter 95
g.op, query parameter 95
generic XML data structure
about 92
appends 111
arr, XML element 92
bool element 92
components 111
date element 93
defaults 111
double element 92
first-components 111
float element 92
int element 92
invariants 111
last-components 111
long element 92
lst, XML element 92
str element 92
Git
URL 11

H

Hadoop 225
HathiTrust 273
Heritrix
using, to download artist pages 226, 227
highlighted field list. *See* **hl.fl**
highlighting component, search components
about 161
configuring 163
example 161, 163
hl 164
hl.fl 164
hl.fragsize 164
hl.highlightMultiTerm 164
hl.mergeContiguous 165
hl.requireFieldMatch 164
hl.snippets 164
hl.usePhraseHighlighter 164
hl alternateField 165
hl formatter 165
hl fragmenter 165
hl maxAnalyzedChars 165
parameters 164
hl, highlighting component 164
hl.fl 161
hl.fl, highlighting component 164
hl.fragsize, highlighting component 164
hl.highlightMultiTerm, highlighting component 164
hl.increment, regex fragmenter 166
hl.mergeContiguous, highlighting component 165
hl.regex.maxAnalyzedChars, regex fragmenter 166
hl.regex.pattern, regex fragmenter 166
hl.regex.slop, regex fragmenter 166
hl.requireFieldMatch, highlighting component 164
hl.snippets, highlighting component 164
hl.usePhraseHighlighter, highlighting component 164
hl alternateField, highlighting component 165
hl formatter, highlighting component
about 165
hl.simple.pre and hl.simple.post 165

hl fragmenter, highlighting component 165
hl maxAlternateFieldLength, highlighting component 165
hl maxAnalyzedChars, highlighting component 165
home directory, Solr
bin 15
conf 15
conf/schema.xml 15
conf/solrconfig.xml 15
conf/xslt 15
data 15
lib 15
HTML, indexing in Solr 227
HTMLStripStandardTokenizerFactory 52
HTMLStripStandardTokenizerFactory tokenizer 227
HTMLStripWhitespaceTokenizerFactory 52
HTTP caching 277-279
HTTP server request access logs, logging
about 201, 202
log directory, creating 201
Tailing 202

I

IDF 33
idf 112
ID field 44
indent, diagnostic parameter 98
index 31
index-time
and query-time, boosting 113
versus query-time 57
index-time boosting 70
IndexBasedSpellChecker options
field 174
sourceLocation 174
thresholdTokenFrequency 175
index data
document access, controlling 221
securing 220
indexed, field option 41
indexed, schema design 282
indexes
sharding 295
indexing strategies
about 283
factors, committing 285
factors, optimizing 285
unique document checking, disabling 285
Index Searchers 280
Information Retrieval. S*ee* IR
int element 92
InternetArchive 226
invariants 111
Inverse Document Frequency. S*ee* IDF
inverse reciprocals 125
IR 8
ISOLatin1AccentFilterFactory filter 62
issue tracker, Solr 27

J

J2SE
with JConsole 212
JARmageddon 205
jarowinkler, spellchecker 172
java.util.logging package 203
Java class names
abbreviated 40
org.apache.solr.schema.BoolField 40
Java Development Kit (JDK)
URL 11
JavaDoc tags 234
Java Management Extensions. S*ee* JMX
Java Naming and Directory Interface. S*ee* JNDI
Java replication
versus script 289
JavaScript Object Notation. S*ee* JSON
Java Server Pages. S*ee* JSPs
JConsole GUI
about 212
URL 212
JDK [1.4] logging 203
JDK logging 203
Jetty
startup integration 205
web.xml, customizing 218
jetty.xml 201
JIRB tool 215
JMX
about 212
access, controlling 220

information extracting, JRuby used 215
Solr, starting with 212-215
Jmx4r 217
JMX Console 212
JNDI 16, 200
JNDI name 200
jQuery 240
jQuery Autocomplete widget 241, 242
JRuby
using, to extract JMS information 215
JRuby Interactive Browser tool. S*ee* JIRB tool
JSON 238
JSONP 242
JSON with Padding. S*ee* JSONP
JSPs 17
JUL 203
JVM
configuration 277

K

KeepWordFilterFactory filter 62
KeywordTokenizerFactory 52
KStem, stemming 55

L

last-components 111
LengthFilterFactory 145
LengthFilterFactory filter 62
LetterTokenizerFactory 52
limited query syntax 131
disabling 132
linear(x,m,c), miscellaneous math 122
Local Params 155
LocalSolr component 194
log(x), mathematical primitives 121
Log4j
configuring, URL 205
logging to 204
Log4j JAR file
URL 204
logarithms 123, 124
Logback
URL 204
logging
about 201
HTTP server request access logs 201, 202
levels. managing at runtime 205, 206
Solr application logging 203
types 201
logging.properties file 204
long element 92
LowerCaseFilterFactory filter 62
LRUCache 280
lst, XML element 92
Lucene
about 8
DisjunctionMaxQuery 130
features 8
scoring 112
Lucene's query syntax
URL 44
LUCENE-1435 45
Lucene search index
and database, differences 9, 10
Lucene syntax
query expression 100
query syntax 99
sub-expressions 101

M

mailing lists, Solr
URL 26
Managed Bean. S*ee* MBeans
mandatory clause, expression query 100
map() function 243
map(x,min,max,target), miscellaneous math 121
master server
indexing into 292
mathematical primitives, function references
abs(x) 121
div(x,y) 121
log(x) 121
pow(x,y) 121
product(x,y,z,...) 121
sqrt(x) 121
sum(x,y,z, ...) 121
Maven 228
max(x,c), miscellaneous math 121
max, Stats component 189
maxGramSize 60

maxScore 93
maxWarmingSearchers 284
mb-dih-artists-jdbc.xml file 75, 76
mb_attributes.txt
content 145
MBeans 212
mean, Stats component 189
member_id field 36
memory usage 272
Metaphone, phonetic encoding algorithms 58
min, Stats component 189
min-should-match
about 138
basic rules 139
multiple rules 139
rules 139
rules, choosing 140
minGramSize 60
miscellaneous math, function references
linear(x,m,c) 122
map(x,min,max,target) 121
max(x,c) 121
recip(x,m,a,c) 122
scale(x,minTarget,maxTarget) 121
missing, Stats component 189
MLT, search components
as dedicated request handler 182
as request handler, with external input document 183
as Solr component 182
configuration parameters 183
mlt 183
mlt.boost 186
mlt.count 183
mlt.fl 185
mlt.maxntp 186
mlt.maxqt 186
mlt.maxwl 185
mlt.mindf 185
mlt.mintf 185
mlt.minwl 185
mlt.qf 185
parameters 185, 186
parameters, specific to MLT request handler 184
results, example 186, 188
specific parameters 183
using, ways 182
mlt.boost 186
mlt.fl 185
mlt.maxntp 186
mlt.maxqt 186
mlt.maxwl 185
mlt.mindf 185
mlt.mintf 185
mlt.minwl 185
mlt.qf 185
mm query parameter 138
mm specification formats
as examples 139
more-like-this search component. S*ee* MLT, search components
more like this plugin 9
multi-word synonyms 56
multicore
need for 210, 211
multiple indices 32
multiple Solr servers
documents, assigning to shards 296
indexes, sharding 295
master server, indexing into 292
replication, configuring 291
script versus Java replication 289
searches, distributing 291
search queries, distributing across slaves 293, 294
shards, searching across 297, 298
slaves, configuring 292, 293
starting 290, 291
multiValued, field option 41
multiValued field 221
MusicBrainz.org 30, 31

N

n-gramming costs
Edge n-gramming costs 62
tokenizer based n-gramming costs 62
N-gramming costs, substring indexing
a_name field 61
a_name field + a_ngram field 61
minGramSize 62
name 173
name attribute 143

name field 33
newSearch query 284
NOT operator 100, 101
numFound 93
Nutch 225
Nutch + Web Archive eXtensions. *See* **NutchWAX**
NutchWAX 225

O

OLTP 78
omitNorms (advanced), field option 41
omitNorms, schema design 282
omitTermFreqAndPositions, schema design 282
Online Transaction Processing systems. *See* **OLTP**
optional clause, expression query 100
ord() function 120, 122
ord(fieldReference) 122
ord/rord 122
ord and rord, function references
  ord(fieldReference) 122
  rord(fieldReference) 122
OR operator 100
OR or || operator 101
output related parameters, query parameters
  fl 96
  sort 96
  version 98
  wt 97
outputUnigrams controls 288

P

parse
  parameter 243
parse() function 244
partial indexing. *See* **substring indexing**
PatternReplaceFilterFactory filter 63
PatternTokenizerFactory 53
pf, tips 134
pf parameter 133
phoneme 58
phonetic encoding algorithms
  DoubleMetaphone 58
  encoder attribute 59
  Metaphone 58
  RefinedSoundex 58
  Soundex 58
PhoneticFilterFactory filter 59
phonetic sounds-like
  about 58
  phonetic encoding algorithms 58
phrase queries 103
phrase search performance
  improving 287
  shingling, solution 287, 288
phrase slop
  configuring 134
Plain Old Java Objects. *See* **POJOs**
POJOs
  indexing 234
PorterStemFilterFactory, stemming 54
positionIncrementGap (advanced), field option 42
pow(x,y), mathematical primitives 121
product(x,y,z, ...), mathematical primitives 121
prohibited clause, expression query 100
PRONOM Unique Identifier. *See* **PUID**
public searches
  securing 219, 220
PUID 31

Q

q parameter
  processing 175
qt, miscellaneous parameter 95
QTime 93
queries, Faceting 146
query-time
  and index-time, boosting 113
  versus index-time 57
query-time boosting 70
query attribute 78
query converter 175
query elevation, search components
  about 166
  config-file 167, 168
  configuration parameters 167
  configuring 167
  elevateArtists.xml 168

forceElevation 168
queryFieldType 168
query expression, clauses
mandatory clause 100
optional clause 100
prohobited clause 100
query parameters
about 95
defType 95
df 95
diagnostic 98
fq 95
output related parameters 96
q 95
q.op 95
qt 95
result paging 96
rows 96
start 96
query parser plugin 128
QueryResponse object 235
queryResultCache 280
query spell checker
indexed content based 8, 9
query syntax
about 99
boosting 107
documents, matching 99
existence (and non-existence) queries 107
field qualifier 102, 103
fuzzy queries 105
phrase queries 103
query expression, clauses 100
special characters 108
sub-expressions 101
term proximity 103
wildcard queries 103, 104

R

r_a_name 42
r_attributes 144
r_event_date_earliest field 138
r_name_facetLetter 148
r_official 144
r_type 144
range queries
[and] brackets 106
{ and } brackets 106
about 105, 106
date math 106, 107
readOnly 77
recip(x,m,a,c), miscellaneous math 122
reciprocals and rord, with dates 126, 127
RecordItem 234
RefinedSoundex, phonetic encoding algorithms 58
regex fragmenter, options
hl.increment 166
hl.regex.pattern 166
hl.regex.slop 166
hl regex.maxAnalyzedChars 166
release's artist's name. S*ee* r_a_name
remote streaming
about 68, 221
disabling 69
enabling 69
remote streaming feature 224
RemoveDuplicatesTokenFilterFactory filter 62
renderResult() method 247
replication
and sharding, combining 298-300
configuring 291
requestHandler 207
request handler
about 110
configuration, creating 110
configuring 110
result() function 243, 244
right field type/analysis, using 109
rOfficial 144
rord() 122
rord(fieldReference) 122
rows parameter 96, 242
rsolr
versus solr-ruby 269
Ruby On Rails integrations
acts_as_solr 254-259
acts_as_solr plugin 253
Blacklight OPAC 263
Convention over Configuration 253
display, customizing 267
fields display, customizing 268, 269
solr-ruby versus rsolr 269
solr_data 257

S

scale() function
example 123
inverse reciprocals, using 124, 125
logarithms, using 123, 124
reciprocals and rord with dates, using 126, 127
scale(x,minTarget,maxTarget), miscellaneous math 121
scale deep 298
scale high 276
scale wide 289
schema, Solr
<copyField> tag 25
<fields> tag 25
<types> tag 25
primary key 25
text, field name 25
schema.xml, settings
defaultSearchField 47
solrconfig.xml 47
solrQueryParser 47
uniqueKey 47
schema.xml file
<fields/> tag 40
<types/> tag 40
field definitions 42, 43
field options 40
field types 40
sample 45
schema design
about 34
compressed field option 282
data, denormalizing 36
entities returned from search, determining 35
inclusion of fields used in search results, omitting 38, 39
indexed 282
omitNorms 282
omitTermFreqAndPositions 282
one to many associated data, denormalizing 36, 37
one to one associated data, denormalizing 36
Solr powered search, determining 35
stored 282
score boosting. S*ee* **boosting**
scoring
about 112
co-ordination factor (coord) 112
factors 112
field length (fieldNorm) 112
Inverse Document Frequency (idf) 112
query-time and index-time, boosting 113
term frequency (tf) 112
troubleshooting 113, 114
script
versus Java replication 289
search, distributing across slaves
about 291
master server, indexing into 292
slaves, configuring 292, 293
search components
about 161
field collapsing 191, 192
highlighting component 161
MLT (more-like-this) 182
query elevation 166
spellcheck 169
Stats component 189
terms component 194
termVector component 194
search engine 161, 223, 237, 266, 272
searcher.num_docs attribute 216
SearchHandler
per search interface 207
search handler 128
searching 89, 90
server access
limiting 217, 219
Servlet container
and Solr, differences 199
installing in 199
solr.home property, defining 199
sharding
and replication, combining 298-300
documents, assigning 296
indexes 295, 296
searching across 297, 298
ShingleFilterFactory 288
shingling 133, 127, 287

Simple Java interface. S*ee* **SolrJ**
Simple Logging Facade for Java package. S*ee* **SLF4J package**
single combined index
issues 34
schema.xml snippet, sample 32
using, issues 33
single Solr server
optimizing 276
single Solr server, optimizing
faceting performance, enhancing 286
HTTP caching 277-279
indexing strategies 283, 284
JVM configuration 277
phrase search performance, improving 287
schema design considerations 282
Solr caching 280, 281
term vectors, using 286, 287
tuning caches 281
slaves
configuring 292
search queries, distributing across slaves 293, 294
SLF4j 20
SLF4J package 203
SnowballPorterFilterFactory, stemming 54
Solr
about 7, 10
and Servlet container, differences 199
building 13
communicating with 65
complex systems, tuning 271, 272
configuration files 25, 26
cores, managing 209, 210
CSV, sending to 72
deploying 17
deployment process 197, 198
directory structure 13
disjunction-max query handler 9
Faceting 141
features 8, 9
filtering 108, 109
function query, incorporating to searches 117
generic XML data structure 92
home directory 15
interacting with, curl used 66, 68
issue tracker 27
local file accessing, example 68
logging 201
mailing list 26
official site, URL 11
powered artists building, autocomplete widget with jQuery used 240, 241, 242
powered artists building, autocomplete widget with JSONP used 243
prerequisites 11
query parameters 95
query syntax 99
remote streaming 68, 69
request handlers 110
resources 26
running 17-19
sample data, loading 20, 21
schema 25
search request handler 128
securing 217
simple query, running 22-24
solr.solr.home, searching for 16
sorting 109
spell check plugin 9
starting 15, 16
starting, with JMX 212-215
statistics page 24
system changes 272
testing 13
tools 58
XML, sending to 69, 70
XML response format 93
Solr's DIH DataImportHandler contrib add-on 66
Solr's Wiki 26
Solr, accessing from PHP applications
about 247, 248
Drupal, options 250
solr-php-client 248-250
Solr, communicating with
convenient client API 65
data formats 66
data streamed remotely 66
Direct HTTP 65
Solr's filesystem 66
Solr, data formats
rich documents 66

Solr-binary 66
Solr-XML 66
Solr, examples
structure 223
summary 224
Solr, filters
CapitalizationFilterFactory 63
CharFilterFactory 62
ISOLatin1AccentFilterFactory 62
KeepWordFilterFactory 62
LengthFilterFactory 62
LowerCaseFilterFactory 62
PatternReplaceFilterFactory 63
RemoveDuplicatesTokenFilterFactory 62
StandardFilterFactory 62
write your own 63
Solr, integrating
JavaScript used 238, 239
Solr, prerequisites
Apache ant 11
Java Development Kit (JDK) 11
Subversion or Git 11
Solr, securing
document access, controlling 221
index data, securing 220
JMX access, controlling 220
server access, limiting 217, 219, 220
SOLR-236 191
solr-balancer 294
Solr-binary 66
solr-php-client
a_member_name array 249
about 248, 249, 250
Apache_Solr_Service, configuration 249
solr-ruby
versus rsolr 269
Solr-XML 66
solr.body feature 68
solr.home property
defining 199
JNDI (Java Naming and Directory Interface) 200
solr.war file 200
solr.setParser(new XMLResponseParser()) 235
solr.solr.home
searching for 16
solr.TextField 48
Solr 1.3 11
Solr 1.4 11
Solr admin
Assistance area 20
example 19
Make a Query text box 20
navigation menu 19
Solr application logging, logging 203
Jetty, startup integration 205
Log4j, logging to 204
logging output, configuring 203
log levels, managing at runtime 205, 206
solrbook-packtpub 273
Solr caching
autowarmCount 281
class 281
configuring 281
documentCache 281
filterCache 280
queryResultCache 280
size 281
Solr cell
binary content, extracting 81, 82
documents, indexing with 81
karaoke lyrics, extracting 83-85
richer documents, indexing 85-87
Solr, configuring 83
Solr cores
cores, managing 209, 210
multicore, need for 210, 211
solr.xml, configuring 208, 209
solrconfig.xml
<requestHandler /> elements 159
about 75
solrconfig.xml, schema.xml settings 47
Solr DIH Wiki page
URL 79
SolrDocumentList object 235
SolrDocument object 235
Solr home 16
SolrIndexSearch Mbean 214
SolrJ
about 65, 224
client API 230-233
CommonsHttpSolrServer 224
embedded Solr, need for 235, 236

EmbeddedSolrServer class 224
Heritrix using, to download artist pages 226, 227
HTML, indexing 227-230
HTMLStripStandardTokenizerFactory tokenizer 227
POJOs, indexing 234, 235
stream.file parameter 224
Solr JIRA
URL 12
SolrJS
about 245, 246
addWidget() method 247
project homepage, URL 245
SolrJS Manager object 247
URL 220
Solrmarc 236
SolrQuery object 235
solrQueryParser, schema.xml settings 47
Solr resources
about 26
issue tracker 27
mailing lists 26
Solr's Wiki 26
Solr search components
LocalSolr component 194
terms component 194
termVector component 194
sort, output related parameter 97
sorting
about 44, 109
limitations 44
string type 45
title_sort type 45
sortMissingFirst, field option 41
sortMissingLast, field option 41
Soundex, phonetic encoding algorithms 58
sourceLocation, FileBasedSpellChecker option 175
sourceLocation, IndexBasedSpellChecker option 174
spellcheck 177
spellcheck, search components
a_spell, spellchecker 172
a_spellPhrase, spellchecker 172
about 169
alternative approach 180, 182
classname 173
dictionary, building from source 176
file, spellchecker 172
FileBasedSpellChecker options 175
IndexBasedSpellChecker options 174
indexed content 169
jarowinkler, spellchecker 172
mispelled query, example 178, 180
name 173
q parameter, processing 175
requests, issuing 177, 178
schema configuration 169-171
solrconfig.xml, configuration in 171, 172
Solr configuring, ways 169
spellcheck.q parameter, processing 176
spellchecker, index and file based 173
spellcheckers (dictionaries), configuring 173
spellcheckIndexDir 173
text file of words 169
spellcheck.collate 178
spellcheck.count 177
spellcheck.dictionary 177
spellcheck.extendedResults 178
spellcheck.onlyMorePopular 178
spellcheck.q 177
spellcheck.q parameter
processing 176
spellchecker, index and file based
accuracy 174
buildOnCommit 174
buildOnOptimize 174
classname 173
distanceMeasure 174
fieldType 174
name 173
spellcheckIndexDir 173
spellcheckIndexDir 173
spell check plugin 9
Splunk 205
sqrt(x), mathematical primitives 121
Squid
URL 279
standard component list 160
StandardFilterFactory filter 62
StandardTokenizerFactory 52
start 93

startEmbeddedSolr() 234
start parameter 96
stats, Stats component 189
stats.facet, Stats component 190
stats.field, Stats component 189
Stats component, search components
about 189
configuring 189
count 189
max 189
mean 189
min 189
missing 189
statistics, for track durations 190
stats 189
stats.facet 190
stats.field 189
stddev 189
sum 189
sumOfSquares 189
status 93
stddev, Stats component 189
stemming
about 54
EnglishPorterFilterFactory 54
implementations 54
KStem 55
PorterStemFilterFactory 54
SnowballPorterFilterFactory 54
StopFilterFactory 186
used, for stop words filtering 57
stop words
filtering, StopFilterFactory used 57
stored, field option 41
stored, schema design 282
stream.body parameter 67
stream.file parameter 67, 224
stream.url parameter 67
StreamingUpdateSolrServer 284
str element 92
string type 45
sub-expressions
about 101
prohibited clause, limitations 102
substring indexing
about 60
analyzer configuration, n-grams used 60
EdgeNGramFilterFactory 61
EdgeNGramTokenizerFactory 61
n-gramming costs 61
NGramFilterFactory, configuring with minGramSize of 2 60
NGramFilterFactory, configuring with minGramSize of 5 60
Subversion
URL 11
sum(x,y,z, ...), mathematical primitives 121
sum, Stats component 189
sumOfSquares, Stats component 189
synonyms
=> 56
about 55
ignoreCase, setting true 56
index-time versus query-time 57
WordNet, thesarus 55

T

t_duration 152
t_shingle 288
t_trm_lookups 118
Tailing 202
term-suggest 141, 156
term frequency. S*ee* tf
term proximity 103
terms component 194
termVector component 194
termVectors 186
term vectors 286, 287
termVectors (advanced), field option 41
text analysis
about 47
experimenting with 50, 51
highlight matches 51
index box 51
multi-word synonyms 56
n-gram 60
n-gramming costs 61, 62
partial indexing 60
phonetic sounds-like 58
query box 51
stemming 54, 55
stop words 58
substring indexing 60
synonyms 55

term text 51
text field type 50
text field type definition, configuration 48
text field type definition, configuring 49
tokenizer 52
verbose output 51
WordDelimiter analyzer 53
WordDelimiterFilterFactory 53
WorkDelimiterFilterFactory 54
text field type 50
tf 112
threaded_test.rb script 283, 284
thresholdTokenFrequency, IndexBasedSpellChecker option 175
title_sort type 45
tokenizer
about 50
HTMLStripStandardTokenizerFactory 52
HTMLStripWhitespaceTokenizerFactory 52
KeywordTokenizerFactory 52
LetterTokenizerFactory 52
PatternTokenizerFactory 53
StandardTokenizerFactory 52
WhitespaceTokenizerFactory 52
Tomcat 199
TPS 272
track_PUID field 33
Transactions Per Second. S*ee* TPS

U

uniqueKey, schema.xml settings 47
uniqueKey field 232, 233

V

version, output related parameter 98
Vigilog
URL 204

W

WAR 199
web.xml
customizing, in Jetty 218
Web application archive. S*ee* WAR
WebTrends 202
WhitespaceTokenizerFactory 52
wildcard queries
about 103, 104
fuzzy queries 105
WordDelimeterFilterFactory 51
WordDelimeterFilterFactory, tokenizer action 50
WordDelimiter analyzer
splitting, ways 53, 54
tokenizing, ways 53, 54
WordDelimiterFilterFactory 53
WordNet thesarus 55
write your own filter 63
wt, output related parameter 97

X

XML, sending to Solr
about 69, 70
changes, committing 71
commit and optimize 71
documents, deleting 70
rollback command 71
uncommitted changes, withdrawing 71
XML response format
<lst name=”response header”> 93
<result name="response" numFound="1002272" start="0" maxScore="1.0"> 93
about 93
maxScore 93
numFound 93
QTime 93
start 93
status 93
URL, parsing 94

Y

y, argument 120

Z

zip format 292

PACKT PUBLISHING

Thank you for buying

Solr 1.4 Enterprise Search Server

Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing Solr 1.4 Enterprise Search Server, Packt will have given some of the money received to the Apache Solr project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

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.

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`.

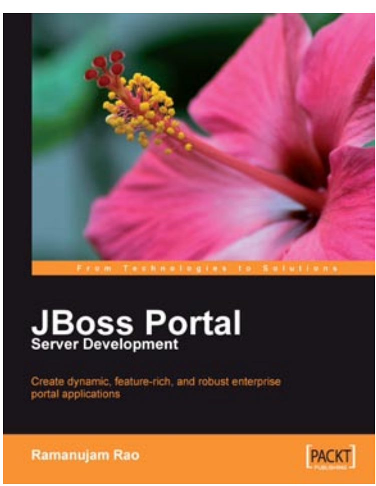
JasperReports for Java Developers

ISBN: 1-904811-90-6 Paperback: 344 pages

Create, Design, Format, and Export Reports with the world's most popular Java reporting library

1. Get started with JasperReports, and develop the skills to get the most from it
2. Create, design, format, and export reports
3. Generate report data from a wide range of datasources
4. Integrate Jasper Reports with Spring, Hibernate, Java Server Faces, or Struts

---



JBoss Portal Server Development

ISBN: 978-1-847194-10-7 Paperback: 276 pages

Create dynamic, feature-rich, and robust enterprise portal applications

1. Complete guide with examples for building enterprise portal applications using the free, open-source standards-based JBoss portal server
2. Quickly build portal applications such as B2B web sites or corporate intranets
3. Practical approach to understanding concepts such as personalization, single sign-on, integration with web technologies, and content management

Please check **www.PacktPub.com** for information on our titles