



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Elasticsearch Server

Second Edition

A practical guide to building fast, scalable, and flexible search solutions with clear and easy-to-understand examples

Rafał Kuć
Marek Rogoziński

[PACKT] open source*
community experience distilled
PUBLISHING

Elasticsearch Server

Second Edition

A practical guide to building fast, scalable, and flexible search solutions with clear and easy-to-understand examples

Rafał Kuć

Marek Rogoziński



BIRMINGHAM - MUMBAI

Elasticsearch Server

Second Edition

Copyright © 2014 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: February 2013

Second edition: April 2014

Production Reference: 1170414

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

ISBN 978-1-78398-052-9

www.packtpub.com

Cover Image by Kannan PM Palanisamy (kannan.pmp@gmail.com)

Credits

Authors

Rafał Kuć
Marek Rogoziński

Reviewers

John Boere
Jetro Coenradie
Clive Holloway
Surendra Mohan
Alberto Paro
Lukáš Vlček

Commissioning Editor

Anthony Alburqueque

Acquisition Editor

Neha Nagwekar

Content Development Editor

Shaon Basu

Technical Editors

Indrajit Das
Menza Mathew
Shali Sasidharan

Copy Editors

Dipti Kapadia
Insiya Morbiwala
Aditya Nair
Adithi Shetty

Project Coordinator

Amey Sawant

Proofreaders

Simran Bhogal
Maria Gould
Bernadette Watkins

Indexer

Priya Subramani

Graphics

Abhinash Sahu

Production Coordinator

Sushma Redkar

Cover Work

Sushma Redkar

About the Author

Rafał Kuć is a born team leader and software developer. He currently works as a consultant and a software engineer at Sematext Group, Inc., where he concentrates on open source technologies such as Apache Lucene and Solr, Elasticsearch, and Hadoop stack. He has more than 12 years of experience in various branches of software, from banking software to e-commerce products. He focuses mainly on Java but is open to every tool and programming language that will make the achievement of his goal easier and faster. Rafał is also one of the founders of the [solr.pl](#) site, where he tries to share his knowledge and help people with the problems they face with Solr and Lucene. Also, he has been a speaker at various conferences around the world, such as Lucene Eurocon, Berlin Buzzwords, ApacheCon, and Lucene Revolution.

Rafał began his journey with Lucene in 2002, and it wasn't love at first sight. When he came back to Lucene in late 2003, he revised his thoughts about the framework and saw the potential in search technologies. Then, Solr came along and this was it. He started working with Elasticsearch in the middle of 2010. Currently, Lucene, Solr, Elasticsearch, and information retrieval are his main points of interest.

Rafał is also the author of *Apache Solr 3.1 Cookbook*, and the update to it, *Apache Solr 4 Cookbook*. Also, he is the author of the previous edition of this book and *Mastering ElasticSearch*. All these books have been published by Packt Publishing.

Acknowledgments

The book you are holding in your hands is an update to *ElasticSearch Server*, published at the beginning of 2013. Since that time, Elasticsearch has changed a lot; there are numerous improvements and massive additions in terms of functionalities, both when it comes to cluster handling and searching. After completing *Mastering ElasticSearch*, which covered Version 0.90 of this great search server, we decided that Version 1.0 would be a perfect time to release the updated version of our first book about Elasticsearch. Again, just like with the original book, we were not able to cover all the topics in detail. We had to choose what to describe in detail, what to mention, and what to omit in order to have a book not more than 1,000 pages long. Nevertheless, I hope that by reading this book, you'll easily learn about Elasticsearch and the underlying Apache Lucene, and that you will get the desired knowledge easily and quickly.

I would like to thank my family for the support and patience during all those days and evenings when I was sitting in front of a screen instead of being with them.

I would also like to thank all the people I'm working with at Sematext, especially Otis, who took out his time and convinced me that Sematext is the right company for me.

Finally, I would like to thank all the people involved in creating, developing, and maintaining Elasticsearch and Lucene projects for their work and passion. Without them, this book wouldn't have been written and open source search would be less powerful.

Once again, thank you all!

About the Author

Marek Rogoziński is a software architect and consultant with more than 10 years of experience. He has specialized in solutions based on open source search engines such as Solr and Elasticsearch, and also the software stack for Big Data analytics including Hadoop, HBase, and Twitter Storm.

He is also the cofounder of the [solr.pl](#) site, which publishes information and tutorials about Solr and the Lucene library. He is also the co-author of some books published by Packt Publishing.

Currently, he holds the position of the Chief Technology Officer in a new company, designing architecture for a set of products that collect, process, and analyze large streams of input data.

Acknowledgments

This is our third book on Elasticsearch and the second edition of the first book, which was published a little over a year ago. This is quite a short period but this is also the year when Elasticsearch changed. Not more than a year ago, we used Version 0.20; now, Version 1.0.1 has been released. This is not only a number. Elasticsearch is now a well-known, widely used piece of software with built-in commercial support and ecosystem—just look at Logstash, Kibana, or any additional plugins. The functionality of this search server is also constantly growing. There are some new features such as the aggregation framework, which opens new use cases—this is where Elasticsearch shines. This development caused the previous book to get outdated quickly. It was also a great challenge to keep up with these changes. The differences between the beta release candidates and the final version caused us to introduce changes several times during the writing.

Now, it is time to say thank you.

Thanks to all the people involved in creating Elasticsearch, Lucene, and all of the libraries and modules published around these projects or used by these projects.

I would also like to thank the team working on this book. First of all, a thank you to the people who worked on the extermination of all my errors, typos, and ambiguities. Many thanks to all the people who send us remarks or write constructive reviews. I was surprised and encouraged by the fact that someone found our work useful.

Last but not least, thanks to all my friends who withstood me and understood my constant lack of time.

About the Reviewers

John Boere is an engineer with 22 years of experience in geospatial database design and development and 13 years of web development experience. He is the founder of two successful startups and has consulted at many others. He is the founder and CEO of Cliffhanger Solutions Inc., a company that offers a geospatial search engine for the companies that need mapping solutions.

John lives in Arizona with his family and enjoys the outdoors – hiking and biking. He can also solve a Rubik's cube.

Jettro Coenradie likes to try out new stuff. That is why he got his motorcycle driver's license recently. On a motorbike, you tend to explore different routes to get the best experience out of your bike and have fun while doing the things you need to do, such as going from A to B. In the past 15 years, while exploring new technologies, he has tried out new routes to find better and more interesting ways to accomplish his goal. Jettro rides an all-terrain bike; he does not like riding on the same ground over and over again. The same is true for his technical interests; he knows about backend (Elasticsearch, MongoDB, Axon Framework, Spring Data, and Spring Integration), as well as frontend (AngularJS, Sass, and Less), and mobile development (iOS and Sencha Touch).

Clive Holloway is a web application developer based in New York City. Over the past 18 years, he has worked on a variety of backend and frontend projects, focusing mainly on Perl and JavaScript.

He lives with his partner, Christine, and his cat, Blueberry (who would have been called Blackberry except for the intervention of his daughter, Abbey, after she pointed out that they could not name a cat after a phone).

In his spare time, he is involved as a part of Thisoneisonus, an international collective of music fans who work together to produce fan-created live show recordings. You can learn more about him at <http://toiou.org>.

Surendra Mohan, who has served a few top-notch software organizations in varied roles, is currently a freelance software consultant. He has been working on various cutting-edge technologies such as Drupal, Moodle, Apache Solr, and Elasticsearch for more than 9 years. He also delivers technical talks at various community events such as Drupal Meetups and Drupal Camps. To know more about him, his write-ups, technical blogs, and many more, log on to <http://www.surendramohan.info/>.

He has also authored the titles, *Administrating Solr* and *Apache Solr High Performance*, published by Packt Publishing, and there are many more in the pipeline to be published soon. He also contributes technical articles to a number of portals, for instance, sitepoint.com.

Additionally, he has reviewed other technical books, such as *Drupal 7 Multi Sites Configuration* and *Drupal Search Engine Optimization*, both by Packt Publishing. He has also reviewed titles on Drupal commerce, Elasticsearch, Drupal-related video tutorials, a title on OpsView, and many more.

I would like to thank my family and friends who supported and encouraged me to complete this book on time with good quality.

Alberto Paro is an engineer, project manager, and software developer. He currently works as a chief technology officer at The Net Planet Europe and as a freelance consultant on software engineering on Big Data and NoSQL Solutions. He loves studying the emerging solutions and applications mainly related to Big Data processing, NoSQL, natural language processing, and neural networks. He started programming in BASIC on a Sinclair Spectrum when he was 8 years old, and in his life, he has gained a lot of experience by using different operative systems, applications, and by doing programming.

In 2000, he graduated from a degree in Computer Science Engineering from Politecnico di Milano with a thesis on designing multiuser and multidevice web applications. He worked as a professor's helper at the university for about one year. Then, having come in contact with The Net Planet company and loving their innovative ideas, he started working on knowledge management solutions and advanced data-mining products.

In his spare time, when he is not playing with his children, he likes working on open source projects. When he was in high school, he started contributing to projects related to the Gnome environment (gtkmm). One of his preferred programming languages was Python, and he wrote one of the first NoSQL backend for Django MongoDB (django-mongodb-engine). In 2010, he started using Elasticsearch to provide search capabilities for some Django e-commerce sites and developed PyES (a pythonic client for Elasticsearch) and the initial part of Elasticsearch MongoDB River. Now, he mainly works on Scala, using the Typesafe Stack and Apache Spark project.

He is the author of *ElasticSearch Cookbook*, Packt Publishing, published in December 2013.

I would like to thank my wife and children for their support.

Lukáš Vlček is a professional open source fan. He has been working with Elasticsearch nearly from the day it was released and enjoys it till today. Currently, Lukáš works for Red Hat, where he uses Elasticsearch hand-in-hand with various JBoss Java technologies on a daily basis. He has been speaking on Elasticsearch and his work at several conferences around Europe. He is also heavy on client-side JavaScript and building frontends for full-text search services.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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

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



<http://PacktLib.PacktPub.com>

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface	1
Chapter 1: Getting Started with the Elasticsearch Cluster	7
Full-text searching	8
The Lucene glossary and architecture	8
Input data analysis	10
Indexing and querying	10
Scoring and query relevance	11
The basics of Elasticsearch	12
Key concepts of data architecture	12
Index	12
Document	12
Document type	13
Mapping	13
Key concepts of Elasticsearch	14
Node and cluster	14
Shard	14
Replica	14
Gateway	15
Indexing and searching	15
Installing and configuring your cluster	17
Installing Java	17
Installing Elasticsearch	17
Installing Elasticsearch from binary packages on Linux	18
Installing Elasticsearch using the RPM package	18
Installing Elasticsearch using the DEB package	18
The directory layout	18
Configuring Elasticsearch	19
Running Elasticsearch	20
Shutting down Elasticsearch	22
Running Elasticsearch as a system service	23

Table of Contents

Elasticsearch as a system service on Linux	23
Elasticsearch as a system service on Windows	24
Manipulating data with the REST API	24
Understanding the Elasticsearch RESTful API	25
Storing data in Elasticsearch	25
Creating a new document	25
Automatic identifier creation	27
Retrieving documents	27
Updating documents	28
Deleting documents	30
Versioning	30
An example of versioning	31
Using the version provided by an external system	31
Searching with the URI request query	32
Sample data	32
The URI request	33
The Elasticsearch query response	33
Query analysis	35
URI query string parameters	37
The Lucene query syntax	41
Summary	42
Chapter 2: Indexing Your Data	43
Elasticsearch indexing	43
Shards and replicas	44
Creating indices	45
Altering automatic index creation	46
Settings for a newly created index	46
Mappings configuration	47
Type determining mechanism	47
Disabling field type guessing	49
Index structure mapping	50
Type definition	51
Fields	52
Core types	52
Multifields	57
The IP address type	57
The token_count type	58
Using analyzers	58
Different similarity models	63
Setting per-field similarity	64
Available similarity models	65
The postings format	66
Configuring the postings format	67
Doc values	68

Table of Contents

Configuring the doc values	68
Doc values formats	69
Batch indexing to speed up your indexing process	69
Preparing data for bulk indexing	69
Indexing the data	70
Even quicker bulk requests	72
Extending your index structure with additional internal information	73
Identifier fields	73
The _type field	74
The _all field	75
The _source field	76
Exclusion and inclusion	76
The _index field	77
The _size field	77
The _timestamp field	78
The _ttl field	79
Introduction to segment merging	80
Segment merging	81
The need for segment merging	81
The merge policy	81
The merge scheduler	82
The merge factor	82
Throttling	83
Introduction to routing	83
Default indexing	84
Default searching	85
Routing	86
The routing parameters	88
Routing fields	89
Summary	90
Chapter 3: Searching Your Data	91
Querying Elasticsearch	91
The example data	92
A simple query	94
Paging and result size	95
Returning the version value	96
Limiting the score	97
Choosing the fields that we want to return	98
The partial fields	100
Using the script fields	100
Passing parameters to the script fields	102

Table of Contents

Understanding the querying process	102
Query logic	103
Search types	104
Search execution preferences	105
The Search shards API	106
Basic queries	108
The term query	108
The terms query	109
The match_all query	110
The common terms query	110
The match query	112
The Boolean match query	112
The match_phrase query	114
The match_phrase_prefix query	114
The multi_match query	115
The query_string query	116
Running the query_string query against multiple fields	118
The simple_query_string query	118
The identifiers query	119
The prefix query	120
The fuzzy_like_this query	121
The fuzzy_like_this_field query	122
The fuzzy query	122
The wildcard query	124
The more_like_this query	125
The more_like_this_field query	126
The range query	127
The dismax query	128
The regular expression query	129
Compound queries	129
The bool query	130
The boosting query	131
The constant_score query	132
The indices query	133
Filtering your results	134
Using filters	134
Filter types	136
The range filter	136
The exists filter	137
The missing filter	138
The script filter	138
The type filter	139
The limit filter	139

Table of Contents

The identifiers filter	139
If this is not enough	140
Combining filters	141
Named filters	143
Caching filters	146
Highlighting	147
Getting started with highlighting	148
Field configuration	149
Under the hood	149
Configuring HTML tags	150
Controlling the highlighted fragments	151
Global and local settings	151
Require matching	152
The postings highlighter	155
Validating your queries	158
Using the validate API	158
Sorting data	161
Default sorting	161
Selecting fields used for sorting	162
Specifying the behavior for missing fields	164
Dynamic criteria	165
Collation and national characters	166
Query rewrite	166
An example of the rewrite process	166
Query rewrite properties	168
Summary	169
Chapter 4: Extending Your Index Structure	171
Indexing tree-like structures	171
Data structure	172
Analysis	173
Indexing data that is not flat	174
Data	174
Objects	175
Arrays	175
Mappings	175
Final mappings	176
Sending the mappings to Elasticsearch	177
To be or not to be dynamic	178
Using nested objects	178
Scoring and nested queries	182
Using the parent-child relationship	182

Table of Contents

Index structure and data indexing	183
Parent mappings	183
Child mappings	183
The parent document	184
The child documents	184
Querying	184
Querying data in the child documents	185
Querying data in the parent documents	187
The parent-child relationship and filtering	188
Performance considerations	188
Modifying your index structure with the update API	189
The mappings	189
Adding a new field	189
Modifying fields	190
Summary	192
Chapter 5: Make Your Search Better	193
An introduction to Apache Lucene scoring	193
When a document is matched	194
Default scoring formula	194
Relevancy matters	195
Scripting capabilities of Elasticsearch	196
Objects available during script execution	196
MVEL	198
Using other languages	198
Using our own script library	199
Using native code	199
Searching content in different languages	202
Handling languages differently	202
Handling multiple languages	203
Detecting the language of the documents	203
Sample document	204
The mappings	204
Querying	206
Queries with the identified language	206
Queries with unknown languages	207
Combining queries	208
Influencing scores with query boosts	209
The boost	209
Adding boost to queries	209
Modifying the score	212
The constant_score query	212
The boosting query	213
The function_score query	213

Table of Contents

Deprecated queries	219
When does index-time boosting make sense?	222
Defining field boosting in input data	222
Defining boosting in mapping	223
Words with the same meaning	223
The synonym filter	224
Synonyms in the mappings	224
Synonyms stored in the filesystem	225
Defining synonym rules	225
Using Apache Solr synonyms	225
Using WordNet synonyms	227
Query- or index-time synonym expansion	227
Understanding the explain information	227
Understanding field analysis	227
Explaining the query	229
Summary	231
Chapter 6: Beyond Full-text Searching	233
Aggregations	233
General query structure	234
Available aggregations	236
Metric aggregations	236
Bucketing	240
Nesting aggregations	255
Bucket ordering and nested aggregations	258
Global and subsets	258
Inclusions and exclusions	261
Faceting	262
The document structure	262
Returned results	263
Using queries for facetting calculations	264
Using filters for facetting calculations	265
Terms facetting	266
Ranges based facetting	268
Choosing different fields for an aggregated data calculation	270
Numerical and date histogram facetting	271
The date_histogram facet	272
Computing numerical field statistical data	272
Computing statistical data for terms	274
Geographical facetting	276
Filtering facetting results	277
Memory considerations	277
Using suggesters	278

Table of Contents

Available suggester types	278
Including suggestions	278
The suggester response	279
The term suggester	281
The term suggester configuration options	281
Additional term suggester options	282
The phrase suggester	283
The completion suggester	284
Percolator	289
The index	289
Percolator preparation	290
Getting deeper	293
Getting the number of matching queries	296
Indexed documents percolation	296
Handling files	297
Adding additional information about the file	300
Geo	301
Mappings preparation for spatial search	301
Example data	302
Sample queries	302
Distance-based sorting	302
Bounding box filtering	304
Limiting the distance	306
Arbitrary geo shapes	307
Point	308
Envelope	308
Polygon	308
Multipolygon	309
An example usage	309
Storing shapes in the index	311
The scroll API	312
Problem definition	313
Scrolling to the rescue	313
The terms filter	316
Terms lookup	317
The terms lookup query structure	320
Terms lookup cache settings	321
Summary	321
Chapter 7: Elasticsearch Cluster in Detail	323
Node discovery	323
Discovery types	324
The master node	324
Configuring the master and data nodes	325
The master-election configuration	325

Table of Contents

Setting the cluster name	326
Configuring multicast	326
Configuring unicast	327
Ping settings for nodes	327
The gateway and recovery modules	328
The gateway	328
Recovery control	328
Additional gateway recovery options	329
Preparing Elasticsearch cluster for high query and indexing throughput	330
The filter cache	330
The field data cache and circuit breaker	330
The circuit breaker	331
The store	331
Index buffers and the refresh rate	332
The index refresh rate	332
The thread pool configuration	333
Combining it all together – some general advice	334
Choosing the right store	335
The index refresh rate	335
Tuning the thread pools	336
Tuning your merge process	336
The field data cache and breaking the circuit	336
RAM buffer for indexing	337
Tuning transaction logging	337
Things to keep in mind	338
Templates and dynamic templates	338
Templates	338
An example of a template	338
Storing templates in files	339
Dynamic templates	340
The matching pattern	341
Field definitions	341
Summary	342
Chapter 8: Administrating Your Cluster	343
The Elasticsearch time machine	343
Creating a snapshot repository	344
Creating snapshots	345
Additional parameters	346
Restoring a snapshot	347
Cleaning up – deleting old snapshots	348
Monitoring your cluster's state and health	348
The cluster health API	348
Controlling information details	349

Table of Contents

Additional parameters	349
The indices stats API	350
Docs	351
Store	351
Indexing, get, and search	352
Additional information	353
The status API	353
The nodes info API	353
The nodes stats API	355
The cluster state API	356
The pending tasks API	357
The indices segments API	357
The cat API	357
Limiting returned information	358
Controlling cluster rebalancing	359
Rebalancing	359
Cluster being ready	359
The cluster rebalance settings	360
Controlling when rebalancing will start	360
Controlling the number of shards being moved between nodes concurrently	360
Controlling the number of shards initialized concurrently on a single node	360
Controlling the number of primary shards initialized concurrently on a single node	360
Controlling types of shards allocation	361
Controlling the number of concurrent streams on a single node	361
Controlling the shard and replica allocation	361
Explicitly controlling allocation	362
Specifying node parameters	362
Configuration	362
Index creation	362
Excluding nodes from allocation	363
Requiring node attributes	364
Using IP addresses for shard allocation	364
Disk-based shard allocation	364
Cluster wide allocation	366
Number of shards and replicas per node	366
Moving shards and replicas manually	366
Moving shards	367
Canceling shard allocation	367
Forcing shard allocation	368
Multiple commands per HTTP request	368
Warming up	369
Defining a new warming query	369
Retrieving the defined warming queries	371
Deleting a warming query	372
Disabling the warming up functionality	372

Table of Contents

Choosing queries	372
Index aliasing and using it to simplify your everyday work	374
An alias	374
Creating an alias	374
Modifying aliases	375
Combining commands	375
Retrieving all aliases	376
Removing aliases	376
Filtering aliases	376
Aliases and routing	377
Elasticsearch plugins	378
The basics	378
Installing plugins	379
Removing plugins	380
The update settings API	380
Summary	381
Index	383

Preface

Welcome to *Elasticsearch Server Second Edition*. In the second edition of the book, we decided not only to do the update to match the latest version of Elasticsearch but also to add some additional important sections that we didn't think of while writing the first book. While reading this book, you will be taken on a journey through a wonderful world of full-text search provided by the Elasticsearch server. We will start with a general introduction to Elasticsearch, which covers how to start and run Elasticsearch, what are the basic concepts of Elasticsearch, and how to index and search your data in the most basic way.

This book will also discuss the query language, so-called QueryDSL, that allows you to create complicated queries and filter the returned results. In addition to all this, you'll see how you can use faceting to calculate aggregated data based on the results returned by your queries, and how to use the newly introduced aggregation framework (the analytics engine allows you to give meaning to your data). We will implement autocomplete functionality together and learn how to use Elasticsearch spatial capabilities and prospective search.

Finally, this book will show you Elasticsearch administration API capabilities with features such as shard placement control and cluster handling.

What this book covers

Chapter 1, Getting Started with the Elasticsearch Cluster, covers what full-text searching, Apache Lucene, and text analysis are, how to run and configure Elasticsearch, and finally, how to index and search your data in the most basic way.

Chapter 2, Indexing Your Data, shows how indexing works, how to prepare an index structure and what data types we are allowed to use, how to speed up indexing, what segments are, how merging works, and what routing is.

Chapter 3, Searching Your Data, introduces the full-text search capabilities of Elasticsearch by discussing how to query, how the querying process works, and what type of basic and compound queries are available. In addition to this, we will learn how to filter our results, use highlighting, and modify the sorting of returned results.

Chapter 4, Extending Your Index Structure, discusses how to index more complex data structures. We will learn how to index tree-like data types, index data with relationships between documents, and modify the structure of an index.

Chapter 5, Make Your Search Better, covers Apache Lucene scoring and how to influence it in Elasticsearch, the scripting capabilities of Elasticsearch, and language analysis.

Chapter 6, Beyond Full-text Searching, shows the details of the aggregation framework functionality, faceting, and how to implement spellchecking and autocomplete using Elasticsearch. In addition to this, readers will learn how to index binary files, work with geospatial data, and efficiently process large datasets.

Chapter 7, Elasticsearch Cluster in Detail, discusses the nodes discovery mechanism, recovery and gateway Elasticsearch modules, templates and cluster preparation for high indexing, and querying use cases.

Chapter 8, Administrating Your Cluster, covers the Elasticsearch backup functionality, cluster monitoring, rebalancing, and moving shards. In addition to this, you will learn how to use the warm-up functionality, work with aliases, install plugins, and update cluster settings with the update API.

What you need for this book

This book was written using Elasticsearch server Version 1.0.0, and all the examples and functions should work with it. In addition to this, you'll need a command that allows you to send HTTP requests such as cURL, which is available for most operating systems. Please note that all the examples in this book use the mentioned cURL tool. If you want to use another tool, please remember to format the request in an appropriate way that can be understood by the tool of your choice.

In addition to this, some chapters may require additional software such as Elasticsearch plugins, but it has been explicitly mentioned when certain types of software are needed.

Who this book is for

If you are a beginner to the world of full-text search and Elasticsearch, this book is for you. You will be guided through the basics of Elasticsearch, and you will learn how to use some of the advanced functionalities.

If you know Elasticsearch and have worked with it, you may find this book interesting as it provides a nice overview of all the functionalities with examples and description.

If you know the Apache Solr search engine, this book can also be used to compare some functionalities of Apache Solr and Elasticsearch. This may give you the knowledge about the tool, which is more appropriate for your use.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"The postings format is a per-field property, just like `type` or `name`."

A block of code is set as follows:

```
{  
    "status" : 200,  
    "name" : "es_server",  
    "version" : {  
        "number" : "1.0.0",  
        "build_hash" : "a46900e9c72c0a623d71b54016357d5f94c8ea32",  
        "build_timestamp" : "2014-02-12T16:18:34Z",  
        "build_snapshot" : false,  
        "lucene_version" : "4.6"  
    "tagline" : "You Know, for Search"  
}
```

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

```
{  
  "mappings" : {  
    "post" : {  
      "properties" : {  
        "id" : { "type" : "long", "store" : "yes",  
                 "precision_step" : "0" },  
        "name" : { "type" : "string", "store" : "yes",  
                   "index" : "analyzed", "similarity" : "BM25" },  
        "contents" : { "type" : "string", "store" : "no",  
                      "index" : "analyzed", "similarity" : "BM25" }  
      }  
    }  
  }  
}
```

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

```
curl -XGET http://localhost:9200/blog/article/1
```



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

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

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1

Getting Started with the Elasticsearch Cluster

Welcome to the wonderful world of Elasticsearch—a great full text search and analytics engine. It doesn't matter if you are new to Elasticsearch and full text search in general or if you have experience. We hope that by reading this book you'll be able to learn and extend your knowledge of Elasticsearch. As this book is also dedicated to beginners, we decided to start with a short introduction to full text search in general and after that, a brief overview of Elasticsearch.

The first thing we need to do with Elasticsearch is install it. With many applications, you start with the installation and configuration and usually forget the importance of those steps. We will try to guide you through these steps so that it becomes easier to remember. In addition to this, we will show you the simplest way to index and retrieve data without getting into too many details. By the end of this chapter, you will have learned the following topics:

- Full-text searching
- Understanding Apache Lucene
- Performing text analysis
- Learning the basic concepts of Elasticsearch
- Installing and configuring Elasticsearch
- Using the Elasticsearch REST API to manipulate data
- Searching using basic URI requests

Full-text searching

Back in the days when full-text searching was a term known to a small percentage of engineers, most of us used SQL databases to perform search operations. Of course, it is ok, at least to some extent. However, as you go deeper and deeper, you start to see the limits of such an approach. Just to mention some of them—lack of scalability, not enough flexibility, and lack of language analysis (of course there were additions that introduced full-text searching to SQL databases). These were the reasons why Apache Lucene (<http://lucene.apache.org>) was created—to provide a library of full text search capabilities. It is very fast, scalable, and provides analysis capabilities for different languages.

The Lucene glossary and architecture

Before going into the details of the analysis process, we would like to introduce you to the glossary for Apache Lucene and the overall architecture of Apache Lucene. The basic concepts of the mentioned library are as follows:

- **Document:** This is a main data carrier used during indexing and searching, comprising one or more fields that contain the data we put in and get from Lucene.
- **Field:** This is a section of the document which is built of two parts; the name and the value.
- **Term:** This is a unit of search representing a word from the text.
- **Token:** This is an occurrence of a term in the text of the field. It consists of the term text, start and end offsets, and a type.

Apache Lucene writes all the information to the structure called **inverted index**. It is a data structure that maps the terms in the index to the documents and not the other way around as the relational database does in its tables. You can think of an inverted index as a data structure where data is term-oriented rather than document-oriented. Let's see how a simple inverted index will look. For example, let's assume that we have the documents with only the title field to be indexed and they look as follows:

- Elasticsearch Server 1.0 (document 1)
- Mastering Elasticsearch (document 2)
- Apache Solr 4 Cookbook (document 3)

So, the index (in a very simplified way) can be visualized as follows:

Term	Count	Document
1.0	1	<1>
4	1	<3>
Apache	1	<3>
Cookbook	1	<3>
Elasticsearch	2	<1>, <2>
Mastering	1	<2>
Server	1	<1>
Solr	1	<3>

Each term points to the number of documents it is present in. This allows a very efficient and fast searching, such as the term-based queries. In addition to this, each term has a number connected to it, **count**, telling Lucene how often the term occurs.

Of course, the actual index created by Lucene is much more complicated and advanced because of additional files that include information such as **term vectors**, **doc values**, and so on. However, all you need to know for now is how the data is organized and not what is exactly stored.

Each index is divided into multiple write once and read many time segments. When indexing, after a single segment is written to the disk, it can't be updated. Therefore, the information on deleted documents is stored in a separate file, but the segment itself is not updated.

However, multiple segments can be merged together through a process called **segments merge**. After forcing the segments to merge or after Lucene decides that it is time to perform merging, the segments are merged together by Lucene to create larger ones. This can demand I/O; however, some information needs to be cleaned up because during this time, information that is not needed anymore will be deleted (for example, the deleted documents). In addition to this, searching with one large segment is faster than searching with multiple smaller ones holding the same data. That's because, in general, to search means to just match the query terms to the ones that are indexed. You can imagine how searching through multiple small segments and merging those results will be slower than having a single segment preparing the results.

Input data analysis

Of course, the question that arises is how the data that is passed in the documents is transformed into the inverted index and how the query text is changed into terms to allow searching. The process of transforming this data is called **analysis**. You may want some of your fields to be processed by a language analyzer so that words such as *car* and *cars* are treated as the same in your index. On the other hand, you may want other fields to be only divided on the white space or only lowercased.

Analysis is done by the **analyzer**, which is built of a **tokenizer** and zero or more **token filters**, and it can also have zero or more character **mappers**.

A tokenizer in Lucene is used to split the text into tokens, which are basically the terms with additional information, such as its position in the original text and its length. The results of the tokenizer's work is called a **token stream**, where the tokens are put one by one and are ready to be processed by the filters.

Apart from the tokenizer, the Lucene analyzer is built of zero or more token filters that are used to process tokens in the token stream. Some examples of filters are as follows:

- **Lowercase filter:** This makes all the tokens lowercased
- **Synonyms filter:** This is responsible for changing one token to another on the basis of synonym rules
- **Multiple language stemming filters:** These are responsible for reducing tokens (actually, the text part that they provide) into their root or base forms, the stem

Filters are processed one after another, so we have almost unlimited analysis possibilities with the addition of multiple filters one after another.

Finally, the character mappers operate on non-analyzed text—they are used before the tokenizer. Therefore, we can easily remove HTML tags from whole parts of text without worrying about tokenization.

Indexing and querying

We may wonder how all the preceding functionalities affect indexing and querying when using Lucene and all the software that is built on top of it. During indexing, Lucene will use an analyzer of your choice to process the contents of your document; of course, different analyzers can be used for different fields, so the `name` field of your document can be analyzed differently compared to the `summary` field. Fields may not be analyzed at all, if we want.

During a query, your query will be analyzed. However, you can also choose not to analyze your queries. This is crucial to remember because some of the Elasticsearch queries are analyzed and some are not. For example, the prefix and the term queries are not analyzed, and the match query is analyzed. Having the possibility to chose from the queries that are analyzed and the ones that are not analyzed are very useful; sometimes, you may want to query a field that is not analyzed, while sometimes you may want to have a full text search analysis. For example, if we search for the `LightRed` term and the query is being analyzed by the standard analyzer, then the terms that would be searched are `light` and `red`. If we use a query type that has not been analyzed, then we will explicitly search for the `LightRed` term.

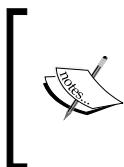
What you should remember about indexing and querying analysis is that the index should match the query term. If they don't match, Lucene won't return the desired documents. For example, if you are using stemming and lowercasing during indexing, you need to ensure that the terms in the query are also lowercased and stemmed, or your queries wouldn't return any results at all. It is important to keep the token filters in the same order during indexing and query time analysis so that the terms resulting of such an analysis are the same.

Scoring and query relevance

There is one additional thing we haven't mentioned till now – **scoring**. What is the score of a document? The **score** is a result of a scoring formula that describes how well the document matches the query. By default, Apache Lucene uses the **TF/IDF (term frequency / inverse document frequency)** scoring mechanism – an algorithm that calculates how relevant the document is in the context of our query. Of course, it is not the only algorithm available, and we will mention other algorithms in the *Mappings configuration* section of *Chapter 2, Indexing Your Data*.

If you want to read more about the Apache Lucene TF/IDF scoring formula, please visit Apache Lucene Javadocs for the `TFIDFSimilarity` class available at http://lucene.apache.org/core/4_6_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

Remember though that the higher the score value calculated by Elasticsearch and Lucene, the more relevant is the document. The score calculation is affected by parameters such as boost, by different query types (we will discuss these query types in the *Basic queries* section of *Chapter 3, Searching Your Data*), or by using different scoring algorithms.



If you want to read more detailed information about how Apache Lucene scoring works, what the default algorithm is, and how the score is calculated, please refer to our book, *Mastering ElasticSearch*, Packt Publishing.



The basics of Elasticsearch

Elasticsearch is an open source search server project started by Shay Banon and published in February 2010. During this time, the project has grown into a major player in the field of search and data analysis solutions and is widely used in many more or lesser-known search applications. In addition, due to its distributed nature and real-time capabilities, many people use it as a document store.

Key concepts of data architecture

Let's go through the basic concepts of Elasticsearch. You can skip this section if you are already familiar with the Elasticsearch architecture. However, if you are not familiar with this architecture, consider reading this section. We will refer to the key words used in the rest of the book.

Index

Index is the logical place where Elasticsearch stores logical data, so that it can be divided into smaller pieces. If you come from the relational database world, you can think of an index like a table. However, the index structure is prepared for fast and efficient full-text searching, and in particular, does not store original values. If you know MongoDB, you can think of the Elasticsearch index as a collection in MongoDB. If you are familiar with CouchDB, you can think about an index as you would about the CouchDB database. Elasticsearch can hold many indices located on one machine or spread over many servers. Every index is built of one or more **shards**, and each shard can have many **replicas**.

Document

The main entity stored in Elasticsearch is a **document**. Using the analogy to relational databases, a document is a row of data in a database table. When you compare an Elasticsearch document to a MongoDB document, you will see that both can have different structures, but the document in Elasticsearch needs to have the same type for all the common fields. This means that all the documents with a field called `title` need to have the same data type for it, for example, `string`.

Documents consist of **fields**, and each field may occur several times in a single document (such a field is called **multivalued**). Each field has a type (text, number, date, and so on). The field types can also be complex: a field can contain other subdocuments or arrays. The field type is important for Elasticsearch because it gives information about how various operations such as analysis or sorting should be performed. Fortunately, this can be determined automatically (however, we still suggest using mappings). Unlike the relational databases, documents don't need to have a fixed structure – every document may have a different set of fields, and in addition to this, fields don't have to be known during application development. Of course, one can force a document structure with the use of schema. From the client's point of view, a document is a JSON object (see more about the JSON format at <http://en.wikipedia.org/wiki/JSON>). Each document is stored in one index and has its own unique identifier (which can be generated automatically by Elasticsearch) and **document type**. A document needs to have a unique identifier in relation to the document type. This means that in a single index, two documents can have the same unique identifier if they are not of the same type.

Document type

In Elasticsearch, one index can store many objects with different purposes. For example, a blog application can store articles and comments. The document type lets us easily differentiate between the objects in a single index. Every document can have a different structure, but in real-world deployments, dividing documents into types significantly helps in data manipulation. Of course, one needs to keep the limitations in mind; that is, different document types can't set different types for the same property. For example, a field called `title` must have the same type across all document types in the same index.

Mapping

In the section about the basics of full-text searching (the *Full-text searching* section), we wrote about the process of analysis – the preparation of input text for indexing and searching. Every field of the document must be properly analyzed depending on its type. For example, a different analysis chain is required for the numeric fields (numbers shouldn't be sorted alphabetically) and for the text fetched from web pages (for example, the first step would require you to omit the HTML tags as it is useless information – noise). Elasticsearch stores information about the fields in the mapping. Every document type has its own mapping, even if we don't explicitly define it.

Key concepts of Elasticsearch

Now, we already know that Elasticsearch stores data in one or more indices. Every index can contain documents of various types. We also know that each document has many fields and how Elasticsearch treats these fields is defined by mappings. But there is more. From the beginning, Elasticsearch was created as a distributed solution that can handle billions of documents and hundreds of search requests per second. This is due to several important concepts that we are going to describe in more detail now.

Node and cluster

Elasticsearch can work as a standalone, single-search server. Nevertheless, to be able to process large sets of data and to achieve fault tolerance and high availability, Elasticsearch can be run on many cooperating servers. Collectively, these servers are called a **cluster**, and each server forming it is called a **node**.

Shard

When we have a large number of documents, we may come to a point where a single node may not be enough—for example, because of RAM limitations, hard disk capacity, insufficient processing power, and inability to respond to client requests fast enough. In such a case, data can be divided into smaller parts called **shards** (where each shard is a separate Apache Lucene index). Each shard can be placed on a different server, and thus, your data can be spread among the cluster nodes. When you query an index that is built from multiple shards, Elasticsearch sends the query to each relevant shard and merges the result in such a way that your application doesn't know about the shards. In addition to this, having multiple shards can speed up the indexing.

Replica

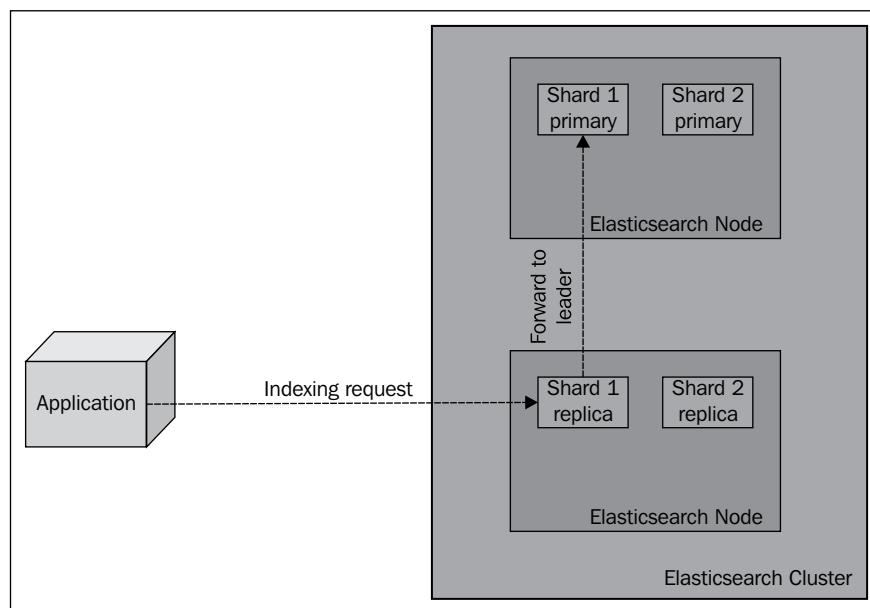
In order to increase query throughput or achieve high availability, shard replicas can be used. A **replica** is just an exact copy of the shard, and each shard can have zero or more replicas. In other words, Elasticsearch can have many identical shards and one of them is automatically chosen as a place where the operations that change the index are directed. This special shard is called a **primary shard**, and the others are called **replica shards**. When the primary shard is lost (for example, a server holding the shard data is unavailable), the cluster will promote the replica to be the new primary shard.

Gateway

Elasticsearch handles many nodes. The cluster state is held by the gateway. By default, every node has this information stored locally, which is synchronized among nodes. We will discuss the gateway module in *The gateway and recovery modules* section of *Chapter 7, Elasticsearch Cluster in Detail*.

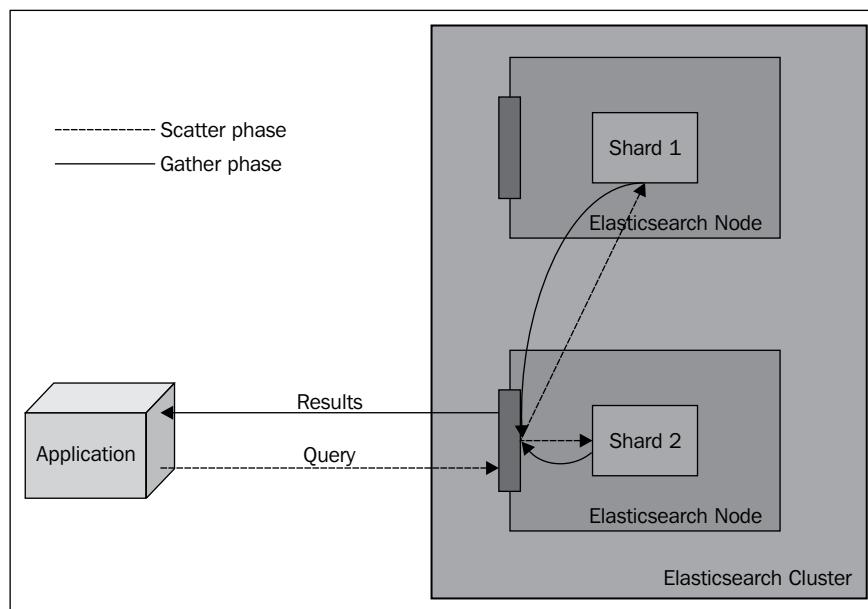
Indexing and searching

You may wonder how you can practically tie all the indices, shards, and replicas together in a single environment. Theoretically, it should be very difficult to fetch data from the cluster when you have to know where is your document, on which server, and in which shard. Even more difficult is searching when one query can return documents from different shards placed on different nodes in the whole cluster. In fact, this is a complicated problem; fortunately, we don't have to care about this—it is handled automatically by Elasticsearch itself. Let's look at the following diagram:



When you send a new document to the cluster, you specify a target index and send it to any of the nodes. The node knows how many shards the target index has and is able to determine which shard should be used to store your document. Elasticsearch can alter this behavior; we will talk about this in the *Routing* section of *Chapter 2, Indexing Your Data*. The important information that you have to remember for now is that Elasticsearch calculates the shard in which the document should be placed using the unique identifier of the document. After the indexing request is sent to a node, that node forwards the document to the target node, which hosts the relevant shard.

Now let's look at the following diagram on searching request execution:



When you try to fetch a document by its identifier, the node you send the query to uses the same routing algorithm to determine the shard and the node holding the document and again forwards the query, fetches the result, and sends the result to you. On the other hand, the querying process is a more complicated one. The node receiving the query forwards it to all the nodes holding the shards that belong to a given index and asks for minimum information about the documents that match the query (identifier and score, by default), unless routing is used, where the query will go directly to a single shard only. This is called the **scatter phase**. After receiving this information, the aggregator node (the node that receives the client request) sorts the results and sends a second request to get the documents that are needed to build the results list (all the other information apart from the document identifier and score).

This is called the **gather phase**. After this phase is executed, the results are returned to the client.

Now the question arises – what is the role of replicas in the process described previously? While indexing, replicas are only used as an additional place to store the data. When executing a query, by default, Elasticsearch will try to balance the load among the shard and its replicas so that they are evenly stressed.

Also, remember that we can change this behavior; we will discuss this in the *Understanding the querying process* section of *Chapter 3, Searching Your Data*.

Installing and configuring your cluster

There are a few steps required to install Elasticsearch, which we will explore in the following sections.

Installing Java

In order to set up Elasticsearch, the first step is to make sure that a Java SE environment is installed properly. Elasticsearch requires Java Version 6 or later to run. You can download it from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. You can also use OpenJDK (<http://openjdk.java.net/>) if you wish. You can, of course, use Java Version 6, but it is not supported with patches by default, so we suggest that you install Java 7.

Installing Elasticsearch

To install Elasticsearch, just download it from <http://www.elasticsearch.org/download/> and unpack it. Choose the last stable version. That's it! The installation is complete.



At the time of writing this book, we used Elasticsearch 1.0.0.GA. This means that we've skipped describing some properties that were marked as deprecated and are or will be removed in the future versions of Elasticsearch.

The main interface to communicate with Elasticsearch is based on an HTTP protocol and REST. This means that you can even use a web browser for some basic queries and requests, but for anything more sophisticated, you'll need to use additional software such as the cURL command. If you use the Linux or OS X command, the curl package should already be available. If you use Windows, you can download it from <http://curl.haxx.se/download.html>.

Installing Elasticsearch from binary packages on Linux

The other way to install Elasticsearch is to use the provided binary packages – the RPM or DEB packages, depending on your Linux distribution. The mentioned binary packages can be found at the following URL address: <http://www.elasticsearch.org/download/>.

Installing Elasticsearch using the RPM package

After downloading the RPM package, you just need to run the following command:

```
sudo yum elasticsearch-1.0.0.noarch.rpm
```

It is as simple as that. If everything went well, Elasticsearch should be installed and its configuration file should be stored in /etc/sysconfig/elasticsearch. If your operating system is based on Red Hat, you will be able to use the init script found at /etc/init.d/elasticsearch. If your operating system is a SUSE Linux, you can use the systemctl file found at /bin to start and stop the Elasticsearch service.

Installing Elasticsearch using the DEB package

After downloading the DEB package, all you need to do is run the following command:

```
sudo dpkg -i elasticsearch-1.0.0.deb
```

It is as simple as that. If everything went well, Elasticsearch should be installed and its configuration file should be stored in /etc/elasticsearch/elasticsearch.yml. The init script that allows you to start and stop Elasticsearch will be found at /etc/init.d/elasticsearch. Also, there will be files containing environment settings at /etc/default/elasticsearch.

The directory layout

Now, let's go to the newly created directory. We should see the following directory structure:

Directory	Description
bin	The scripts needed for running Elasticsearch instances and for plugin management
config	The directory where configuration files are located
lib	The libraries used by Elasticsearch

After Elasticsearch starts, it will create the following directories (if they don't exist):

Directory	Description
data	Where all the data used by Elasticsearch is stored
logs	The files with information about events and errors
plugins	The location for storing the installed plugins
work	The temporary files used by Elasticsearch

Configuring Elasticsearch

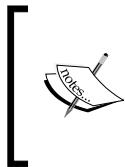
One of the reasons – of course, not the only one – why Elasticsearch is gaining more and more popularity is that getting started with Elasticsearch is quite easy. Because of the reasonable default values and automatic settings for simple environments, we can skip the configuration and go straight to the next chapter without changing a single line in our configuration files. However, in order to truly understand Elasticsearch, it is worth understanding some of the available settings.

We will now explore the default directories and layout of the files provided with the Elasticsearch `.tar.gz` archive. The whole configuration is located in the `config` directory. We can see two files there: `elasticsearch.yml` (or `elasticsearch.json`, which will be used if present) and `logging.yml`. The first file is responsible for setting the default configuration values for the server. This is important because some of these values can be changed at runtime and can be kept as a part of the cluster state, so the values in this file may not be accurate. The two values that we cannot change at runtime are `cluster.name` and `node.name`.

The `cluster.name` property is responsible for holding the name of our cluster. The cluster name separates different clusters from each other. Nodes configured with the same cluster name will try to form a cluster.

The second value is the instance (the node) name. We can leave this parameter undefined. In this case, Elasticsearch automatically chooses a unique name for itself. Note that this name is chosen during every startup, so the name can be different on each restart. Defining the name can help when referring to concrete instances by the API or when using monitoring tools to see what is happening to a node during long periods of time and between restarts. Think about giving descriptive names to your nodes.

Other parameters are well commented in the file, so we advise you to look through it; don't worry if you do not understand the explanation. We hope that everything will become clear after reading the next few chapters.



Remember that most of the parameters that have been set in the `elasticsearch.yml` file can be overwritten with the use of Elasticsearch REST API. We will talk about this API in the *The update settings API* section of *Chapter 8, Administrating Your Cluster*.



The second file (`logging.yml`) defines how much information is written to system logs, defines the logfiles, and creates new files periodically. Changes in this file are usually required only when you need to adapt to monitoring or backup solutions or during system debugging; however, if you want to have a more detailed logging, you need to adjust it accordingly.

Let's leave the configuration files for now. An important part of the configuration is tuning your operating system. During the indexing, especially when having many shards and replicas, Elasticsearch will create many files; so, the system cannot limit the open file descriptors to less than 32,000. For Linux servers, this can be usually changed in `/etc/security/limits.conf` and the current value can be displayed using the `ulimit` command. If you end up reaching the limit, Elasticsearch will not be able to create new files; so, merging will fail, indexing may fail, and new indices will not be created.

The next set of settings is connected to the **Java Virtual Machine (JVM)** heap memory limit for a single Elasticsearch instance. For small deployments, the default memory limit (1024 MB) will be sufficient, but for large ones, it will not be enough. If you spot entries that indicate the `OutOfMemoryError` exceptions in a logfile, set the `ES_HEAP_SIZE` variable to a value greater than 1024. When choosing the right amount of memory size to be given to the JVM, remember that, in general, no more than 50 percent of your total system memory should be given. However, as with all the rules, there are exceptions. We will discuss this in greater detail later, but you should always monitor your JVM heap usage and adjust it when needed.

Running Elasticsearch

Let's run our first instance that we just downloaded as the ZIP archive and unpacked. Go to the `bin` directory and run the following commands depending on the OS:

- Linux or OS X: `./elasticsearch`
- Windows: `elasticsearch.bat`

Congratulations! Now, we have our Elasticsearch instance up and running. During its work, the server usually uses two port numbers: the first one for communication with the REST API using the HTTP protocol, and the second one for the transport module used for communication in a cluster and in between the native Java client and the cluster. The default port used for the HTTP API is 9200, so we can check the search readiness by pointing the web browser to `http://127.0.0.1:9200/`. The browser should show a code snippet similar to the following:

```
{
  "status" : 200,
  "name" : "es_server",
  "version" : {
    "number" : "1.0.0",
    "build_hash" : "a46900e9c72c0a623d71b54016357d5f94c8ea32",
    "build_timestamp" : "2014-02-12T16:18:34Z",
    "build_snapshot" : false,
    "lucene_version" : "4.6"
  },
  "tagline" : "You Know, for Search"
}
```

The output is structured as a **JSON (JavaScript Object Notation)** object. If you are not familiar with JSON, please take a minute and read the article available at <http://en.wikipedia.org/wiki/JSON>.

Elasticsearch is smart. If the default port is not available, the engine binds to the next free port. You can find information about this on the console during booting as follows:



```
[2013-11-16 11:56:12,101] [INFO] [[http] [Red Lotus]
  bound_address {inet[/0:0:0:0:0:0:0%0:9200]},
  publish_address {inet[/192.168.1.101:9200}]}
```

Note the fragment with `[http]`. Elasticsearch uses a few ports for various tasks. The interface that we are using is handled by the HTTP module.

Now, we will use the cURL program. For example, to check cluster health, we will use the following command:

```
curl -XGET http://127.0.0.1:9200/_cluster/health?pretty
```

The `-x` parameter is a request method. The default value is `GET` (so, in this example, we can omit this parameter). Temporarily, do not worry about the `GET` value; we will describe it in more detail later in this chapter.

As a standard, the API returns information in a JSON object in which new line characters are omitted. The `pretty` parameter added to our requests forces Elasticsearch to add a new line character to the response, making the response more human friendly. You can try running the preceding query with and without the `?pretty` parameter to see the difference.

Elasticsearch is useful in small- and medium-sized applications, but it has been built with large clusters in mind. So, now we will set up our big, two-node cluster. Unpack the Elasticsearch archive in a different directory and run the second instance. If we look at the log, we see what is shown as follows:

```
[2013-11-16 11:55:16,767] [INFO ] [cluster.service] 
[Stane, Obadiah] detected_master [Martha Johansson]
[vswsFRWTSjOa_fy7uPuOMA]
[inet[/192.168.1.19:9300]], added {[Martha Johansson]
[vswsFRWTSjOa_fy7uPuOMA]
[inet[/192.168.1.19:9300]],}, reason: zen-disco-receive(from master
[[Martha Johansson] [vswsFRWTSjOa_fy7uPuOMA]
[inet[/192.168.1.19:9300]])
```

This means that our second instance (named `Stane`, `Obadiah`) discovered the previously running instance (named `Martha Johansson`). Here, Elasticsearch automatically formed a new, two-node cluster.



Note that on some systems, the firewall software may be enabled by default, which may result in the nodes not being able to discover themselves.



Shutting down Elasticsearch

Even though we expect our cluster (or node) to run flawlessly for a lifetime, we may need to restart it or shut it down properly (for example, for maintenance). The following are three ways in which we can shut down Elasticsearch:

- If your node is attached to the console, just press `Ctrl + C`
- The second option is to kill the server process by sending the `TERM` signal (see the `kill` command on the Linux boxes and Program Manager on Windows)
- The third method is to use a REST API

We will focus on the last method now. It allows us to shut down the whole cluster by executing the following command:

```
curl -XPOST http://localhost:9200/_cluster/nodes/_shutdown
```

To shut down just a single node, for example, a node with the BlrmMvBdSKiCeYGsiHijdg identifier, we will execute the following command:

```
curl -XPOST  
http://localhost:9200/_cluster/nodes/BlrmMvBdSKiCeYGsiHijdg/_shutdown
```

The identifier of the node can be read either from the logs or using the _cluster/nodes API, with the following command:

```
curl -XGET http://localhost:9200/_cluster/nodes/
```

Running Elasticsearch as a system service

Elasticsearch 1.0 can run as a service both on Linux-based systems as well as on Windows-based ones.

Elasticsearch as a system service on Linux

If you have installed Elasticsearch from the provided binary packages, you are already good to go and don't have to worry about anything. However, if you have just downloaded the archive and unpacked Elasticsearch to the directories of your choice, you'll need to put some additional effort. To install Elasticsearch as a Linux system service, we will use the Elasticsearch service wrapper that can be downloaded from <https://github.com/elasticsearch/elasticsearch-servicewrapper>.

Let's look at the steps to use the Elasticsearch service wrapper in order to set up a Linux service for Elasticsearch. First, we will run the following command to download the wrapper:

```
curl -L http://github.com/elasticsearch/elasticsearch-  
servicewrapper/tarball/master | tar -xz
```

Assuming that Elasticsearch has been installed in /usr/local/share/elasticsearch, we will run the following command to move the needed service wrapper files:

```
sudo mv *servicewrapper*/service /usr/local/share/elasticsearch/bin/
```

We will remove the remaining wrapper files by running the following command:

```
rm -Rf *servicewrapper*
```

Finally, we will install the service by running the `install` command as follows:

```
sudo /usr/local/share/elasticsearch/bin/service/elasticsearch install
```

After this, we need to create a symbolic link to the `/usr/local/share/elasticsearch/bin/service/elasticsearch` script in `/usr/local/bin/rceasticsearch`. We do this by running the following command:

```
sudo ln -s 'readlink -f  
/usr/local/share/elasticsearch/bin/service/elasticsearch'  
/usr/local/bin/rceasticsearch
```

And that's all. If you want to start Elasticsearch, just run the following command:

```
/etc/init.d/elasticsearch start
```

Elasticsearch as a system service on Windows

Installing Elasticsearch as a system service on Windows is very easy. You just need to go to your Elasticsearch installation directory, then go to the `bin` subdirectory, and run the following command:

```
service.bat install
```

You'll be asked about the permission to do so. If you allow the script to run, Elasticsearch will be installed as a Windows service.

If you would like to see all the commands exposed by the `service.bat` script file, just run the following command in the same directory as earlier:

```
service.bat
```

For example, to start Elasticsearch, we will just run the following command:

```
service.bat start
```

Manipulating data with the REST API

The Elasticsearch REST API can be used for various tasks. Thanks to this, we can manage indices, change instance parameters, check nodes and cluster status, index data, search the data, or retrieve documents via the GET API. But for now, we will concentrate on using the **CRUD (create-retrieve-update-delete)** part of the API, which allows you to use Elasticsearch in a similar way to how you would use a NoSQL database.

Understanding the Elasticsearch RESTful API

In a REST-like architecture, every request is directed to a concrete object indicated by the path of the address. For example, if `/books/` is a reference to a list of books in our library, `/books/1` is the reference to the book with the identifier 1. Note that these objects can be nested. The `/books/1/chapter/6` reference denotes the sixth chapter of the first book in the library, and so on. We have a subject for our API call. What about an operation that we would like to execute, such as `GET` or `POST`? To indicate this, request types are used. The HTTP protocol gives us quite a long list of types that can be used as verbs in the API calls. Logical choices are `GET` in order to obtain the current state of the requested object, `POST` to change the object state, `PUT` to create an object, and `DELETE` to destroy objects. There is also a `HEAD` request that is only used to fetch the base information of an object.

If we look at the following examples of the operations discussed in the *Shutting down Elasticsearch* section, everything should make more sense:

- `GET http://localhost:9000/`: This command retrieves basic information about Elasticsearch
- `GET http://localhost:9200/_cluster/state/nodes/`: This command retrieves the information about the nodes in the cluster
- `POST http://localhost:9200/_cluster/nodes/_shutdown`: This command sends a shutdown request to all the nodes in the cluster

We now know what REST means, at least in general (you can read more about REST at http://en.wikipedia.org/wiki/Representational_state_transfer). Now, we can proceed and learn how to use the Elasticsearch API to store, fetch, alter, and delete data.

Storing data in Elasticsearch

As we have already discussed, in Elasticsearch, every piece of data – each document – has a defined index and type. Each document can contain one or more fields that will hold your data. We will start by showing you how to index a simple document using Elasticsearch.

Creating a new document

Now, we will try to index some of the documents. For example, let's imagine that we are building some kind of CMS system for our blog. One of the entities in this blog is articles (surprise!).

Using the JSON notation, a document can be presented as shown in the following example:

```
{  
  "id": "1",  
  "title": "New version of Elasticsearch released!",  
  "content": "Version 1.0 released today!",  
  "priority": 10,  
  "tags": ["announce", "elasticsearch", "release"]  
}
```

As we can see, the JSON document contains a set of fields, where each field can have a different form. In our example, we have a number (`priority`), text (`title`), and an array of strings (`tags`). In the following examples, we will show you the other types. As mentioned earlier in this chapter, Elasticsearch can guess these types (because JSON is semi-typed; for example, the numbers are not in quotation marks) and automatically customize how this data will be stored in its internal structures.

Of course, we would like to index our example document and make it available for searching. We will use an index named `blog` and a type named `article`. In order to index our example document to this index under the given type and with the identifier of `1`, we will execute the following command:

```
curl -XPUT http://localhost:9200/blog/article/1 -d '{"title": "New  
version of Elasticsearch released!", "content": "Version 1.0  
released today!", "tags": ["announce", "elasticsearch", "release"] }'
```

Note a new option to the cURL command: the `-d` parameter. The value of this option is the text that will be used as a request payload—a request body. This way, we can send additional information such as document definition. Also, note that the unique identifier is placed in the URL and not in the body. If you omit this identifier (while using the HTTP PUT request), the indexing request will return the following error:

```
No handler found for uri [/blog/article/] and method [PUT]
```

If everything is correct, Elasticsearch will respond with a JSON response similar to the following output:

```
{  
  "_index": "blog",  
  "_type": "article",  
  "_id": "1",  
  "_version": 1  
}
```

In the preceding response, Elasticsearch includes the information about the status of the operation and shows where a new document was placed. There is information about the document's unique identifier and current version, which will be incremented automatically by Elasticsearch every time it is updated.

Automatic identifier creation

In the last example, we specified the document identifier ourselves. However, Elasticsearch can generate this automatically. This seems very handy, but only when index is the only source of data. If we use a database to store data and Elasticsearch for full-text searching, the synchronization of this data will be hindered unless the generated identifier is stored in the database as well. The generation of a unique identifier can be achieved by using the POST HTTP request type and by not specifying the identifier in the URL. For example, look at the following command:

```
curl -XPOST http://localhost:9200/blog/article/ -d '{"title": "New  
version of Elasticsearch released!", "content": "Version 1.0  
released today!", "tags": ["announce", "elasticsearch", "release"] }'
```

Note the use of the POST HTTP request method instead of PUT in comparison to the previous example. Referring to the previous description of REST verbs, we wanted to change the list of documents in the index rather than create a new entity, and that's why we used POST instead of PUT. The server should respond with a response similar to the following output:

```
{  
  "_index" : "blog",  
  "_type" : "article",  
  "_id" : "XQmdeSe_RVamFgRHMqcZQg",  
  "_version" : 1  
}
```

Note the highlighted line, which holds the unique identifier generated automatically by Elasticsearch.

Retrieving documents

We already have documents stored in our instance. Now let's try to retrieve them by using their identifiers. We will start by executing the following command:

```
curl -XGET http://localhost:9200/blog/article/1
```

Elasticsearch will return a response similar to the following output:

```
{  
  "_index": "blog",  
  "_type": "article",  
  "_id": "1",  
  "_version": 1,  
  "exists": true,  
  "_source": {  
    "title": "New version of Elasticsearch released!",  
    "content": "Version 1.0 released today!",  
    "tags": ["announce", "elasticsearch", "release"]  
  }  
}
```

In the preceding response, besides the index, type, identifier, and version, we can also see the information that says that the document was found (the `exists` property) and the source of this document (in the `_source` field). If document is not found, we get a reply as follows:

```
{  
  "_index": "blog",  
  "_type": "article",  
  "_id": "9999",  
  "exists": false  
}
```

Of course, there is no information about the version and source because no document was found.

Updating documents

Updating documents in the index is a more complicated task. Internally, Elasticsearch must first fetch the document, take its data from the `_source` field, remove the old document, apply changes to the `_source` field, and then index it as a new document. It is so complicated because we can't update the information once it is stored in the Lucene inverted index. Elasticsearch implements this through a script given as an update request parameter. This allows us to do more a sophisticated document transformation than simple field changes. Let's see how it works in a simple case.

Please recall the example blog article that we've indexed previously. We will try to change its content field from the old one to new content. To do this, we will run the following command:

```
curl -XPOST http://localhost:9200/blog/article/1/_update -d '{  
  "script": "ctx._source.content = \"new content\""  
}'
```

Elasticsearch will reply with the following response:

```
{"_index": "blog", "_type": "article", "_id": "1", "_version": 2}
```

It seems that the update operation was executed successfully. To be sure, let's retrieve the document by using its identifier. To do this, we will run the following command:

```
curl -XGET http://localhost:9200/blog/article/1
```

The response from Elasticsearch should include the changed content field, and indeed, it includes the following information:

```
{  
  "_index" : "blog",  
  "_type" : "article",  
  "_id" : "1",  
  "_version" : 2,  
  "exists" : true,  
  "_source" : {  
    "title": "New version of Elasticsearch released!",  
    "content": "new content",  
    "tags": ["announce", "elasticsearch", "release"]  
}
```

Elasticsearch changed the contents of our article and the version number for this document. Note that we didn't have to send the whole document, only the changed parts. However, remember that to use the update functionality, we need to use the `_source` field—we will describe how to use the `_source` field in the *Extending your index structure with additional internal information* section in *Chapter 2, Indexing Your Data*.

There is one more thing about document updates; if your script uses a field value from a document that is to be updated, you can set a value that will be used if the document doesn't have that value present. For example, if you want to increment the counter field of the document and it is not present, you can use the upsert section in your request to provide the default value that will be used. For example, look at the following lines of command:

```
curl -XPOST http://localhost:9200/blog/article/1/_update -d '{  
  "script": "ctx._source.counter += 1",  
  "upsert": {  
    "counter" : 0  
  }  
'
```

If you execute the preceding example, Elasticsearch will add the counter field with the value of 0 to our example document. This is because our document does not have the counter field present and we've specified the upsert section in the update request.

Deleting documents

We have already seen how to create (PUT) and retrieve (GET) documents. We also know how to update them. It is not difficult to guess that the process to remove a document is similar; we need to send a proper HTTP request using the DELETE request type. For example, to delete our example document, we will run the following command:

```
curl -XDELETE http://localhost:9200/blog/article/1
```

The response from Elasticsearch will be as follows:

```
{"found":true,"_index":"blog","_type":"article","_id":"1","_version":3}
```

This means that our document was found and it was deleted.

Now we can use the CRUD operations. This lets us create applications using Elasticsearch as a simple key-value store. But this is only the beginning!

Versioning

In the examples provided, you might have seen information about the version of the document, which looked like the following:

```
"_version" : 1
```

If you look carefully, you will notice that after updating the document with the same identifier, this version is incremented. By default, Elasticsearch increments the version when a document is added, changed, or deleted. In addition to informing us about the number of changes made to the document, it also allows us to implement **optimistic locking** (http://en.wikipedia.org/wiki/Optimistic_concurrency_control). This allows us to avoid issues when processing the same document in parallel. For example, we read the same document in two different applications, modify it differently, and then try to update the one in Elasticsearch. Without versioning the version, we will see the one sent for indexation as the last version. Using optimistic locking, Elasticsearch guards the data accuracy – every attempt to write the document that has been already changed will fail.

An example of versioning

Let's look at an example that uses versioning. Let's assume that we want to delete a document with the `book` type from the `library` index. We also want to be sure that the delete operation is successful if the document was not updated. What we need to do is add the `version` parameter with the value of 1 as follows:

```
curl -XDELETE 'localhost:9200/library/book/1?version=1'
```

If the version of the document in the index is different from 1, the following error will be returned by Elasticsearch:

```
{
  "error": "VersionConflictEngineException[[library] [4] [book] [1]]: 
    version conflict, current [2], provided [1]]",
  "status": 409
}
```

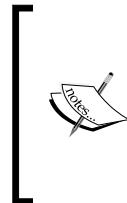
In our example, Elasticsearch compared the version number declared by us and saw that this version is not the same in comparison to the version of the document in Elasticsearch. That's why the operation failed.

Using the version provided by an external system

Elasticsearch can also be based on the version number provided by us. It is necessary when the version is stored in the external system – in this case, when you index a new document, you should provide the `version` parameter as in the preceding example. In such cases, Elasticsearch will only check if the version provided with the operation is greater (it is not important how much) than the one saved in the index. If it is, the operation will be successful, and if not, it will fail. To inform Elasticsearch that we want to use external version tracking, we need to add the `version_type=external` parameter in addition to the `version` parameter.

For example, if we want to add a document that has a version 123456 in our system, we will run a command as follows:

```
curl -XPUT 'localhost:9200/library/book/1?version=123456' -d {...}
```



Elasticsearch can check the version number even after the document is removed. That's because Elasticsearch keeps information about the version of the deleted document. By default, this information is available for 60 seconds after the deletion of the document. This time value can be changed by using the `index.gc_deletes` configuration parameter.

Searching with the URI request query

Before going into the details of Elasticsearch querying, we will use its capabilities of using a simple URI request to search. Of course, we will extend our search knowledge using Elasticsearch in *Chapter 3, Searching Your Data*, but for now, we will stick to the simplest approach.

Sample data

For the purpose of this section of the book, we will create a simple index with two document types. To do this, we will run the following commands:

```
curl -XPOST 'localhost:9200/books/es/1' -d '{"title":"Elasticsearch Server", "published": 2013}'  
curl -XPOST 'localhost:9200/books/es/2' -d '{"title":"Mastering Elasticsearch", "published": 2013}'  
curl -XPOST 'localhost:9200/books/solr/1' -d '{"title":"Apache Solr 4 Cookbook", "published": 2012}'
```

Running the preceding commands will create the `books` index with two types: `es` and `solr`. The `title` and `published` fields will be indexed. If you want to check this, you can do so by running the mappings API call using the following command (we will talk about the mappings in the *Mappings configuration* section of *Chapter 2, Indexing Your Data*):

```
curl -XGET 'localhost:9200/books/_mapping?pretty'
```

This will result in Elasticsearch returning the mappings for the whole index.

The URI request

All the queries in Elasticsearch are sent to the `_search` endpoint. You can search a single index or multiple indices, and you can also narrow down your search only to a given document type or multiple types. For example, in order to search our `books` index, we will run the following command:

```
curl -XGET 'localhost:9200/books/_search?pretty'
```

If we have another index called `clients`, we can also run a single query against these two indices as follows:

```
curl -XGET 'localhost:9200/books,clients/_search?pretty'
```

In the same manner, we can also choose the types we want to use during searching. For example, if we want to search only in the `es` type in the `books` index, we will run a command as follows:

```
curl -XGET 'localhost:9200/books/es/_search?pretty'
```

 Please remember that in order to search for a given type, we need to specify the index or indices. If we want to search for any index, we just need to set `*` as the index name or omit the index name totally. Elasticsearch allows quite a rich semantics when it comes to choosing index names. If you are interested, please refer to <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/multi-index.html>.

We can also search all the indices by omitting the indices and types. For example, the following command will result in a search through all the data in our cluster:

```
curl -XGET 'localhost:9200/_search?pretty'
```

The Elasticsearch query response

Let's assume that we want to find all the documents in our `books` index that contain the `elasticsearch` term in the `title` field. We can do this by running the following query:

```
curl -XGET  
'localhost:9200/books/_search?pretty&q=title:elasticsearch'
```

The response returned by Elasticsearch for the preceding request will be as follows:

```
{  
  "took" : 4,
```

```
"timed_out" : false,
"_shards" : {
  "total" : 5,
  "successful" : 5,
  "failed" : 0
},
"hits" : {
  "total" : 2,
  "max_score" : 0.625,
  "hits" : [ {
    "_index" : "books",
    "_type" : "es",
    "_id" : "1",
    "_score" : 0.625, "_source" : {"title":"Elasticsearch Server",
      "published": 2013}
  }, {
    "_index" : "books",
    "_type" : "es",
    "_id" : "2",
    "_score" : 0.19178301, "_source" : {"title":"Mastering
      Elasticsearch", "published": 2013}
  } ]
}
}
```

The first section of the response gives us the information on how much time the request took (the `took` property is specified in milliseconds); whether it was timed out (the `timed_out` property); and information on the shards that were queried during the request execution—the number of queried shards (the `total` property of the `_shards` object), the number of shards that returned the results successfully (the `successful` property of the `_shards` object), and the number of failed shards (the `failed` property of the `_shards` object). The query may also time out if it is executed for a longer time than we want. (We can specify the maximum query execution time using the `timeout` parameter.) The failed shard means that something went wrong on that shard or it was not available during the search execution.

Of course, the mentioned information can be useful, but usually, we are interested in the results that are returned in the `hits` object. We have the total number of documents returned by the query (in the `total` property) and the maximum score calculated (in the `max_score` property). Finally, we have the `hits` array that contains the returned documents. In our case, each returned document contains its index name (the `_index` property), type (the `_type` property), identifier (the `_id` property), score (the `_score` property), and the `_source` field (usually, this is the JSON object sent for indexing; we will discuss this in the *Extending your index structure with additional internal information* section in *Chapter 2, Indexing Your Data*).

Query analysis

You may wonder why the query we've run in the previous section worked. We indexed the `Elasticsearch` term and ran a query for `elasticsearch` and even though they differ (capitalization), relevant documents were found. The reason for this is the analysis. During indexing, the underlying Lucene library analyzes the documents and indexes the data according to the Elasticsearch configuration. By default, Elasticsearch will tell Lucene to index and analyze both string-based data as well as numbers. The same happens during querying because the URI request query maps to the `query_string` query (which will be discussed in *Chapter 3, Searching Your Data*), and this query is analyzed by Elasticsearch.

Let's use the `indices analyze` API (<http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/indices-analyze.html>). It allows us to see how the analysis process is done. With it, we can see what happened to one of the documents during indexing and what happened to our query phrase during querying.

In order to see what was indexed in the `title` field for the `Elasticsearch Server` phrase, we will run the following command:

```
curl -XGET 'localhost:9200/books/_analyze?field=title' -d  
'Elasticsearch Server'
```

The response will be as follows:

```
{  
  "tokens" : [ {  
    "token" : "elasticsearch",  
    "start_offset" : 0,  
    "end_offset" : 13,
```

```
"type" : "<ALPHANUM>",
"position" : 1
}, {
"token" : "server",
"start_offset" : 14,
"end_offset" : 20,
"type" : "<ALPHANUM>",
"position" : 2
} ]
}
```

We can see that Elasticsearch has divided the text into two terms – the first one has a token value of `elasticsearch` and the second one has a token value of `server`.

Now let's look at how the query text was analyzed. We can do that by running the following command:

```
curl -XGET 'localhost:9200/books/_analyze?pretty&field=title' -d
'elasticsearch'
```

The response of the request looks as follows:

```
{
  "tokens" : [ {
    "token" : "elasticsearch",
    "start_offset" : 0,
    "end_offset" : 13,
    "type" : "<ALPHANUM>",
    "position" : 1
  } ]
}
```

We can see that the word is the same as the original one that we passed to the query. We won't get into Lucene query details and how the query parser constructed the query, but in general, the indexed term after analysis was the same as the one in the query after analysis; so, the document matched the query and the result was returned.

URI query string parameters

There are a few parameters that we can use to control the URI query behavior, which we will discuss now. Each parameter in the query should be concatenated with the & character, as shown in the following example:

```
curl -XGET  
'localhost:9200/books/_search?pretty&q=published:  
2013&df=title&explain=true&default_operator=AND'
```

Please also remember about the ' characters because on Linux-based systems, the & character will be analyzed by the Linux shell.

The query

The q parameter allows us to specify the query that we want our documents to match. It allows us to specify the query using the Lucene query syntax described in the *The Lucene query syntax* section in this chapter. For example, a simple query could look like q=title:elasticsearch.

The default search field

By using the df parameter, we can specify the default search field that should be used when no field indicator is used in the q parameter. By default, the _all field will be used (the field that Elasticsearch uses to copy the content of all the other fields. We will discuss this in greater depth in the *Extending your index structure with additional internal information* section in Chapter 2, Indexing Your Data). An example of the df parameter value can be df=title.

Analyzer

The analyzer property allows us to define the name of the analyzer that should be used to analyze our query. By default, our query will be analyzed by the same analyzer that was used to analyze the field contents during indexing.

The default operator

The default_operator property which can be set to OR or AND allows us to specify the default Boolean operator used for our query. By default, it is set to OR, which means that a single query term match will be enough for a document to be returned. Setting this parameter to AND for a query will result in the returning of documents that match all the query terms.

Query explanation

If we set the `explain` parameter to `true`, Elasticsearch will include additional `explain` information with each document in the result—such as the shard, from which the document was fetched, and detailed information about the scoring calculation (we will talk more about it in the *Understanding the explain information* section in *Chapter 5, Make Your Search Better*). Also remember not to fetch the `explain` information during normal search queries because it requires additional resources and adds performance degradation to the queries. For example, a single result can look like the following code:

```
{  
  "_shard" : 3,  
  "_node" : "kyuzK62NQcGJyhc2gI1P2w",  
  "_index" : "books",  
  "_type" : "es",  
  "_id" : "2",  
  "_score" : 0.19178301, "_source" : {"title":"Mastering  
  Elasticsearch", "published": 2013},  
  "_explanation" : {  
    "value" : 0.19178301,  
    "description" : "weight(title:elasticsearch in 0)  
      [PerFieldSimilarity], result of:",  
    "details" : [ {  
      "value" : 0.19178301,  
      "description" : "fieldWeight in 0, product of:",  
      "details" : [ {  
        "value" : 1.0,  
        "description" : "tf(freq=1.0), with freq of:",  
        "details" : [ {  
          "value" : 1.0,  
          "description" : "termFreq=1.0"  
        } ]  
      }, {  
        "value" : 0.30685282,  
        "description" : "norm(1.0)"  
      } ]  
    } ]  
  } }  
}
```

```

    "description" : "idf(docFreq=1, maxDocs=1)"
}, {
    "value" : 0.625,
    "description" : "fieldNorm(doc=0)"
} ]
} ]
}
}

```

The fields returned

By default, for each document returned, Elasticsearch will include the index name, type name, document identifier, score, and the `_source` field. We can modify this behavior by adding the `fields` parameter and specifying a comma-separated list of field names. The field will be retrieved from the stored fields (if they exist) or from the internal `_source` field. By default, the value of the `fields` parameter is `_source`. An example can be like this `fields=title`.



We can also disable the fetching of the `_source` field by adding the `_source` parameter with its value set to `false`.



Sorting the results

By using the `sort` parameter, we can specify custom sorting. The default behavior of Elasticsearch is to sort the returned documents by their score in the descending order. If we would like to sort our documents differently, we need to specify the `sort` parameter. For example, adding `sort=published:desc` will sort the documents by the `published` field in the descending order. By adding the `sort=published:asc` parameter, we will tell Elasticsearch to sort the documents on the basis of the `published` field in the ascending order.

If we specify custom sorting, Elasticsearch will omit the `_score` field calculation for documents. This may not be the desired behavior in your case. If you want to still keep a track of the scores for each document when using custom sort, you should add the `track_scores=true` property to your query. Please note that tracking the scores when doing custom sorting will make the query a little bit slower (you may even not notice it) due to the processing power needed to calculate the score.

The search timeout

By default, Elasticsearch doesn't have timeout for queries, but you may want your queries to timeout after a certain amount of time (for example, 5 seconds). Elasticsearch allows you to do this by exposing the `timeout` parameter. When the `timeout` parameter is specified, the query will be executed up to a given `timeout` value, and the results that were gathered up to that point will be returned. To specify a timeout of 5 seconds, you will have to add the `timeout=5s` parameter to your query.

The results window

Elasticsearch allows you to specify the results window (the range of documents in the results list that should be returned). We have two parameters that allow us to specify the results window size: `size` and `from`. The `size` parameter defaults to 10 and defines the maximum number of results returned. The `from` parameter defaults to 0 and specifies from which document the results should be returned. In order to return five documents starting from the eleventh one, we will add the following parameters to the query: `size=5&from=10`.

The search type

The URI query allows us to specify the search type by using the `search_type` parameter, which defaults to `query_then_fetch`. There are six values that we can use: `dfs_query_then_fetch`, `dfs_query_and_fetch`, `query_then_fetch`, `query_and_fetch`, `count`, and `scan`. We'll learn more about search types in the *Understanding the querying process* section in *Chapter 3, Searching Your Data*.

Lowercasing the expanded terms

Some of the queries use query expansion, such as the prefix query. We will discuss this in the *Query rewrite* section of *Chapter 3, Searching Your Data*. We are allowed to define whether the expanded terms should be lowercased or not by using the `lowercase_expanded_terms` property. By default, the `lowercase_expanded_terms` property is set to `true`, which means that the expanded terms will be lowercased.

Analyzing the wildcard and prefixes

By default, the wildcard queries and the prefix queries are not analyzed. If we want to change this behavior, we can set the `analyzeWildcard` property to `true`.

The Lucene query syntax

We thought that it will be good to know a bit more about what syntax can be used in the `q` parameter passed in the URI query. Some of the queries in Elasticsearch (such as the one currently discussed) support the Lucene query parsers syntax—the language that allows you to construct queries. Let's take a look at it and discuss some basic features. To read about the full Lucene query syntax, please go to the following web page: http://lucene.apache.org/core/4_6_1/queryparser/org/apache/lucene/queryparser/classic/package-summary.html.

A query that we pass to Lucene is divided into terms and operators by the query parser. Let's start with the terms—you can distinguish them into two types—single terms and phrases. For example, to query for a term `book` in the `title` field, we will pass the following query:

```
title:book
```

To query for a phrase `elasticsearch book` in the `title` field, we will pass the following query:

```
title:"elasticsearch book"
```

You may have noticed the name of the field in the beginning and in the term or phrase later.

As we already said, the Lucene query syntax supports operators. For example, the `+` operator tells Lucene that the given part must be matched in the document. The `-` operator is the opposite, which means that such a part of the query can't be present in the document. A part of the query without the `+` or `-` operator will be treated as the given part of the query that can be matched but it is not mandatory. So, if we would like to find a document with the term `book` in the `title` field and without the term `cat` in the `description` field, we will pass the following query:

```
+title:book -description:cat
```

We can also group multiple terms with parenthesis, as shown in the following query:

```
title:(crime punishment)
```

We can also boost parts of the query with the `^` operator and the boost value after it, as shown in the following query:

```
title:book^4
```

Summary

In this chapter, we learned what full text search is and how Apache Lucene fits in there. In addition to this, we are now familiar with the basic concepts of Elasticsearch and its top-level architecture. We used the Elasticsearch REST API not only to index data but also to update it, retrieve it, and finally delete it. Finally, we searched our data using the simple URI query. In the next chapter, we'll focus on indexing our data. We will see how Elasticsearch indexing works and what is the role of primary shard and its replicas. We'll see how Elasticsearch handles the data that it doesn't know or how to create our own mappings—the JSON structure that describes the structure of our index. We'll also learn how to use batch indexing to speed up the indexing process and what additional information can be stored along with our index to help us achieve our goal. In addition, we will discuss what an index segment is, what segment merging is, and how to tune the segment. Finally, we'll see how routing works in Elasticsearch and what options we have when it comes to both indexing and querying routing.

2

Indexing Your Data

In the previous chapter, we learned the basics about full text search and Elasticsearch. We also saw what Apache Lucene is. In addition to that, we saw how to install Elasticsearch, what the standard directory layout is, and what to pay attention to. We created an index, and we indexed and updated our data. Finally, we used the simple URI query to get data from Elasticsearch. By the end of this chapter, you will learn the following topics:

- Elasticsearch indexing
- Configuring your index structure mappings and knowing what field types we are allowed to use
- Using batch indexing to speed up the indexing process
- Extending your index structure with additional internal information
- Understanding what segment merging is, how to configure it, and what throttling is
- Understanding how routing works and how we can configure it to our needs

Elasticsearch indexing

We have our Elasticsearch cluster up and running, and we also know how to use the Elasticsearch REST API to index our data, delete it, and retrieve it. We also know how to use search to get our documents. If you are used to SQL databases, you might know that before you can start putting the data there, you need to create a structure, which will describe what your data looks like. Although Elasticsearch is a schema-less search engine and can figure out the data structure on the fly, we think that controlling the structure and thus defining it ourselves is a better way. In the following few pages, you'll see how to create new indices (and how to delete them). Before we look closer at the available API methods, let's see what the indexing process looks like.

Shards and replicas

As you recollect from the previous chapter, the Elasticsearch index is built of one or more shards and each of them contains part of your document set. Each of these shards can also have replicas, which are exact copies of the shard. During index creation, we can specify how many shards and replicas should be created. We can also omit this information and use the default values either defined in the global configuration file (`elasticsearch.yml`) or implemented in Elasticsearch internals. If we rely on Elasticsearch defaults, our index will end up with five shards and one replica. What does that mean? To put it simply, we will end up with having 10 Lucene indices distributed among the cluster.

 Are you wondering how we did the calculation and got 10 Lucene indices from five shards and one replica? The term "replica" is somewhat misleading. It means that *every* shard has its copy, so it means there are five shards and five copies.

Having a shard and its replica, in general, means that when we index a document, we will modify them both. That's because to have an exact copy of a shard, Elasticsearch needs to inform all the replicas about the change in shard contents. In the case of fetching a document, we can use either the shard or its copy. In a system with many physical nodes, we will be able to place the shards and their copies on different nodes and thus use more processing power (such as disk I/O or CPU). To sum up, the conclusions are as follows:

- More shards allow us to spread indices to more servers, which means we can handle more documents without losing performance.
- More shards means that fewer resources are required to fetch a particular document because fewer documents are stored in a single shard compared to the documents stored in a deployment with fewer shards.
- More shards means more problems when searching across the index because we have to merge results from more shards and thus the aggregation phase of the query can be more resource intensive.
- Having more replicas results in a fault tolerance cluster, because when the original shard is not available, its copy will take the role of the original shard. Having a single replica, the cluster may lose the shard without data loss. When we have two replicas, we can lose the primary shard and its single replica and still everything will work well.

- The more the replicas, the higher the query throughput will be. That's because the query can use either a shard or any of its copies to execute the query.

Of course, these are not the only relationships between the number of shards and replicas in Elasticsearch. We will talk about most of them later in the book.

So, how many shards and replicas should we have for our indices? That depends. We believe that the defaults are quite good but nothing can replace a good test. Note that the number of replicas is less important because you can adjust it on a live cluster after index creation. You can remove and add them if you want and have the resources to run them. Unfortunately, this is not true when it comes to the number of shards. Once you have your index created, the only way to change the number of shards is to create another index and reindex your data.

Creating indices

When we created our first document in Elasticsearch, we didn't care about index creation at all. We just used the following command:

```
curl -XPUT http://localhost:9200/blog/article/1 -d '{"title": "New version of Elasticsearch released!", "content": "...", "tags": ["announce", "elasticsearch", "release"] }'
```

This is fine. If such an index does not exist, Elasticsearch automatically creates the index for us. We can also create the index ourselves by running the following command:

```
curl -XPUT http://localhost:9200/blog/
```

We just told Elasticsearch that we want to create the index with the `blog` name. If everything goes right, you will see the following response from Elasticsearch:

```
{"acknowledged":true}
```

When is manual index creation necessary? There are many situations. One of them can be the inclusion of additional settings such as the index structure or the number of shards.

Altering automatic index creation

Sometimes, you can come to the conclusion that automatic index creation is a bad thing. When you have a big system with many processes sending data into Elasticsearch, a simple typo in the index name can destroy hours of script work. You can turn off automatic index creation by adding the following line in the `elasticsearch.yml` configuration file:

```
action.auto_create_index: false
```

Note that `action.auto_create_index` is more complex than it looks. The value can be set to not only `false` or `true`. We can also use index name patterns to specify whether an index with a given name can be created automatically if it doesn't exist. For example, the following definition allows automatic creation of indices with the names beginning with `a`, but disallows the creation of indices starting with `an`. The other indices aren't allowed and must be created manually (because of `-*`).

```
action.auto_create_index: -an*,+a*,-*
```

Note that the order of pattern definitions matters. Elasticsearch checks the patterns up to the first pattern that matches, so if you move `-an*` to the end, it won't be used because of `+a*`, which will be checked first.



Settings for a newly created index

The manual creation of an index is also necessary when you want to set some configuration options, such as the number of shards and replicas. Let's look at the following example:

```
curl -XPUT http://localhost:9200/blog/ -d '{  
  "settings" : {  
    "number_of_shards" : 1,  
    "number_of_replicas" : 2  
  }  
}'
```

The preceding command will result in the creation of the `blog` index with one shard and two replicas, so it makes a total of three physical Lucene indices. Also, there are other values that can be set in this way; we will talk about those later in the book.

So, we already have our new, shiny index. But there is a problem; we forgot to provide the mappings, which are responsible for describing the index structure. What can we do? Since we have no data at all, we'll go for the simplest approach – we will just delete the index. To do that, we will run a command similar to the preceding one, but instead of using the PUT HTTP method, we use DELETE. So the actual command is as follows:

```
curl -XDELETE http://localhost:9200/posts
```

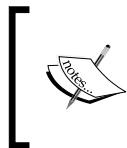
And the response will be the same as the one we saw earlier, as follows:

```
{"acknowledged":true}
```

Now that we know what an index is, how to create it, and how to delete it, we are ready to create indices with the mappings we have defined. It is a very important part because data indexation will affect the search process and the way in which documents are matched.

Mappings configuration

If you are used to SQL databases, you may know that before you can start inserting the data in the database, you need to create a schema, which will describe what your data looks like. Although Elasticsearch is a schema-less search engine and can figure out the data structure on the fly, we think that controlling the structure and thus defining it ourselves is a better way. In the following few pages, you'll see how to create new indices (and how to delete them) and how to create mappings that suit your needs and match your data structure.



Note that we didn't include all the information about the available types in this chapter and some features of Elasticsearch, such as nested type, parent-child handling, storing geographical points, and search, are described in the following chapters of this book.

Type determining mechanism

Before we start describing how to create mappings manually, we wanted to write about one thing. Elasticsearch can guess the document structure by looking at JSON, which defines the document. In JSON, strings are surrounded by quotation marks, Booleans are defined using specific words, and numbers are just a few digits. This is a simple trick, but it usually works. For example, let's look at the following document:

```
{
  "field1": 10,
```

```
"field2": "10"  
}
```

The preceding document has two fields. The `field1` field will be determined as a number (to be precise, as `long` type), but `field2` will be determined as a string, because it is surrounded by quotation marks. Of course, this can be the desired behavior, but sometimes the data source may omit the information about the data type and everything may be present as strings. The solution to this is to enable more aggressive text checking in the mapping definition by setting the `numeric_detection` property to `true`. For example, we can execute the following command during the creation of the index:

```
curl -XPUT http://localhost:9200/blog/?pretty -d '{  
  "mappings" : {  
    "article": {  
      "numeric_detection" : true  
    }  
  }  
}'
```

Unfortunately, the problem still exists if we want the Boolean type to be guessed. There is no option to force the guessing of Boolean types from the text. In such cases, when a change of source format is impossible, we can only define the field directly in the `mappings` definition.

Another type that causes trouble is a date-based one. Elasticsearch tries to guess dates given as timestamps or strings that match the date format. We can define the list of recognized date formats using the `dynamic_date_formats` property, which allows us to specify the formats array. Let's look at the following command for creating the index and type:

```
curl -XPUT 'http://localhost:9200/blog/' -d '{  
  "mappings" : {  
    "article" : {  
      "dynamic_date_formats" : ["yyyy-MM-dd hh:mm"]  
    }  
  }  
}'
```

The preceding command will result in the creation of an index called `blog` with the single type called `article`. We've also used the `dynamic_date_formats` property with a single date format that will result in Elasticsearch using the `date` core type (please refer to the *Core types* section in this chapter for more information about field types) for fields matching the defined format. Elasticsearch uses the `joda-time` library to define date formats, so please visit <http://joda-time.sourceforge.net/api-release/org/joda/time/format/DateTimeFormat.html> if you are interested in finding out more about them.



Remember that the `dynamic_date_format` property accepts an array of values. That means that we can handle several date formats simultaneously.



Disabling field type guessing

Let's think about the following case. First we index a number, an integer.

Elasticsearch will guess its type and will set the type to `integer` or `long` (refer to the *Core types* section in this chapter for more information about field types). What will happen if we index a document with a floating point number into the same field? Elasticsearch will just remove the decimal part of the number and store the rest. Another reason for turning it off is when we don't want to add new fields to an existing index – the fields that were not known during application development.

To turn off automatic field adding, we can set the `dynamic` property to `false`.

We can add the `dynamic` property as the `type` property. For example, if we would like to turn off automatic field type guessing for the `article` type in the `blog` index, our command will look as follows:

```
curl -XPUT 'http://localhost:9200/blog/' -d '{
  "mappings" : {
    "article" : {
      "dynamic" : "false",
      "properties" : {
        "id" : { "type" : "string" },
        "content" : { "type" : "string" },
        "author" : { "type" : "string" }
      }
    }
  }
}'
```

```
        }
    }
}
}'
```

After creating the `blog` index using the preceding command, any field that is not mentioned in the `properties` section (we will discuss this in the next section) will be ignored by Elasticsearch. So any field apart from `id`, `content`, and `author` will just be ignored. Of course, this is only true for the `article` type in the `blog` index.

Index structure mapping

The schema mapping (or in short, mappings) is used to define the index structure. As you may recall, each index can have multiple types, but we will concentrate on a single type for now—just for simplicity. Let's assume that we want to create an index called `posts` that will hold data for `blog` posts. It could have the following structure:

- Unique identifier
- Name
- Publication date
- Contents

In Elasticsearch, mappings are sent as JSON objects in a file. So, let's create a mapping file that will match the aforementioned needs—we will call it `posts.json`. Its content is as follows:

```
{
  "mappings": {
    "post": {
      "properties": {
        "id": {"type": "long", "store": "yes",
        "precision_step": "0" },
        "name": {"type": "string", "store": "yes",
        "index": "analyzed" },
        "published": {"type": "date", "store": "yes",
        "precision_step": "0" },
        "contents": {"type": "string", "store": "no",
        "index": "analyzed" }
      }
    }
  }
}
```

To create our `posts` index with the preceding file, run the following command (assuming that we stored the mappings in the `posts.json` file):

```
curl -XPOST 'http://localhost:9200/posts' -d @posts.json
```



Note that you can store your mappings and set a file named anyway you want.



And again, if everything goes well, we see the following response:

```
{"acknowledged":true}
```

Now we have our index structure and we can index our data. Let's take a break to discuss the contents of the `posts.json` file.

Type definition

As you can see, the contents of the `posts.json` file are JSON objects and therefore it starts and ends with curly brackets (if you want to learn more about JSON, please visit <http://www.json.org/>). All the type definitions inside the mentioned file are nested in the `mappings` object. You can define multiple types inside the `mappings` JSON object. In our example, we have a single `post` type. But, for example, if we would also like to include the `user` type, the file will look as follows:

```
{
  "mappings": {
    "post": {
      "properties": {
        "id": { "type": "long", "store": "yes",
        "precision_step": "0" },
        "name": { "type": "string", "store": "yes",
        "index": "analyzed" },
        "published": { "type": "date", "store": "yes",
        "precision_step": "0" },
        "contents": { "type": "string", "store": "no",
        "index": "analyzed" }
      }
    },
    "user": {
      "properties": {
        "id": { "type": "long", "store": "yes",
        "precision_step": "0" },
        "name": { "type": "string", "store": "yes",
        "index": "analyzed" }
      }
    }
  }
}
```

```
        "index": "analyzed" }  
    }  
}  
}  
}
```

Fields

Each type is defined by a set of properties – fields that are nested inside the `properties` object. So let's concentrate on a single field now; for example, the `contents` field, whose definition is as follows:

```
"contents": { "type": "string", "store": "yes", "index": "analyzed" }
```

It starts with the name of the field, which is `contents` in the preceding case. After the name of the field, we have an object defining the behavior of the field. The attributes are specific to the types of fields we are using and we will discuss them in the next section. Of course, if you have multiple fields for a single type (which is what we usually have), remember to separate them with a comma.

Core types

Each field type can be specified to a specific core type provided by Elasticsearch. The core types in Elasticsearch are as follows:

- String
- Number
- Date
- Boolean
- Binary

So, now let's discuss each of the core types available in Elasticsearch and the attributes it provides to define their behavior.

Common attributes

Before continuing with all the core type descriptions, we would like to discuss some common attributes that you can use to describe all the types (except for the binary one).

- `index_name`: This defines the name of the field that will be stored in the index. If this is not defined, the name will be set to the name of the object that the field is defined with.

- **index**: This can take the values `analyzed` and `no`. Also, for string-based fields, it can also be set to `not_analyzed`. If set to `analyzed`, the field will be indexed and thus searchable. If set to `no`, you won't be able to search on such a field. The default value is `analyzed`. In the case of string-based fields, there is an additional option, `not_analyzed`. This, when set, will mean that the field will be indexed but not analyzed. So, the field is written in the index as it was sent to Elasticsearch and only a perfect match will be counted during a search. Setting the `index` property to `no` will result in the disabling of the `include_in_all` property of such a field.
- **store**: This can take the values `yes` and `no` and specifies if the original value of the field should be written into the index. The default value is `no`, which means that you can't return that field in the results (although, if you use the `_source` field, you can return the value even if it is not stored), but if you have it indexed, you can still search the data on the basis of it.
- **boost**: The default value of this attribute is `1`. Basically, it defines how important the field is inside the document; the higher the boost, the more important the values in the field.
- **null_value**: This attribute specifies a value that should be written into the index in case that field is not a part of an indexed document. The default behavior will just omit that field.
- **copy_to**: This attribute specifies a field to which all field values will be copied.
- **include_in_all**: This attribute specifies if the field should be included in the `_all` field. By default, if the `_all` field is used, all the fields will be included in it. The `_all` field will be described in more detail in the *Extending your index structure with additional internal information* section.

String

String is the most basic text type, which allows us to store one or more characters inside it. A sample definition of such a field can be as follows:

```
"contents" : { "type" : "string", "store" : "no", "index" :  
  "analyzed" }
```

In addition to the common attributes, the following attributes can also be set for string-based fields:

- **term_vector**: This attribute can take the values `no` (the default one), `yes`, `with_offsets`, `with_positions`, and `with_positions_offsets`. It defines whether or not to calculate the Lucene term vectors for that field. If you are using highlighting, you will need to calculate the term vector.

- `omit_norms`: This attribute can take the value `true` or `false`. The default value is `false` for string fields that are analyzed and `true` for string fields that are indexed but not analyzed. When this attribute is set to `true`, it disables the Lucene norms calculation for that field (and thus you can't use index-time boosting), which can save memory for fields used only in filters (and thus not being taken into consideration when calculating the score of the document).
- `analyzer`: This attribute defines the name of the analyzer used for indexing and searching. It defaults to the globally-defined analyzer name.
- `index_analyzer`: This attribute defines the name of the analyzer used for indexing.
- `search_analyzer`: This attribute defines the name of the analyzer used for processing the part of the query string that is sent to a particular field.
- `norms.enabled`: This attribute specifies whether the norms should be loaded for a field. By default, it is set to `true` for analyzed fields (which means that the norms will be loaded for such fields) and to `false` for non-analyzed fields.
- `norms.loading`: This attribute takes the values `eager` and `lazy`. The first value means that the norms for such fields are always loaded. The second value means that the norms will be loaded only when needed.
- `position_offset_gap`: This attribute defaults to `0` and specifies the gap in the index between instances of the given field with the same name. Setting this to a higher value may be useful if you want position-based queries (like phrase queries) to match only inside a single instance of the field.
- `index_options`: This attribute defines the indexing options for the postings list—the structure holding the terms (we will talk about this more in *The postings format* section of this chapter). The possible values are `docs` (only document numbers are indexed), `freqs` (document numbers and term frequencies are indexed), `positions` (document numbers, term frequencies, and their positions are indexed), and `offsets` (document numbers, term frequencies, their positions, and offsets are indexed). The default value for this property is `positions` for analyzed fields and `docs` for fields that are indexed but not analyzed.
- `ignore_above`: This attribute defines the maximum size of the field in characters. Fields whose size is above the specified value will be ignored by the analyzer.

Number

This is the core type that gathers all numeric field types that are available to be used. The following types are available in Elasticsearch (we specify them by using the `type` property):

- `byte`: This type defines a `byte` value; for example, `1`
- `short`: This type defines a `short` value; for example, `12`
- `integer`: This type defines a `integer` value; for example, `134`
- `long`: This type defines a `long` value; for example, `123456789`
- `float`: This type defines a `float` value; for example, `12.23`
- `double`: This type defines a `double` value; for example, `123.45`

 You can learn more about the mentioned Java types at <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>.

A sample definition of a field based on one of the numeric types is as follows:

```
"price" : { "type" : "float", "store" : "yes", "precision_step" : "4" }
```

In addition to the common attributes, the following ones can also be set for the numeric fields:

- `precision_step`: This attribute specifies the number of terms generated for each value in a field. The lower the value, the higher the number of terms generated. For fields with a higher number of terms per value, range queries will be faster at the cost of a slightly larger index. The default value is `4`.
- `ignore_malformed`: This attribute can take the value `true` or `false`. The default value is `false`. It should be set to `true` in order to omit badly formatted values.

Boolean

The `boolean` core type is designed for indexing Boolean values (`true` or `false`). A sample definition of a field based on the `boolean` type can be as follows:

```
"allowed" : { "type" : "boolean", "store": "yes" }
```

Binary

The binary field is a Base64 representation of the binary data stored in the index. You can use it to store data that is normally written in binary form, such as images. Fields based on this type are by default stored and not indexed, so you can only retrieve them and cannot perform search operations on them. The binary type only supports the `index_name` property. The sample field definition based on the `binary` field may look like the following:

```
"image" : { "type" : "binary" }
```

Date

The date core type is designed to be used for date indexing. It follows a specific format that can be changed and is stored in UTC by default.

The default date format understood by Elasticsearch is quite universal and allows the specifying of the date and optionally the time, for example, `2012-12-24T12:10:22`. A sample definition of a field based on the `date` type is as follows:

```
"published" : { "type" : "date", "store" : "yes", "format" :  
  "YYYY-mm-dd" }
```

A sample document that uses the preceding field is as follows:

```
{  
  "name" : "Sample document",  
  "published" : "2012-12-22"  
}
```

In addition to the common attributes, the following ones can also be set for the fields based on the `date` type:

- `format`: This attribute specifies the format of the date. The default value is `dateOptionalTime`. For a full list of formats, please visit <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/mapping-date-format.html>.
- `precision_step`: This attribute specifies the number of terms generated for each value in that field. The lower the value, the higher the number of terms generated, and thus the faster the range queries (but with a higher index size). The default value is 4.
- `ignore_malformed`: This attribute can take the value `true` or `false`. The default value is `false`. It should be set to `true` in order to omit badly formatted values.

Multifields

Sometimes, you would like to have the same field values in two fields; for example, one for searching and one for sorting, or one analyzed with the language analyzer and one only on the basis of whitespace characters. Elasticsearch addresses this need by allowing the addition of the `fields` object to the field definition. It allows the mapping of several core types into a single field and having them analyzed separately. For example, if we would like to calculate facetting and search on our name field, we can define the following field:

```
"name": {
  "type": "string",
  "fields": {
    "facet": { "type" : "string", "index": "not_analyzed" }
  }
}
```

The preceding definition will create two fields: we will refer to the first as `name` and the second as `name.facet`. Of course, you don't have to specify two separate fields during indexing—a single one named `name` is enough; Elasticsearch will do the rest, which means copying the value of the field to all the fields from the preceding definition.

The IP address type

The `ip` field type was added to Elasticsearch to simplify the use of IPv4 addresses in a numeric form. This field type allows us to search data that is indexed as an IP address, sort on this data, and use range queries using IP values.

A sample definition of a field based on one of the numeric types is as follows:

```
"address" : { "type" : "ip", "store" : "yes" }
```

In addition to the common attributes, the `precision_step` attribute can also be set for the numeric fields. This attribute specifies the number of terms generated for each value in a field. The lower the value, the higher the number of terms generated. For fields with a higher number of terms per value, range queries will be faster at the cost of a slightly larger index. The default value is 4.

A sample document that uses the preceding field is as follows:

```
{
  "name" : "Tom PC",
  "address" : "192.168.2.123"
}
```

The `token_count` type

The `token_count` field type allows us to store index information about how many words the given field has instead of storing and indexing the text provided to the field. It accepts the same configuration options as the `number` type, but in addition to that, it allows us to specify the analyzer by using the `analyzer` property.

A sample definition of a field based on the `token_count` field type looks as follows:

```
"address_count" : { "type" : "token_count", "store" : "yes" }
```

Using analyzers

As we mentioned, for the fields based on the `string` type, we can specify which analyzer the Elasticsearch should use. As you remember from the *Full-text searching* section of *Chapter 1, Getting Started with the Elasticsearch Cluster*, the analyzer is a functionality that is used to analyze data or queries in a way we want. For example, when we divide words on the basis of whitespaces and lowercase characters, we don't have to worry about users sending words in lowercase or uppercase. Elasticsearch allows us to use different analyzers at the time of indexing and different analyzers at the time of querying – we can choose how we want our data to be processed at each stage of the search process. To use one of the analyzers, we just need to specify its name to the correct property of the field and that's all.

Out-of-the-box analyzers

Elasticsearch allows us to use one of the many analyzers defined by default. The following analyzers are available out of the box:

- `standard`: This is a standard analyzer that is convenient for most European languages (please refer to <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-standard-analyzer.html> for the full list of parameters).
- `simple`: This is an analyzer that splits the provided value depending on non-letter characters and converts them to lowercase.
- `whitespace`: This is an analyzer that splits the provided value on the basis of whitespace characters
- `stop`: This is similar to a `simple` analyzer, but in addition to the functionality of the `simple` analyzer, it filters the data on the basis of the provided set of stop words (please refer to <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-stop-analyzer.html> for the full list of parameters).

- keyword: This is a very simple analyzer that just passes the provided value. You'll achieve the same by specifying a particular field as `not_analyzed`.
- pattern: This is an analyzer that allows flexible text separation by the use of regular expressions (please refer to <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-pattern-analyzer.html> for the full list of parameters).
- language: This is an analyzer that is designed to work with a specific language. The full list of languages supported by this analyzer can be found at <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-lang-analyzer.html>.
- snowball: This is an analyzer that is similar to `standard`, but additionally provides the stemming algorithm (please refer to <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-snowball-analyzer.html> for the full list of parameters).



Stemming is the process of reducing inflected and derived words to their stem or base form. Such a process allows for the reduction of words, for example, with cars and car. For the mentioned words, **stemmer** (which is an implementation of the stemming algorithm) will produce a single stem, car. After indexing, documents containing such words will be matched while using any of them. Without stemming, documents with the word "cars" will only be matched by a query containing the same word.

Defining your own analyzers

In addition to the analyzers mentioned previously, Elasticsearch allows us to define new ones without the need to write a single line of Java code. In order to do that, we need to add an additional section to our `mappings` file; that is, the `settings` section, which holds useful information required by Elasticsearch during index creation. The following is how we define our custom `settings` section:

```
"settings" : {
  "index" : {
    "analysis": {
      "analyzer": {
        "en": {
          "tokenizer": "standard",
          "filter": [
            "asciifolding",
            "lowercase",

```

```
        "ourEnglishFilter"
    ]
}
},
"filter": {
    "ourEnglishFilter": {
        "type": "kstem"
    }
}
}
}
```

We specified that we want a new analyzer named `en` to be present. Each analyzer is built from a single tokenizer and multiple filters. A complete list of default filters and tokenizers can be found at <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis.html>. Our `en` analyzer includes the standard tokenizer and three filters: `asciifolding` and `lowercase`, which are the ones available by default, and `ourEnglishFilter`, which is a filter we have defined.

To define a filter, we need to provide its name, its type (the `type` property), and any number of additional parameters required by that filter type. The full list of filter types available in Elasticsearch can be found at <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis.html>. This list changes constantly, so we'll skip commenting on it.

So, the final `mappings` file with the analyzer defined will be as follows:

```
{
    "settings" : {
        "index" : {
            "analysis": {
                "analyzer": {
                    "en": {
                        "tokenizer": "standard",
                        "filter": [
                            "asciifolding",
                            "lowercase",
                            "ourEnglishFilter"
                        ]
                    }
                },
                "filter": {
                    "ourEnglishFilter": {

```

```
        "type": "kstem"
    }
}
}
},
"mappings" : {
    "post" : {
        "properties" : {
            "id": { "type" : "long", "store" : "yes",
            "precision_step" : "0" },
            "name": { "type" : "string", "store" : "yes", "index" :
            "analyzed", "analyzer": "en" }
        }
    }
}
}
```

We can see how our analyzer works by using the Analyze API (<http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/indices-analyze.html>). For example, let's look at the following command:

```
curl -XGET 'localhost:9200/posts/_analyze?pretty&field=post.name' -d
'robots cars'
```

The command asks Elasticsearch to show the content of the analysis of the given phrase (robots cars) with the use of the analyzer defined for the post type and its name field. The response that we will get from Elasticsearch is as follows:

```
{
    "tokens" : [ {
        "token" : "robot",
        "start_offset" : 0,
        "end_offset" : 6,
        "type" : "<ALPHANUM>",
        "position" : 1
    }, {
        "token" : "car",
        "start_offset" : 7,
        "end_offset" : 11,
        "type" : "<ALPHANUM>",

[ 61 ]
```

```
    "position" : 2
  } ]
}
```

As you can see, the `robots cars` phrase was divided into two tokens. In addition to that, the `robots` word was changed to `robot` and the `cars` word was changed to `car`.

Analyzer fields

An analyzer field (`_analyzer`) allows us to specify a field value that will be used as the analyzer name for the document to which the field belongs. Imagine that you have a software running that detects the language the document is written in and you store that information in the `language` field in the document. In addition to that, you would like to use that information to choose the right analyzer. To do that, just add the following lines to your `mappings` file:

```
"_analyzer" : {
  "path" : "language"
}
```

The `mappings` file that includes the preceding information is as follows:

```
{
  "mappings" : {
    "post" : {
      "_analyzer" : {
        "path" : "language"
      },
      "properties" : {
        "id": { "type" : "long", "store" : "yes",
        "precision_step" : "0" },
        "name": { "type" : "string", "store" : "yes",
        "index" : "analyzed" },
        "language": { "type" : "string", "store" : "yes",
        "index" : "not_analyzed" }
      }
    }
  }
}
```

Note that there has to be an analyzer defined with the same name as the value provided in the `language` field or else the indexing will fail.

Default analyzers

There is one more thing to say about analyzers—the ability to specify the analyzer that should be used by default if no analyzer is defined. This is done in the same way as we configured a custom analyzer in the `settings` section of the `mappings` file, but instead of specifying a custom name for the analyzer, a `default` keyword should be used. So to make our previously defined analyzer the default, we can change the `en` analyzer to the following:

```
{
  "settings" : {
    "index" : {
      "analysis": {
        "analyzer": {
          "default": {
            "tokenizer": "standard",
            "filter": [
              "asciifolding",
              "lowercase",
              "ourEnglishFilter"
            ]
          }
        },
        "filter": {
          "ourEnglishFilter": {
            "type": "kstem"
          }
        }
      }
    }
  }
}
```

Different similarity models

With the release of Apache Lucene 4.0 in 2012, all the users of this great full text search library were given the opportunity to alter the default TF/IDF-based algorithm (we've mentioned it in the *Full-text searching* section of *Chapter 1, Getting Started with the Elasticsearch Cluster*. However, it was not the only change. Lucene 4.0 was shipped with additional similarity models, which basically allows us to use different scoring formulas for our documents.

Setting per-field similarity

Since Elasticsearch 0.90, we are allowed to set a different similarity for each of the fields that we have in our `mappings` file. For example, let's assume that we have the following simple mappings that we use in order to index blog posts:

```
{  
  "mappings" : {  
    "post" : {  
      "properties" : {  
        "id" : { "type" : "long", "store" : "yes",  
                 "precision_step" : "0" },  
        "name" : { "type" : "string", "store" : "yes", "index" :  
                   "analyzed" },  
        "contents" : { "type" : "string", "store" : "no",  
                      "index" : "analyzed" }  
      }  
    }  
  }  
}
```

To do this, we will use the `BM25` similarity model for the `name` field and the `contents` field. In order to do that, we need to extend our field definitions and add the `similarity` property with the value of the chosen similarity name. Our changed mappings will look like the following:

```
{  
  "mappings" : {  
    "post" : {  
      "properties" : {  
        "id" : { "type" : "long", "store" : "yes",  
                  "precision_step" : "0" },  
        "name" : { "type" : "string", "store" : "yes",  
                   "index" : "analyzed", "similarity" : "BM25" },  
        "contents" : { "type" : "string", "store" : "no",  
                      "index" : "analyzed", "similarity" : "BM25" }  
      }  
    }  
  }  
}
```

And that's all, nothing more is needed. After the preceding change, Apache Lucene will use the `BM25` similarity to calculate the score factor for the `name` and `contents` fields.

Available similarity models

The three new similarity models available are as follows:

- **Okapi BM25 model:** This similarity model is based on a probabilistic model that estimates the probability of finding a document for a given query. In order to use this similarity in Elasticsearch, you need to use the `BM25` name. The Okapi BM25 similarity is said to perform best when dealing with short text documents where term repetitions are especially hurtful to the overall document score. To use this similarity, you need to set the `similarity` property for a field to `BM25`. This similarity is defined out of the box and doesn't need additional properties to be set.
- **Divergence from randomness model:** This similarity model is based on the probabilistic model of the same name. In order to use this similarity in Elasticsearch, you need to use the `DFR` name. It is said that the divergence from randomness similarity model performs well on text that is similar to natural language.
- **Information-based model:** This is the last model of the newly introduced similarity models and is very similar to the divergence from randomness model. In order to use this similarity in Elasticsearch, you need to use the `IB` name. Similar to the `DFR` similarity, it is said that the information-based model performs well on data similar to natural language text.

Configuring DFR similarity

In the case of the `DFR` similarity, we can configure the `basic_model` property (which can take the value `be`, `d`, `g`, `if`, `in`, or `ine`), the `after_effect` property (with values of `no`, `b`, or `l`), and the `normalization` property (which can be `no`, `h1`, `h2`, `h3`, or `z`). If we choose a normalization value other than `no`, we need to set the normalization factor. Depending on the chosen normalization value, we should use `normalization.h1.c` (float value) for `h1` normalization, `normalization.h2.c` (float value) for `h2` normalization, `normalization.h3.c` (float value) for `h3` normalization, and `normalization.z.z` (float value) for `z` normalization. For example, the following is how the example similarity configuration will look (we put this into the `settings` section of our `mappings` file):

```
"similarity" : {
  "esserverbook_dfr_similarity" : {
    "type" : "DFR",
    "basic_model" : "g",
    "after_effect" : "l",
    "normalization" : "h2",
    "normalization.h2.c" : "2.0"
```

```
    }  
}
```

Configuring IB similarity

In case of `IB` similarity, we have the following parameters through which we can configure the `distribution` property (which can take the value of `ll` or `spl`) and the `lambda` property (which can take the value of `df` or `tff`). In addition to that, we can choose the normalization factor, which is the same as for the `DFR` similarity, so we'll omit describing it the second time. The following is how the example `IB` similarity configuration will look (we put this into the `settings` section of our `mappings` file):

```
"similarity" : {  
    "esserverbook_ib_similarity" : {  
        "type" : "IB",  
        "distribution" : "ll",  
        "lambda" : "df",  
        "normalization" : "z",  
        "normalization.z.z" : "0.25"  
    }  
}
```



The similarity model is a fairly complicated topic and will require a whole chapter to be properly described. If you are interested in it, please refer to our book, *Mastering Elasticsearch*, Packt Publishing, or go to <http://elasticsearchserverbook.com/elasticsearch-0-90-similarities/> to read more about them.

The postings format

One of the most significant changes introduced with Apache Lucene 4.0 was the ability to alter how index files are written. Elasticsearch leverages this functionality by allowing us to specify the postings format for each field. You may want to change how fields are indexed for performance reasons; for example, to have faster primary key lookups.

The following postings formats are included in Elasticsearch:

- `default`: This is a postings format that is used when no explicit format is defined. It provides on-the-fly stored fields and term vectors compression. If you want to read about what to expect from the compression, please refer to <http://solr.pl/en/2012/11/19/solr-4-1-stored-fields-compression/>.

- **pulsing**: This is a postings format that encodes the post listing into the terms array for high cardinality fields, which results in one less seek that Lucene needs to perform when retrieving a document. Using this postings format for high cardinality fields can speed up queries on such fields.
- **direct**: This is a postings format that loads terms into arrays during read operations. These arrays are held in the memory uncompressed. This format may give you a performance boost on commonly used fields, but should be used with caution as it is very memory intensive, because the terms and postings arrays needs to be stored in the memory. Please remember that since all the terms are held in the byte array, you can have up to 2.1 GB of memory used for this per segment.
- **memory**: This postings format, as its name suggests, writes all the data to disk, but reads the terms and post listings into the memory using a structure called **FST (Finite State Transducers)**. You can read more about this structure in a great post by Mike McCandless, available at <http://blog.mikemccandless.com/2010/12/using-finite-state-transducers-in.html>. Because the data is stored in memory, this postings format may result in a performance boost for commonly used terms.
- **bloom_default**: This is an extension of the default postings format that adds the functionality of a **bloom filter** written to disk. While reading, the bloom filter is read and held into memory to allow very fast checking if a given value exists. This postings format is very useful for high cardinality fields such as the primary key. If you want to know more about what the bloom filter is, please refer to http://en.wikipedia.org/wiki/Bloom_filter. This postings format uses the bloom filter in addition to what the default format does.
- **bloom_pulsing**: This is an extension of the pulsing postings format and uses the bloom filter in addition to what the pulsing format does.

Configuring the postings format

The postings format is a per-field property, just like type or name. In order to configure our field to use a different format than the default postings format, we need to add a property called `postings_format` with the name of the chosen postings format as a value. For example, if we would like to use the pulsing postings format for the `id` field, the mappings will look as follows:

```
{  
  "mappings" : {  
    "post" : {  
      "properties" : {  
        "id" : { "type" : "long", "store" : "yes",  
                 "postings_format" : "pulsing" }  
      }  
    }  
  }  
}
```

```
        "precision_step" : "0", "postings_format" : "pulsing" },
      "name" : { "type" : "string", "store" : "yes",
                 "index" : "analyzed" },
      "contents" : { "type" : "string", "store" : "no",
                     "index" : "analyzed" }
    }
  }
}
```

Doc values

The doc values is the last field property we will discuss in this section. The doc values format is another new feature introduced in Lucene 4.0. It allows us to define that a given field's values should be written in a memory efficient, column-stride structure for efficient sorting and faceting. The field with doc values enabled will have a dedicated field data cache instances that doesn't need to be inverted (so that they won't be stored in a way we described in the *Full-text searching* section in *Chapter 1, Getting Started with the Elasticsearch Cluster*) like standard fields. Therefore, it makes the index refresh operation faster and allows you to store the field data for such fields on disk and thus save heap memory for such fields.

Configuring the doc values

Let's extend our posts index example by adding a new field called votes. Let's assume that the newly added field contains the number of votes a given post was given and we want to sort on it. Because we are sorting on it, it is a good candidate for doc values. To use doc values on a given field, we need to add the `doc_values_format` property to its definition and specify the format. For example, our mappings will look as follows:

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
                  "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes",
                   "index" : "analyzed" },
        "contents" : { "type" : "string", "store" : "no",
                      "index" : "analyzed" },
        "votes" : { "type" : "integer",
                    "doc_values_format" : "memory" }
      }
    }
}
```

```
    }  
}  
}
```

As you can see, the definition is very simple. So let's see what options we have when it comes to the value of the `doc_values_format` property.

Doc values formats

Currently, there are three values for the `doc_values_format` property that can be used, as follows:

- `default`: This is a doc values format that is used when no format is specified. It offers good performance with low memory usage.
- `disk`: This is a doc values format that stores the data on disk. It requires almost no memory. However, there is a slight performance degradation when using this data structure for operations like faceting and sorting. Use this doc values format if you are struggling with memory issues while using faceting or sorting operations.
- `memory`: This is a doc values format that stores data in memory. Using this format will result in sorting and faceting functions that give performance that is comparable to standard inverted index fields. However, because the data structure is stored in memory, the index refresh operation will be faster, which can help with rapidly changing indices and short index refresh values.

Batch indexing to speed up your indexing process

In the first chapter, we've seen how to index a particular document into Elasticsearch. Now, it's time to find out how to index many documents in a more convenient and efficient way than doing it one by one.

Preparing data for bulk indexing

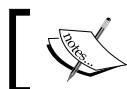
Elasticsearch allows us to merge many requests into one packet. These packets can be sent as a single request. In this way, we can mix the following operations:

- Adding or replacing the existing documents in the index (`index`)
- Removing documents from the index (`delete`)
- Adding new documents to the index when there is no other definition of the document in the index (`create`)

The format of the request was chosen for processing efficiency. It assumes that every line of the request contains a JSON object with the description of the operation followed by the second line with a JSON object itself. We can treat the first line as a kind of information line and the second as the data line. The exception to this rule is the delete operation, which contains only the information line. Let's look at the following example:

```
{ "index": { "_index": "addr", "_type": "contact", "_id": 1 } }
{ "name": "Fyodor Dostoevsky", "country": "RU" }
{ "create": { "_index": "addr", "_type": "contact", "_id": 2 } }
{ "name": "Erich Maria Remarque", "country": "DE" }
{ "create": { "_index": "addr", "_type": "contact", "_id": 2 } }
{ "name": "Joseph Heller", "country": "US" }
{ "delete": { "_index": "addr", "_type": "contact", "_id": 4 } }
{ "delete": { "_index": "addr", "_type": "contact", "_id": 1 } }
```

It is very important that every document or action description be placed in one line (ended by a newline character). This means that the document cannot be pretty-printed. There is a default limitation on the size of the bulk indexing file, which is set to 100 megabytes and can be changed by specifying the `http.max_content_length` property in the Elasticsearch configuration file. This lets us avoid issues with possible request timeouts and memory problems when dealing with requests that are too large.



Note that with a single batch indexing file, we can load the data into many indices and documents can have different types.



Indexing the data

In order to execute the bulk request, Elasticsearch provides the `_bulk` endpoint. This can be used as `/_bulk`, with the index name `/index_name/_bulk`, or even with a type and index name `/index_name/type_name/_bulk`. The second and third forms define the default values for the index name and type name. We can omit these properties in the information line of our request and Elasticsearch will use the default values.

Assuming we've stored our data in the `documents.json` file, we can run the following command to send this data to Elasticsearch:

```
curl -XPOST 'localhost:9200/_bulk?pretty' --data-binary
@documents.json
```

The `?pretty` parameter is of course not necessary. We've used this parameter only for the ease of analyzing the response of the preceding command. In this case, using `curl` with the `--data-binary` parameter instead of using `-d` is important. This is because the standard `-d` parameter ignores newline characters, which, as we said earlier, are important for parsing Elasticsearch's bulk request content. Now let's look at the response returned by Elasticsearch:

```
{  
  "took" : 139,  
  "errors" : true,  
  "items" : [ {  
    "index" : {  
      "_index" : "addr",  
      "_type" : "contact",  
      "_id" : "1",  
      "_version" : 1,  
      "status" : 201  
    }  
  }, {  
    "create" : {  
      "_index" : "addr",  
      "_type" : "contact",  
      "_id" : "2",  
      "_version" : 1,  
      "status" : 201  
    }  
  }, {  
    "create" : {  
      "_index" : "addr",  
      "_type" : "contact",  
      "_id" : "2",  
      "status" : 409,  
      "error" : "DocumentAlreadyExistsException[[addr] [3]  
      [contact] [2]: document already exists]"  
    }  
  }]
```

```
}, {  
  "delete" : {  
    "_index" : "addr",  
    "_type" : "contact",  
    "_id" : "4",  
    "_version" : 1,  
    "status" : 404,  
    "found" : false  
  }  
, {  
  "delete" : {  
    "_index" : "addr",  
    "_type" : "contact",  
    "_id" : "1",  
    "_version" : 2,  
    "status" : 200,  
    "found" : true  
  }  
} ]  
}
```

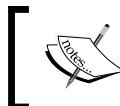
As we can see, every result is a part of the `items` array. Let's briefly compare these results with our input data. The first two commands, named `index` and `create`, were executed without any problems. The third operation failed because we wanted to create a record with an identifier that already existed in the index. The next two operations were deletions. Both succeeded. Note that the first of them tried to delete a nonexistent document; as you can see, this wasn't a problem for Elasticsearch. As you can see, Elasticsearch returns information about each operation, so for large bulk requests, the response can be massive.

Even quicker bulk requests

Bulk operations are fast, but if you are wondering if there is a more efficient and quicker way of indexing, you can take a look at the **User Datagram Protocol (UDP)** bulk operations. Note that using UDP doesn't guarantee that no data will be lost during communication with the Elasticsearch server. So, this is useful only in some cases where performance is critical and more important than accuracy and having all the documents indexed.

Extending your index structure with additional internal information

Apart from the fields that are used to hold data, we can store additional information along with the documents. We already talked about different mapping options and what data type we can use. We would like to discuss in more detail some functionalities of Elasticsearch that are not used every day, but can make your life easier when it comes to data handling.



Each of the field types discussed in the following sections should be defined on an appropriate type level – they are not index wide



Identifier fields

As you may recall, each document indexed in Elasticsearch has its own identifier and type. In Elasticsearch, there are two types of internal identifiers for the documents.

The first one is the `_uid` field, which is the unique identifier of the document in the index and is composed of the document's identifier and document type. This basically means that documents of different types that are indexed into the same index can have the same document identifier, yet Elasticsearch will be able to distinguish them. This field doesn't require any additional settings; it is always indexed, but it's good to know that it exists.

The second field that holds the identifier is the `_id` field. This field stores the actual identifier set during index time. In order to enable the `_id` field indexing (and storing, if needed), we need to add the `_id` field definition just like any other property in our `mappings` file (although, as said before, add it in the body of the type definition). So, our sample book type definition will look like the following:

```
{
  "book" : {
    "_id" : {
      "index": "not_analyzed",
      "store" : "no"
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

```
    }  
}
```

As you can see, in the preceding example, we coded that we want our `_id` field to be indexed but not analyzed and we don't want to store the index.

In addition to specifying the identifier during indexing time, we can specify that we want it to be fetched from one of the fields of the indexed documents (although it will be slightly slower because of the additional parsing needed). In order to do that, we need to specify the `path` property and set its value to the name of the field whose value we want to set as the identifier. For example, if we have the `book_id` field in our index and we want to use it as the value for the `_id` field, we can change the preceding `mappings` file as follows:

```
{  
  "book" : {  
    "_id" : {  
      "path": "book_id"  
    },  
    "properties" : {  
      .  
      .  
      .  
    }  
  }  
}
```

One last thing—remember that even when disabling the `_id` field, all the functionalities requiring the document's unique identifier will still work, because they will be using the `_uid` field instead.

The `_type` field

We already said that each document in Elasticsearch is at least described by its identifier and type. By default, the document type is indexed but not analyzed and stored. If we would like to store that field, we will change our `mappings` file as follows:

```
{  
  "book" : {  
    "_type" : {  
      "store" : "yes"  
    },  
    "properties" : {  
      .  
    }  
  }  
}
```

```
    .
    .
    }
}
```

We can also change the `_type` field to not be indexed, but then some queries such as term queries and filters will not work.

The `_all` field

The `_all` field is used by Elasticsearch to store data from all the other fields in a single field for ease of searching. This kind of field may be useful when we want to implement a simple search feature and we want to search all the data (or only the fields we copy to the `_all` field), but we don't want to think about field names and things like that. By default, the `_all` field is enabled and contains all the data from all the fields from the index. However, this field will make the index a bit bigger and that is not always needed. We can either disable the `_all` field completely or exclude the copying of certain fields to it. In order not to include a certain field in the `_all` field, we will use the `include_in_all` property, which was discussed earlier in this chapter. To completely turn off the `_all` field functionality, we will modify our mappings file as follows:

```
{
  "book" : {
    "_all" : {
      "enabled" : "false"
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

In addition to the `enabled` property, the `_all` field supports the following ones:

- `store`
- `term_vector`
- `analyzer`
- `index_analyzer`
- `search_analyzer`

For information about the preceding properties, please refer to the *Mappings configuration* section in this chapter.

The `_source` field

The `_source` field allows us to store the original JSON document that was sent to Elasticsearch during indexation. By default, the `_source` field is turned on because some of the Elasticsearch functionalities depend on it (for example, the partial update feature). In addition to that, the `_source` field can be used as the source of data for the highlighting functionality if a field is not stored. But if we don't need such a functionality, we can disable the `_source` field as it causes some storage overhead. In order to do that, we need to set the `_source` object's `enabled` property to `false`, as follows:

```
{  
  "book" : {  
    "_source" : {  
      "enabled" : false  
    },  
    "properties" : {  
      .  
      .  
      .  
    }  
  }  
}
```

Exclusion and inclusion

We can also tell Elasticsearch which fields we want to exclude from the `_source` field and which fields we want to include. We do that by adding the `includes` and `excludes` properties to the `_source` field definition. For example, if we want to exclude all the fields in the `author` path from the `_source` field, our mappings will look as follows:

```
{  
  "book" : {  
    "_source" : {  
      "excludes" : [ "author.*" ]  
    },  
    "properties" : {  
      .  
      .  
      .  
    }  
  }  
}
```

```
        }
    }
}
```

The `_index` field

Elasticsearch allows us to store the information about the index that the documents are indexed to. We can do that by using the internal `_index` field. Imagine that we create daily indices, we use aliasing, and we are interested in the daily index in which the returned document is stored. In such cases, the `_index` field can be useful, because it may help us identify the index the document comes from.

By default, the indexing of the `_index` field is disabled. In order to enable it, we need to set the `enabled` property of the `_index` object to `true`, as follows:

```
{
  "book" : {
    "_index" : {
      "enabled" : true
    },
    "properties" : {
      .
      .
      .
    }
  }
}
```

The `_size` field

By default, the `_size` field is not enabled; it enables us to automatically index the original size of the `_source` field and store it along with the documents. If we would like to enable the `_size` field, we need to add the `_size` property and wrap the `enabled` property with the value `true`. In addition to that, we can also set the `_size` field to be stored by using the usual `store` property. So, if we want our mapping to include the `_size` field and we want it to be stored, we will change our `mappings` file to something like the following:

```
{
  "book" : {
    "_size" : {
      "enabled": true,
      "store" : "yes"
    },
  }
}
```

```
"properties" : {  
    .  
    .  
    .  
}  
}  
}
```

The `_timestamp` field

Disabled by default, the `_timestamp` field allows us to store when the document was indexed. Enabling this functionality is as simple as adding the `_timestamp` section to our `mappings` file and setting the `enabled` property to `true`, as follows:

```
{  
    "book" : {  
        "_timestamp" : {  
            "enabled" : true  
        },  
        "properties" : {  
            .  
            .  
            .  
        }  
    }  
}
```

The `_timestamp` field is, by default, not stored, indexed, but not analyzed and you can change these two parameters to match your needs. In addition to that, the `_timestamp` field is just like the normal date field, so we can change its format just like we do with usual date-based fields. In order to change the format, we need to specify the `format` property with the desired format (please refer to the `date` core type description earlier in this chapter to read more about date formats).

One more thing – instead of automatically creating the `_timestamp` field during document indexation, we can add the `path` property and set it to the name of the field, which should be used to get the date. So if we want our `_timestamp` field to be based on the `year` field, we will modify our `mappings` file to something like the following:

```
{  
    "book" : {  
        "_timestamp" : {  
            "enabled" : true,  
            "path" : "year"  
        }  
    }  
}
```

```
        "path" : "year",
        "format" : "YYYY"
    },
    "properties" : {
        .
        .
        .
    }
}
```

As you may have noticed, we also modify the format of the `_timestamp` field in order to match the values stored in the `year` field.



If you use the `_timestamp` field and you let Elasticsearch create it automatically, the value of that field will be set to the time of indexation of that document. Please note that when using the partial document update functionality, the `_timestamp` field will also be updated.

The `_ttl` field

The `_ttl` field stands for **time to live**, a functionality that allows us to define the life period of a document, after which it will be automatically deleted. As you may expect, by default, the `_ttl` field is disabled. And to enable it, we need to add the `_ttl` JSON object and set its `enabled` property to `true`, just as in the following example:

```
{  
    "book" : {  
        "_ttl" : {  
            "enabled" : true  
        },  
        "properties" : {  
            .  
            .  
            .  
        }  
    }  
}
```

If you need to provide the default expiration time for documents, just add the `default` property to the `_ttl` field definition with the desired expiration time. For example, to have our documents deleted after 30 days, we will set the following:

```
{  
  "book" : {  
    "_ttl" : {  
      "enabled" : true,  
      "default" : "30d"  
    },  
    "properties" : {  
      .  
      .  
      .  
    }  
  }  
}
```

The `_ttl` value, by default, is stored and indexed, but not analyzed and you can change these two parameters, but remember that this field needs to be not analyzed to work.

Introduction to segment merging

In the *Full-text searching* section of *Chapter 1, Getting Started with the Elasticsearch Cluster*, we mentioned segments and their immutability. We wrote that the Lucene library, and thus Elasticsearch, writes data to certain structures that are written once and never changed. This allows for some simplification, but also introduces the need for additional work. One such example is deletion. Because a segment cannot be altered, information about deletions must be stored alongside and dynamically applied during search. This is done to eliminate deleted documents from the returned result set. The other example is the inability to modify documents (however, some modifications are possible, such as modifying numeric doc values). Of course, one can say that Elasticsearch supports document updates (please refer to the *Manipulating data with the REST API* section of *Chapter 1, Getting Started with the Elasticsearch Cluster*). However, under the hood, the old document is deleted and the one with the updated contents is indexed.

As time passes and you continue to index your data, more and more segments are created. Because of that, the search performance may be lower and your index may be larger than it should be—it still contains the deleted documents. This is when segment merging comes into play.

Segment merging

Segment merging is the process during which the underlying Lucene library takes several segments and creates a new segment based on the information found in them. The resulting segment has all the documents stored in the original segments except the ones that were marked for deletion. After the merge operation, the source segments are deleted from the disk. Because segment merging is rather costly in terms of CPU and I/O usage, it is crucial to appropriately control when and how often this process is invoked.

The need for segment merging

You may ask yourself why you have to bother with segment merging. First of all, the more segments the index is built of, the slower the search will be and the more memory Lucene will use. The second is the disk space and resources, such as file descriptors, used by the index. If you delete many documents from your index, until the merge happens, those documents are only marked as deleted and not deleted physically. So, it may happen that most of the documents that use our CPU and memory don't exist! Fortunately, Elasticsearch uses reasonable defaults for segment merging and it is very probable that no changes are necessary.

The merge policy

The merge policy describes when the merging process should be performed. Elasticsearch allows us to configure the following three different policies:

- `tiered`: This is the default merge policy that merges segments of approximately similar size, taking into account the maximum number of segments allowed per tier.
- `log_byte_size`: This is a merge policy that, over time, will produce an index that will be built of a logarithmic size of indices. There will be a few large segments, a few merge factor smaller segments, and so on.
- `log_doc`: This policy is similar to the `log_byte_size` merge policy, but instead of operating on the actual segment size in bytes, it operates on the number of documents in the index.

Each of the preceding policies has their own parameters, which define their behavior and whose default values can be overridden. In this book, we will skip the detailed description. If you want to learn more, check our book, *Mastering ElasticSearch, Packt Publishing*, or go to <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/index-modules-merge.html>.

The merge policy we want to use can be set using the `index.merge.policy.type` property as follows:

```
index.merge.policy.type: tiered
```

It is worth mentioning that the value cannot be changed after index creation.

The merge scheduler

The merge scheduler tells Elasticsearch how the merge process should occur. There are two possibilities:

- **Concurrent merge scheduler:** This is the default merge process that starts in a separate thread, and the defined number of threads can do the merges in parallel.
- **Serial merge scheduler:** This process of merging runs in the calling thread (the one executing indexing). The merge process will block the thread until the merge is finished.

The scheduler can be set using the `index.merge.scheduler.type` parameter. The values that we can use are `serial` for the serial merge scheduler or `concurrent` for the concurrent one. For example, consider the following scheduler:

```
index.merge.scheduler.type: concurrent
```

The merge factor

Each of the policies has several settings. We already told that we don't want to describe them, but there is an exception – the **merge factor**, which specifies how often segments are merged during indexing. With a smaller merge factor value, the searches are faster and less memory is used, but that comes with the cost of slower indexing. With larger values, it is the opposite—indexing is faster (because less merging being done), but the searches are slower and more memory is used. This factor can be set for the `log_byte_size` and `log_doc` merge policies using the `index.merge.policy.merge_factor` parameter as follows:

```
index.merge.policy.merge_factor: 10
```

The preceding example will set the merge factor to 10, which is also the default value. It is advised to use larger values of `merge_factor` for batch indexing and lower values of this parameter for normal index maintenance.

Throttling

As we have already mentioned, merging may be expensive when it comes to server resources. The merge process usually works in parallel to other operations, so theoretically it shouldn't have too much influence. In practice, the number of disk input/output operations can be so large that it will significantly affect the overall performance. In such cases, **throttling** is something that may help. In fact, this feature can be used for limiting the speed of the merge, but also may be used for all the operations using the data store. Throttling can be set in the Elasticsearch configuration file (the `elasticsearch.yml` file) or dynamically using the settings API (refer to the *The update settings API* section of *Chapter 8, Administrating Your Cluster*). There are two settings that adjust throttling: `type` and `value`.

To set the throttling type, set the `indices.store.throttle.type` property, which allows us to use the following values:

- `none`: This value defines that no throttling is on
- `merge`: This value defines that throttling affects only the merge process
- `all`: This value defines that throttling is used for all data store activities

The second property – `indices.store.throttle.max_bytes_per_sec` – describes how much the throttling limits I/O operations. As its name suggests, it tells us how many bytes can be processed per second. For example, let's look at the following configuration:

```
indices.store.throttle.type: merge
indices.store.throttle.max_bytes_per_sec: 10mb
```

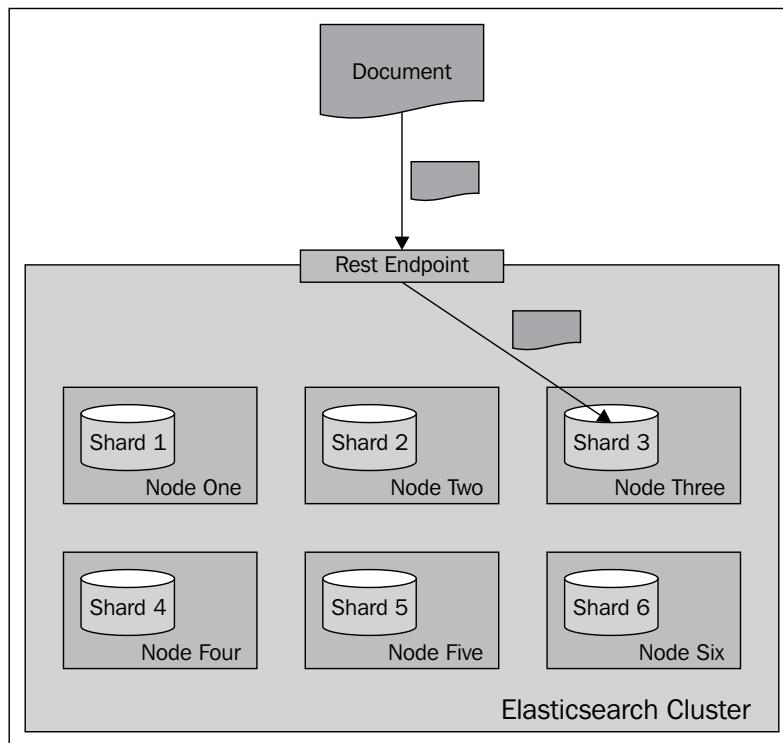
In this example, we limit the merge operations to 10 megabytes per second. By default, Elasticsearch uses the `merge` throttling type with the `max_bytes_per_sec` property set to `20mb`. That means that all the merge operations are limited to 20 megabytes per second.

Introduction to routing

By default, Elasticsearch will try to distribute your documents evenly among all the shards of the index. However, that's not always the desired situation. In order to retrieve the documents, Elasticsearch must query all the shards and merge the results. However, if you can divide your data on some basis (for example, the client identifier), you can use a powerful document and query distribution control mechanism – routing. In short, it allows us to choose a shard that will be used to index or search data.

Default indexing

During indexing operations, when you send a document for indexing, Elasticsearch looks at its identifier to choose the shard in which the document should be indexed. By default, Elasticsearch calculates the hash value of the document's identifier and on the basis of that, it puts the document in one of the available primary shards. Then, those documents are redistributed to the replicas. The following diagram shows a simple illustration of how indexing works by default:



Default searching

Searching is a bit different from indexing, because in most situations, you need to query all the shards to get the data you are interested in. Imagine a situation when you have the following mappings describing your index:

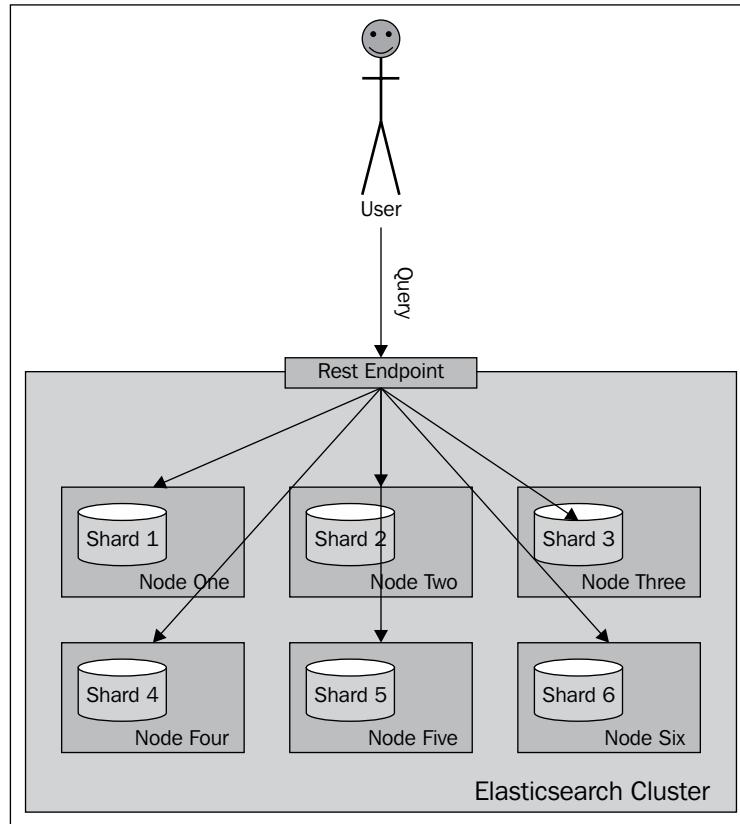
```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
                  "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes",
                   "index" : "analyzed" },
        "contents" : { "type" : "string", "store" : "no",
                      "index" : "analyzed" },
        "userId" : { "type" : "long", "store" : "yes",
                     "precision_step" : "0" }
      }
    }
  }
}
```

As you can see, our index consists of four fields—the identifier (the `id` field), name of the document (the `name` field), contents of the document (the `contents` field), and the identifier of the user to which the documents belong (the `userId` field). To get all the documents for a particular user—one with `userId` equal to 12—you can run the following query:

```
curl -XGET 'http://localhost:9200/posts/_search?q=userId:12'
```

In general, you will send your query to one of the Elasticsearch nodes and Elasticsearch will do the rest. Depending on the search type (we will talk more about it in *Chapter 3, Searching Your Data*), Elasticsearch will run your query, which usually means that it will first query all the nodes for the identifier and score of the matching documents. Then, it will send an internal query again, but only to the relevant shards (the ones containing the needed documents) to get the documents needed to build the response.

A very simplified view of how default routing works during searching is shown in the following illustration:



What if we could put all the documents for a single user into a single shard and query on that shard? Wouldn't that be wise for performance? Yes, that is handy and that is what routing allows you do to.

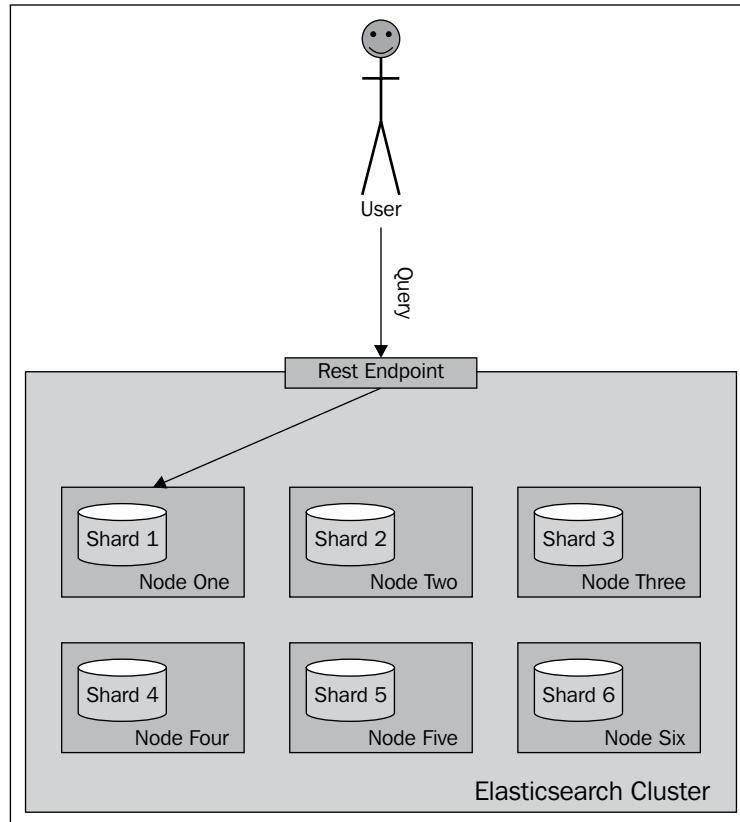
Routing

Routing can control to which shard your documents and queries will be forwarded. By now, you will probably have guessed that we can specify the routing value both during indexing and during querying, and in fact if you decide to specify explicit routing values, you'll probably do that during indexing and searching.

In our case, we will use the `userId` value to set routing during indexing and the same value during searching. You can imagine that for the same `userId` value, the same hash value will be calculated and thus all the documents for that particular user will be placed in the same shard. Using the same value during search will result in searching a single shard instead of the whole index.

Remember that when using routing, you should still add a filter for the same value as the routing one. This is because you'll probably have more distinct routing values than the number of shards your index will be built with. Because of that, a few distinct values can point to the same shard, and if you omit filtering, you will get data not for a single value you route on, but for all those that reside in a particular shard.

The following diagram shows a very simple illustration of how searching works with a provided custom routing value:



As you can see, Elasticsearch will send our query to a single shard. Now let's look at how we can specify the routing values.

The routing parameters

The simplest way (but not always the most convenient one) is to provide the routing value using the `routing` parameter. When indexing or querying, you can add the `routing` parameter to your HTTP or set it by using the client library of your choice.

So, in order to index a sample document to the previously shown index, we will use the following lines of command:

```
curl -XPUT 'http://localhost:9200/posts/post/1?routing=12' -d '{  
  "id": "1",  
  "name": "Test document",  
  "contents": "Test document",  
  "userId": "12"  
'
```

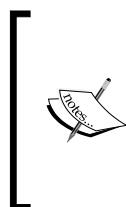
This is how our previous query will look if we add the `routing` parameter:

```
curl -XGET  
'http://localhost:9200/posts/_search?routing=12&q=userId:12'
```

As you can see, the same routing value was used during indexing and querying. We did that because we knew that during indexing we have used the value `12`, so we wanted to point our query to the same shard and therefore we used exactly the same value.

Please note that you can specify multiple routing values separated by commas. For example, if we want the preceding query to be additionally routed with the use of the `section` parameter (if it existed) and we also want to filter by this parameter, our query will look like the following:

```
curl -XGET  
'http://localhost:9200/posts/_search?routing=12,  
6654&q=userId:12+AND+section:6654'
```



Remember that routing is not the only thing that is required to get results for a given user. That's because usually we have less shards that have unique routing values. This means that we will have data from multiple users in a single shard. So when using routing, you should also filter the results. You'll learn more about filtering in the *Filtering your results* section in *Chapter 3, Searching Your Data*.

Routing fields

Specifying the routing value with each request that we send to Elasticsearch works, but it is not convenient. In fact, Elasticsearch allows us to define a field whose value will be used as the routing value during indexing, so we only need to provide the routing parameter during querying. To do that, we need to add the following section to our type definition:

```
"_routing" : {
    "required" : true,
    "path" : "userId"
}
```

The preceding definition means that the routing value needs to be provided (the "required": true property); without it, an index request will fail. In addition to that, we specified the path attribute, which says which field value of the document will be used as the routing value. In our case, the userId field value will be used. These two parameters mean that each document we send for indexing needs to have the userId field defined. This is convenient, because we can now use batch indexing without the limitation of having all the documents from a single branch using the same routing value (which would be the case with the routing parameter). However, please remember that when using the routing field, Elasticsearch needs to do some additional parsing, and therefore it's a bit slower than when using the routing parameter.

After adding the routing part, the whole updated `mappings` file will be as follows:

```
{
  "mappings" : {
    "post" : {
      "_routing" : {
        "required" : true,
        "path" : "userId"
      },
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
                  "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes",
                   "index" : "analyzed" },
        "contents" : { "type" : "string", "store" : "no",
                      "index" : "analyzed" },
        "userId" : { "type" : "long", "store" : "yes",
                     "precision_step" : "0" }
      }
    }
}
```

```
        }
    }
}
```

If we want to create the `posts` index using the preceding mappings, use the following command to index a single, test document:

```
curl -XPOST 'localhost:9200/posts/post/1' -d '{
  "id":1,
  "name": "New post",
  "contents": "New test post",
  "userId":1234567
}'
```

Elasticsearch will end up using `1234567` as the routing value for indexing.

Summary

In this chapter, we learned how Elasticsearch indexing works. We learned to create our own mappings that define index structure and create indices using them. We learned what batch indexing is and how to use it, and how we can index our data efficiently. We also learned what additional information can be stored along with the documents. In addition to that, we've seen what segment merging is, how to configure it, and what throttling is. Finally, we used and configured routing.

In the next chapter, we will concentrate on searching. We will start with how to query Elasticsearch and what the basic queries we can use are. In addition to that, we will use filters and learn why they are important. We will see how we can validate our queries and how to use the highlighting functionality. Finally, we will use compound queries, we will get into querying internals, and we will sort our results.

3

Searching Your Data

In the previous chapter, we learned how Elasticsearch indexing works, how to create our own mappings, and what data types we can use. We also stored additional information in our index and used routing, both default and nondefault. By the end of this chapter, we will have learned about the following topics:

- Querying Elasticsearch and choosing the data to be returned
- The working of the Elasticsearch querying process
- Understanding the basic queries exposed by Elasticsearch
- Filtering our results
- Understanding how highlighting works and how to use it
- Validating our queries
- Exploring compound queries
- Sorting our data

Querying Elasticsearch

So far, when we searched our data we used the REST API and a simple query or the GET request. Similarly, when we changed the index, we also used the REST API and sent the JSON-structured data to Elasticsearch, regardless of the type of operation we wanted to perform – whether it was a mapping change or document indexation. A similar situation happens when we want to send more than a simple query to Elasticsearch – we structure it using JSON objects and send it to Elasticsearch. This is called the **query DSL**. In a broader view, Elasticsearch supports two kinds of queries: basic ones and compound ones. Basic queries such as the `term` query are used for querying the actual data. We will cover these in the *Basic queries* section of this chapter. The second type of query is the compound query, such as the `bool` query, which can combine multiple queries. We will cover these in the *Compound queries* section of this chapter.

However, this is not the whole picture. In addition to these two types of queries, your query can have **filter queries** that are used to narrow down your results with certain criteria. Unlike other queries, filter queries don't affect scoring and are usually very efficient.

To make it even more complicated, queries can contain other queries. (Don't worry; we will try to explain this!) Furthermore, some queries can contain filters and others can contain both queries and filters. Although this is not everything, we will stick with this working explanation for now. We will go over this in detail in the *Compound queries* and *Filtering your results* sections of this chapter.

The example data

If not stated otherwise, the following mappings will be used for the rest of the chapter:

```
{  
    "book" : {  
        "_index" : {  
            "enabled" : true  
        },  
        "_id" : {  
            "index": "not_analyzed",  
            "store" : "yes"  
        },  
        "properties" : {  
            "author" : {  
                "type" : "string"  
            },  
            "characters" : {  
                "type" : "string"  
            },  
            "copies" : {  
                "type" : "long",  
                "ignore_malformed" : false  
            },  
            "otitle" : {  
                "type" : "string"  
            },  
            "tags" : {  
                "type" : "string"  
            },  
            "title" : {  
                "type" : "string"  
            }  
        }  
    }  
}
```

```
        "type" : "string"
    },
    "year" : {
        "type" : "long",
        "ignore_malformed" : false,
        "index" : "analyzed"
    },
    "available" : {
        "type" : "boolean"
    }
}
```



The string-based fields will be analyzed if not stated otherwise.

J

The preceding mappings (which are stored in the `mapping.json` file) were used to create the library index. In order to run them, use the following commands:

```
curl -XPOST 'localhost:9200/library'  
curl -XPUT 'localhost:9200/library/book/_mapping' -d @mapping.json
```

If not stated otherwise, the following data will be used for the rest of the chapter:

```
{ "index": {"_index": "library", "_type": "book", "_id": "1"} }
{ "title": "All Quiet on the Western Front", "otitle": "Im Westen nichts Neues", "author": "Erich Maria Remarque", "year": 1929, "characters": ["Paul Bäumer", "Albert Kropp", "Haie Westhus", "Fredrich Müller", "Stanislaus Katczinsky", "Tjaden"], "tags": ["novel"], "copies": 1, "available": true, "section": 3}
{ "index": {"_index": "library", "_type": "book", "_id": "2"} }
{ "title": "Catch-22", "author": "Joseph Heller", "year": 1961, "characters": ["John Yossarian", "Captain Aardvark", "Chaplain Tappman", "Colonel Cathcart", "Doctor Daneeka"], "tags": ["novel"], "copies": 6, "available": false, "section": 1}
{ "index": {"_index": "library", "_type": "book", "_id": "3"} }
{ "title": "The Complete Sherlock Holmes", "author": "Arthur Conan Doyle", "year": 1936, "characters": ["Sherlock Holmes", "Dr. Watson", "G. Lestrade"], "tags": [], "copies": 0, "available": false, "section": 12}
{ "index": {"_index": "library", "_type": "book", "_id": "4"} }
```

```
{ "title": "Crime and Punishment", "otitle": "Преступление и  
наказание", "author": "Fyodor Dostoevsky", "year":  
1886, "characters": ["Raskolnikov", "Sofia Semyonovna  
Marmeladova"], "tags": [], "copies": 0, "available": true}
```

We stored our data in the `documents.json` file, and we use the following command to index it:

```
curl -s -XPOST 'localhost:9200/_bulk' --data-binary @documents.json
```

This command runs bulk indexing. You can learn more about it in the *Batch indexing to speed up your indexing process* section in *Chapter 2, Indexing Your Data*.

A simple query

The simplest way to query Elasticsearch is to use the URI request query. We already discussed it in the *Searching with the URI request query* section of *Chapter 1, Getting Started with the Elasticsearch Cluster*. For example, to search for the word `crime` in the `title` field, send a query using the following command:

```
curl -XGET 'localhost:9200/library/book/_search?q=title:crime&pretty=true'
```

This is a very simple, but limited, way of submitting queries to Elasticsearch. If we look from the point of view of the Elasticsearch query DSL, the preceding query is the `query_string` query. It searches for the documents that have the `crime` term in the `title` field and can be rewritten as follows:

```
{
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

Sending a query using the query DSL is a bit different, but still not rocket science. We send the GET HTTP request to the `_search` REST endpoint as before, and attach the query to the request body. Let's take a look at the following command:

```
curl -XGET 'localhost:9200/library/book/_search?pretty=true' -d '{
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}'
```

As you can see, we used the request body (the `-d` switch) to send the whole JSON-structured query to Elasticsearch. The `pretty=true` request parameter tells Elasticsearch to structure the response in such a way that we humans can read it more easily. In response to the preceding command, we get the following output:

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641, "_source" : { "title": "Crime and Punishment", "otitle": "Преступлénie и Наказánie", "author": "Fyodor Dostoevsky", "year": 1886, "characters": ["Raskolnikov", "Sofia Semyonovna Marmeladova"], "tags": [], "copies": 0, "available": true}
    } ]
  }
}
```

Nice! We got our first search results with the query DSL.

Paging and result size

As we expected, Elasticsearch allows us to control how many results we want to get (at most) and from which result we want to start. The following are the two additional properties that can be set in the request body:

- `from`: This property specifies the document that we want to have our results from. Its default value is `0`, which means that we want to get our results from the first document.

- **size:** This property specifies the maximum number of documents we want as the result of a single query (which defaults to 10). For example, if we are only interested in faceting results and don't care about the documents returned by the query, we can set this parameter to 0.

If we want our query to get documents starting from the tenth item on the list and get 20 of items from there on, we send the following query:

```
{  
    "from" : 9,  
    "size" : 20,  
    "query" : {  
        "query_string" : { "query" : "title:crime" }  
    }  
}
```

Downloading the example code



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

Returning the version value

In addition to all the information returned, Elasticsearch can return the version of the document. To do this, we need to add the `version` property with the value of `true` to the top level of our JSON object. So, the final query, which requests for version information, will look as follows:

```
{  
    "version" : true,  
    "query" : {  
        "query_string" : { "query" : "title:crime" }  
    }  
}
```

After running the preceding query, we get the following results:

```
{  
    "took" : 2,  
    "timed_out" : false,  
    "_shards" : {
```

```

    "total" : 5,
    "successful" : 5,
    "failed" : 0
},
{
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_version" : 1,
      "_score" : 0.15342641, "_source" : { "title": "Crime and
        Punishment", "otitle": "Преступлénie и Наказánie",
        "author": "Fyodor Dostoevsky", "year": 1886,
        "characters": ["Raskolnikov", "Sofia Semyonovna
        Marmeladova"], "tags": [], "copies": 0, "available" : true}
    } ]
  }
}

```

As you can see, the `_version` section is present for the single hit we got.

Limiting the score

For nonstandard use cases, Elasticsearch provides a feature that lets us filter the results on the basis of the minimum score value that the document must have to be considered a match. In order to use this feature, we must provide the `min_score` value at the top level of our JSON object with the value of the minimum score. For example, if we want our query to only return documents with a score higher than 0.75, we send the following query:

```
{
  "min_score" : 0.75,
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

We get the following response after running the preceding query:

```
{  
    "took" : 1,  
    "timed_out" : false,  
    "_shards" : {  
        "total" : 5,  
        "successful" : 5,  
        "failed" : 0  
    },  
    "hits" : {  
        "total" : 0,  
        "max_score" : null,  
        "hits" : [ ]  
    }  
}
```

If you look at the previous examples, the score of our document was `0.15342641`, which is lower than `0.75`, and thus we didn't get any documents in response.

Limiting the score doesn't make much sense, usually because comparing scores between queries is quite hard. However, maybe in your case, this functionality will be needed.

Choosing the fields that we want to return

With the use of the `fields` array in the request body, Elasticsearch allows us to define which fields to include in the response. Remember that you can only return those fields if they are marked as stored in the mappings used to create the index, or if the `_source` field was used (Elasticsearch uses the `_source` field to provide the stored values).

So, for example, to return only the `title` and `year` fields in the results (for each document), send the following query to Elasticsearch:

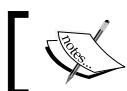
```
{  
    "fields" : [ "title", "year" ],  
    "query" : {  
        "query_string" : { "query" : "title:crime" }  
    }  
}
```

And in response, we get the following output:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641,
      "fields" : {
        "title" : [ "Crime and Punishment" ],
        "year" : [ 1886 ]
      }
    } ]
  }
}
```

As you can see, everything worked as we wanted it to. There are three things we would like to share with you, which are as follows:

- If we don't define the `fields` array, it will use the default value and return the `_source` field if available
- If we use the `_source` field and request a field that is not stored, then that field will be extracted from the `_source` field (however, this requires additional processing)
- If we want to return all stored fields, we just pass an asterisk (*) as the field name



From a performance point of view, it's better to return the `_source` field instead of multiple stored fields.

The partial fields

In addition to choosing which fields are returned, Elasticsearch allows us to use the so-called **partial fields**. They allow us to control how fields are loaded from the `_source` field. Elasticsearch exposes the `include` and `exclude` properties of the `partial_fields` object so we can include and exclude fields on the basis of these properties. For example, for our query to include the fields that start with `titl` and exclude the ones that start with `chara`, we send the following query:

```
{  
    "partial_fields" : {  
        "partial1" : {  
            "include" : [ "titl*" ],  
            "exclude" : [ "chara*" ]  
        }  
    },  
    "query" : {  
        "query_string" : { "query" : "title:crime" }  
    }  
}
```

Using the script fields

Elasticsearch allows us to use script-evaluated values that will be returned with result documents. To use the script fields, we add the `script_fields` section to our JSON query object and an object with a name of our choice for each scripted value that we want to return. For example, to return a value named `correctYear`, which is calculated as the `year` field minus `1800`, we run the following query:

```
{  
    "script_fields" : {  
        "correctYear" : {  
            "script" : "doc['year'].value - 1800"  
        }  
    },  
    "query" : {  
        "query_string" : { "query" : "title:crime" }  
    }  
}
```

Using the `doc` notation, like we did in the preceding example, allows us to catch the results returned, which thereby results in faster script execution, but it also leads to higher memory consumption. We are also limited to single-valued and single term fields. If we care about memory usage, or we are using more complicated field values, we can always use the `_source` field. Our query using this field looks as follows:

```
{
  "script_fields" : {
    "correctYear" : {
      "script" : "_source.year - 1800"
    }
  },
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

The following response is returned by Elasticsearch for the preceding query:

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641,
      "fields" : {
        "title" : [
          "The Mystery of the Old Clock"
        ],
        "year" : [
          1850
        ]
      }
    } ]
  }
}
```

```
        "correctYear" : [ 86 ]
    }
}
}
}
```

As you can see, we got the calculated `correctYear` field in response.

Passing parameters to the script fields

Let's take a look at one more feature of the script fields: the passing of additional parameters. Instead of having the value `1800` in the equation, we can use a variable name and pass its value in the `params` section. If we do this, our query will look as follows:

```
{
  "script_fields" : {
    "correctYear" : {
      "script" : "_source.year - paramYear",
      "params" : {
        "paramYear" : 1800
      }
    }
  },
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

As you can see, we added the `paramYear` variable as a part of the scripted equation and provided its value in the `params` section.

You can learn more about the use of scripts in the *Scripting capabilities of Elasticsearch* section of *Chapter 5, Make Your Search Better*.

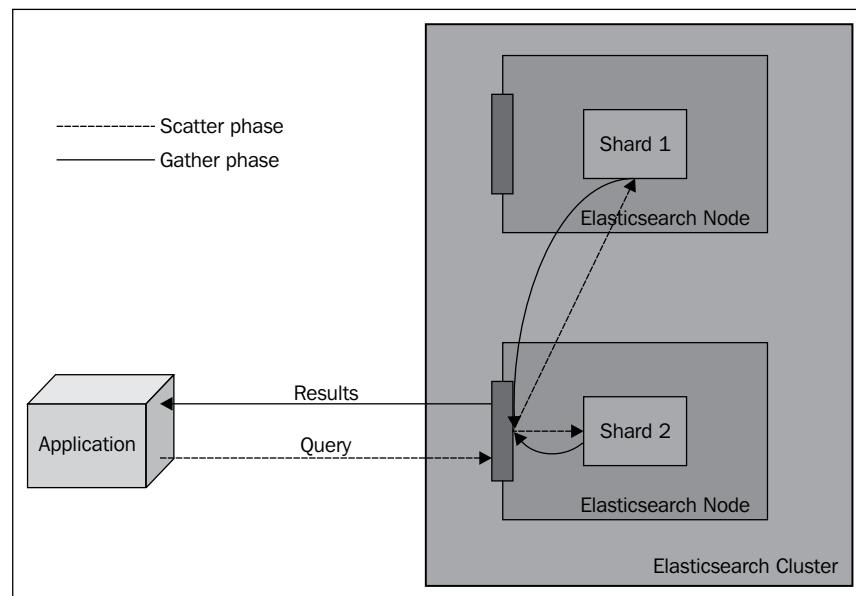
Understanding the querying process

After reading the previous section, we now know how querying works in Elasticsearch. You know that Elasticsearch, in most cases, needs to scatter the query across multiple nodes, get the results, merge them, fetch for relevant documents, and return the results. What we didn't talk about is three additional things that define how queries behave: query rewrite, search type, and query execution preference. We will now concentrate on these functionalities of Elasticsearch and also try to show you how querying works.

Query logic

Elasticsearch is a distributed search engine, and so all functionality provided must be distributed in its nature. It is exactly the same with querying. Since we want to discuss some more advanced topics on how to control the query process, we first need to know how it works.

By default, if we don't alter anything, the query process will consist of two phases as shown in the following diagram:



When we send a query, we send it to one of the Elasticsearch nodes. What is occurring now is a so-called **scatter phase**. The query is distributed to all the shards that our index is built of. For example, if it is built of five shards and one replica, then five physical shards will be queried (we don't need to query both a shard and its replica because they contain the same data). Each of the queried shards will only return the document identifier and the score of the document. The node that sent the scatter query will wait for all the shards to complete their task, gather the results, and sort them appropriately (in this case, from the top scoring to the lowest scoring ones).

After that, a new request will be sent to build the search results. However, for now, the request will be sent only to those shards that held the documents to build the response. In most cases, Elasticsearch won't send the request to all the shards but only to its subset. This is because we usually don't get the entire result of the query but only a portion of it. This phase is called the **gather phase**. After all the documents have been gathered, the final response is built and returned as the query result.

Of course, the preceding behavior is the default in Elasticsearch and can be altered. The following section will describe how to change this behavior.

Search types

Elasticsearch allows us to choose how we want our query to be processed internally. We can do this by specifying the search type. There are different situations where different search types are appropriate; you may only care about performance, while sometimes, query relevance may be the most important factor. You should remember that each shard is a small Lucene index, and in order to return more relevant results, some information, such as frequencies, needs to be transferred between shards. To control how queries are executed, we can pass the `search_type` request parameter and set it to one of the following values:

- `query_then_fetch`: In the first step, the query is executed to get the information needed to sort and rank the documents. This step is executed against all the shards. Then, only the relevant shards are queried for the actual content of the documents. Different from `query_and_fetch`, the maximum number of results returned by this query type will be equal to the `size` parameter. This is the search type used by default if no search type has been provided with the query, and this is the query type we described earlier.
- `query_and_fetch`: This is usually the fastest and simplest search type implementation. The query is executed against all the shards (of course, only a single replica of a given primary shard will be queried) in parallel, and all the shards return the number of results equal to the value of the `size` parameter. The maximum number of returned documents will be equal to the value of `size` multiplied by the number of shards.
- `dfs_query_and_fetch`: This is similar to the `query_and_fetch` search type, but it contains an additional phase compared to `query_and_fetch`. The additional part is the initial query phase that is executed to calculate distributed term frequencies to allow more precise scoring of returned documents and thus more relevant query results.
- `dfs_query_then_fetch`: As with the previous `dfs_query_and_fetch` search type, the `dfs_query_then_fetch` search type is similar to its counterpart: `query_then_fetch`. However, it contains an additional phase compared to `query_then_fetch`, just like `dfs_query_and_fetch`.
- `count`: This is a special search type that only returns the number of documents that matched the query. If you only need to get the number of results but do not care about the documents, you should use this search type.

- `scan`: This is another special search type. Only use it if you expect your query to return a large amount of results. It differs a bit from the usual queries because after sending the first request, Elasticsearch responds with a scroll identifier, similar to a cursor in relational databases. All the queries need to be run against the `_search/scroll` REST endpoint and need to send the returned scroll identifier in the request body. You can learn more about this functionality in the *The scroll API* section of *Chapter 6, Beyond Full-text Searching*.

So if we want to use the simplest search type, we run the following command:

```
curl -XGET 'localhost:9200/library/book/_search?pretty=true&search_type=query_and_fetch' -d '{
  "query" : {
    "term" : { "title" : "crime" }
  }
}'
```

Search execution preferences

In addition to the possibility of controlling how the query is executed, we can also control the shards that we want to execute the query on. By default, Elasticsearch uses shards and replicas, both the ones available on the node that we've sent the request on and the other nodes in the cluster. And, the default behavior is in most cases the best query preference method. However, there may be times when we want to change the default behavior. For example, we may want the search to be executed on only the primary shards. To do this, we can set the `preference` request parameter to one of the following values shown in the table:

Parameter values	Description
<code>_primary</code>	This search operation will only be executed on the primary shards, so the replicas won't be used. This can be useful when we need to use the latest information from the index, but our data is not replicated right away.
<code>_primary_first</code>	This search operation will be executed on the primary shards if they are available. If not, it will be executed on the other shards.
<code>_local</code>	This search operation will only be executed on the shards available on the node that we are sending the request to (if possible).
<code>_only_node:node_id</code>	This search operation will be executed on the node with the provided node identifier.

Parameter values	Description
_prefer_node:node_id	Elasticsearch will try to execute this search operation on the node with the provided identifier. However, if the node is not available, it will be executed on the nodes that are available.
_shards:1,2	Elasticsearch will execute the operation on the shards with the given identifiers (in this case, on the shards with the identifiers 1 and 2). The _shards parameter can be combined with other preferences, but the shard identifiers need to be provided first, for example, _shards:1,2;_local.
Custom value	Any custom string value may be passed. The requests provided with the same values will be executed on the same shards.

For example, if we want to execute a query only on the local shards, we run the following command:

```
curl -XGET 'localhost:9200/library/_search?preference=_local' -d '{  
    "query" : {  
        "term" : { "title" : "crime" }  
    }  
}'
```

The Search shards API

When discussing the search preference, we would also like to mention the Search shards API exposed by Elasticsearch. This API allows us to check the shards that the query will be executed on. In order to use this API, run a request against the _search_shards REST endpoint. For example, to see how the query is executed, we run the following command:

```
curl -XGET 'localhost:9200/library/_search_shards?pretty' -d  
    '{"query":"match_all":{}}'
```

And, the response to the preceding command is as follows:

```
{  
    "nodes" : {  
        "N0iP_bh3QriX4NpqsqSUAg" : {  
            "name" : "Oracle",  
            "transport_address" : "inet[/192.168.1.19:9300]"  
    }  
}
```

```
        },
    ],
  "shards" : [ [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAg",
    "relocating_node" : null,
    "shard" : 0,
    "index" : "library"
} ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAg",
    "relocating_node" : null,
    "shard" : 1,
    "index" : "library"
} ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAg",
    "relocating_node" : null,
    "shard" : 4,
    "index" : "library"
} ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAg",
    "relocating_node" : null,
    "shard" : 3,
    "index" : "library"
} ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAg",
    "relocating_node" : null,
```

```
    "shard" : 2,  
    "index" : "library"  
} ] ]  
}
```

As you can see, in the response returned by Elasticsearch, we have the information about the shards that will be used during the query process. Of course, with the Search shards API, you can use all the parameters, such as routing or preference, and see how it affects your search execution.

Basic queries

Elasticsearch has extensive search and data analysis capabilities that are exposed in the form of different queries, filters, and aggregates, and so on. In this section, we will concentrate on the basic queries provided by Elasticsearch.

The term query

The `term` query is one of the simplest queries in Elasticsearch. It just matches the document that has a term in a given field – the exact, not analyzed term. The simplest `term` query is as follows:

```
{  
  "query" : {  
    "term" : {  
      "title" : "crime"  
    }  
  }  
}
```

The preceding query will match the documents that have the `crime` term in the `title` field. Remember that the `term` query is not analyzed, so you need to provide the exact term that will match the term in the indexed document. Please note that in our input data, we have the `title` field with the `Crime and Punishment` term, but we are searching for `crime` because the `Crime` term becomes `crime` after analysis during indexing.

In addition to the term we want to find, we can also include the `boost` attribute to our `term` query; it will affect the importance of the given term. We will talk more about boosts in the *An introduction to Apache Lucene scoring* section of *Chapter 5, Make Your Search Better*. For now, we just need to remember that it changes the importance of the given part of the query.

For example, to change our previous query and give a boost of `10.0` to our term query, we send the following query:

```
{  
  "query" : {  
    "term" : {  
      "title" : {  
        "value" : "crime",  
        "boost" : 10.0  
      }  
    }  
  }  
}
```

As you can see, the query changed a bit. Instead of a simple term value, we nested a new JSON object that contains the `value` property and the `boost` property. The value of the `value` property contains the term we are interested in, and the `boost` property is the boost value we want to use.

The terms query

The `terms` query allows us to match documents that have certain terms in their contents. The `term` query allowed us to match a single, not analyzed term, and the `terms` query allows us to match multiples of these. For example, let's say that we want to get all the documents that have the terms `novel` or `book` in the `tags` field. To achieve this, we run the following query:

```
{  
  "query" : {  
    "terms" : {  
      "tags" : [ "novel", "book" ],  
      "minimum_match" : 1  
    }  
  }  
}
```

The preceding query returns all the documents that have one or both of the searched terms in the `tags` field. Why is that? It is because we set the `minimum_match` property to `1`; this basically means that one term should match. If we want the query to match the document with all the provided terms, we set the `minimum_match` property to `2`.

The `match_all` query

The `match_all` query is one of the simplest queries available in Elasticsearch. It allows us to match all the documents in the index. If we want to get all the documents from our index, we just run the following query:

```
{  
  "query" : {  
    "match_all" : {}  
  }  
}
```

We can also include `boost` in the query, which will be given to all the documents matched by it. For example, if we want to add a boost of `2.0` to all the documents in our `match_all` query, we send the following query to Elasticsearch:

```
{  
  "query" : {  
    "match_all" : {  
      "boost" : 2.0  
    }  
  }  
}
```

The common terms query

The common terms query is a modern Elasticsearch solution for improving query relevance and precision with common words when we are not using stop words (http://en.wikipedia.org/wiki/Stop_words). For example, `crime` and `punishment` can be translated to three term queries and each of those term queries have a cost in terms of performance (the more the terms, the lower the performance of the query). However, the `and` term is a very common one, and its impact on the document score will be very low. The solution is the common terms query that divides the query into two groups. The first group is the one with important terms; these are the ones that have lower frequency. The second group is the less important terms that have higher frequency. The first query is executed first, and Elasticsearch calculates the score for all the terms from the first group. This way, the low frequency terms, which are usually the ones that have more importance, are always taken into consideration. Then, Elasticsearch executes the second query for the second group of terms but calculates the score only for the documents that were matched for the first query. This way, the score is only calculated for the relevant documents and thus, higher performance is achieved.

An example of the common terms query is as follows:

```
{
  "query" : {
    "common" : {
      "title" : {
        "query" : "crime and punishment",
        "cutoff_frequency" : 0.001
      }
    }
  }
}
```

The query can take the following parameters:

- **query**: This parameter defines the actual query contents.
- **cutoff_frequency**: This parameter defines the percentage (0.001 means 0.1 percent) or an absolute value (when the property is set to a value equal to or larger than 1). High and low frequency groups are constructed using this value. Setting this parameter to 0.001 means that the low frequency terms group will be constructed for terms that have a frequency of 0.1 percent and lower.
- **low_freq_operator**: This parameter can be set to `or` or `and` (defaults to `or`). It specifies the Boolean operator used to construct queries in the low frequency term group. If we want all of the terms to be present in a document for it to be considered a match, we should set this parameter to `and`.
- **high_freq_operator**: This parameter can be set to `or` or `and` (it defaults to `or`). It specifies the Boolean operator used to construct queries in the high frequency term group. If we want all of the terms to be present in a document for it to be considered a match, we should set this parameter to `and`.
- **minimum_should_match**: Instead of using the `low_freq_operator` and `high_freq_operator` parameters, we can use `minimum_should_match`. Just like with other queries, it allows us to specify the minimum number of terms that should be found in a document for it to be considered a match.
- **boost**: This parameter defines the boost given to the score of the documents.
- **analyzer**: This parameter defines the name of the analyzer that will be used to analyze the query text and defaults to the default analyzer.

- `disable_coord`: This parameter value defaults to `false` and allows us to enable or disable the score factor computation that is based on the fraction of all the query terms that a document contains. Set it to `true` for less precise scoring but slightly faster queries.



Unlike the `term` and `terms` queries, the common `terms` query is analyzed by Elasticsearch.



The match query

The `match` query takes the values given in the `query` parameter, analyzes them, and constructs the appropriate query out of them. When using a `match` query, Elasticsearch will choose the proper analyzer for a field we've chosen, so we can be sure that the terms passed to the `match` query will be processed by the same analyzer that was used during indexing. Please remember that the `match` query (and the `multi_match` query that will be explained later) doesn't support the Lucene query syntax; however, it fits perfectly as a query handler for our search box. The simplest `match` (and the default) query can look like the following:

```
{  
  "query" : {  
    "match" : {  
      "title" : "crime and punishment"  
    }  
  }  
}
```

The preceding query will match all the documents that have the terms, `crime`, and, or `punishment` in the `title` field. However, the previous query is only the simplest one; there are multiple types of `match` queries that we will discuss now.

The Boolean match query

The Boolean match query is a query that analyzes the provided text and makes a Boolean query out of it. There are a few parameters that allow us to control the behavior of the Boolean match queries; they are as follows:

- `operator`: This parameter can take the value of `or` or `and` and control the Boolean operator that is used to connect the created Boolean clauses. The default value is `or`. If we want all the terms in our query to match, we use the `and` Boolean operator.

- **analyzer:** This parameter specifies the name of the analyzer that will be used to analyze the query text and defaults to the default analyzer.
- **fuzziness:** Providing the value of this parameter allows us to construct fuzzy queries. It should take values from 0.0 to 1.0 for a `string` type. While constructing fuzzy queries, this parameter will be used to set the similarity.
- **prefix_length:** This parameter allows us to control the behavior of the fuzzy query. For more information on the value of this parameter, refer to the *The fuzzy_like_this query* section in this chapter.
- **max_expansions:** This parameter allows us to control the behavior of the fuzzy query. For more information on the value of this parameter, please refer to the *The fuzzy_like_this query* section in this chapter.
- **zero_terms_query:** This parameter allows us to specify the behavior of the query when all the terms are removed by the analyzer (for example, because of stop words). It can be set to `none` or `all`, with `none` as the default value. When set to `none`, no documents will be returned when the analyzer removes all the query terms. All the documents will be returned on setting this parameter to `all`.
- **cutoff_frequency:** This parameter allows us to divide the query into two groups: one with high frequency terms and one with low frequency terms. Refer to the description of the common terms query to see how this parameter can be used.

The parameters should be wrapped in the name of the field that we are running the query for. So if we wish to run a sample Boolean match query against the `title` field, we send a query as follows:

```
{
  "query" : {
    "match" : {
      "title" : {
        "query" : "crime and punishment",
        "operator" : "and"
      }
    }
  }
}
```

The `match_phrase` query

A `match_phrase` query is similar to the Boolean query, but instead of constructing the Boolean clauses from the analyzed text, it constructs the phrase query. The following parameters are available for this query:

- `slop`: This is an integer value that defines how many unknown words can be put between terms in the text query for a match to be considered a phrase. The default value of this parameter is 0, which means that no additional words are allowed.
- `analyzer`: This parameter specifies the name of the analyzer that will be used to analyze the query text and defaults to the default analyzer.

A sample `match_phrase` query against the `title` field looks like the following code:

```
{  
    "query" : {  
        "match_phrase" : {  
            "title" : {  
                "query" : "crime punishment",  
                "slop" : 1  
            }  
        }  
    }  
}
```

Note that we removed the `and` term from our query, but since the `slop` parameter is set to 1, it will still match our document.

The `match_phrase_prefix` query

The last type of the match query is the `match_phrase_prefix` query. This query is almost the same as the `match_phrase` query, but in addition, it allows prefix matches on the last term in the query text. Also, in addition to the parameters exposed by the `match_phrase` query, it exposes an additional one: the `max_expansions` parameter. This controls how many prefixes will be rewritten to the last terms. Our example query when changed to the `match_phrase_prefix` query will look like the following:

```
{  
    "query" : {  
        "match_phrase_prefix" : {  
            "title" : {  
                "query" : "crime punis",  
                "max_expansions" : 1  
            }  
        }  
    }  
}
```

```
        "query" : "crime and punishment",
        "slop" : 1,
        "max_expansions" : 20
    }
}
}
}
```

Note that we didn't provide the full crime and punishment phrase but only crime and punishment, and the query still matches our document.

The multi_match query

The `multi_match` query is the same as the `match` query, but instead of running against a single field, it can be run against multiple fields with the use of the `fields` parameter. Of course, all the parameters you use with the `match` query can be used with the `multi_match` query. So, if we want to modify our `match` query to be run against the `title` and `otitle` fields, we run the following query:

```
{
    "query" : {
        "multi_match" : {
            "query" : "crime punishment",
            "fields" : [ "title", "otitle" ]
        }
    }
}
```

In addition to the previously mentioned parameters, the `multi_match` query exposes the following additional parameters that allow more control over its behavior:

- `use_dis_max`: This parameter defines the Boolean value that allows us to set whether the `dismax` (`true`) or `boolean` (`false`) queries should be used. It defaults to `true`. You can read more about the `dismax` query in the *The dismax query* section of this chapter.
 - `tie_breaker`: This parameter is only used when the `use_dis_max` parameter is set to `true` and allows us to specify the balance between lower and maximum scoring query items. You can read more about it in the *The dismax query* section of this chapter.

The query_string query

In comparison to the other queries available, the `query_string` query supports the full Apache Lucene query syntax, which we discussed earlier in the *The Lucene query syntax* section in *Chapter 1, Getting Started with the Elasticsearch Cluster*. It uses a query parser to construct an actual query using the provided text. An example of the `query_string` query is as follows:

```
{  
  "query" : {  
    "query_string" : {  
      "query" : "title:crime^10 +title:punishment -otitle:cat  
      +author:(+Fyodor +dostoevsky)",  
      "default_field" : "title"  
    }  
  }  
}
```

Because we are familiar with the basics of the Lucene query syntax, we can discuss how the preceding query works. As you can see, we wanted to get the documents that may have the term `crime` in the `title` field, and such documents should be boosted with the value of `10`. Next, we want only the documents that have the `punishment` term in the `title` field and not the documents with the `cat` term in the `otitle` field. Finally, we tell Lucene that we only want the documents that have the `Fyodor` and `dostoevsky` terms in the `author` field.

Like most of the queries in Elasticsearch, the `query_string` query provides the following parameters that allow us to control the query behavior:

- `query`: This parameter specifies the query text.
- `default_field`: This parameter specifies the default field that the query will be executed against. It defaults to the `index.query.default_field` property, which is set to `_all` by default.
- `default_operator`: This parameter specifies the default logical operator (`or` or `and`) that is used when no operator is specified. The default value of this parameter is `or`.
- `analyzer`: This parameter specifies the name of the analyzer that is used to analyze the query provided in the `query` parameter.
- `allow_leading_wildcard`: This parameter specifies whether a wildcard character is allowed as the first character of a term. It defaults to `true`.

- `lowercase_expand_terms`: This parameter specifies whether the terms that are a result of a query rewrite should be lowercased. It defaults to `true`, which means that the rewritten terms will be lowercased.
- `enable_position_increments`: This parameter specifies whether the position increments should be turned on in the result query. It defaults to `true`.
- `fuzzy_max_expansions`: This parameter specifies the maximum terms that the fuzzy query will be expanded into if used. It defaults to 50.
- `fuzzy_prefix_length`: This parameter specifies the prefix length for the generated fuzzy queries and defaults to 0. To learn more about it, refer to the *The fuzzy query* section.
- `fuzzy_min_sim`: This parameter specifies the minimum similarity for the fuzzy queries and defaults to 0.5. To learn more about it, refer to the *The fuzzy query* section.
- `phrase_slop`: This parameter specifies the phrase slop value and defaults to 0. To learn more about it, refer to the *The match_phrase query* section.
- `boost`: This parameter specifies the boost value that will be used and defaults to 1.0.
- `analyze_wildcard`: This parameter specifies whether the terms generated by the wildcard query should be analyzed. It defaults to `false`, which means that the terms won't be analyzed.
- `auto_generate_phrase_queries`: This parameter specifies whether phrase queries will be generated from the query automatically. It defaults to `false`, which means that the phrase queries won't be generated automatically.
- `minimum_should_match`: This parameter controls how many generated Boolean should clauses must match against a document for it to be considered a hit. The value can be provided as a percentage, for example, 50%. This means that at least 50 percent of the given terms should match. It can also be provided as an integer value, such as 2, which means that at least two terms must match.
- `lenient`: This parameter takes the value of `true` or `false`. If set to `true`, format-based failures will be ignored.

DisMax is an abbreviation of Disjunction Max. **Disjunction** refers to the fact that the search is executed across multiple fields, and the fields can be given different boost weights. **Max** means that only the maximum score for a given term will be included in a final document score and not the sum of all the scores from all fields that have the matched term (like a simple Boolean query would do).

Note that Elasticsearch can rewrite the `query_string` query, and because of this, Elasticsearch allows us to pass additional parameters that control the `rewrite` method. However, for more details on this process, refer to the *Understanding the querying process* section in this chapter.

Running the `query_string` query against multiple fields

It is possible to run the `query_string` query against multiple fields. In order to do this, one needs to provide the `fields` parameter in the query body, which holds the array of field names. There are two methods of running the `query_string` query against multiple fields; the default method uses the Boolean query to make queries and the other method uses the `dismax` query.

In order to use the `dismax` query, you should add the `use_dis_max` property in the query body and set it to `true`. An example query is as follows:

```
{
  "query" : {
    "query_string" : {
      "query" : "crime punishment",
      "fields" : [ "title", "otitle" ],
      "use_dis_max" : true
    }
  }
}
```

The `simple_query_string` query

The `simple_query_string` query uses one of the newest query parsers in Lucene: `SimpleQueryParser`. Similar to the `query_string` query, it accepts the Lucene query syntax as the query; however unlike it, the `simple_query_string` query never throws an exception when an error is parsed. Instead of throwing an exception, it discards the invalid parts of the query and runs the rest of the parts.

An example of the `simple_query_string` query is as follows:

```
{
  "query" : {
    "simple_query_string" : {
      "query" : "title:crime^10 +title:punishment -otitle:cat
+author:(+Fyodor +dostoevsky)",
      ...
    }
  }
}
```

```
        "default_operator" : "and"
    }
}
}
```

This query supports parameters similar to the ones exposed by the `query_string` query, and similarly, it can also be run against multiple fields using the `fields` property.

The identifiers query

The identifiers query is a simple query that filters the returned documents to only those queries with provided identifiers. This query works on the internal `_uid` field, so it doesn't require the `_id` field to be enabled. The simplest version of such a query looks like the following:

```
{
  "query" : {
    "ids" : {
      "values" : [ "10", "11", "12", "13" ]
    }
  }
}
```

This query only returns documents that have one of the identifiers present in the `values` array. We can complicate the identifiers query a bit and also limit the documents on the basis of their type. For example, if we want to only include documents from the `book` type, we send the following query:

```
{
  "query" : {
    "ids" : {
      "type" : "book",
      "values" : [ "10", "11", "12", "13" ]
    }
  }
}
```

As you can see, we added the `type` property to our query and set its value to the type we are interested in.

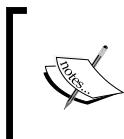
The prefix query

The `prefix` query is similar to the `term` query in terms of its configuration. The `prefix` query allows us to match documents that have the value in a certain field and starts with a given prefix. For example, if we want to find all the documents that have values starting with `cri` in the `title` field, we run the following query:

```
{  
  "query" : {  
    "prefix" : {  
      "title" : "cri"  
    }  
  }  
}
```

Similar to the `term` query, we can also include the `boost` attribute to our `prefix` query; this will affect the importance of the given prefix. For example, if we want to change our previous query and give it a boost of `3.0`, we send the following query:

```
{  
  "query" : {  
    "prefix" : {  
      "title" : {  
        "value" : "cri",  
        "boost" : 3.0  
      }  
    }  
  }  
}
```



The `prefix` query is rewritten by Elasticsearch; therefore, Elasticsearch allows us to pass an additional parameter by controlling the `rewrite` method. However, for more details on this process, refer to the *Understanding the querying process* section of this chapter.

The fuzzy_like_this query

The `fuzzy_like_this` query is similar to the `more_like_this` query. It finds all the documents that are similar to the provided text, but it works a bit differently than the `more_like_this` query. It makes use of fuzzy strings and picks the best differencing terms that were produced. For example, if we want to run a `fuzzy_like_this` query against the `title` and `otitle` fields to find all the documents similar to the `crime punishment` query, we run the following query:

```
{
  "query" : {
    "fuzzy_like_this" : {
      "fields" : ["title", "otitle"],
      "like_text" : "crime punishment"
    }
  }
}
```

The following query parameters are supported by the `fuzzy_like_this` query:

- `fields`: This parameter defines an array of fields against which the query should be run. It defaults to the `_all` field.
- `like_text`: This is a required parameter that holds the text that we compare the documents to.
- `ignore_tf`: This parameter specifies whether term frequencies should be ignored during similarity computation. It defaults to `false`, which means that the term frequencies will be used.
- `max_query_terms`: This parameter specifies the maximum number of query terms that will be included in a generated query. It defaults to 25.
- `min_similarity`: This parameter specifies the minimum similarity that differencing terms should have. It defaults to 0.5.
- `prefix_length`: This parameter specifies the length of the common prefix of the differencing terms. It defaults to 0.

- **boost:** This parameter specifies the boost value that will be used when boosting a query. It defaults to 1.0.
- **analyzer:** This parameter specifies the name of the analyzer that will be used to analyze the text we provided.

The `fuzzy_like_this_field` query

The `fuzzy_like_this_field` query is similar to the `fuzzy_like_this` query, but it only works against a single field. Because of this, it doesn't support the `fields` property. Instead of specifying the fields that should be used for query analysis, we should wrap the query parameters into that field name. Our example query to a `title` field should look like the following:

```
{  
  "query" : {  
    "fuzzy_like_this_field" : {  
      "title" : {  
        "like_text" : "crime and punishment"  
      }  
    }  
  }  
}
```

All the other parameters from the `fuzzy_like_this_field` query work the same for this query.

The `fuzzy` query

The `fuzzy` query is the third type of fuzzy query; it matches the documents on the basis of the edit distance algorithm. The edit distance is calculated on the basis of terms we provide in the query and against the searched documents. This query can be expensive when it comes to CPU resources, but it can help us when we need fuzzy matching, for example, when users make spelling mistakes. In our example, let's assume that instead of `crime`, our user enters the word `crme` into the search box and we want to run the simplest form of fuzzy query. Such a query would look like the following:

```
{  
  "query" : {  
    "fuzzy" : {  
      "title" : "crme"  
    }  
  }  
}
```

```
    }
}
```

And, the response for such a query would be as follows:

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641, "_source" : { "title": "Crime and Punishment", "otitle": "Преступлénie и наказánie", "author": "Fyodor Dostoevsky", "year": 1886, "characters": ["Raskolnikov", "Sofia Semyonovna Marmeladova"], "tags": [], "copies": 0, "available": true}
    } ]
  }
}
```

Even though we made a typo, Elasticsearch managed to find the documents we were interested in.

We can control the behavior of the `fuzzy` query using the following parameters:

- `value`: This parameter specifies the actual query.
- `boost`: This parameter specifies the boost value for the query. It defaults to 1.0.
- `min_similarity`: This parameter specifies the minimum similarity a term must have to count as a match. In the case of string fields, this value should be between 0 and 1 inclusive. For numeric fields, this value can be greater than one; for example, if the query value is equal to 20 and `min_similarity` is set to 3, we get values from 17 to 23. For date fields, we can have the `min_similarity` values that include 1d, 2d, 1m, and so on. These values correspond to one day, two days, and one month, respectively.

- `prefix_length`: This parameter defines the length of the common prefix of the differencing terms; it defaults to 0.
- `max_expansions`: This parameter specifies the maximum number of terms that the query will be expanded to. The default value is unbounded.

The parameters should be wrapped in the name of the field that we are running the query against. So if we want to modify the previous query and add additional parameters, the query will look like the following code:

```
{  
    "query" : {  
        "fuzzy" : {  
            "title" : {  
                "value" : "crme",  
                "min_similarity" : 0.2  
            }  
        }  
    }  
}
```

The wildcard query

The wildcard query allows us to use the * and ? wildcards in the values that we search for. Apart from this, the wildcard query is very similar to the `term` query in terms of its content. To send a query that matches all the documents with the value of the `cr?me` term, where ? means any character, we send the following query:

```
{  
    "query" : {  
        "wildcard" : {  
            "title" : "cr?me"  
        }  
    }  
}
```

This will match the documents that have all the terms that match `cr?me` in the `title` field. However, you can also include the `boost` attribute to your wildcard query; it will affect the importance of each term that matches the given value. For example, if we want to change our previous query and give a boost of 20.0 to our `term` query, we send the following query:

```
{  
    "query" : {  
        "wildcard" : {
```

```
        "title" : {  
            "value" : "cr?me",  
            "boost" : 20.0  
        }  
    }  
}
```



Note that wildcard queries are not very performance-oriented queries and should be avoided if possible; in particular, avoid leading wildcards (the terms that start with wildcards). Also, note that the wildcard query is rewritten by Elasticsearch, and because of this, Elasticsearch allows us to pass an additional parameter that controls the rewrite method. For more details on this process, refer to the *Understanding the querying process* section of this chapter.

The more_like_this query

The `more_like_this` query allows us to get documents that are similar to the provided text. Elasticsearch supports a few parameters to define how the `more_like_this` query should work; they are as follows:

- **fields**: This parameter defines an array of fields that the query should be run against. It defaults to the `_all` field.
 - **like_text**: This is a required parameter that holds the text that we compare the documents to.
 - **percent_terms_to_match**: This parameter specifies the percentage of terms from the query that need to match in a document for that document to be considered similar. It defaults to `0.3`, which means 30 percent.
 - **min_term_freq**: This parameter specifies the minimum term frequency (for the terms in the documents) below which terms will be ignored. It defaults to `2`.
 - **max_query_terms**: This parameter specifies the maximum number of terms that will be included in any generated query. It defaults to `25`. A higher value may mean higher precision, but lower performance.
 - **stop_words**: This parameter defines an array of words that will be ignored when comparing the documents and the query. It is empty by default.
 - **min_doc_freq**: This parameter defines the minimum number of documents in which the term has to be present to not be ignored. It defaults to `5`, which means that a term needs to be present in at least five documents.

- `max_doc_freq`: This parameter defines the maximum number of documents in which a term may be present in order to not be ignored. By default, it is unbounded.
- `min_word_len`: This parameter defines the minimum length of a single word below which the word will be ignored. It defaults to 0.
- `max_word_len`: This parameter defines the maximum length of a single word above which the word will be ignored. It is unbounded by default.
- `boost_terms`: This parameter defines the boost value that will be used to boost each term. It defaults to 1.
- `boost`: This parameter defines the boost value that will be used to boost the query. It defaults to 1.
- `analyzer`: This parameter defines the name of the analyzer that will be used to analyze the text we provided.

An example of the `more_like_this` query is as follows:

```
{  
    "query" : {  
        "more_like_this" : {  
            "fields" : [ "title", "otitle" ],  
            "like_text" : "crime and punishment",  
            "min_term_freq" : 1,  
            "min_doc_freq" : 1  
        }  
    }  
}
```

The `more_like_this_field` query

The `more_like_this_field` query is similar to the `more_like_this` query, but it works only against a single field. Because of this, it doesn't support the `fields` property. Instead of specifying the fields that should be used for query analysis, we wrap the query parameters into the field name. So, our example query of the `title` field is as follows:

```
{  
    "query" : {  
        "more_like_this_field" : {  
            "title" : {  
                "like_text" : "crime and punishment",  
                "min_term_freq" : 1,  
            }  
        }  
    }  
}
```

```
        "min_doc_freq" : 1
    }
}
}
}
```

All the other parameters from the `more_like_this` query work in the same way for this query.

The range query

The range query allows us to find documents that have a field value within a certain range and work both for numerical fields as well as for string-based fields (it just maps to a different Apache Lucene query). The range query should be run against a single field, and the query parameters should be wrapped in the field name.

The following parameters are supported by the range query:

- `gte`: The range query will match the documents with a value greater than or equal to the ones provided with this parameter
- `gt`: The range query will match the documents with a value greater than the one provided with this parameter
- `lte`: The range query will match the documents with a value lower than or equal to the ones provided with this parameter
- `lt`: The range query will match the documents with a value lower than the one provided with this parameter

So, for example, if we want to find all the books that have a value from 1700 to 1900 in the `year` field, we run the following query:

```
{
  "query" : {
    "range" : {
      "year" : {
        "gte" : 1700,
        "lte" : 1900
      }
    }
  }
}
```

The dismax query

The `dismax` query is very useful as it generates a union of documents returned by all of the subqueries and returns it as the result. The good thing about this query is the fact that we can control how the lower scoring subqueries affect the final score of the documents.

The final document score is calculated as the sum of scores of the maximum scoring query and the sum of scores returned from the rest of the queries, multiplied by the value of the `tie` parameter. So, the `tie_breaker` parameter allows us to control how the lower scoring queries affect the final score. If we set the `tie_breaker` parameter to `1.0`, we get the exact sum, while setting the `tie` parameter to `0.1` results in only 10 percent of the scores (of all the scores apart from the maximum scoring query) being added to the final score.

An example of the `dismax` query is as follows:

```
{  
    "query" : {  
        "dismax" : {  
            "tie_breaker" : 0.99,  
            "boost" : 10.0,  
            "queries" : [  
                {  
                    "match" : {  
                        "title" : "crime"  
                    }  
                },  
                {  
                    "match" : {  
                        "author" : "fyodor"  
                    }  
                }  
            ]  
        }  
    }  
}
```

As you can see, we included the `tie_breaker` and `boost` parameters. In addition to that, we specified the `queries` parameter that holds the array of queries that will be run and used to generate the union of documents for results.

The regular expression query

The regular expression query allows us to use regular expressions as the query text. Remember that the performance of such queries depends on the chosen regular expression. If our regular expression matches many terms, the query will be slow. The general rule is that the higher the volume of the terms matched by the regular expression, the slower the query will be.

An example of the regular expression query is as follows:

```
{
  "query" : {
    "regexp" : {
      "title" : {
        "value" : "cr.m[ae]",
        "boost" : 10.0
      }
    }
  }
}
```

The preceding query will result in Elasticsearch rewriting the query to a number of term queries depending on the content of our index that matches the given regular expression. The `boost` parameter seen in the query specifies the boost value for the generated queries.



The full regular expression syntax accepted by Elasticsearch can be found at <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-regexp-query.html#regexp-syntax>.

Compound queries

In the *Basic queries* section of this chapter, we discussed the simplest queries exposed by Elasticsearch. However, the simple ones are not the only queries that Elasticsearch provides. The compound queries, as we call them, allow us to connect multiple queries together or alter the behavior of other queries. You may wonder if you need such functionality. A simple exercise to determine this would be to combine a simple `term` query with a phrase query in order to get better search results.

The `bool` query

The `bool` query allows us to wrap a virtually unbounded number of queries and connect them with a logical value using one of the following sections:

- `should`: The `bool` query when wrapped into this section may or may not match—the number of `should` sections that have to match is controlled by the `minimum_should_match` parameter
- `must`: The `bool` query when wrapped into this section must match in order for the document to be returned
- `must_not`: The `bool` query when wrapped into this section must not match in order for the document to be returned

Each of these sections can be present multiple times in a single `bool` query.

This allows us to build very complex queries that have multiple levels of nesting (you can include the `bool` query in another `bool` query). Remember that the score of the resulting document will be calculated by taking a sum of all the wrapped queries that the document matched.

In addition to the preceding sections, we can add the following parameters to the query body to control its behavior:

- `boost`: This parameter specifies the boost used in the query, defaulting to `1.0`. The higher the boost, the higher the score of the matching document.
- `minimum_should_match`: The value of this parameter describes the minimum number of `should` clauses that have to match in order for the checked document to be counted as a match. For example, it can be an integer value such as `2` or a percentage value such as `75%`. For more information, refer to <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-minimum-should-match.html>.
- `disable_coord`: This parameter defaults to `false` and allows us to enable or disable the score factor computation that is based on the fraction of all the query terms that a document contains. We should set it to `true` for less precise scoring, but slightly faster queries.

Imagine that we want to find all the documents that have the term `crime` in the `title` field. In addition, the documents may or may not have a range of `1900` to `2000` in the `year` field and may not have the `nothing` term in the `otitle` field. Such a query made with the `bool` query will look like the following code:

```
{  
  "query" : {  
    "bool" : {  
      "must" : {
```

```
        "term" : {
            "title" : "crime"
        }
    },
    "should" : {
        "range" : {
            "year" : {
                "from" : 1900,
                "to" : 2000
            }
        }
    },
    "must_not" : {
        "term" : {
            "otitle" : "nothing"
        }
    }
}
```



Note that the must, should, and must_not sections can contain a single query or an array of multiple queries.

The boosting query

The boosting query wraps around two queries and lowers the score of the documents returned by one of the queries. There are three sections of the boosting query that need to be defined—the positive section that holds the query whose document score will be left unchanged, the negative section whose resulting documents will have their score lowered, and the `negative_boost` section that holds the boost value that will be used to lower the second section's query score. The advantage of the boosting query is that the results of both the queries included in it (the negative and the positive ones) will be present in the results, although the scores of some queries will be lowered. For example, if we were to use the `bool` query with the `must_not` section, we wouldn't get the results for such a query.

Let's assume that we want to have the results of a simple `term` query for the term `crime` in the `title` field and want the score of such documents to not be changed. However, we also want to have the documents that range from 1800 to 1900 in the `year` field and the scores of documents returned by such a query to have an additional boost of 0.5. Such a query will look like the following:

```
{  
  "query" : {
```

```
    "boosting" : {
      "positive" : {
        "term" : {
          "title" : "crime"
        }
      },
      "negative" : {
        "range" : {
          "year" : {
            "from" : 1800,
            "to" : 1900
          }
        }
      },
      "negative_boost" : 0.5
    }
  }
}
```

The `constant_score` query

The `constant_score` query wraps another query (or filter) and returns a constant score for each document returned by the wrapped query (or filter). It allows us to strictly control the score value assigned for a document matched by a query or filter. For example, if we want to have a score of `2.0` for all the documents that have the term `crime` in the `title` field, we send the following query to Elasticsearch:

```
{
  "query" : {
    "constant_score" : {
      "query" : {
        "term" : {
          "title" : "crime"
        }
      },
      "boost" : 2.0
    }
  }
}
```

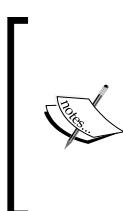
The indices query

The `indices` query is useful when executing a query against multiple indices. It allows us to provide an array of indices (the `indices` property) and two queries, one that will be executed if we query the index from the list (the `query` property) and the second that will be executed on all the other indices (the `no_match_query` property). For example, assume we have an alias name, `books`, holding two indices—`library` and `users`—and we want to use this alias; however, we want to run different queries on those indices. To do this, we send the following query:

```
{
  "query" : {
    "indices" : {
      "indices" : [ "library" ],
      "query" : {
        "term" : {
          "title" : "crime"
        }
      },
      "no_match_query" : {
        "term" : {
          "user" : "crime"
        }
      }
    }
  }
}
```

In the preceding query, the query described in the `query` property was run against the `library` index and `no_match_query` was run against all the other indices present in the cluster.

The `no_match_query` property can also have a string value instead of a query. This string value can either be `all` or `none` and will default to `all`. If the `no_match_query` property is set to `all`, the documents from the indices that don't match will be returned. Setting the `no_match_query` property to `none` will result in no documents from the indices that don't match.



Some of the queries exposed by Elasticsearch, such as the `custom_score` query, the `custom_boost_factor` query, and the `custom_filters_score` query, are replaced by the `function_score` query, which we describe in the *The function_score query* section of *Chapter 5, Make Your Search Better*. We decided to omit the description of these queries as they will probably be removed in the future versions of Elasticsearch.

Filtering your results

We already know how to build queries and search for data using different criteria and queries. We are also familiar with scoring (refer to the *Scoring and query relevance* section of *Chapter 1, Getting Started with the Elasticsearch Cluster*), which tells us which document is more important in a given query and how our query text affects ordering. However, sometimes we may want to choose only a subset of our index without influencing the final score. This is where filters should be used (of course, this is not the only reason why).

To be perfectly honest, use filters whenever possible. Filters don't affect scoring, and score calculation complicates searches and requires CPU power. On the other hand, filtering is a relatively simple operation. Due to the fact that filtering is applied on the contents of the whole index, the result of the filtering is independent of the documents that were found and the relationship between them. Filters can easily be cached, further improving the overall performance of the filtered queries.

In the following sections about filters, we've used the `post_filter` parameter to keep the examples as simple as possible. However, remember that if possible, you should always use the `filtered` query instead of `post_filter` because query execution using `filtered` will be faster.

Using filters

To use a filter in any search, just add a `filter` section on the same level as the `query` section. You can also omit the `query` section completely if you only want to have filters. Let's take an example query that searches for `Catch-22` in the `title` field and add a filter to it as follows:

```
{  
  "query" : {  
    "match" : { "title" : "Catch-22" }  
  },  
  "post_filter" : {  
    "term" : { "year" : 1961 }  
  }  
}
```

This returned all the documents with the given title, but that result was narrowed only to the books published in 1961. There is also a second way to include a filter in our query: using the `filtered` query. So our preceding query can be rewritten as follows:

```
{  
  "query": {  
    "filtered": {  
      "filter": {  
        "term": { "year": 1961 }  
      }  
    }  
  }  
}
```

```
    "filtered" : {
      "query" : {
        "match" : { "title" : "Catch-22" }
      },
      "filter" : {
        "term" : { "year" : 1961 }
      }
    }
  }
}
```

If you run both the queries by sending the `curl -XGET localhost:9200/library/_search?pretty -d @query.json` command, you will see that both the responses are exactly the same (except, perhaps, the response time):

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.2712221,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "2",
      "_score" : 0.2712221, "_source" : { "title": "Catch-22", "author": "Joseph Heller", "year": 1961, "characters": ["John Yossarian", "Captain Aardvark", "Chaplain Tappman", "Colonel Cathcart", "Doctor Daneeka"], "tags": ["novel"], "copies": 6, "available": false}
    } ]
  }
}
```

This suggests that both forms are equivalent. This is not true because of the different orders that the filter and search are applied in. In the first case, filters are applied to all the documents found by the query. In the second case, the documents are filtered *before* the query is run. This yields better performance. As mentioned earlier, filters are fast, so a filtered query is more efficient. We will return to this in the *Faceting* section of *Chapter 6, Beyond Full-text Searching*.

Filter types

We now know how to use filters. We also know what the differences between the mentioned filtering methods are. Let's now take a look at the filter types provided by Elasticsearch.

The range filter

The range filter allows us to limit searching to only those documents where the value of a field is between the given boundaries. For example, to construct a filter that will filter the results to the books published only between 1930 and 1990, we have the following part of the query:

```
{  
  "post_filter" : {  
    "range" : {  
      "year" : {  
        "gte": 1930,  
        "lte": 1990  
      }  
    }  
  }  
}
```

Using `gte` and `lte`, we indicate that the left and right boundaries of the field are inclusive. If we want to exclude any of the bounds, we can use the `gt` and `lt` version of the parameter. For example, if we want to have documents from 1930 (including the ones with this value) to 1990 (excluding the ones with this value), we construct the following filter:

```
{  
  "post_filter" : {  
    "range" : {  
      "year" : {  
        "gt": 1930,  
        "lt": 1990  
      }  
    }  
  }  
}
```

```
        "gte": 1930,  
        "lt": 1990  
    }  
}  
}  
}
```

Let's summarize this as follows:

- `gt`: This means greater than
- `lt`: This means lower than
- `gte`: This means greater or equals to
- `lte`: This means lower or equals to

You can also use the execution parameter. This is a hint to the engine on how to execute the filter. The available values are `fielddata` and `index`. The rule of thumb is that the `fielddata` value should increase performance (and memory usage) when there are many values in the range, and when there are less values in the range, the `index` value should be better.

There is also a second variant of this filter: `numeric_filter`. It is a specialized version that has been designed to filter on the range values that are numerical. This filter is faster but comes with a requirement for additional memory usage—Elasticsearch needs to load the values of a field that we filter on. Note that sometimes these values will be loaded independent of the range filter. In such cases, there is no reason not to use this filter. This happens if we use the same field for faceting or sorting.

The exists filter

The `exists` filter is a very simple one. It filters out documents that don't have a value in the given field. For example, consider the following code:

```
{  
  "post_filter" : {  
    "exists" : { "field": "year" }  
  }  
}
```

The preceding filter results in a query that returns documents with a value in the `year` field.

The missing filter

The `missing` filter is the opposite of the `exists` filter; it filters out documents with a value in a given field. However, it has a few additional features. Besides selecting the documents where the specified fields are missing, we can define what Elasticsearch should treat as an empty field. This helps in situations where the input data contains tokens such as `null`, `EMPTY`, and `not-defined`. Let's change our previous example to find all the documents without the `year` field defined or the ones that have the `year` field equal to 0. So, the modified filter will look like the following:

```
{  
  "post_filter" : {  
    "missing" : {  
      "field": "year",  
      "null_value": 0,  
      "existence": true  
    }  
  }  
}
```

In the preceding example, you see two parameters in addition to the previous ones. The `existence` parameter tells Elasticsearch that it should check the documents with a value that exists in the specified field, and the `null_value` parameter defines the additional value to be treated as empty. If you didn't define `null_value`, the `existence` value will be set by default; so, you can omit `existence` in this case.

The script filter

Sometimes, we want to filter our documents by a computed value. A good example for our case can be to filter out all the books that were published more than a century ago. We do this using the `script` filter, as follows:

```
{  
  "post_filter" : {  
    "script" : {  
      "script" : "now - doc['year'].value > 100",  
      "params" : {  
        "now" : 2012  
      }  
    }  
  }  
}
```

As you can see, we used a simple script to calculate the value and filter data in this calculation. We will talk more about the scripting capabilities of Elasticsearch in the *Scripting capabilities of Elasticsearch* section of *Chapter 5, Make Your Search Better*.

The type filter

The type filter is dedicated to limiting documents by type. It can be useful when our query is run against several indices or an index with numerous types. For example, if we want to limit the returned documents to the ones with the `book` type, we use the following filter:

```
{
  "post_filter": {
    "type": {
      "value": "book"
    }
  }
}
```

The limit filter

The limit filter limits the number of documents returned by a single shard. This should not be confused with the `size` parameter. For example, let's take a look at the following filter:

```
{
  "post_filter": {
    "limit": {
      "value": 1
    }
  }
}
```

When we use the default settings for a number of shards, the preceding filter returns up to five documents. This is because indices in Elasticsearch are divided into five shards by default. Each shard is queried separately, and each shard may return one document at most.

The identifiers filter

The `ids` filter helps when we have to filter out several concrete documents. For example, if we need to exclude one document with an identifier that is equal to 1, the filter will look like the following code:

```
{
  "post_filter": {
```

```
        "ids" : {  
          "type": ["book"],  
          "values": [1]  
        }  
      }  
    }
```

Note that the `type` parameter is not required. However, it is useful when we are searching among several indices to specify a type that we are interested in.

If this is not enough

So far, we discussed a few examples of the filters used in Elasticsearch. However, this is only the tip of the iceberg. You can wrap almost every query into a filter. For example, let's take a look at the following query:

```
{  
  "query" : {  
    "multi_match" : {  
      "query" : "novel erich",  
      "fields" : [ "tags", "author" ]  
    }  
  }  
}
```

The preceding example shows a simple `multi_match` query that we are already familiar with. This query can be rewritten as a filter, as follows:

```
{  
  "post_filter" : {  
    "query" : {  
      "multi_match" : {  
        "query" : "novel erich",  
        "fields" : [ "tags", "author" ]  
      }  
    }  
  }  
}
```

Of course, the only difference in the result will be the scoring. Every document returned by the filter will have a score of `1.0`. Note that Elasticsearch has a few dedicated filters that act this way (for example, the `term` query and the `term` filter). So, you don't have to always use wrapped query syntax. In fact, you should always use a dedicated version wherever possible.

The following dedicated filters are available in Elasticsearch:

- The `bool` filter
- The `geo_shape` filter
- The `has_child` filter
- The `has_parent` filter
- The `ids` filter
- The `indices` filter
- The `match_all` filter
- The `nested` filter
- The `prefix` filter
- The `range` filter
- The `regexp` filter
- The `term` filter
- The `terms` filter

Combining filters

Now, it's time to combine some filters together. The first option is to use the `bool` filter, which can group filters on the same basis as described in *The bool query* section. The second option is to use `and`, `or`, and `not` filters. The `and` filter takes an array of filters and returns the documents that match all of the filters in the array. The `or` filter also takes an array of filters, but it only returns the documents matching any of the defined filters. In the case of the `not` filter, the returned documents are the ones that were not matched by the enclosed filter. Of course, all these filters may be nested, as shown in the following example:

```
{  
  "post_filter": {  
    "not": {  
      "and": [  
        {  
          "term": {  
            "title": "Catch-22"  
          }  
        },  
        {  
          "or": [  
            {  
              "not": {  
                "term": {  
                  "title": "The Catcher in the Rye"  
                }  
              }  
            }  
          ]  
        }  
      ]  
    }  
  }  
}
```

```
{  
    "range": {  
        "year": {  
            "gte": 1930,  
            "lte": 1990  
        }  
    },  
    {  
        "term": {  
            "available": true  
        }  
    }  
}  
]  
}  
]  
}
```

A word about the bool filter

Of course, you should ask about the difference between the `bool` filter and the `and`, `or`, and `not` filters. The first thing is that the filters can be used interchangeably. Of course, it is true from the perspective of the returned results but not the performance.

If we look at the Elasticsearch internals, we will see that for every filter, a structure called **bitset** is built. It holds information about whether a subsequent document in the index matches the filter. The bitset can easily be cached and reused for all the queries using the same filter. This is an easy and efficient task for Elasticsearch.

In conclusion, use the `bool` filter whenever possible. Unfortunately, real life is not so simple. Some types of filters do not have the ability to create the bitset directly. In this rare situation, the `bool` filter will be less effective. You already know of two of these filters: the numeric `range` filter and the `script` filter. The third is a whole group of filters that use geographic coordinates; we will visit these in the *Geo* section of *Chapter 6, Beyond Full-text Searching*.

Named filters

Looking at how complicated setting filters may be, sometimes it would be useful to know which filters were used to determine that a document should be returned by a query. Fortunately, it is possible to give every filter a name. This name will be returned with a document that was matched during the query. Let's check how this works. The following query will return every book that is available and tagged as novel or every book from the nineteenth century:

```
{
  "query": {
    "filtered": {
      "query": { "match_all" : {} },
      "filter": {
        "or": [
          { "and": [
            { "term": { "available" : true } },
            { "term": { "tags" : "novel" } }
          ] },
          { "range": { "year" : { "gte": 1800, "lte" : 1899 } } }
        ]
      }
    }
  }
}
```

We used the `filtered` version of the query because this is the only version where Elasticsearch can add information about the filters that were used. Let's rewrite this query to add a name to each filter as follows:

```
{
  "query": {
    "filtered": {
      "query": { "match_all" : {} },
      "filter": {
        "or": {
          "filters": [
            {
              "and": {

```

As you can see, we added the `_name` property to every filter. In the case of the `and` and `or` filters, we needed to change the syntax. So, we wrapped the enclosed filters by an additional object so that JSON is properly formatted. After sending a query to Elasticsearch, we should get a response similar to the following one:

```
{  
    "took" : 2,  
    "timed_out" : false,  
    "shards" : {
```

```

    "total" : 2,
    "successful" : 2,
    "failed" : 0
},
"hits" : [
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
        "_index" : "library",
        "_type" : "book",
        "_id" : "1",
        "_score" : 1.0, "_source" : { "title": "All Quiet on the
            Western Front", "otitle": "Im Westen nichts Neues",
            "author": "Erich Maria Remarque", "year": 1929,
            "characters": ["Paul Bäumer", "Albert Kropp",
            "Haie Westhus", "Fredrich Müller",
            "Stanislaus Katczinsky", "Tjaden"],
            "tags": ["novel"], "copies": 1, "available": true},
        "matched_queries" : [ "or", "tag", "avail", "and" ]
    }, {
        "_index" : "library",
        "_type" : "book",
        "_id" : "4",
        "_score" : 1.0, "_source" : { "title": "Crime and
            Punishment", "otitle": "Преступлénie и наказáниe",
            "author": "Fyodor Dostoevsky", "year": 1886,
            "characters": ["Raskolnikov", "Sofia Semyonovna
            Marmeladova"], "tags": [], "copies": 0, "available": true},
        "matched_queries" : [ "or", "year", "avail" ]
    } ]
}

```

You can see that in addition to standard information, each document contains a table with the name of the filters that were matched for that particular document.



Remember that in most cases, the filtered query will be faster than the post_filter query. Therefore, the filtered query should be used instead of post_filter whenever possible.

Caching filters

The last thing to be mentioned about filters in this chapter is **caching**. Caching increases the speed of the queries that use filters, but at the cost of memory and query time, during the first execution of such a filter. Because of this, the best candidates for caching are the filters that can be reused, for example, the ones that we will use frequently that also include the parameters' values.

Caching can be turned on for the `and`, `bool`, and `or` filters (but usually, it is a better idea to cache the enclosed filters instead). In this case, the required syntax is the same as described in the named filters, as follows:

```
{  
  "post_filter": {  
    "script": {  
      "_cache": true,  
      "script": "now - doc['year'].value > 100",  
      "params": {  
        "now": 2012  
      }  
    }  
  }  
}
```

Some filters don't support the `_cache` parameter because their results are always cached. By default, the following filters are the ones that are always cached:

- `exists`
- `missing`
- `range`
- `term`
- `terms`

This behavior can be modified and caching can be turned off using the following code:

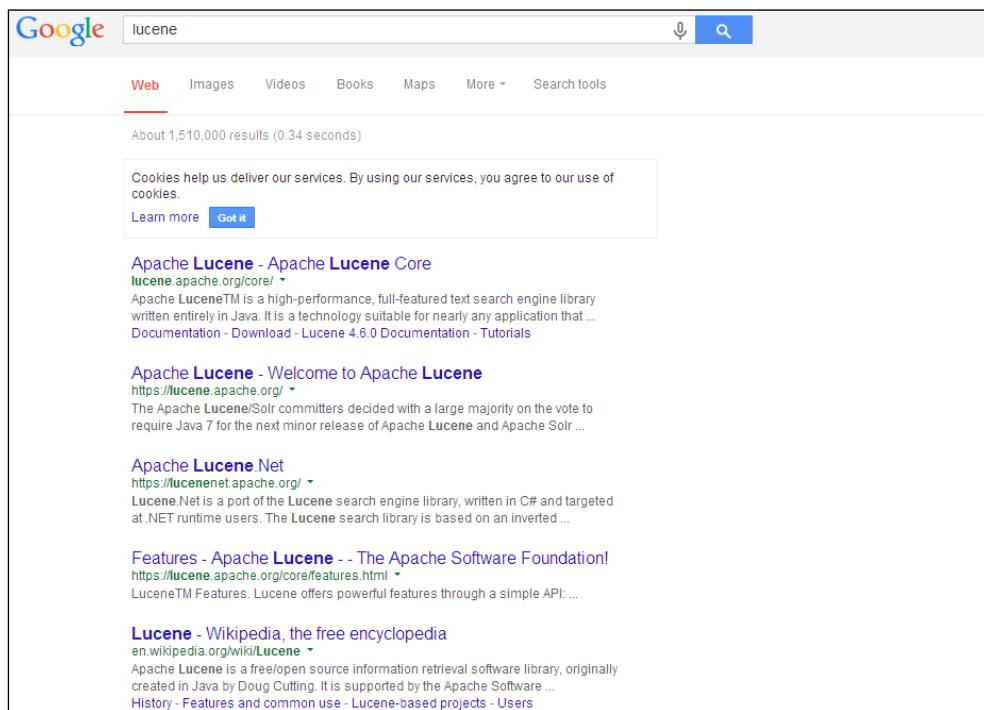
```
{  
  "post_filter": {  
    "term": {  
      "_cache": false,  
      "year": 1961  
    }  
  }  
}
```

```
}
```

Caching is not able to sense the `ids`, `match_all`, and `limit` filters.

Highlighting

You have probably heard of **highlighting**, or even if you are not familiar with the name, you've probably seen highlighted results on the usual web pages you visit. Highlighting is the process of showing which word or words from the query were matched in the resulting documents. For example, if we want to do a search on Google for the word `lucene`, we will see it in bold in the list of results as shown in the following screenshot:



In this chapter, we will see how to use the Elasticsearch highlighting capabilities to enhance our application with highlighted results.

Getting started with highlighting

There is no better way of showing how highlighting works besides creating a query and looking at the results returned by Elasticsearch. So let's assume that we want to highlight the words that were matched in the `title` field of our documents to enhance our users' search experience. We search for the `crime` word again, and to get the results of the highlighting, we send the following query:

```
{  
  "query" : {  
    "term" : {  
      "title" : "crime"  
    }  
  },  
  "highlight" : {  
    "fields" : {  
      "title" : {}  
    }  
  }  
}
```

The response for such a query should be like the following:

```
{  
  "took" : 2,  
  "timed_out" : false,  
  "_shards" : {  
    "total" : 5,  
    "successful" : 5,  
    "failed" : 0  
  },  
  "hits" : {  
    "total" : 1,  
    "max_score" : 0.19178301,  
    "hits" : [ {  
      "_index" : "library",  
      "_type" : "book",  
      "_id" : "4",  
      "_score" : 0.19178301,  
      "highlight" : {  
        "title" : "Crime and Punishment"  
      }  
    }]  
  }  
}
```

```

    "_score" : 0.19178301, "_source" : { "title": "Crime and
    Punishment", "otitle": "Преступлénie и наказánie",
    "author": "Fyodor Dostoevsky", "year": 1886,
    "characters": ["Raskolnikov", "Sofia Semyonovna
    Marmeladova"], "tags": [], "copies": 0, "available" : true},
    "highlight" : {
        "title" : [ "<em>Crime</em> and Punishment" ]
    }
}
}
}

```

As you can see, apart from the standard information we got from Elasticsearch, there is a new section called `highlight`. Elasticsearch used the `` HTML tag at the beginning of the highlighting section and its closing counterpart to close the section. This is the default behavior of Elasticsearch, but we will learn how to change it.

Field configuration

In order to perform highlighting, the original content of the field needs to be present—we have to set the fields that we will use for highlighting either to be stored, or we should use the `_source` field with those fields included.

Under the hood

Elasticsearch uses Apache Lucene under the hood, and highlighting is one of the features of that library. Lucene provides three types of highlighting implementations: the standard one, which we just used; the second one called `FastVectorHighlighter`, which needs term vectors and positions in order to work; and the third one called `PostingsHighlighter`, which we will discuss at the end of this chapter. Elasticsearch chooses the correct highlighter implementation automatically—if the field is configured with the `term_vector` property set to `with_positions_offsets`, `FastVectorHighlighter` will be used.

However, you have to remember that having term vectors will cause your index to be larger, but the highlighting will take less time to be executed. Also, `FastVectorHighlighter` is recommended for fields that store a lot of data in them.

Configuring HTML tags

As we already mentioned, it is possible to change the default HTML tags to the ones we would like to use. For example, let's assume that we want to use the standard HTML **** tag for highlighting. In order to do this, we should set the `pre_tags` and `post_tags` properties (these are arrays) to `` and ``. Since the two mentioned properties are arrays, we can include more than one tag, and Elasticsearch will use each of the defined tags to highlight different words. So, our example query will be like the following:

```
{  
  "query" : {  
    "term" : {  
      "title" : "crime"  
    }  
  },  
  "highlight" : {  
    "pre_tags" : [ "<b>" ],  
    "post_tags" : [ "</b>" ],  
    "fields" : {  
      "title" : {}  
    }  
  }  
}
```

The result returned by Elasticsearch to the preceding query will be as follows:

```
{  
  "took" : 2,  
  "timed_out" : false,  
  "_shards" : {  
    "total" : 5,  
    "successful" : 5,  
    "failed" : 0  
  },  
  "hits" : {  
    "total" : 1,  
    "max_score" : 0.19178301,  
    "hits" : [ {  
      "_index" : "library",  
      "_type" : "book",  
      "_id" : "4",  
      "_score" : 0.19178301,  
      "highlight" : {  
        "title" : "Crime and Punishment"  
      }  
    }  
  }  
}
```

```

    "_score" : 0.19178301, "_source" : { "title": "Crime and
    Punishment", "otitle": "Преступлénie и наказánie",
    "author": "Fyodor Dostoevsky", "year": 1886,
    "characters": ["Raskolnikov", "Sofia Semyonovna
    Marmeladova"], "tags": [], "copies": 0, "available" : true},
    "highlight" : {
      "title" : [ "<b>Crime</b> and Punishment" ]
    }
  }
}

```

As you can see, the `Crime` word in the `title` field was surrounded by the tags of our choice.

Controlling the highlighted fragments

Elasticsearch allows us to control the number of highlighted fragments returned and their size, and exposes the two properties we are allowed to use. The first one, `number_of_fragments`, defines the number of fragments returned by Elasticsearch and defaults to 5. Setting this property to 0 causes the whole field to be returned, which can be handy for short fields; however, it can be expensive for longer fields. The second property, `fragment_size`, lets us specify the maximum length of the highlighted fragments in characters and defaults to 100.

Global and local settings

The highlighting properties discussed earlier can both be set on a global basis and on a per-field basis. The global ones will be used for all the fields that don't override them and should be placed on the same level as the `fields` section of your highlighting, as follows:

```

{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "highlight" : {
    "pre_tags" : [ "<b>" ],
    "post_tags" : [ "</b>" ],
    "fields" : {
      "title" : {}
    }
  }
}

```

```
        }
    }
}
```

We can also set the properties for each field. For example, if we want to keep the default behavior for all the fields except for our `title` field, we use the following code:

```
{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "highlight" : {
    "fields" : {
      "title" : {
        "pre_tags" : [ "<b>" ], "post_tags" : [ "</b>" ]
      }
    }
  }
}
```

As you can see, instead of placing the properties on the same level as the `fields` section, we placed it inside the empty JSON object that specifies the `title` field behavior. Of course, each field can be configured using different properties.

Require matching

Sometimes, there may be a need (especially when using multiple highlighted fields) to show only the fields that match our query. In order to cause such behavior, we need to set the `require_field_match` property to `true`. Setting this property to `false` will cause all the terms to be highlighted even if a field didn't match the query.

To see how this works, let's create a new index called `users` and index a single document there. We will do this by sending the following command:

```
curl -XPUT 'http://localhost:9200/users/user/1' -d '{
  "name" : "Test user",
  "description" : "Test document"
}'
```

Now, let's assume that we want to highlight the hits in both the `name` and `description` fields; our query could look like the following code:

```
{
  "query" : {
    "term" : {
      "name" : "test"
    }
  },
  "highlight" : {
    "fields" : {
      "name" : { "pre_tags" : [ "<b>" ], "post_tags" : [ "</b>" ] },
      "description" : { "pre_tags" : [ "<b>" ], "post_tags" : [
        "</b>" ] }
    }
  }
}
```

The result of the preceding query will be as follows:

```
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
      "_index" : "users",
      "_type" : "user",
      "_id" : "1",
      "_score" : 0.19178301, "_source" : {"name" : "Test
        user", "description" : "Test document"},
      "highlight" : {
        "name" : "<b>test</b>",
        "description" : "<b>Test user</b>, <b>Test document</b>"}
    } ]
  }
}
```

```
        "description" : [ "<b>Test</b> document" ],
        "name" : [ "<b>Test</b> user" ]
    }
}
}
}
```

Note that even though we only matched the `name` field, we got the results of the highlighting in both the fields. In most cases, we want to avoid this. So now, let's modify our query to use the `require_field_match` property as follows:

```
{
  "query" : {
    "term" : {
      "name" : "test"
    }
  },
  "highlight" : {
    "require_field_match" : "true",
    "fields" : {
      "name" : { "pre_tags" : [ "<b>" ], "post_tags" : [ "</b>" ] },
      "description" : { "pre_tags" : [ "<b>" ], "post_tags" : [
        "</b>" ] }
    }
  }
}
```

Let's take a look at the modified query results as follows:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
```

```
"total" : 1,
"max_score" : 0.19178301,
"hits" : [ {
  "_index" : "users",
  "_type" : "user",
  "_id" : "1",
  "_score" : 0.19178301, "_source" : {"name" : "Test
  user","description" : "Test document"},
  "highlight" : {
    "name" : [ "<b>Test</b> user" ]
  }
} 1
}]
```

As you can see, Elasticsearch returned only the field that was matched, in our case, the `name` field.

The postings highlighter

It is time to discuss the third highlighter available in Elasticsearch. It was added in Elasticsearch 0.90.6 and is slightly different from the previous ones. Let's see these differences in the following example. `PostingsHighlighter` is automatically used when a field definition has the `index_options` attribute set to `offsets`. So again, to illustrate how `PostingsHighlighter` works, we will create a simple index with a proper mapping. We will do this using the following commands:

```
curl -XPUT 'localhost:9200/hl_test'
curl -XPOST 'localhost:9200/hl_test/doc/_mapping' -d '{
  "doc" : {
    "properties" : {
      "contents" : {
        "type" : "string",
        "fields" : {
          "ps" : { "type" : "string", "index_options" : "offsets" }
        }
      }
    }
}
```

```
        }
    }
}
}'
}'
```



Remember that similar to `FastVectorHighlighter`, the offsets required for `PostingsHighlighter` will result in an increase in the index size. However, this will be a smaller increase than using term vectors. In addition to that, indexing offsets is faster than indexing term vectors, and `PostingsHighlighter` is better when it comes to query performance.

If everything goes well, we will have a new index and the mappings. The mappings have two fields defined: one named `contents` and the second one named `contents.ps`. In this second case, we turn on the offsets by using the `index_options` property. This means that Elasticsearch will use the standard highlighter for the `contents` field and the postings highlighter for the `contents.ps` field.

To see the difference, we will index a single document with a fragment from Wikipedia that describes the history of Birmingham. We do this by running the following command:

```
curl -XPUT localhost:9200/hl_test/doc/1 -d '{
  "contents" : "Birmingham's early history is that of a remote and
               marginal area. The main centers of population, power and wealth
               in the pre-industrial English Midlands lay in the fertile and
               accessible river valleys of the Trent, the Severn and the Avon.
               The area of modern Birmingham lay in between, on the upland
               Birmingham Plateau and within the densely wooded and sparsely
               populated Forest of Arden."
}'
```

The last step is to send a query using both the highlighters. We can do this in a single request using the following command:

```
curl 'localhost:9200/hl_test/_search?pretty' -d '{
  "query": {
    "term": {
      "contents": "modern"
    }
  },
}'
```

```
"highlight": {  
    "fields": {  
        "contents": {},  
        "contents.ps" : {}  
    }  
}  
}'
```

If everything is all right, we will find the following snippet in response:

```
"highlight" : {  
    "contents" : [ "valleys of the Trent, the Severn and the  
    Avon. The area of <em>modern</em> Birmingham lay in  
    between, on the upland" ],  
    "contents.ps" : [ "The area of <em>modern</em> Birmingham lay  
    in between, on the upland Birmingham Plateau and within the  
    densely wooded and sparsely populated Forest of Arden." ]  
}
```

As you see, both highlighters found the occurrence of the desired word. The difference is that the postings highlighter returns the smarter snippet—it checks for the sentence boundaries.

Let's try one more query using the following command:

```
curl 'localhost:9200/hl_test/_search?pretty' -d '{  
    "query": {  
        "match_phrase": {  
            "contents": "centers of"  
        }  
    },  
    "highlight": {  
        "fields": {  
            "contents": {},  
            "contents.ps": {}  
        }  
    }  
}'
```

We searched for a particular phrase, `centers of`. As you may expect, the results for these two highlighters will differ. For standard highlighting, you will find the following phrase in response:

```
"Birmingham's early history is that of a remote and marginal area.  
The main <em>centers</em> <em>of</em> population"
```

As you can clearly see, the standard highlighter divided the given phrase and highlighted individual terms. Not all occurrences of the `centers` and `of` terms were highlighted and only the ones that form the phrase were.

On the other hand, the postings highlighter returned the following highlighted fragment:

```
"Birmingham's early history is that <em>of</em> a remote and marginal  
area.",  
"The main <em>centers</em> <em>of</em> population, power and wealth  
in the pre-industrial English Midlands lay in the fertile and  
accessible river valleys <em>of</em> the Trent, the Severn and the  
Avon.",  
"The area <em>of</em> modern Birmingham lay in between, on the upland  
Birmingham Plateau and within the densely wooded and sparsely  
populated Forest <em>of</em> Arden."
```

This is the significant difference; the postings highlighter highlighted all the terms that match the terms from the query and not only those that formed the phrase.

Validating your queries

Sometimes, the queries that your application sends to Elasticsearch are generated automatically from multiple criteria or even worse; they are generated by some kind of wizard, where the end user can create complicated queries. The issue is that sometimes it is not easy to tell if the query is correct or not. To help with this, Elasticsearch exposes the validate API.

Using the validate API

The validate API is very simple. Instead of sending the query to the `_search` endpoint, we send it to the `_validate/query` endpoint. And that's it. Let's look at the following query:

```
{  
  "query" : {
```

```
"bool" : {
    "must" : {
        "term" : {
            "title" : "crime"
        }
    },
    "should" : {
        "range" : {
            "year" : {
                "from" : 1900,
                "to" : 2000
            }
        }
    },
    "must_not" : {
        "term" : {
            "otitle" : "nothing"
        }
    }
}
```

This query has already been used in this book. We know that everything is right with this query, but let's check it with the following command (we've stored the query in the `query.json` file):

```
curl -XGET 'localhost:9200/library/_validate/query?pretty' -d
@query.json
```

The query seems all right, but let's look at what the validate API has to say. The response returned by Elasticsearch is as follows:

```
{
    "valid" : false,
    "_shards" : {
        "total" : 1,
        "successful" : 1,
        "failed" : 0
    }
}
```

Let's take a look at the `valid` attribute. It is set to `false`. Something has gone wrong. Let's execute the query validation once again with the `explain` parameter added in the query as follows:

```
curl -XGET 'localhost:9200/library/_validate/query?pretty&explain' --  
  data-binary @query.json
```

Now, the result returned from Elasticsearch is more verbose, as follows:

```
{  
  "valid" : false,  
  "_shards" : {  
    "total" : 1,  
    "successful" : 1,  
    "failed" : 0  
  },  
  "explanations" : [ {  
    "index" : "library",  
    "valid" : false,  
    "error" : "org.elasticsearch.index.query.QueryParsingException:  
      [library] Failed to parse;  
      org.elasticsearch.common.jackson.core.JsonParseException:  
      Illegal unquoted character ((CTRL-CHAR, code 10)): has to be  
      escaped using backslash to be included in name\\n at [Source:  
      [B@6456919f; line: 10, column: 18]"  
  } ]  
}
```

Now everything is clear. In our example, we improperly quoted the `range` attribute.



You may wonder why we used the `--data-binary` parameter in our curl query. This parameter properly preserves the new line character when sending a query to Elasticsearch. This means that the line and column number will be intact, and it'll be easier to find errors. In the other cases, the `-d` parameter is more convenient because it's shorter.

The validate API can also detect other errors, for example, the incorrect format of a number or other mapping-related issues. Unfortunately, for our application, it is not easy to detect what the problem is because of a lack of structure in the error messages.

Sorting data

We now know how to build queries and filter the results. We also know what the search types are and why they matter. We can send these queries to Elasticsearch and analyze the returned data. For now, this data was organized in the order determined by the scoring. This is exactly what we want in most cases. The search operation should give us the most relevant documents first. However, what we can do if we want to use our search more like a database or set a more sophisticated algorithm to order data? Let's check what Elasticsearch can do with a sorting function.

Default sorting

Let's look at the following query that returns all the books with at least one of the specified words:

```
{
  "query" : {
    "terms" : {
      "title" : [ "crime", "front", "punishment" ],
      "minimum_match" : 1
    }
  }
}
```

Under the hood, Elasticsearch sees this as follows:

```
{
  "query" : {
    "terms" : {
      "title" : [ "crime", "front", "punishment" ],
      "minimum_match" : 1
    }
  },
  "sort" : { "_score" : "desc" }
}
```

Note the highlighted section in the preceding query. This is the default sorting used by Elasticsearch. More verbose, this fragment may be shown as follows:

```
"sort" : [
  { "_score" : "desc" }
]
```

The preceding section defines how the documents should be sorted in the results list. In this case, Elasticsearch will show the documents with the highest score on top of the results list. The simplest modification is to reverse the ordering by changing the sort section to the following one:

```
"sort" : [
  { "_score" : "asc" }
]
```

Selecting fields used for sorting

Default sorting is boring, isn't it? So, let's change it to sort one of the fields present in the documents as follows:

```
"sort" : [
  { "title" : "asc" }
]
```

Unfortunately, this doesn't work as expected. Although Elasticsearch sorted the documents, the ordering is somewhat strange. Look closer at the response. With every document, Elasticsearch returns information about the sorting; for example, for the `Catch-22` book, the returned document looks like the following code:

```
{
  "_index": "library",
  "_type": "book",
  "_id": "2",
  "_score": null,
  "_source": {
    "title": "Catch-22",
    "author": "Joseph Heller",
    "year": 1961,
    "characters": [
      "John Yossarian",
      "Captain Aardvark",
      "Chaplain Tappman",
      "Colonel Cathcart",
      "Doctor Daneeka"
    ],
    "tags": [
      "war",
      "satire",
      "military"
    ]
  }
}
```

```
    "novel"
  ],
  "copies": 6,
  "available": false,
  "section": 1
},
"sort": [
  "22"
]
}
```

If you compare the `title` field and the returned sorting information, everything should be clear. Elasticsearch, during the analysis process, splits the field into several tokens. Since sorting is done using a single token, Elasticsearch chooses one of those produced tokens. It does the best that it can by sorting these tokens alphabetically and choosing the first one. This is the reason why, in the sorting value, we find only a single word instead of the whole contents of the `title` field. In your spare time, you can check how Elasticsearch will behave when sorting on the `characters` field.

In general, it is a good idea to have a not analyzed field for sorting. We can use fields with multiple values for sorting, but in most cases, it doesn't make much sense and has limited usage. As an example of using two different fields, one for sorting and another for searching, let's change our `title` field. The changed `title` field definition could look like the following code:

```
"title" : {
  "type": "string",
  "fields": {
    "sort": { "type" : "string", "index": "not_analyzed" }
  }
}
```

After changing the `title` field in the mappings, we've shown in the beginning of the chapter that we can try sorting the `title.sort` field and see whether it will work. To do this, we will need to send the following query:

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : [
    {"title.sort" : "asc" }
```

```
]  
}
```

Now, it works properly. As you can see, we used the new field, `title.sort`. We've set it to be not analyzed; so, there is a single value for that field in the index.

In the response from Elasticsearch, every document contains information about the value used for sorting; it is as follows:

```
"_index" : "library",  
"_type" : "book",  
"_id" : "1",  
"_score" : null, "_source" : { "title": "All Quiet on the  
Western Front", "otitle": "Im Westen nichts Neues",  
"author": "Erich Maria Remarque", "year":  
1929, "characters": ["Paul Bäumer", "Albert Kropp",  
"Haie Westhus", "Fredrich Müller", "Stanislaus  
Katczinsky", "Tjaden"], "tags": ["novel"], "copies": 1,  
"available": true, "section": 3},  
"sort" : [ "All Quiet on the Western Front" ]
```

Note that `sort`, in request and response, is given as an array. This suggests that we can use several different orderings. Elasticsearch will use the following elements from the list to determine ordering between documents that have the same previous field value. So, if we have the same value in the `title` field, documents will be sorted by the next field that we specify.

Specifying the behavior for missing fields

What about when some of the documents that match the query don't have the field we want to sort on? By default, documents without the given field are returned first in the case of ascending order and last in the case of descending order. However, sometimes this is not exactly what we want to achieve.

When we use sorting on numeric fields, we can change the default Elasticsearch behavior for documents with missing fields. For example, let's take a look at the following query:

```
{  
  "query" : {  
    "match_all" : { }  
  },  
  "sort" : [
```

```
{
  "section" : { "order" : "asc", "missing" : "_last" } }
]
```

Note the extended form of the `sort` section of our query. We've added the `missing` parameter to it. By setting the `missing` parameter to `_last`, Elasticsearch will place the documents without the given field at the bottom of the results list. Setting the `missing` parameter to `_first` will result in Elasticsearch placing the documents without the given field at the top of the results list. It is worth mentioning that besides the `_last` and `_first` values, Elasticsearch allows us to use any number. In such a case, a document without a defined field will be treated as the document with this given value.

Dynamic criteria

As we've mentioned in the previous section, Elasticsearch allows us to sort using fields that have multiple values. We can control how the comparison is made using scripts for sorting. We do that by showing Elasticsearch how to calculate the value that should be used for sorting. Let's assume that we want to sort by the first value indexed in the `tags` field. Let's take a look at the following example query:

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "doc['tags'].values.length > 0 ?
        doc['tags'].values[0] : '\u19999'",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

In the preceding example, we replaced every nonexistent value by the Unicode code of a character that should be low enough in the list. The main idea of this code is to check if our array contains at least a single element. If it does, then the first value from the array is returned. If the array is empty, we return the Unicode character that should be placed at the bottom of the results list. Besides the `script` parameter, this option of sorting requires us to specify the `order` (ascending, in our case) and `type` parameters that will be used for the comparison (we return `string` from our script).

Collation and national characters

If we want to use languages other than English, we can face the problem of an incorrect order of characters. It happens because many languages have a different alphabetical order defined. Elasticsearch supports many languages, but proper collation requires an additional plugin. It's easy to install and configure, but we will discuss this further in the *Elasticsearch plugins* section in *Chapter 8, Administrating Your Cluster*.

Query rewrite

Queries such as the prefix query and the wildcard query – basically, any query that is said to be multiterm – use query rewriting. Elasticsearch does this because of performance reasons. The rewrite process is about changing the original, expensive query to a set of queries that are far less expensive from the Lucene point of view.

An example of the rewrite process

The best way to illustrate how the rewrite process is carried out internally is to look at an example and see what terms are used instead of the original query term. Let's suppose that we have the following data in our index:

```
curl -XPOST 'localhost:9200/library/book/1' -d '{"title": "Solr 4 Cookbook"}'  
curl -XPOST 'localhost:9200/library/book/2' -d '{"title": "Solr 3.1 Cookbook"}'  
curl -XPOST 'localhost:9200/library/book/3' -d '{"title": "Mastering Elasticsearch"}'
```

What we want is to find all the documents that start with the letter s. It's as simple as that; we run the following query against our library index:

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{  
  "query" : {  
    "prefix" : {  
      "title" : "s",  
      "rewrite" : "constant_score_boolean"  
    }  
  }'  
'
```

Here, we used a simple prefix query. We mentioned that we want to find all the documents containing the s letter in the title field. We also used the rewrite property to specify the query rewrite method, but let's skip it for now as we will discuss the possible values of this parameter in the latter part of this section.

As a response to the preceding query, we get the following output:

```
{  
    "took" : 22,  
    "timed_out" : false,  
    "_shards" : {  
        "total" : 5,  
        "successful" : 5,  
        "failed" : 0  
    },  
    "hits" : {  
        "total" : 2,  
        "max_score" : 1.0,  
        "hits" : [ {  
            "_index" : "library",  
            "_type" : "book",  
            "_id" : "2",  
            "_score" : 1.0, "_source" : {"title": "Solr 3.1 Cookbook"}  
        }, {  
            "_index" : "library",  
            "_type" : "book",  
            "_id" : "1",  
            "_score" : 1.0, "_source" : {"title": "Solr 4 Cookbook"}  
        } ]  
    }  
}
```

As you can see, in response, we got the two documents that have the contents of the title field that begin with the desired character. If we take a look at the Lucene-level query, we notice that the prefix query has been rewritten into a query similar to the following one:

```
ConstantScore(title:solr)
```

This is because `solr` is the only term that starts with the letter `s`. And this is what query rewrite is all about: to find the relevant terms, and instead of running an expensive query, just rewrite it to something more performance friendly.

Query rewrite properties

As we already said, we can use the `rewrite` parameter of any multiterm query (such as the Elasticsearch prefix and wildcard queries) to control how we want the query to be rewritten. We place the `rewrite` parameter inside the JSON object responsible for the actual query, as follows:

```
{  
  "query" : {  
    "prefix" : {  
      "title" : "s",  
      "rewrite" : "constant_score_boolean"  
    }  
  }  
}
```

Now, let's look at the options we have when it comes to the value of this parameter:

- `scoring_boolean`: This rewrite method translates each generated term into a Boolean `should` clause in the Boolean query. This query rewrite method may be CPU intensive (because the score for each term is calculated and stored), and queries that have many terms may exceed the Boolean query limit, which is set to 1024. Also, this query stores the computed score.
- `constant_score_boolean`: This rewrite method is similar to the `scoring_boolean` rewrite method described earlier, but it is less demanding of the CPU because the scoring is not computed. Instead of that, each term receives a score equal to the query boost, which is 1 by default, and can be set using the `boost` property. Similar to the `scoring_boolean` rewrite method, this method can also hit the maximum limit of the Boolean clauses.
- `constant_score_filter`: As Apache Lucene Javadocs states, this rewrite method rewrites the query by creating a private filter by visiting each term in a sequence and marking all the documents for that term. Matching documents are given a constant score equal to the query boost. This method is faster than the `scoring_boolean` and `constant_score_boolean` methods when the number of matching terms or documents is large.

- `top_terms_N`: This is a rewrite method that translates each generated term into a Boolean `should` clause in a Boolean query and keeps the scores as computed by the query. However, unlike the `scoring_boolean` rewrite method, it only keeps the `N` number of top scoring terms to avoid hitting the maximum limit of the Boolean clauses.
- `top_terms_boost_N`: This is a rewrite method similar to `top_terms_N`. However, unlike the `top_terms_N` rewrite method, the scores are only computed as the boost and not the query.

 When the `rewrite` property is set to either `constant_score_auto` or not set at all, the value of `constant_score_filter` or `constant_score_boolean` will be used depending on the query and how it is constructed.

Before we finish the query rewrite part of this chapter, we should ask ourselves one last question, "When do we use which type of rewrite?". The answer to such a question depends largely on our use case, but just to summarize if we can live with lower precision (but higher performance), we can go for the `top N` rewrite method. If we need high precision (but lower performance), we choose the Boolean approach.

Summary

In this chapter, we learned how Elasticsearch querying works and how to choose the data we want returned. We saw how query rewrite works, what the search types are, and what search preference is. We learned about the basic queries available in Elasticsearch and filtered our results using filters. In addition to this, we discussed the highlighting functionality, which allowed us to highlight matches in our documents, and we validated our queries. We learned about compound queries that can group multiple queries together, and finally, we saw how to configure sorting to match our needs.

In the next chapter, we'll focus on indices again, but not only on indices. We'll learn how to index tree-like structures. We will see how to index data that is not flat by storing JSON objects in Elasticsearch, and how to modify the structure of an already created index. We'll also see how to handle relationships between documents using nested documents and parent-child functionality.

4

Extending Your Index Structure

In the previous chapter, we learned many things about querying Elasticsearch. We saw how to choose fields that will be returned and learned how querying works in Elasticsearch. In addition to that, we now know the basic queries that are available and how to filter our data. What's more, we saw how to highlight the matches in our documents and how to validate our queries. In the end, we saw the compound queries of Elasticsearch and learned how to sort our data. By the end of this chapter, you will have learned the following topics:

- Indexing tree-like structured data
- Indexing data that is not flat
- Modifying your index structure when possible
- Indexing data with relationships by using nested documents
- Indexing data with relationships between them by using the parent-child functionality

Indexing tree-like structures

Trees are everywhere. If you develop a shop application, you would probably have categories. If you look at the filesystem, the files and directories are arranged in tree-like structures. This book can also be represented as a tree: chapters contain topics and topics are divided into subtopics. As you can imagine, Elasticsearch is also capable of indexing tree-like structures. Let's check how we can navigate through this type of data using `path_analyzer`.

Data structure

First, let's create a simple index structure by using the following lines of code:

```
curl -XPUT 'localhost:9200/path' -d '{  
    "settings" : {  
        "index" : {  
            "analysis" : {  
                "analyzer" : {  
                    "path_analyzer" : { "tokenizer" : "path_hierarchy" }  
                }  
            }  
        }  
    },  
    "mappings" : {  
        "category" : {  
            "properties" : {  
                "category" : {  
                    "type" : "string",  
                    "fields" : {  
                        "name" : { "type" : "string",  
                                   "index" : "not_analyzed" },  
                        "path" : { "type" : "string",  
                                   "analyzer" : "path_analyzer",  
                                   "store" : true }  
                    }  
                }  
            }  
        }  
    }  
'
```

As you can see, we have a single type created – the `category` type. We will use it to store the information about the location of our document in the tree structure. The idea is simple – we can show the location of the document as a path, in the exact same manner as files and directories are presented on your hard disk drive. For example, in an automotive shop we can have `/cars/passenger/sport`, `/cars/passenger/camper`, or `/cars/delivery_truck/`. However, we need to index this path in three ways. We will use a field named `name`, which doesn't have any additional processing, and an additional field called `path`, which will use `path_analyzer`, which we defined. We will also leave the original value as it is, just in case we want to search it.

Analysis

Now, let's see what Elasticsearch will do with the category path during the analysis process. To see this, we will use the following command line, which uses the analysis API described in the *Understanding field analysis* section in *Chapter 5, Make Your Search Better*:

```
curl -XGET 'localhost:9200/path/_analyze?field=category.path&pretty' -d '/cars/passenger/sport'
```

The following results were returned by Elasticsearch:

```
{
  "tokens": [
    {
      "token": "/cars",
      "start_offset": 0,
      "end_offset": 5,
      "type": "word",
      "position": 1
    },
    {
      "token": "/cars/passenger",
      "start_offset": 0,
      "end_offset": 15,
      "type": "word",
      "position": 1
    },
    {
      "token": "/cars/passenger/sport",
      "start_offset": 0,
      "end_offset": 21,
      "type": "word",
      "position": 1
    }
  ]
}
```

As we can see, our category path `/cars/passenger/sport` was processed by Elasticsearch and divided into three tokens. Thanks to this, we can simply find every document that belongs to a given category or its subcategories using the term filter. An example of using filters is as follows:

```
{  
  "filter" : {  
    "term" : { "category.path" : "/cars" }  
  }  
}
```

Note that we also have the original value indexed in the category.name field. This is handy when we want to find documents from a particular path, ignoring documents that are deeper in the hierarchy.

Indexing data that is not flat

Not all data is flat like the data we have been using so far in this book. Of course, if we are building the system that Elasticsearch will be a part of, we can create a structure that is convenient for Elasticsearch. Of course, the structure can't always be flat, because not all use cases allow that. Let's see how to create mappings that use fully-structured JSON objects.

Data

Let's assume that we have the following data (we will store it in the file named `structured_data.json`):

```
{
  "book" : {
    "author" : {
      "name" : {
        "firstName" : "Fyodor",
        "lastName" : "Dostoevsky"
      }
    },
    "isbn" : "123456789",
    "englishTitle" : "Crime and Punishment",
    "year" : 1866,
    "characters" : [
      {
        "name" : "Raskolnikov"
      },
      {
        "name" : "Sonya"
      }
    ]
  }
}
```

```
{  
    "name" : "Sofia"  
}  
]  
,"copies" : 0  
}  
}
```

As you can see in the preceding code, the data is not flat; it contains arrays and nested objects. If we would like to create mappings and use the knowledge that we've obtained so far, we will have to flatten the data. However, Elasticsearch allows some degree of structure to be present in the documents and we should be able to create mappings that will be able to handle the preceding example.

Objects

The preceding example shows the structured JSON file. As you can see, the root object in our example file is `book`. The `book` object has some additional, simple properties, such as `englishTitle`. Those will be indexed as normal fields. In addition to that, it has the `characters` array type, which we will discuss in the next paragraph. For now, let's focus on `author`. As you can see, `author` is an object, which has another object nested within it—the `name` object, which has two properties, `firstName` and `lastName`.

Arrays

We already used the array type data, but we didn't discuss it in detail. By default, all fields in Lucene and thus in Elasticsearch are multivalued, which means that they can store multiple values. In order to send such fields to be indexed, we use the JSON array type, which is nested within opening and closing square brackets `[]`. As you can see in the preceding example, we used the array type for `characters` within the `book`.

Mappings

To index arrays, we just need to specify the properties for such fields inside the array name. So, in our case in order to index the `characters` data, we would need to add the following mappings:

```
"characters" : {  
    "properties" : {  
        "name" : {"type" : "string", "store" : "yes"}  
    }  
}
```

Nothing strange, we just nest the `properties` section inside the array's name (which is `characters` in our case) and we define the fields there. As a result of the preceding mappings, we would get `characters.name` as a multivalued field in the index.

Similarly, for the `author` object, we will call the section with the same name as it is present in the data, but in addition to the `properties` section, we also inform Elasticsearch that it should expect an object type by adding the `type` property with the value as `object`. We have the `author` object, but it also has the `name` object nested within it, so we just nest another object inside it. So, our mappings for the `author` field would look like the following:

```
"author" : {
    "type" : "object",
    "properties" : {
        "name" : {
            "type" : "object",
            "properties" : {
                "firstName" : {"type" : "string", "index" : "analyzed"},
                "lastName" : {"type" : "string", "index" : "analyzed"}
            }
        }
    }
}
```

The `firstName` and `lastName` fields appear in the index as `author.name.firstName` and `author.name.lastName`.

The rest of the fields are simple core types, so I'll skip discussing them as they were already discussed in the *Mappings configuration* section of *Chapter 2, Indexing Your Data*.

Final mappings

So, our final mappings file, which we've named `structured_mapping.json`, looks as follows:

```
{
    "book" : {
        "properties" : {
            "author" : {
                "type" : "object",
                "properties" : {

```

```
        "name" : {
            "type" : "object",
            "properties" : {
                "firstName" : {"type" : "string", "store": "yes"},
                "lastName" : {"type" : "string", "store": "yes"}
            }
        },
        "isbn" : {"type" : "string", "store": "yes"},
        "englishTitle" : {"type" : "string", "store": "yes"},
        "year" : {"type" : "integer", "store": "yes"},
        "characters" : {
            "properties" : {
                "name" : {"type" : "string", "store": "yes"}
            }
        },
        "copies" : {"type" : "integer", "store": "yes"}
    }
}
```

As you can see, we set the `store` property to `yes` for all of the fields. This is just to show you that the fields were properly indexed.

Sending the mappings to Elasticsearch

Now that we have done our mappings, we would like to test if all of them actually work. This time we will use a slightly different technique to create an index and put the mappings. First, let's create the `library` index using the following command line:

```
curl -XPUT 'localhost:9200/library'
```

Now, let's send our mappings for the book type, using the following command line:

```
curl -XPUT 'localhost:9200/library/book/_mapping' -d @structured_mapping.json
```

We can now index our example data using the following command line:

```
curl -XPOST 'localhost:9200/library/book/1' -d @structured_data.json
```

To be or not to be dynamic

As we already know, Elasticsearch is schemaless, which means it can index data without the need to create the mappings upfront. The dynamic behavior of Elasticsearch is turned on by default, but there may be situations where you may want to turn it off for some parts of your index. In order to do that, you should add the `dynamic` property to the given field and set it to `false`. This should be done on the same level of nesting as the `type` property for objects that shouldn't be dynamic. For example, if we would like our `author` and `name` objects to not be dynamic, we should modify the relevant part of the mappings file so that it looks similar to the following lines of code:

```
"author" : {  
    "type" : "object",  
    "dynamic" : false,  
    "properties" : {  
        "name" : {  
            "type" : "object",  
            "dynamic" : false,  
            "properties" : {  
                "firstName" : {"type" : "string", "index" : "analyzed"},  
                "lastName" : {"type" : "string", "index" : "analyzed"}  
            }  
        }  
    }  
}
```

However, please remember that in order to add new fields for such objects we will have to update the mappings.



You can also turn off the dynamic mappings functionality by adding the `index.mapper.dynamic` property to your `elasticsearch.yml` configuration file and setting it to `false`.

Using nested objects

Nested objects can come in handy in certain situations. Basically, with nested objects, Elasticsearch allows us to connect multiple documents together – one main document and multiple dependent ones. The main document and the nested ones will be indexed together and they will be placed in the same segment of the index (actually, in the same block), which guarantees the best performance we can get for data structure. The same goes for changing the document; unless you are using the update API, you need to index the parent document and all the other nested documents at the same time.



If you would like to read more about how nested objects work on the Lucene level, there is a very good blog post by *Mike McCandless* at <http://blog.mikemccandless.com/2012/01/searching-relational-content-with.html>.

Now, let's get to our example use case. Imagine that we have a shop with clothes and we store the size and color of each t-shirt. Our standard, nonnested mappings will look similar to the following lines of code (stored in `cloth.json`):

```
{
  "cloth" : {
    "properties" : {
      "name" : {"type" : "string"},
      "size" : {"type" : "string", "index" : "not_analyzed"},
      "color" : {"type" : "string", "index" : "not_analyzed"}
    }
  }
}
```

Imagine that we have a red t-shirt only in the XXL size and a black one only in the XL size in our shop. So our example document will look like the following code:

```
{
  "name" : "Test shirt",
  "size" : [ "XXL", "XL" ],
  "color" : [ "red", "black" ]
}
```

However, there is a problem with this data structure. What if one of our clients searches our shop in order to find the XXL t-shirt in black? Let's check that by running the following query (we assume that we've used our mappings to create the index and we've indexed our example document):

```
curl -XGET 'localhost:9200/shop/cloth/_search?pretty=true' -d '{
  "query" : {
    "bool" : {
      "must" : [
        {
          "term" : { "size" : "XXL" }
        },
        {
          "term" : { "color" : "black" }
        }
      ]
    }
}
```

```
    1
    }
}
}'
```

We should get no results right? But, in fact, Elasticsearch returned the following document:

```
{
  ...
  "hits" : {
    "total" : 1,
    "max_score" : 0.4339554,
    "hits" : [ {
      "_index" : "shop",
      "_type" : "cloth",
      "_id" : "1",
      "_score" : 0.4339554,
      "_source" : { "name" : "Test shirt",
                    "size" : [ "XXL", "XL" ],
                    "color" : [ "red", "black" ] }
    } ]
  }
}
```

This is because the document was compared; we have the value we are searching for in the `size` field and in the `color` field. Of course, this is not what we would like to get.

So, let's modify our mappings to use nested objects to separate `color` and `size` to different, nested documents. The final mapping looks like the following (we store these mappings in the `cloth_nested.json` file):

```
{
  "cloth" : {
    "properties" : {
      "name" : { "type" : "string", "index" : "analyzed" },
      "variation" : {
        "type" : "nested",
        "properties" : {
          "size" : { "type" : "string", "index" : "not_analyzed" },
          "color" : { "type" : "string", "index" : "not_analyzed" }
        }
      }
    }
  }
}
```

```
        }
    }
}
}
```

As you can see, we've introduced a new object, `variation`, inside our `cloth` type, which is a nested one (the `type` property set to `nested`). It basically says that we will want to index nested documents. Now, let's modify our document. We will add the `variation` object to it and that object will store objects with two properties: `size` and `color`. So, our example product will look as follows:

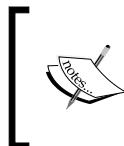
```
{
  "name" : "Test shirt",
  "variation" : [
    { "size" : "XXL", "color" : "red" },
    { "size" : "XL", "color" : "black" }
  ]
}
```

We've structured the document so that each size and its matching color is a separate document. However, if you would run our previous query, it wouldn't return any documents. This is because in order to query for nested documents, we need to use a specialized query. So, now our query looks as follows (of course we've created our index and type again):

```
curl -XGET 'localhost:9200/shop/cloth/_search?pretty=true' -d '{
  "query" : {
    "nested" : {
      "path" : "variation",
      "query" : {
        "bool" : {
          "must" : [
            { "term" : { "variation.size" : "XXL" } },
            { "term" : { "variation.color" : "black" } }
          ]
        }
      }
    }
  }
}'
```

And now, the preceding query wouldn't return the indexed document, because we don't have a nested document that has a size equal to xxL and the color black.

Let's get back to the query for a second to discuss it briefly. As you can see, we've used the nested query in order to search in the nested documents. The path property specifies the name of the nested object (yes, we can have multiple). As you can see, we just included a standard query section under the nested type. Please also note that we specified the full path for the field names in the nested objects, which is handy when you have multilevel nesting, which is also possible.



If you would like to filter your data on the basis of nested objects, you can do it – there is a nested filter, which has the same functionality as the nested query. Please refer to the *Filtering your results* section in *Chapter 3, Searching Your Data*, for more information about filtering.

Scoring and nested queries

There is an additional property when it comes to handling nested documents during queries. In addition to the path property, there is the score_mode property, which allows us to define how the score is calculated from the nested queries. Elasticsearch allows us to set this property to one of the following values:

- avg: This is the default value; using it for the score_mode property will result in Elasticsearch taking the average value calculated from the scores of the defined nested queries. The calculated average will be included in the score of the main query.
- total: This value is used for the score_mode property and it will result in Elasticsearch taking a sum of the scores for each nested query and including it in the score of the main query.
- max: This value is used for the score_mode property and it will result in Elasticsearch taking the score of the maximum scoring nested query and including it in the score of the main query.
- none: This value is used for the score_mode property and it will result in no score being taken from the nested query.

Using the parent-child relationship

In the previous section, we discussed the ability to index nested documents along with the parent one. However, even though the nested documents are indexed as separate documents in the index, we can't change a single nested document (unless we use the update API). However, Elasticsearch allows us to have a real parent-child relationship and we will look at it in the following section.

Index structure and data indexing

Let's use the same example that we used when discussing the nested documents—the hypothetical cloth store. However, what we would like to have is the ability to update sizes and colors without the need to index the whole document after each change.

Parent mappings

The only field we need to have in our parent document is name. We don't need anything more than that. So, in order to create our cloth type in the shop index, we will run the following commands:

```
curl -XPOST 'localhost:9200/shop'
curl -XPUT 'localhost:9200/shop/cloth/_mapping' -d '{
  "cloth" : {
    "properties" : {
      "name" : {"type" : "string"}
    }
  }
}'
```

Child mappings

To create child mappings, we need to add the _parent property with the name of the parent type—cloth, in our case. So, the command that will create the variation type would look as follows:

```
curl -XPUT 'localhost:9200/shop/variation/_mapping' -d '{
  "variation" : {
    "_parent" : { "type" : "cloth" },
    "properties" : {
      "size" : {"type" : "string", "index" : "not_analyzed"},
      "color" : {"type" : "string", "index" : "not_analyzed"}
    }
  }
}'
```

And, that's all. You don't need to specify which field will be used to connect child documents to the parent ones because, by default, Elasticsearch will use the unique identifier for that. If you remember from the previous chapters, the information about a unique identifier is present in the index by default.

The parent document

Now, we are going to index our parent document. It's very simple; to do that, we just run the usual indexing command, for example, the one as follows:

```
curl -XPOST 'localhost:9200/shop/cloth/1' -d '{  
  "name" : "Test shirt"  
}'
```

If you look at the preceding command, you'll notice that our document will be given the identifier 1.

The child documents

To index child documents, we need to provide information about the parent document with the use of the `parent` request parameter and set that parameter value to the identifier of the parent document. So, to index two child documents to our parent document, we would need to run the following command lines:

```
curl -XPOST 'localhost:9200/shop/variation/1000?parent=1' -d '{  
  "color" : "red",  
  "size" : "XXL"  
}'
```

Also, we need to run the following command lines to index the second child document:

```
curl -XPOST 'localhost:9200/shop/variation/1001?parent=1' -d '{  
  "color" : "black",  
  "size" : "XL"  
}'
```

And that's all. We've indexed two additional documents, which are of a new type, but we've specified that our documents have a parent—the document with an identifier of 1.

Querying

We've indexed our data and now we need to use appropriate queries to match documents with the data stored in their children. Of course, we can also run queries against the child documents and check their parent's existence. However, please note that when running queries against parents, child documents won't be returned, and vice versa.

Querying data in the child documents

So, if we would like to get clothes that are of the XXL size and in red, we would run the following command lines:

```
curl -XGET 'localhost:9200/shop/_search?pretty' -d '{
  "query" : {
    "has_child" : {
      "type" : "variation",
      "query" : {
        "bool" : {
          "must" : [
            { "term" : { "size" : "XXL" } },
            { "term" : { "color" : "red" } }
          ]
        }
      }
    }
  }
}'
```

The query is quite simple; it is of the `has_child` type, which tells Elasticsearch that we want to search in the child documents. In order to specify which type of children we are interested in, we specify the `type` property with the name of the child type. Then we have a standard `bool` query, which we've already discussed. The result of the query will contain only parent documents, which in our case will look as follows:

```
{
  (...)

  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "shop",
      "_type" : "cloth",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "name" : "Test shirt" }
    } ]
  }
}
```

```
    }  
}
```

The top_children query

In addition to the `has_child` query, Elasticsearch exposes one additional query that returns parent documents, but is run against the child documents – the `top_children` query. That query can be used to run against a specified number of child documents. Let's look at the following query:

```
{  
  "query" : {  
    "top_children" : {  
      "type" : "variation",  
      "query" : {  
        "term" : { "size" : "XXL" }  
      },  
      "score" : "max",  
      "factor" : 10,  
      "incremental_factor" : 2  
    }  
  }  
}
```

The preceding query will be run first against a total of 100 child documents (`factor` multiplied by the default `size` parameter of 10). If there are 10 parent documents found (because of the default `size` parameter being equal to 10), then those will be returned and the query execution will end. However, if fewer parents are returned and there are still child documents that were not queried, another 20 documents will be queried (the `incremental_factor` parameter multiplied by the result's size), and so on, until the requested amount of parent documents will be found or there are no child documents left to be queried.

The `top_children` query offers the ability to specify how the score should be calculated with the use of the `score` parameter, with the value of `max` (maximum of all the scores of child queries), `sum` (sum of all the scores of child queries), or `avg` (average of all the scores of child queries) as the possible ones.

Querying data in the parent documents

If you would like to return child documents that match a given data in the parent document, you should use the `has_parent` query. It is similar to the `has_child` query; however, instead of the `type` property, we specify the `parent_type` property with the value of the parent document type. For example, the following query will return both the child documents that we've indexed, but not the parent document:

```
curl -XGET 'localhost:9200/shop/_search?pretty' -d '{
  "query" : {
    "has_parent" : {
      "parent_type" : "cloth",
      "query" : {
        "term" : { "name" : "test" }
      }
    }
  }
}'
```

The response from Elasticsearch should be similar to the following one:

```
{
  (...)

  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "shop",
      "_type" : "variation",
      "_id" : "1000",
      "_score" : 1.0, "_source" : {"color" : "red","size" : "XXL"}
    }, {
      "_index" : "shop",
```

```
    "_type" : "variation",
    "_id" : "1001",
    "_score" : 1.0, "_source" : {"color" : "black", "size" : "XL"}
  } ]
}
}
```

The parent-child relationship and filtering

If you would like to use the parent-child queries as filters, you can; there are `has_child` and `has_parent` filters that have the same functionality as queries with corresponding names. Actually, Elasticsearch wraps those filters in the constant score query to allow them to be used as queries.

Performance considerations

When using the Elasticsearch parent-child functionality, you have to be aware of the performance impact that it has. The first thing you need to remember is that the parent and the child documents need to be stored in the same shard in order for the queries to work. If you happen to have a high number of children for a single parent, you may end up with shards not having a similar number of documents. Because of that, your query performance can be lower on one of the nodes, resulting in the whole query being slower. Also, please remember that the parent-child queries will be slower than the ones that run against documents that don't have a relationship between them.

The second very important thing is that when running queries, like the `has_child` query, Elasticsearch needs to preload and cache the document identifiers. Those identifiers will be stored in the memory and you have to be sure that you have given Elasticsearch enough memory to store those identifiers. Otherwise, you can expect `OutOfMemory` exceptions to be thrown and your nodes or the whole cluster not being operational.

Finally, as we mentioned, the first query will preload and cache the document identifiers. This takes time. In order to improve the performance of initial queries that use the parent-child relationship, **Warmer API** can be used. You can find more information about how to add warming queries to Elasticsearch in the *Warming up* section of *Chapter 8, Administrating Your Cluster*.

Modifying your index structure with the update API

In the previous chapters, we discussed how to create index mappings and index the data. But what if you already have the mappings created and data indexed, but want to modify the structure of the index? This is possible to some extent. For example, by default, if we index a document with a new field, Elasticsearch will add that field to the index structure. Let's now look at how to modify the index structure manually.

The mappings

Let's assume that we have the following mappings for our `users` index stored in the `user.json` file:

```
{
  "user" : {
    "properties" : {
      "name" : {"type" : "string"}
    }
  }
}
```

As you can see, it is very simple. It just has a single property that will hold the username. Now, let's create an index called `users`, and use the previous mappings to create our own type. To do that, we will run the following commands:

```
curl -XPOST 'localhost:9200/users'
curl -XPUT 'localhost:9200/users/_mapping' -d @user.json
```

If everything functions correctly, we will have our index and type created. So now, let's try to add a new field to the mappings.

Adding a new field

In order to illustrate how to add a new field to our mappings, we assume that we want to add a phone number to the data stored for each user. In order to do that, we need to send an `HTTP PUT` command to the `/index_name/type_name/_mapping` REST endpoint with the proper body that will include our new field. For example, to add the `phone` field, we would run the following command:

```
curl -XPUT 'http://localhost:9200/users/_mapping' -d '{
  "user" : {
```

```
"properties" : {  
    "phone" : {"type" : "string",  
              "store" : "yes",  
              "index" : "not_analyzed"}  
}  
}  
}'
```

And again, if everything functions correctly, we should have a new field added to our index structure. To ensure everything is all right, we can run the `GET /_mapping` REST endpoint and Elasticsearch will return the appropriate mappings. An example command to get the mappings for our `user` type in the `users` index could look as follows:

```
curl -XGET 'localhost:9200/users/_mapping?pretty'
```

 After adding a new field to the existing type, we need to index all the documents again, because Elasticsearch didn't update them automatically. This is crucial to remember. You can use your primary source of data to do that or use the `_source` field to get the original data from it and index it once again.

Modifying fields

So now, our index structure contains two fields: `name` and `phone`. We indexed some data, but after a while, we decided that we want to search on the `phone` field and we would like to change the `index` property from `not_analyzed` to `analyzed`. So, we run the following command:

```
curl -XPUT 'http://localhost:9200/users/_mapping' -d '{  
    "user" : {  
        "properties" : {  
            "phone" : {"type" : "string",  
                      "store" : "yes",  
                      "index" : "analyzed"}  
        }  
    }  
}'
```

After running the preceding command lines, Elasticsearch returns the following output:

```
{"error": "MergeMappingException[Merge failed with failures { [mapper [phone] has different index values, mapper [phone] has different 'norms. enabled' values, mapper [phone] has different tokenize values, mapper [phone] has different index_analyzer]}]", "status": 400}
```

This is because we can't change the `not_analyzed` field to `analyzed`. And not only that, in most cases you won't be able to update the fields mapping. This is a good thing, because if we would be allowed to change such settings, we would confuse Elasticsearch and Lucene. Imagine that we already have many documents with the `phone` field set to `not_analyzed` and we are allowed to change the mappings to `analyzed`. Elasticsearch wouldn't change the data that was already indexed, but the queries that are analyzed would be processed with a different logic and thus you wouldn't be able to properly find your data.

However, to give you some examples of what is prohibited and what is not, we will mention some of the operations for both cases. For example, the following modifications can be safely made:

- Adding a new type definition
- Adding a new field
- Adding a new analyzer

The following modifications are prohibited or will not work:

- Changing the type of the field (for example from `text` to `numeric`)
- Changing `stored` to `field` to `not` to be stored and vice versa
- Changing the value of the `indexed` property
- Changing the analyzer of already indexed documents

Please remember that the preceding mentioned examples of allowed and not allowed updates do not mention all of the possibilities of the Update API usage and you have to try for yourself if the update you are trying to do will work.

If you want to ignore conflicts and just put the new mappings, you can set the `ignore_conflicts` parameter to `true`. This will cause Elasticsearch to overwrite your mappings with the one you send. So, our preceding command with the additional parameter would look as follows:

```
curl -XPUT 'http://localhost:9200/users/_user/_mapping?ignore_conflicts=true' -d '...'
```

Summary

In this chapter, we learned how to index tree-like structures using Elasticsearch. In addition to that, we indexed data that is not flat and modified the structure of already-created indices. Finally, we learned how to handle relationships by using nested documents and by using the Elasticsearch parent-child functionality.

In the next chapter, we'll focus on making our search even better. We will see how Apache Lucene scoring works and why it matters so much. We will learn how to use the Elasticsearch function-score query to adjust the importance of our documents using different functions and we'll leverage the provided scripting capabilities. We will search the content in different languages and discuss when index time-boosting makes sense. We'll use synonyms to match words with the same meaning and we'll learn how to check why a given document was found by a query. Finally, we'll influence queries with boosts, and we will learn how to understand the score calculation done by Elasticsearch.

5

Make Your Search Better

In the previous chapter, we learned how Elasticsearch indexing works when it comes to data that is not flat. We saw how to index tree-like structures. In addition to that, we indexed data that had an object-oriented structure. We also learned how to modify the structure of already created indices. Finally, we saw how to handle relationships in Elasticsearch by using nested documents as well as the parent-child functionality. By the end of this chapter, you will have learned the following topics:

- Apache Lucene scoring
- Using the scripting capabilities of Elasticsearch
- Indexing and searching data in different languages
- Using different queries to influence the score of the returned documents
- Using index-time boosting
- Words having the same meaning
- Checking why a particular document was returned
- Checking score calculation details

An introduction to Apache Lucene scoring

When talking about queries and their relevance, we can't omit information about scoring and where it comes from. But what is the score? The **score** is a parameter that describes the relevance of a document against a query. In the following section, we will discuss the default Apache Lucene scoring mechanism, the TF/IDF algorithm, and how it affects the returned document.



The TF/IDF algorithm is not the only available algorithm exposed by Elasticsearch. For more information about available models, refer to the *Different similarity models* section in *Chapter 2, Indexing Your Data*, and our book, *Mastering ElasticSearch*, Packt Publishing.

When a document is matched

When a document is returned by Lucene, it means that Lucene matched the query we sent and that document has been given a score. The higher the score, the more relevant the document is from the search engine point of view. However, the score calculated for the same document on two different queries will be different.

Because of that, comparing scores between queries usually doesn't make much sense. However, let's get back to the scoring. Multiple factors are taken into account to calculate the score property for a document, which are as follows:

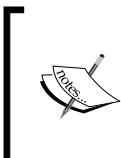
- **Document boost:** This is the boost value given to a document during indexing.
- **Field boost:** This is the boost value given to a field during querying and indexing.
- **Coord:** This is the coordination factor that is based on the number of terms the document has. It is responsible for giving more value to the documents that contain more search terms compared to other documents.
- **Inverse document frequency:** This is a term-based factor that tells the scoring formula how rare the given term is. The higher the inverse document frequency, the rarer the term.
- **Length norm:** This is a field-based factor for normalization based on the number of terms the given field contains. The longer the field, the smaller boost this factor will give. It basically means that shorter documents will be favored.
- **Term frequency:** This is a term-based factor that describes how many times the given term occurs in a document. The higher the term frequency, the higher the score of the document.
- **Query norm:** This is a query-based normalization factor that is calculated as the sum of the squared weight of each of the query terms. Query norm is used to allow score comparison between queries, which is not always easy and possible.

Default scoring formula

The practical formula for the TF/IDF algorithm looks as follows:

$$score(q, d) = coord(q, d) * queryNorm(q) * \sum_{t \in q} (tf(t \text{ in } d) * idf(t)^2 * boost(t) * norm(t, d))$$

To adjust your query relevance, you don't need to remember the details of the equation, but it is very important to at least know how it works. We can see that the score factor for the document is a function of query q and document d . There are also two factors that are not dependent directly on the query terms, coord and queryNorm . These two elements of the formula are multiplied by the sum calculated for each term in the query. The sum, on the other hand, is calculated by multiplying the term frequency for the given term, its inverse document frequency, term boost, and the norm, which is the length norm we've discussed earlier.



Note that the preceding formula is a practical one. You can find more information about the conceptual formula in the Lucene Javadocs, which is available at http://lucene.apache.org/core/4_7_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

The good thing about the preceding rules is that you don't need to remember all of them. What you should be aware of is what matters when it comes to the document score. Basically, the following are a few rules that are derived from the preceding equation:

- The more rare the term matched is, the higher score the document will have.
- The smaller the document fields are, the higher the score the document will have.
- The higher the boost for fields is, the higher the score the document will have.
- As we can see, Lucene will give the higher score to the documents that have the highest number of query terms matched in the document contents and have shorter fields (less terms indexed). Also, rarer terms will be favored instead of the common ones.

Relevancy matters

In most of the cases, we want to get the best matching documents. However, the most relevant documents don't always mean the same. Some use cases define very strict rules on why a given document should be at a higher level on the results list. For example, one can say that in addition for the document to be a perfect match in terms of the TF/IDF similarity, we have customers, who pay for their documents to be higher in the results. Depending on the customer plan, we want to give more importance to such documents. In such cases, we would want the documents for the customers that pay the most to be at the top in our search results. Of course, this is not relevant in TF/IDF.

This is a very simple example, but Elasticsearch queries can become really complicated. We will discuss those queries in the *Influencing scores with query boosts* section of this chapter.

When working on search relevance, you should always remember that it is not a one-time process. Your data will change with time and your queries will need to be adjusted accordingly. In most cases, tuning query relevancy will be constant work. You will need to react to your business rules and needs, to how users behave, and so on. It is very important to remember that this is not a one-time process which you can forget about once you set it.

Scripting capabilities of Elasticsearch

Elasticsearch has a few functionalities where scripts can be used. You've already seen examples such as updating documents, filtering, and searching. Regardless of the fact that this seems to be advanced, we will take a look at the possibilities offered by Elasticsearch, because scripts are priceless for some use cases.

If we look at any request made to Elasticsearch that uses scripts, we will notice some similar properties, which are as follows:

- `Script`: This property contains the actual script code.
- `Lang`: This property defines the field that provides information about the script language. If it is omitted, Elasticsearch assumes `mvel`.
- `Params`: This object contains parameters and their values. Every defined parameter can be used inside the script by specifying that parameter name. Using parameters, we can write cleaner code. Scripts using parameters are executed faster than code with embedded constants because of caching.

Objects available during script execution

During different operations, Elasticsearch allows us to use different objects in the scripts. To develop a script that fits our use case, we should be familiar with those objects.

For example, during a search operation the following objects are available:

- `_doc` (also available as `doc`): This is an instance of the `org.elasticsearch.search.lookup.DocLookup` object. It gives us access to the current document found with the calculated score and field values.

- `_source`: This is an instance of the `org.elasticsearch.search.lookup.SourceLookup` object. This object provides access to the source of the current document and the values defined in that source.
- `_fields`: This is an instance of the `org.elasticsearch.search.lookup.FieldsLookup` object. Again, it can be used to access the values of document fields.

On the other hand, during a document update operation, Elasticsearch exposes only the `ctx` object with the `_source` property, which provides access to the current document.

As we have previously seen, several methods are mentioned in the context of document fields and their values. Let's now look at the following examples of how to get the value for the `title` field. In the brackets, you can see what Elasticsearch will return for one of our example documents from the `library` index:

- `_doc.title.value (crime)`
- `_source.title (Crime and Punishment)`
- `_fields.title.value (null)`

A bit confusing, isn't it? During indexing, a field value is sent to Elasticsearch as a part of the `_source` document. Elasticsearch can store this information and does that by default. In addition to that, the document is parsed and every field may be stored in an index if it is marked as stored (that is, if the `store` property is set to `true`; otherwise, by default, the fields are not stored). Finally, the field value may be configured as indexed. This means that the field value is analyzed, divided into tokens, and placed in the index. To sum up, one field may be stored in an index as follows:

- A part of the `_source` document
- A stored and unparsed value
- An indexed value that is parsed into tokens

In scripts, we have access to all these representations except updating. You may wonder which version we should use. Well, if we want access to the processed form, the answer would be as simple as `_doc`. What about `_source` and `_fields`? In most cases, `_source` is a good choice. It is usually fast and needs less disk operations than reading the original field values from the index.

MVEL

Elasticsearch can use several languages for scripting. When not explicitly declared, it assumes that **MVEL (MVFLEX Expression Language)** is used. MVEL is fast, easy to use and embed, and a simple but powerful expression language used in open source projects. It allows us to use Java objects, automatically maps properties to a getter/setter call, converts simple types and maps collections, and maps to arrays and associative arrays. For more information about MVEL, refer to <http://mvel.codehaus.org/Language+Guide+for+2.0>.

Using other languages

Using MVEL for scripting is a simple and sufficient solution, but if you would like to use something different, you can choose among JavaScript, Python, or Groovy. Before using other languages, we must install an appropriate plugin. You can read more about plugins in the *Elasticsearch Plugins* section of *Chapter 8, Administering Your Cluster*. For now, we'll just run the following command from the Elasticsearch directory:

```
bin/plugin -install elasticsearch/elasticsearch-lang-
    javascript/2.0.0.RC1
```

The preceding command will install a plugin that will allow us to use JavaScript. The only change we should make in the request is to add the additional information of the language we are using for scripting, and, of course, modify the script itself to be correct in the new language. Look at the following example:

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "doc.tags.values.length > 0 ? doc.tags.values[0]
        : '\u19999';",
      "lang" : "javascript",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

As you can see, we used JavaScript for scripting instead of the default MVEL. The `lang` parameter informs Elasticsearch about the language being used.

Using our own script library

Usually, scripts are small and it is quite convenient to put them in the request. But sometimes applications grow and you want to give the developers something that they can reuse in their modules. If the scripts are large and complicated, it is generally better to place them in files and only refer them in API requests. The first thing to do is to place our script in the proper place with a proper name. Our tiny script should be placed in the Elasticsearch directory, `config/scripts`. Let's name our example script file `text_sort.js`. Note that the extension of the file should indicate the language used for scripting; in our case, we will use JavaScript.

The content of this example file is very simple and looks as follows:

```
doc.tags.values.length > 0 ? doc.tags.values[0] : '\u19999';
```

And the query using the preceding script will look as follows:

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "text_sort",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

As you can see, we can now use `text_sort` as the script name. In addition, we can omit the script language; Elasticsearch will figure it out from the file extension.

Using native code

In case the scripts are too slow or you don't like scripting languages, Elasticsearch allows you to write Java classes and use them instead of scripts.

The factory implementation

We need to implement at least two classes to create a new native script. The first one is a factory for our script. For now, let's focus on it. The following sample code illustrates the factory for our script:

```
package pl.solr.elasticsearch.examples.scripts;
import java.util.Map;
```

```
import org.elasticsearch.common.Nullable;
import org.elasticsearch.script.ExecutableScript;
import org.elasticsearch.script.NativeScriptFactory;

public class HashCodeSortNativeScriptFactory implements
    NativeScriptFactory {

    @Override
    public ExecutableScript newScript(@Nullable Map<String, Object>
        params) {
        return new HashCodeSortScript(params);
    }

}
```

The essential parts are highlighted in the code snippet. This class should implement the `org.elasticsearch.script.NativeScriptFactory` class. The interface forces us to implement the `newScript()` method. It takes parameters defined in the API call and returns an instance of our script.

Implementing the native script

Now let's look at the implementation of our script. The idea is simple—our script will be used for sorting. Documents will be ordered by the `hashCode()` value of the chosen field. Documents without a field defined will be the first. We know the logic doesn't make too much sense, but it is good for presentation as it is simple. The source code for our native script looks as follows:

```
package pl.solr.elasticsearch.examples.scripts;

import java.util.Map;
import org.elasticsearch.script.AbstractSearchScript;

public class HashCodeSortScript extends AbstractSearchScript {
    private String field = "name";

    public HashCodeSortScript(Map<String, Object> params) {
        if (params != null && params.containsKey("field")) {
            this.field = params.get("field").toString();
        }
    }

    @Override
    public Object run() {
        Object value = source().get(field);
```

```
    if (value != null) {
        return value.hashCode();
    }
    return 0;
}

}
```

First of all, our class inherits from the `org.elasticsearch.script.AbstractSearchScript` class and implements the `run()` method. This is where we get the appropriate values from the current document, process it according to our strange logic, and return the result. You may notice the `source()` call. Yes, it is exactly the same `_source` parameter that we met in the non-native scripts. The `doc()` and `fields()` methods are also available and they follow the same logic we described earlier.

The thing worth looking at is how we've used the parameters. We assume that a user can put the `field` parameter, telling us which document field will be used for manipulation. We also provide a default value for this parameter.

Installing scripts

Now it's time to install our native script. After packing the compiled classes as a JAR archive, we should put it in the Elasticsearch `lib` directory. This makes our code visible to the class loader. What we should then do is register our script. This can be done by using the settings API call or by adding a single line to the `elasticsearch.yml` configuration file. We've chosen to put the script in the `elasticsearch.yml` configuration file by adding the following line to the mentioned file:

```
script.native.native_sort.type:
  pl.solv.elasticsearch.examples.scripts.
    HashCodeSortNativeScriptFactory
```

Note the `native_sort` fragment. This is the script name that will be used during requests and will be passed to the `script` parameter. The value for this property is the full classname of the factory we implemented and will be used for script initialization. The last thing we need is to restart Elasticsearch.

Running the script

We've restarted Elasticsearch so that we can start sending the queries that use our native script. For example, we will send a query that uses our previously indexed data from the `library` index. This example query looks as follows:

```
{
  "query" : {
```

```
"match_all" : { }
},
"sort" : {
  "_script" : {
    "script" : "native_sort",
    "params" : {
      "field" : "otitle"
    },
    "lang" : "native",
    "type" : "string",
    "order" : "asc"
  }
}
}
```

Note the `params` part of the query. In this call, we want to sort on the `otitle` field. We provide the script name `native_sort` and the script language `native`. This is required. If everything goes well, we should see our results sorted by our custom sort logic. If we will look at the response from Elasticsearch, we will see that documents without the `otitle` field are at the first few positions of the results list and their sort value is 0.

Searching content in different languages

Till now, when discussing language analysis, we've talked mostly in theory. We didn't see an example regarding language analysis, handling multiple languages that our data can consist of, and so on. Now this will change, as we will discuss how we can handle data in multiple languages.

Handling languages differently

As you already know, Elasticsearch allows us to choose different analyzers for our data. We can have our data divided on the basis of whitespaces, have them lowercased, and so on. This can usually be done with the data regardless of the language—you should have the same tokenization on the basis of whitespaces for English, German, and Polish (that doesn't apply to Chinese, though). However, what if you want to find documents that contain words such as `cat` and `cats` by only sending the word `cat` to Elasticsearch? This is where language analysis comes into play with stemming algorithms for different languages, which allow the analyzed words to be reduced into their root forms

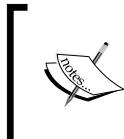
And now the worst part—we can't use one general stemming algorithm for all the languages in the world; we have to choose one appropriate language. The following sections in the chapter will help you with some parts of the language analysis process.

Handling multiple languages

There are a few ways of handling multiple languages in Elasticsearch, and all of them have some pros and cons. We won't be discussing everything, but just for the purpose of giving you an idea, a few of those methods are as follows:

- Storing documents in different languages as different types
- Storing documents in different languages in separate indices
- Storing different versions of fields in a single document so that they contain different languages

However, we will focus on a single method that allows us to store documents in different languages in a single index. We will focus on a problem where we have a single type of document, but they may come from all over the world, and thus can be written in multiple languages. Also, we would like to enable our users to use all the analysis capabilities, such as stemming and stop words for different languages, not only for English.



Note that stemming algorithms perform differently for different languages—both in terms of analysis performance and the resulting terms. For example, English stemmers are very good, but you can run into issues with European languages, such as German.



Detecting the language of the documents

If you don't know the language of your documents and queries (and this is mostly the case), you can use software for language detection that can be used to detect (with some probability) the language of your documents and queries.

If you use Java, you can use one of the few available language detection libraries. Some of them are as follows:

- Apache Tika (<http://tika.apache.org/>)
- Language detection (<http://code.google.com/p/language-detection/>)

The language detection library claims to have over 99 percent precision for 53 languages; that's a lot if you ask us.

You should remember, though, that data language detection will be more precise for longer text. However, because the text of queries is usually short, you'll probably have some degree of errors during query language identification.

Sample document

Let's start with introducing a sample document, which is as follows:

```
{  
    "title" : "First test document",  
    "content" : "This is a test document",  
    "lang" : "english"  
}
```

As you can see, the document is pretty simple; it contains the following three fields:

- **Title**: This field holds the title of the document
- **Content**: This field holds the actual content of the document
- **Lang**: This field defines the identified language

The first two fields are created from our user's documents and the third one is the language that our hypothetical user has chosen when he or she uploaded the document.

To inform Elasticsearch which analyzer should be used, we map the `lang` field to one of the analyzers that exist in Elasticsearch (a full list of these analyzers can be found at <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-lang-analyzer.html>), and if the user enters a language that is not supported, we don't specify the `lang` field at all so that Elasticsearch uses the default analyzer.

The mappings

Let's now look at the mappings created to hold the preceding documents (we've stored them in the `mappings.json` file), as follows:

```
{  
    "mappings" : {  
        "doc" : {  
            "_analyzer" : {  
                "path" : "lang"  
            },  
            "properties" : {  
                ...  
            }  
        }  
    }  
}
```

```
    "title" : {
        "type" : "string",
        "index" : "analyzed",
        "store" : "no",
        "fields" : {
            "default" : {
                "type" : "string",
                "index" : "analyzed",
                "store" : "no",
                "analyzer" : "simple"
            }
        }
    },
    "content" : {
        "type" : "string",
        "index" : "analyzed",
        "store" : "no",
        "fields" : {
            "default" : {
                "type" : "string",
                "index" : "analyzed",
                "store" : "no",
                "analyzer" : "simple"
            }
        }
    },
    "lang" : {
        "type" : "string",
        "index" : "not_analyzed",
        "store" : "yes"
    }
}
```

In the preceding mappings, we are most interested in the analyzer definition and the `title` and `description` fields (if you are not familiar with any aspect of mappings, refer to the *Mappings configuration* section of *Chapter 2, Indexing Your Data*). We want the analyzer to be based on the `lang` field. Therefore, we need to add a value in the `lang` field that is equal to one of the names of the analyzers known to Elasticsearch (the default one or another defined by us).

After that comes the definitions of two fields that hold the actual data. As you can see, we used the multifield definition in order to index the `title` and `description` fields. The first one of the multifields is indexed with the analyzer specified by the `lang` field (because we didn't specify the exact analyzer name, the one defined globally will be used). We will use that field when we know in which language the query is specified. The second of the multifields uses a simple analyzer and will be used to search when a query language is unknown. However, the simple analyzer is only an example and you can also use a standard analyzer or any other of your choice.

In order to create a sample index called `docs` that use our mappings, we will use the following command:

```
curl -XPUT 'localhost:9200/docs' -d @mappings.json
```

Querying

Now let's see how we can query our data. We can divide the querying situation into two different cases.

Queries with the identified language

The first case is when we have our query language identified. Let's assume that the identified language is English and we know that English matches the `english` analyzer. In such cases, our query is as follows:

```
curl -XGET 'localhost:9200/docs/_search?pretty=true' -d '{
  "query" : {
    "match" : {
      "content" : {
        "query" : "documents",
        "analyzer" : "english"
      }
    }
  }
}'
```

Note the `analyzer` parameter, which indicates which analyzer we want to use. We set that parameter to the name of the analyzer corresponding to the identified language. Note that the term we are looking for is `documents`, while the term in the document is `document`, but the `english` analyzer should take care of it and find that document. The response returned by Elasticsearch will be as follows:

```
{  
    "took" : 2,  
    "timed_out" : false,  
    "_shards" : {  
        "total" : 5,  
        "successful" : 5,  
        "failed" : 0  
    },  
    "hits" : {  
        "total" : 1,  
        "max_score" : 0.19178301,  
        "hits" : [ {  
            "_index" : "docs",  
            "_type" : "doc",  
            "_id" : "1",  
            "_score" : 0.19178301  
        } ]  
    }  
}
```

Queries with unknown languages

Now let's assume that we don't know the language used for the user's query. In such cases, we can't use the field analyzed with the analyzer specified by our `lang` field, because we don't want to analyze the query with an analyzer that is language specific. In that case, we will use our standard simple analyzer, and we will send the query to the `contents.default` field instead of `content`. The query will be as follows:

```
curl -XGET 'localhost:9200/docs/_search?pretty=true' -d '{  
    "query" : {  
        "match" : {  
            "content.default" : {  
                "query" : "documents",  
                "analyzer" : "simple"  
            }  
        }  
    }'  
}'
```

However, we didn't get any results this time, because the simple analyzer can't deal with a singular form of a word when we are searching with a plural form.

Combining queries

To additionally boost the documents that perfectly match with our default analyzer, we can combine the two preceding queries with the `bool` query, as follows:

```
curl -XGET 'localhost:9200/docs/_search?pretty=true' -d '{
  "query" : {
    "bool" : {
      "minimum_should_match" : 1,
      "should" : [
        {
          "match" : {
            "content" : {
              "query" : "documents",
              "analyzer" : "english"
            }
          }
        },
        {
          "match" : {
            "content.default" : {
              "query" : "documents",
              "analyzer" : "simple"
            }
          }
        }
      ]
    }
  }
}'
```

For the document to be returned, at least one of the defined queries must match. If they both match, the document will have a higher score value and will be placed higher in the results.

There is one additional advantage of the preceding combined query – if our language analyzer won't find a document (for example, when the analysis is different from the one used during indexing), the second query has a chance to find the terms that are tokenized only by whitespace characters and lowercased.

Influencing scores with query boosts

In the previous chapter, we learned what scoring is and how Elasticsearch calculates it. When an application grows, the need for improving the quality of search also increases. We call it the search experience. We need to gain knowledge about what is more important to the user and see how users use the search functionality. This leads to various conclusions; for example, we see that some parts of the documents are more important than the others or that particular queries emphasize one field at the cost of others. This is where boosting can be used.

The boost

Boost is an additional value used in the process of scoring. We already know it can be applied to the following:

- **query**: This is a way to inform the search engine that the given query is a part of the complex query and is more significant than the others.
- **field**: Several document fields are important for the user. For example, searching e-mails by Bill should probably list those from Bill first, followed by those with Bill in the subject, and then the e-mails mentioning Bill in the content.

The values assigned by us to a query or field are only one of the factors used when we calculate the resulting score and we are aware of this. We will now look at a few examples of query boosting.

Adding boost to queries

Let's imagine that our index has two documents. The first document is as follows:

```
{  
  "id" : 1,  
  "to" : "John Smith",  
  "from" : "David Jones",  
  "subject" : "Top secret!"  
}
```

And, the second document is as follows:

```
{  
    "id" : 2,  
    "to" : "David Jones",  
    "from" : "John Smith",  
    "subject" : "John, read this document"  
}
```

This data is simple, but it should describe our problem very well. Now, let's assume we have the following query:

```
{  
    "query" : {  
        "query_string" : {  
            "query" : "john",  
            "use_dis_max" : false  
        }  
    }  
}
```

In this case, Elasticsearch will create a query for the `_all` field and will find documents that contain the desired words. We also said that we don't want the disjunction query to be used by specifying the `use_dis_max` parameter to `false` (if you don't remember the disjunction query, refer to the *The dismax query* and *The query_string query* sections in *Chapter 3, Searching Your Data*). As we can easily guess, both of our records will be returned and the record with the identifier equal to 2 will be returned first. This is because of the two occurrences of `John` in the `from` and `subject` fields. Let's check this out in the following result:

```
"hits" : {  
    "total" : 2,  
    "max_score" : 0.13561106,  
    "hits" : [ {  
        "_index" : "messages",  
        "_type" : "email",  
        "_id" : "2",  
        "_score" : 0.13561106, "_source" :  
        { "id" : 1, "to" : "David Jones", "from" :  
          "John Smith", "subject" : "John, read this document"}  
    }, {  
        "_index" : "messages",  
        "_type" : "email",
```

```
    "_id" : "1",
    "_score" : 0.11506981, "_source" :
    { "id" : 2, "to" : "John Smith", "from" :
      "David Jones", "subject" : "Top secret!" }
  } ]
}
```

Is everything all right? Technically, yes. But I think that the second document should be positioned as the first one in the result list, because when searching for something, the most important factor (in many cases) is matching people, rather than the subject of the message. You may disagree, but this is exactly why full-text searching relevance is a difficult topic—sometimes, it is hard to tell which ordering is better for a particular case. What can we do? First, let's rewrite our query to implicitly inform Elasticsearch what fields should be used for searching, as follows:

```
{
  "query" : {
    "query_string" : {
      "fields" : ["from", "to", "subject"],
      "query" : "john",
      "use_dis_max" : false
    }
  }
}
```

This is not exactly the same query as the previous one. If we run it, we will get the same results (in our case), but if you will look carefully, you will notice differences in scoring. In the previous example, Elasticsearch only used one field, `_all`. Now we are searching in three fields. This means that several factors, such as field lengths, are changed. Anyway, this is not so important in our case. Under the hood, Elasticsearch generates a complex query made up of three queries—one to each field. Of course, the score contributed by each query depends on the number of terms found in this field and the length of this field. Let's introduce some differences between the fields. Compare the following query to the preceding one:

```
{
  "query" : {
    "query_string" : {
      "fields" : ["from^5", "to^10", "subject"],
      "query" : "john",
      "use_dis_max" : false
    }
  }
}
```

Look at the highlighted parts (^5 and ^10). In this manner, we can tell Elasticsearch how important a given field is. We see that the most important field is the `to` field, and the `from` field is less important. The `subject` field has a default value for `boost`, which is `1.0`. Always remember that this value is only one of the various factors. You may be wondering why we choose `5`, and not `1000` or `1.23`. Well, this value depends on the effect we want to achieve, what query we have, and most importantly, what data we have in our index. Typically, when data changes in the meaningful parts, we should probably check and tune our relevance once again.

Finally, let's look at a similar example, but using the `bool` query, as follows:

```
{  
  "query" : {  
    "bool" : {  
      "should" : [  
        { "term" : { "from": { "value" : "john", "boost" : 5 } } },  
        { "term" : { "to": { "value" : "john", "boost" : 10 } } },  
        { "term" : { "subject": { "value" : "john" } } }  
      ]  
    }  
  }  
}
```

Modifying the score

The preceding example shows how to affect the result list by boosting particular query components. Another technique is to run a query and affect the score of the matched documents. In the following sections, we will summarize the possibilities offered by Elasticsearch. In the examples, we will use the library data that we already used in *Chapter 3, Searching Your Data*.

The `constant_score` query

A `constant_score` query allows us to take any filter or query and explicitly set the value that should be used as the score, which will be given for each matching document by using the `boost` parameter.

Initially, this query doesn't seem to be practical. But when we think about building complex queries, this query allows us to set how many documents matching this query can affect the total score. Look at the following example:

```
{  
  "query" : {  
    "constant_score" : {  
      "query": {
```

```
        "query_string" : {
            "query" : "available:false author:heller"
        }
    }
}
```

In our data, we have two documents with the available field set to `false`. One of these documents has an additional value in the author field. But, thanks to the `constant_score` query, Elasticsearch will ignore that information during scoring. Both documents will be given a score of `1.0`.

The boosting query

The next type of query related to boosting is the boosting query. The idea is to allow us to define an additional part of a query when every matched document score decreases. The following example lists all available books, but books written by E. M. Remarque will have a score that is 10 times lower:

```
{
  "query" : {
    "boosting" : {
      "positive" : {
        "term" : {
          "available" : true
        }
      },
      "negative" : {
        "match" : {
          "author" : "remarque"
        }
      },
      "negative_boost" : 0.1
    }
  }
}
```

The function_score query

Until now, we've seen two examples of queries that allow us to alter the score of the returned documents. The third example we want to talk about, the `function_score` query, is way more complicated compared to the previous queries. This query is very useful when the score calculation is expensive, because it will compute the score on the filtered documents.

The structure of the function query

The structure of the function query is quite simple and looks as follows:

```
{  
  "query" : {  
    "function_score" : {  
      "query" : { ... },  
      "filter" : { ... },  
      "functions" : [  
        {  
          "filter" : { ... },  
          "FUNCTION" : { ... }  
        }  
      ],  
      "boost_mode" : " ... ",  
      "score_mode" : " ... ",  
      "max_boost" : " ... ",  
      "boost" : " ... "  
    }  
  }  
}
```

In general, the `function_score` query can use `query` or `filter`, one of several functions, and additional parameters. Each function can have a filter defined to filter the results on which it will be applied. If no filter is defined for a function, it will be applied to all documents.

The logic behind the `function_score` query is quite simple. First of all, the functions are matched against the documents and the score is calculated based on the `score_mode` parameter. Then, the query score for the document is combined with the score calculated for the functions and combined together on the basis of the `boost_mode` parameter.

Let's now discuss the parameters:

- `boost_mode`: The `boost_mode` parameter allows us to define how the score computed by the function queries will be combined with the score of the query. The following values are allowed:
 - `multiply`: This is the default behavior, which results in the query score being multiplied by the score computed from the functions
 - `replace`: This value causes the query score to be totally ignored and the document score to be equal to the score calculated by the functions

- sum: This value causes the document score to be calculated as the sum of the query and function scores
 - avg: This value returns an average of the query score and the function score
 - max: This value returns the maximum of the query score and the function score to the document
 - min: This value gives a minimum of the query score and the function score to the document
- score_mode: The `score_mode` parameter defines how the score computed by the functions are combined together. The following are the values of the `score_mode` parameter:
 - multiply: This is the default behavior, which results in the scores returned by the functions being multiplied
 - sum: This value sums up the scores returned by the defined functions
 - avg: The score returned by the functions is an average of all the scores of the matching functions
 - first: This value returns the score of the first function with a filter matching the document
 - max: This value returns the maximum score of functions
 - min: This value returns the minimum score of functions

There is one thing to remember – we can limit the maximum calculated score value by using the `max_boost` parameter in the `function_score` query. By default, this parameter is set to `Float.MAX_VALUE`, which means the maximum float value.

The `boost` parameter allows us to set a query-wide boost for the documents.

What we haven't talked about yet are the function scores that we can include in the `functions` section of our query. The currently available functions are as follows:

- The `boost_factor` function: This function allows us to multiply the score of the document by a given value. The value of the `boost_factor` parameter is not normalized and is taken as is. The following is an example using the `boost_factor` parameter:

```
{  
    "query" : {  
        "function_score" : {  
            "query" : {  
                "term" : {
```

```
        "available" : true
    },
},
"functions" : [
    { "boost_factor" : 20 }
]
}
}
```

- The `script_score` function: This function allows us to use a script to calculate the score that will be used as a score returned by a function (and thus will fall into the behavior defined by the `boost_mode` parameter). An example of the usage of the `script_score` function is as follows:

```
{
  "query" : {
    "function_score" : {
      "query" : {
        "term" : {
          "available" : true
        }
      },
      "functions" : [
        {
          "script_score" : {
            "script" : "_score * _source.copies * parameter1",
            "params" : {
              "parameter1" : 12
            }
          }
        ]
      }
    }
  }
}
```

- The `random_score` function: Using this function, we can generate a pseudo-random score by specifying a `seed` value. In order to simulate randomness, we should specify a new seed every time. An example of the usage of this function is as follows:

```
{
  "query" : {
    "function_score" : {
```

```

    "query" : {
      "term" : {
        "available" : true
      }
    },
    "functions" : [
      {
        "random_score" : {
          "seed" : 12345
        }
      }
    ]
  }
}

```

- The decay functions: In addition to the previously mentioned scoring functions, Elasticsearch includes additional functions called decay functions. They differ from the previously described functions, and the difference is that the score given by those functions lowers with distance. A distance is calculated on a basis of single-valued numeric field (such as date, geographical point, or standard numeric field). The simplest example that comes to mind is boosting documents on the basis of distance from a given point.

For example, let's assume that we have a point field that stores the location and we want our document score to be affected by the distance from a point where the user stands (for example, our user sends a query from a mobile device). Assuming the user is at 52, 21, we can send the following query:

```

{
  "query" : {
    "function_score" : {
      "query" : {
        "term" : {
          "available" : true
        }
      },
      "functions" : [
        {
          "linear" : {
            "point" : {
              "origin" : "52, 21",
              "scale" : "1km",

```

```
        "offset" : 0,  
        "decay" : 0.2  
    }  
}  
}  
]  
}  
}
```

In the preceding example, `linear` is the name of the decay function. The value will decay linearly when using it. The other possible values are `gauss` and `exp`. We've chosen the `linear` decay function because it sets the score to 0 when the field value exceeds the given origin value twice. This is useful when you want to lower the value of the documents that are too far away.

We have presented the relevant equations to give you an idea of how the score is calculated by the given function. The `linear` decay function calculates the score of the document using the following equation:

$$score = \max\left(0, \frac{scale - |field\ value - origin|}{scale}\right)$$

The `gauss` decay function calculates the score of the document using the following equation:

$$score = \exp\left(-\frac{(field\ value - origin)^2}{2scale^2}\right)$$

The `exp` decay function calculates the score of the document using the following equation:

$$score = \exp\left(-\frac{|field\ value - origin|}{scale}\right)$$

Of course, you don't need to calculate your document scores using pen and paper every time, but you may need it once in a while, and these equations may come in handy at such times.

Now, let's discuss the rest of the query structure. The field we want to use for score calculation is named `point`. If the document doesn't have a value in the defined field, it will be given a value of `1` at the time of calculation.

In addition to that, we've provided additional parameters. The `origin` and `scale` parameters are required. The `origin` parameter is the central point from which the calculation will be performed and `scale` is the rate of decay. By default, the `offset` parameter is set to `0`; if defined, the decay function will only compute a score for documents with values greater than the value of this parameter. The `decay` parameter tells Elasticsearch how much the score should be lowered; it is set to `0.5` by default. In our case, we've said that at the distance of 1 kilometer, the score should be reduced by 20 percent (`0.2`).



We expect the number of function scores available to be extended with newer versions of Elasticsearch and we suggest following the official documentation and the page dedicated to the `function_score` query available at <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-function-score-query.html>.

Deprecated queries

After an introduction to the `function_score` query, the `custom_boost`, `custom_score`, and `custom_filters_score` queries were deprecated. The following section shows how to achieve the same results as we did with the mentioned queries by using the `function_score` query. This section is provided as a reference for those who want to migrate from older versions of Elasticsearch and alter their queries to remove the deprecated ones.

Replacing the `custom_boost_factor` query

Let's assume that we have the following `custom_boost_factor` query:

```
{
  "query" : {
    "custom_boost_factor" : {
      "query": {
        "term" : { "author" : "heller" }
      },
      "boost_factor": 5.0
    }
  }
}
```

To replace the preceding query with the `function_score` query, we would have to use the following query:

```
{  
  "query" : {  
    "function_score" : {  
      "query": {  
        "term" : { "author" : "heller" }  
      },  
      "functions" : [  
        { "boost_factor": 5.0 }  
      ]  
    }  
  }  
}
```

Replacing the `custom_score` query

The second type of deprecated queries is the `constant_score` query. Let's assume that we have the following `custom_score` query:

```
{  
  "query" : {  
    "custom_score" : {  
      "query" : { "match_all" : {} },  
      "script" : "_source.copies * 0.5"  
    }  
  }  
}
```

If we want to replace it with the `function_score` query, it will look as follows:

```
{  
  "query" : {  
    "function_score" : {  
      "boost_mode" : "replace",  
      "query" : { "match_all" : {} },  
      "functions" : [  
        {  
          "script_score" : {  
            "script" : "_source.copies * 0.5"  
          }  
        }  
      ]  
    }  
  }  
}
```

```
        ]
    }
}

}
```

Replacing the custom_filters_score query

The last query replacement we will discuss is the `custom_filters_score` query. Let's assume we have the following query:

```
{
  "query" : {
    "custom_filters_score" : {
      "query" : { "match_all" : {} },
      "filters" : [
        {
          "filter" : { "term" : { "available" : true } },
          "boost" : 10
        }
      ],
      "score_mode" : "first"
    }
  }
}
```

If we want to replace it with the `function_score` query, it will look as follows:

```
{
  "query" : {
    "function_score" : {
      "query" : { "match_all" : {} },
      "functions" : [
        {
          "filter" : { "term" : { "available" : true } },
          "boost_factor" : 10
        }
      ],
      "score_mode" : "first"
    }
  }
}
```

When does index-time boosting make sense?

In the previous section, we discussed boosting queries. This type of boosting is very handy and powerful and fulfills its role in most situations. However, there is one case when the more convenient way is to use index-time boosting. This is the situation when we know which documents are important during the index phase. We gain a boost that is independent from a query at the cost of reindexing (we need to reindex the document when the boost value is changed). In addition to that, the performance is slightly better because some parts needed in the boosting process are already calculated at index time. Elasticsearch stores information about the boost as a part of normalization information. This is important because if we set `omit_norms` to `true`, we can't use index-time boosting.

Defining field boosting in input data

Let's look at the typical document definition, which looks as follows:

```
{  
    "title" : "The Complete Sherlock Holmes",  
    "author" : "Arthur Conan Doyle",  
    "year" : 1936  
}
```

If we want to boost the author field for this particular document, the structure should be slightly changed and the document should look as follows:

```
{  
    "title" : "The Complete Sherlock Holmes",  
    "author" : {  
        "_value" : "Arthur Conan Doyle",  
        "_boost" : 10.0,  
    },  
    "year": 1936  
}
```

And that's all. After indexing the preceding document, we will let Elasticsearch know that the importance of the author field is greater than the rest of the fields.

 In older versions of Elasticsearch, setting document-wide boost was possible. However, starting with 4.0, Lucene doesn't support whole document boosting and Elasticsearch emulated it by boosting all the fields in the document. In Elasticsearch 1.0, the document boost was deprecated and we decided not to write about it, because it will be removed in the future.

Defining boosting in mapping

It is worth mentioning that it is possible to directly define the field's boost in our mappings. The following example illustrates this:

```
{
  "mappings" : {
    "book" : {
      "properties" : {
        "title" : { "type" : "string" },
        "author" : { "type" : "string", "boost" : 10.0 }
      }
    }
  }
}
```

Thanks to the preceding boost, all queries will favor values found in the field named `author`. This also applies to queries using the `_all` field.

Words with the same meaning

You may have heard about synonyms—words that have the same or similar meaning. Sometimes, you will want to have some words match when one of those words is entered into the search box. Let's recall our sample data from *The example data section of Chapter 3, Searching Your Data*; there was a book called `Crime and Punishment`. What if we want that book to be matched not only when the words `crime` or `punishment` are used, but also when using words like `criminality` and `abuse`. To perform this, we will use synonyms.

The synonym filter

In order to use the `synonym` filter, we need to define our own analyzer. Our analyzer will be called `synonym` and will use the whitespace tokenizer and a single filter called `synonym`. Our filter's `type` property needs to be set to `synonym`, which tells Elasticsearch that this filter is a synonym filter. In addition to that, we want to ignore case so that upper- and lowercase synonyms will be treated equally (set the `ignore_case` property to `true`). To define our custom synonym analyzer that uses a `synonym` filter, we need to have the following mappings:

```
{  
  "index" : {  
    "analysis" : {  
      "analyzer" : {  
        "synonym" : {  
          "tokenizer" : "whitespace",  
          "filter" : [  
            "synonym"  
          ]  
        }  
      },  
      "filter" : {  
        "synonym" : {  
          "type" : "synonym",  
          "ignore_case" : true,  
          "synonyms" : [  
            "crime => criminality"  
          ]  
        }  
      }  
    }  
  }  
}
```

Synonyms in the mappings

In the preceding definition, we specified the synonym rule in the mappings we send to Elasticsearch. In order to do that, we need to add the `synonyms` property, which is an array of synonym rules. For example, the following part of the mappings definition defines a single synonym rule:

```
"synonyms" : [  
  "crime => criminality"  
]
```

We will discuss how to define the synonym rules in just a second.

Synonyms stored in the filesystem

Elasticsearch also allows us to use file-based synonyms. To use a file, we need to specify the `synonyms_path` property instead of the `synonyms` property. The `synonyms_path` property should be set to the name of the file that holds the synonym's definition and the specified file path should be relative to the Elasticsearch config directory. So, if we store our synonyms in the `synonyms.txt` file and save that file in the config directory, in order to use it, we should set `synonyms_path` to the value of `synonyms.txt`.

For example, the following shows how the `synonym` filter (the one from the preceding mappings) will be if we want to use the synonyms stored in a file:

```
"filter" : {
    "synonym" : {
        "type" : "synonym",
        "synonyms_path" : "synonyms.txt"
    }
}
```

Defining synonym rules

Till now, we discussed what we have to do in order to use synonym expansions in Elasticsearch. Now, let's see what formats of synonyms are allowed.

Using Apache Solr synonyms

The most common synonym structure in the Apache Lucene world is probably the one used by Apache Solr—the search engine built on top of Lucene, just like Elasticsearch. This is the default way to handle synonyms in Elasticsearch, and the possibilities of defining a new synonym are discussed in the following sections.

Explicit synonyms

A simple mapping allows us to map a list of words into other words. So, in our case, if we want the word `criminality` to be mapped to `crime` and the word `abuse` to be mapped to `punishment`, we need to define the following entries:

```
criminality => crime
abuse => punishment
```

Of course, a single word can be mapped into multiple words and multiple ones can be mapped into a single word, as follows:

```
star wars, wars => starwars
```

The preceding example means that `star wars` and `wars` will be changed to `starwars` by the `synonym` filter.

Equivalent synonyms

In addition to explicit mapping, Elasticsearch allows us to use equivalent synonyms. For example, the following definition will make all the words exchangeable so that you can use any of them to match a document that has one of them in its contents:

```
star, wars, star wars, starwars
```

Expanding synonyms

A `synonym` filter allows us to use one additional property when it comes to the synonyms of the Apache Solr format—the `expand` property. When the `expand` property is set to `true` (by default, it is set to `false`), all synonyms will be expanded by Elasticsearch to all equivalent forms. For example, let's say we have the following filter configuration:

```
"filter" : {  
    "synonym" : {  
        "type" : "synonym",  
        "expand": false,  
        "synonyms" : [  
            "one, two, three"  
        ]  
    }  
}
```

Elasticsearch will map the preceding synonym definition to the following:

```
one, two, thee => one
```

This means that the words `one`, `two`, and `three` will be changed to `one`. However, if we set the `expand` property to `true`, the same synonym definition will be interpreted in the following way:

```
one, two, three => one, two, three
```

This basically means that each of the words from the left-hand side of the definition will be expanded to all the words on the right-hand side.

Using WordNet synonyms

If we want to use WordNet-structured synonyms (to learn more about WordNet, visit <http://wordnet.princeton.edu/>), we need to provide an additional property for our `synonym` filter. The property name is `format` and we should set its value to `wordnet` in order for Elasticsearch to understand that format.

Query- or index-time synonym expansion

As with all analyzers, one can wonder when we should use our `synonym` filter – during indexing, during querying, or maybe during indexing and querying. Of course, it depends on your needs; however, remember that using index-time synonyms requires data reindexing after each synonym change. That's because they need to be reapplied to all the documents. If we use only query-time synonyms, we can update the synonym lists and have them applied during the query.

Understanding the explain information

Compared to databases, using systems that are capable of performing full-text search can often be anything other than obvious. We can search in many fields simultaneously, and the data in the index can vary from the ones provided as the values of the document fields (because of the analysis process, synonyms, abbreviations, and others). It's even worse; by default, search engines sort data by relevance – a number that indicates how similar the document is to the query. The key here is *how similar*. As we already discussed, scoring takes many factors into account: how many searched words were found in the document, how frequent the word was, how many terms were present in the field, and so on. This seems complicated, and finding out why a document was found and why another document is *better* is not easy. Fortunately, Elasticsearch has some tools that can answer these questions, and we will look at them now.

Understanding field analysis

One of the common questions asked is why a given document was not found. In many cases, the problem lies in the mappings definition and the analysis process configuration. For debugging the analysis process, Elasticsearch provides a dedicated REST API endpoint, `_analyze`.

Let's start with looking at the information returned by Elasticsearch for the default analyzer. To do that, we will run the following command:

```
curl -XGET 'localhost:9200/_analyze?pretty' -d 'Crime and Punishment'
```

In response, we will get the following data:

```
{  
  "tokens" : [ {  
    "token" : "crime",  
    "start_offset" : 0,  
    "end_offset" : 5,  
    "type" : "<ALPHANUM>",  
    "position" : 1  
  }, {  
    "token" : "punishment",  
    "start_offset" : 10,  
    "end_offset" : 20,  
    "type" : "<ALPHANUM>",  
    "position" : 3  
  } ]  
}
```

As we can see, Elasticsearch divided the input phrase into two tokens. During processing, the *and* common word was omitted (because it belongs to the stop words list) and the other words were turned into lowercase. This shows us exactly what would be happening during the analysis process. We can also provide the name of the analyzer, for example, we can change the preceding command as follows:

```
curl -XGET 'localhost:9200/_analyze?analyzer=standard&pretty' -d  
'Crime and Punishment'
```

The preceding command will allow us to check how the standard analyzer analyzes the data (it will be a bit different from the response we've seen previously).

It is worth noting that there is another form of analysis API available—the one that allows us to provide tokenizers and filters. It is very handy when we want to experiment with the configuration before creating the target mappings. An example of such a call is as follows:

```
curl -XGET  
  'localhost:9200/library/_analyze?tokenizer=whitespace&  
  filters=lowercase,kstem&pretty' -d 'John Smith'
```

In the preceding example, we used the analyzer, which was built with the whitespace tokenizer and two filters, lowercase and kstem.

As we can see, an analysis API can be very useful for tracking down the bugs in the mapping configuration. It is also very useful for when we want to solve problems with queries and search relevance. It can show us how our analyzers work, what terms they produce, and what the attributes of those terms are. With such information, analyzing query problems will be easier to track down.

Explaining the query

In addition to looking at what happened during analysis, Elasticsearch allows us to explain how the score was calculated for a particular query and document. Let's look at the following example:

```
curl -XGET 'localhost:9200/library/book/1/_explain?pretty&q=quiet'
```

In the preceding call, we provided a specific document and a query to run. Using the `_explain` endpoint, we ask Elasticsearch for an explanation on how the document was matched by Elasticsearch (or not matched). For example, should the preceding document be found by the provided query? If it is, Elasticsearch will provide information on why the document was matched, along with details about how its score was calculated.

The result returned by Elasticsearch for the preceding command is as follows:

```
{
  "_index": "library",
  "_type": "book",
  "_id": "1",
  "matched": true,
  "explanation": {
    "value": 0.057534903,
    "description": "weight(_all:quiet in 0) [PerFieldSimilarity],\n      result of:",
    "details": [
      {
        "value": 0.057534903,
        "description": "fieldWeight in 0, product of:",
        "details": [
          {
            "value": 1.0,
            "description": "tf(freq=1.0), with freq of:",
            "details": [
              {
                "value": 1.0,
                "description": "termFreq=1.0"
              }
            ]
          }
        ]
      }
    ]
  }
}
```

```
        } ]
    }, {
        "value" : 0.30685282,
        "description" : "idf(docFreq=1, maxDocs=1)"
    }, {
        "value" : 0.1875,
        "description" : "fieldNorm(doc=0)"
    } ]
}
}
```

Looks complicated, and, well, it is complicated! What's even worse is that this is only a simple query! Elasticsearch, and more specifically, the Lucene library, shows the internal information about the scoring process. We will only scratch the surface and will explain the most important things about the preceding response.

The most important part is the total score calculated for a document (the `value` property of the `explanation` object). If it is equal to 0, the document didn't match the given query. Another important element is the `description` section that tells us which similarity was used. In our example, we were looking for the `quiet` term. It was found in the `_all` field. It is obvious because we searched in the default field, which is `_all` (you should remember this field from the *Extending your index structure with additional internal information* section in *Chapter 2, Indexing Your Data*).

The `details` section provides us with information about components and where we should seek explanation about why our document matches the query. When it comes to scoring, we have a single object present—a single component that was responsible for document score calculation. The `value` property is the score calculated by this component, and again we see the `description` and `details` section. As you can see in the `description` field, the final score is the product of (`fieldWeight` in 0, `product of`) all the scores calculated by each element in the inner `details` array ($1.0 * 0.30685282 * 0.1875$).

In the inner `details` array, we can see three objects. The first one shows information about the term frequency in the given field (which was 1 in our case). This means that the field contained only a single occurrence of the searched term. The second object shows the inverse document frequency. Note the `maxDocs` property, which is equal to 1. This means that only one document was found with the specified term. The third object is responsible for the field norm for that field.

Note that the preceding response will be different for each query. What's more, the more complicated the query will be, the more complicated the returned information will be.

Summary

In this chapter, we learned how Apache Lucene scoring works internally. We've also seen how to use the scripting capabilities of Elasticsearch and how to index and search documents in different languages. We've used different queries to alter the score of our documents and modify it so it fits our use case. We've learned about index-time boosting, what synonyms are, and how they can help us. Finally, we've seen how to check why a particular document was a part of the result set and how its score was calculated.

In the next chapter, we'll go beyond full-text searching. We'll see what aggregations are and how we can use them to analyze our data. We'll also see faceting, which also allows us to aggregate our data and bring meaning to it. We'll use suggesters to implement spellchecking and autocomplete, and we'll use prospective search to find out which documents match particular queries. We'll index binary files and use geospatial capabilities to search our data with the use of geographical data. Finally, we'll use the scroll API to efficiently fetch a large number of results and we'll see how to make Elasticsearch use a list of terms (a list that is loaded automatically) in a query.

6

Beyond Full-text Searching

In the previous chapter, we saw how Apache Lucene scoring works internally. We saw how to use the scripting capabilities of Elasticsearch and how to index and search documents in different languages. We learned how to use different queries in order to alter the score of our documents, and we used index-time boosting. We learned what synonyms are and finally, we saw how to check why a particular document was a part of the result set and how its score was calculated. By the end of this chapter, you will have learned the following topics:

- Using aggregations to aggregate our indexed data and calculate useful information from it
- Employing faceting to calculate different statistics from our data
- Implementing the spellchecking and autocomplete functionalities by using Elasticsearch suggesters
- Using prospective search to match documents against queries
- Indexing binary files
- Indexing and searching geographical data
- Efficiently fetching large datasets
- Automatically loading terms and using them in our query

Aggregations

Apart from the improvements and new features that Elasticsearch 1.0 brings, it also includes a highly anticipated framework, which moves Elasticsearch to a new position—a full-featured analysis engine. Now, you can use Elasticsearch as a key part of various systems that process massive volumes of data, allow you to extract conclusions, and visualize that data in a human-readable way. Let's see how this functionality works and what we can achieve by using it.

General query structure

To use aggregation, we need to add an additional section in our query. In general, our queries with aggregations will look like the following code snippet:

```
{  
    "query": { ... },  
    "aggs" : { ... }  
}
```

In the `aggs` property (you can use aggregations if you want; `aggs` is just an abbreviation), you can define any number of aggregations. One thing to remember though is that the key defines the name of the aggregation (you will need it to distinguish particular aggregations in the server response). Let's take our `library` index and create the first query that will use aggregations. A command to send such a query is as follows:

```
curl 'localhost:9200/_search?search_type=count&pretty' -d '{  
    "aggs": {  
        "years": {  
            "stats": {  
                "field": "year"  
            }  
        },  
        "words": {  
            "terms": {  
                "field": "copies"  
            }  
        }  
    }'  
}'
```

This query defines two aggregations. The aggregation named `years` shows the statistics for the `year` field. The `words` aggregation contains information about the terms used in a given field.



In our examples, we assumed that we do aggregation in addition to searching. If we don't need the documents that are found, a better idea is to use the `search_type=count` parameter. This omits some unnecessary work and is more efficient. In such a case, the endpoint should be `/library/_search?search_type=count`. You can read more about the search types in the *Understanding the querying process* section of *Chapter 3, Searching Your Data*.

Now let's look at the response returned by Elasticsearch for the preceding query:

```
{  
  "took": 2,  
  "timed_out": false,  
  "_shards": {  
    "total": 5,  
    "successful": 5,  
    "failed": 0  
  },  
  "hits": {  
    "total": 4,  
    "max_score": 0,  
    "hits": []  
  },  
  "aggregations": {  
    "words": {  
      "buckets": [  
        {  
          "key": 0,  
          "doc_count": 2  
        },  
        {  
          "key": 1,  
          "doc_count": 1  
        },  
        {  
          "key": 6,  
          "doc_count": 1  
        }  
      ]  
    },  
    "years": {  
      "count": 4,  
      "min": 1886,  
      "max": 1961,  
      "avg": 1928,  
      "sum": 7712  
    }  
  }  
}
```

As you can see, both the aggregations (`years` and `words`) were returned. The first aggregation we defined in our query (`years`) returned general statistics for the given field, which was gathered across all the documents that matched our query. The second aggregation (`words`) is a bit different. It created several sets called **buckets** that are calculated on the returned documents, and each of the aggregated values is present within one of these sets. As you can see, there are multiple aggregation types available and they return different results. We will see the differences later in this section.

Available aggregations

After the previous example, you shouldn't be surprised that aggregations are divided into groups. Currently, there are two groups – metric aggregations and bucketing aggregations.

Metric aggregations

Metric aggregations take an input document set and generate at least a single statistic. As you will see, these aggregations are mostly self-explanatory.

Min, max, sum, and avg aggregations

Usage of the `min`, `max`, `sum`, and `avg` aggregations is very similar. For the given field, they return a minimum value, a maximum value, a sum of all the values, and an average value, respectively. Any numeric field can be used as a source for these values. For example, to calculate the minimum value for the `year` field, we will construct the following aggregation:

```
{  
  "aggs": {  
    "min_year": {  
      "min": {  
        "field": "year"  
      }  
    }  
  }  
}
```

The returned result will be similar to the following one:

```
"min_year": {  
  "value": 1886  
}
```

Using scripts

The input values can also be generated by a script. For example, if we want to find a minimum value from all the values in the year field, but we also want to subtract 1000 from these values, we will send an aggregation similar to the following one:

```
{
  "aggs": {
    "min_year": {
      "min": {
        "script": "doc['year'].value - 1000"
      }
    }
  }
}
```

In this case, the value that the aggregations will use is the original year field value reduced by 1000. The other notation that we can use to achieve the same response is to provide the field name and the `script` property, as follows:

```
{
  "aggs": {
    "min_year": {
      "min": {
        "field": "year",
        "script": "_value - 1000"
      }
    }
  }
}
```

The field name is given outside the script. If we like, we can be even more verbose, as follows:

```
{
  "aggs": {
    "min_year": {
      "min": {
        "field": "year",
        "script": "_value - mod",
        "params": {
          "mod": 1000
        }
      }
    }
  }
}
```

```
        }
    }
}
```

As you can see, we've added the `params` section with additional parameters. You can read more about scripts in the *Scripting capabilities of Elasticsearch* section of *Chapter 5, Make Your Search Better*.

The `value_count` aggregation

The `value_count` aggregation is similar to the ones we described previously, but the input field doesn't have to be numeric. An example of this aggregation is as follows:

```
{
  "aggs": {
    "number_of_items": {
      "value_count": {
        "field": "characters"
      }
    }
  }
}
```

Let's stop here for a moment. It is a good opportunity to look at which values are counted by Elasticsearch aggregation in this case. If you run the preceding query on your index with `books` (the `library` index), the response will be something as follows:

```
"number_of_items": {
  "value": 31
}
```

Elasticsearch counted all the tokens from the `characters` field across all the documents. This number makes sense when we keep in mind that, for example, our `Sofia Semyonovna Marmeladova` term will become `sofia, semyonovna, and marmeladova` after analysis. In most of the cases, such a behavior is not what we are aiming at. For such cases, we should use a not-analyzed version of the `characters` field.

The `stats` and `extended_stats` aggregations

The `stats` and `extended_stats` aggregations can be treated as aggregations that return all the previously described aggregations but within a single aggregation object. For example, if we want to calculate statistics for the `year` field, we can use the following code:

```
{
  "aggs": {
```

```
        "stats_year": {
            "stats": {
                "field": "year"
            }
        }
    }
}
```

The relevant part of the results returned by Elasticsearch will be as follows:

```
"stats_year": {  
    "count": 4,  
    "min": 1886,  
    "max": 1961,  
    "avg": 1928,  
    "sum": 7712  
}
```

Of course, the `extended_stats` aggregation returns statistics that are even more extended. Let's look at the following query:

```
{  
  "aggs": {  
    "stats_year": {  
      "extended_stats": {  
        "field": "year"  
      }  
    }  
  }  
}
```

In the returned response, we will see the following output:

```
"stats_year": {  
    "count": 4,  
    "min": 1886,  
    "max": 1961,  
    "avg": 1928,  
    "sum": 7712,  
    "sum_of_squares": 14871654,  
    "variance": 729.5,  
    "std_deviation": 27.00925767213901  
}
```

As you can see, in addition to the already known values, we also got the sum of squares, variance, and the standard deviation statistics.

Bucketing

Bucketing aggregations return many subsets and qualify the input data to a particular subset called **bucket**. You can think of the bucketing aggregations as something similar to the former faceting functionality described in the *Faceting* section. However, the aggregations are more powerful and just easier to use. Let's go through the available bucketing aggregations.

The terms aggregation

The `terms` aggregation returns a single bucket for each term available in a field. This allows you to generate the statistics of the field value occurrences. For example, the following are the questions that can be answered by using this aggregation:

- How many books were published each year?
- How many books were available for borrowing?
- How many copies of the books do we have the most?

To get the answer for the last question, we can send the following query:

```
{  
  "aggs": {  
    "availability": {  
      "terms": {  
        "field": "copies"  
      }  
    }  
  }  
}
```

The response returned by Elasticsearch for our `library` index is as follows:

```
"availability": {  
  "buckets": [  
    {  
      "key": 0,  
      "doc_count": 2  
    },  
    {  
      "key": 1,  
      "doc_count": 1  
    }  
  ]  
}
```

```

        "doc_count": 1
    },
{
    "key": 6,
    "doc_count": 1
}
]
}

```

We see that we have two books without copies available (bucket with the `key` property equal to 0), one book with one copy (bucket with the `key` property equal to 1), and a single book with six copies (bucket with the `key` property equal to 6). By default, Elasticsearch returns the buckets sorted by the value of the `doc_count` property in descending order. We can change this by adding the `order` attribute. For example, to sort our aggregations by using the `key` property values, we will send the following query:

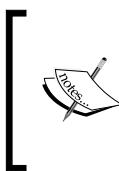
```

{
  "aggs": {
    "availability": {
      "terms": {
        "field": "copies",
        "size": 40,
        "order": { "_term": "asc" }
      }
    }
  }
}

```

We can sort in incremental order (`asc`) or in decremental order (`desc`). In our example, we sorted the values by using their `key` properties (`_term`). The other option available is `_count`, which tells Elasticsearch to sort by the `doc_count` property.

In the preceding example, we also added the `size` attribute. As you can guess, it defines how many buckets should be returned at the maximum.



You should remember that when the field is analyzed, you will get buckets from the analyzed terms as shown in the example with the value count. This probably is not what you want. The answer to such a problem is just to add an additional, not-analyzed version of your field to the index and to use it for the aggregation calculation.

The range aggregation

In the range aggregation, buckets are created using defined ranges. For example, if we want to check how many books were published in the given period of time, we can create the following query:

```
{  
    "aggs": {  
        "years": {  
            "range": {  
                "field": "year",  
                "ranges": [  
                    { "to" : 1850 },  
                    { "from": 1851, "to": 1900 },  
                    { "from": 1901, "to": 1950 },  
                    { "from": 1951, "to": 2000 },  
                    { "from": 2001 }  
                ]  
            }  
        }  
    }  
}
```

For the data in the library index, the response should look like the following output:

```
"years": {  
    "buckets": [  
        {  
            "to": 1850,  
            "doc_count": 0  
        },  
        {  
            "from": 1851,  
            "to": 1900,  
            "doc_count": 1  
        },  
        {  
            "from": 1901,  
            "to": 1950,  
            "doc_count": 2  
        },  
        {  
            "from": 1951,  
            "to": 2000,  
            "doc_count": 1  
        }  
    ]  
}
```

```

    "from": 1951,
    "to": 2000,
    "doc_count": 1
},
{
    "from": 2001,
    "doc_count": 0
}
]
}

```

For example, from the preceding output, we know that between 1901 and 1950, we released two books.

If you create the user interface, it is possible to automatically generate a label for every bucket. Turning on this feature is simple—we just need to add the `keyed` attribute and set it to `true`, just like in the following example:

```

{
  "aggs": {
    "years": {
      "range": {
        "field": "year",
        "keyed": true,
        "ranges": [
          { "to" : 1850 },
          { "from": 1851, "to": 1900 },
          { "from": 1901, "to": 1950 },
          { "from": 1951, "to": 2000 },
          { "from": 2001 }
        ]
      }
    }
  }
}

```

The highlighted part in the preceding code causes the results to contain labels, just as we can see in the following response returned by Elasticsearch:

```

"years": {
  "buckets": {
    "*-1850.0": {

```

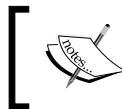
```
        "to": 1850,
        "doc_count": 0
    },
    "1851.0-1900.0": {
        "from": 1851,
        "to": 1900,
        "doc_count": 1
    },
    "1901.0-1950.0": {
        "from": 1901,
        "to": 1950,
        "doc_count": 2
    },
    "1951.0-2000.0": {
        "from": 1951,
        "to": 2000,
        "doc_count": 1
    },
    "2001.0-*": {
        "from": 2001,
        "doc_count": 0
    }
}
}
```

As you probably noticed, the structure is slightly changed – now, the `buckets` field is not a table but a map where the key is generated from the range. This works, but it is not so pretty. For our case, giving a name for every bucket will be more useful. Fortunately, it is possible and we can do this by adding the `key` attribute for every range and setting its value to the desired name. Consider the following example:

```
{
    "aggs": {
        "years": {
            "range": {
                "field": "year",
                "keyed": true,
                "ranges": [
                    { "key": "Before 18th century", "to": 1799 },
                    { "key": "18th century", "from": 1800, "to": 1899 },
                    { "key": "19th century", "from": 1900, "to": 1999 },
                    { "key": "After 19th century", "from": 2000 }
                ]
            }
        }
}
```

```

        }
    }
}
```



It is important and quite useful that ranges need not be disjoint. In such cases, Elasticsearch will properly count the document for multiple buckets.



The date_range aggregation

The `date_range` aggregation is similar to the previously discussed `range` aggregation, but it is designed for the fields that use date types. Although the `library` index documents have the years mentioned in them, the field is a number and not a date.

To test this, let's imagine that we want to extend our `library` index to support newspapers. To do this, we will create a new index called `library2` by using the following command:

```
curl -XPOST localhost:9200/_bulk --data-binary '{
  "index": {"_index": "library2", "_type": "book", "_id": "1"}
}
{ "title": "Fishing news", "published": "2010/12/03 10:00:00",
  "copies": 3, "available": true }

{ "index": {"_index": "library2", "_type": "book", "_id": "2"}}
{ "title": "Knitting magazine", "published": "2010/11/07 11:32:00",
  "copies": 1, "available": true }

{ "index": {"_index": "library2", "_type": "book", "_id": "3"}}
{ "title": "The guardian", "published": "2009/07/13 04:33:00",
  "copies": 0, "available": false }

{ "index": {"_index": "library2", "_type": "book", "_id": "4"}}
{ "title": "Hadoop World", "published": "2012/01/01 04:00:00",
  "copies": 6, "available": true }

'
```

In the `library2` index, we leave the mapping for Elasticsearch discovery mechanisms—this is sufficient in this case. Let's start with the first query using the `date_range` aggregation, which is as follows:

```
{
  "aggs": {
    "years": {
      "date_range": {
        "field": "published",
        "ranges": [
          { "to" : "2009/12/31" },

```

```
        { "from": "2010/01/01", "to": "2010/12/31" },
        { "from": "2011/01/01" }
    ]
}
}
}
}
```

Comparing with the ordinary range aggregation, the only thing that changed is the aggregation type (`date_range`). The dates can be passed in a string format recognized by Elasticsearch (refer to *Chapter 2, Indexing Your Data*, for more information) or as a number value – the number of milliseconds since 1970-01-01). The response returned by Elasticsearch is as follows:

```
"years": {
  "buckets": [
    {
      "to": 1262217600000,
      "to_as_string": "2009/12/31 00:00:00",
      "doc_count": 1
    },
    {
      "from": 1262304000000,
      "from_as_string": "2010/01/01 00:00:00",
      "to": 1293753600000,
      "to_as_string": "2010/12/31 00:00:00",
      "doc_count": 2
    },
    {
      "from": 1293840000000,
      "from_as_string": "2011/01/01 00:00:00",
      "doc_count": 1
    }
  ]
}
```

The only difference in the preceding response compared to the response given by the range aggregation is that the information about the range boundaries is split into two attributes. The attributes named `from` or `to` present the number of milliseconds from 1970-01-01. The properties `from_as_string` and `to_as_string` present the date in a human-readable form. Of course, the `keyed` and `key` attributes in the definition of the `date_range` aggregation work as already described.

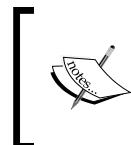
Elasticsearch also allows us to define the format of the presented dates by using the `format` attribute. In our example, we presented the dates with year resolution, so mentioning the day and time were unnecessary. If we want to show month names, we can send a query like the following:

```
{
  "aggs": {
    "years": {
      "date_range": {
        "field": "published",
        "format": "MMMM YYYY",
        "ranges": [
          { "to" : "2009/12/31" },
          { "from": "2010/01/01", "to": "2010/12/31" },
          { "from": "2011/01/01" }
        ]
      }
    }
  }
}
```

One of the returned ranges looks as follows:

```
{
  "from": 1262304000000,
  "from_as_string": "January 2010",
  "to": 1293753600000,
  "to_as_string": "December 2010",
  "doc_count": 2
}
```

Looks better, doesn't it?



The available formats that we can use in the `format` parameter are defined in the Joda Time library. The full list is available at <http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html>.

There is one more thing about the `date_range` aggregation. Sometimes, we may want to build an aggregation that can change with time. For example, we want to see how many newspapers were published in every quarter. This is possible without modifying our query. To do this, consider the following example:

```
{
  "aggs": {
```

```
"years": {
    "date_range": {
        "field": "published",
        "format": "dd-MM-YYYY",
        "ranges": [
            { "to" : "now-9M/M" },
            { "to" : "now-9M" },
            { "from": "now-6M/M", "to": "now-9M/M" },
            { "from": "now-3M/M" }
        ]
    }
}
```

The keys are the expressions such as now-9M. Elasticsearch does the math and generates the appropriate value. You can use y (year), M (month), w (week), d (day), h (hour), m (minute), and s (second). For example, the expression now+3d means three days from now. The /M expression in our example takes only the dates that have been rounded to months. Thanks to such notations, we count only full months. The second advantage is that the calculated date is more cache-friendly – without rounding off, the date changes every millisecond, which causes every cache based on the range to become irrelevant.

IPv4 range aggregation

The last form of the range aggregation is aggregation based on Internet addresses. It works on the fields defined with the ip type and allows you to define ranges given by the IP range in the CIDR notation (http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing). An example of the ip_range aggregation looks as follows:

```
{
    "aggs": {
        "access": {
            "ip_range": {
                "field": "ip",
                "ranges": [
                    { "from": "192.168.0.1", "to": "192.168.0.254" },
                    { "mask": "192.168.1.0/24" }
                ]
            }
        }
    }
}
```

The response to the preceding query can be as follows:

```

"access": {
  "buckets": [
    {
      "from": 3232235521,
      "from_as_string": "192.168.0.1",
      "to": 3232235774,
      "to_as_string": "192.168.0.254",
      "doc_count": 0
    },
    {
      "key": "192.168.1.0/24",
      "from": 3232235776,
      "from_as_string": "192.168.1.0",
      "to": 3232236032,
      "to_as_string": "192.168.2.0",
      "doc_count": 4
    }
  ]
}

```

Again, the `keyed` and `key` attributes here work just like in the `range` aggregation.

The missing aggregation

Let's get back to our library index and check how many entries have no original title defined (the `otitle` field). To do this, we will use the `missing` aggregation, which will be a good friend in this case. An example query will look as follows:

```

{
  "aggs": {
    "missing_original_title": {
      "missing": {
        "field": "otitle"
      }
    }
  }
}

```

The relevant response part looks as follows:

```
"missing_original_title": {  
    "doc_count": 2  
}
```

We have two documents without the `otitle` field.



The missing aggregation is aware of the fact that the mapping definition may have `null_value` defined and will need to count the documents independently from this definition.



Nested aggregation

In the *Using nested objects* section of *Chapter 4, Extending Your Index Structure*, we learned about nested documents. Let's use this data to look into the next type of aggregation—the nested aggregation. Let's create the simplest working query, which will look as follows:

```
{  
    "aggs": {  
        "variations": {  
            "nested": {  
                "path": "variation"  
            }  
        }  
    }  
}
```

The preceding query is similar in structure to any other aggregation. It contains a single parameter—`path`, which points to the nested document. In the response, we get a number, as shown in the following output:

```
"variations": {  
    "doc_count": 2  
}
```

The preceding response means that we have two nested documents in the index with the provided `variation` type.

The histogram aggregation

The `histogram` aggregation is an aggregation that defines the buckets. The simplest form of a query that uses this aggregation looks as follows:

```
{
  "aggs": {
    "years": {
      "histogram": {
        "field": "year",
        "interval": 100
      }
    }
  }
}
```

The new piece of information here is `interval`, which defines the length of every range that will be used to create a bucket. Because of this, in our example, buckets will be created for periods of 100 years. The aggregation part of the response to the preceding query that was sent to our `library` index is as follows:

```
"years": {
  "buckets": [
    {
      "key": 1800,
      "doc_count": 1
    },
    {
      "key": 1900,
      "doc_count": 3
    }
  ]
}
```

As in the `range` aggregation, `histogram` also allows us to use the `keyed` property. The other available option is `min_doc_count`, which allows us to control what is the minimal number of documents required to create a bucket. If we set the `min_doc_count` property to zero, Elasticsearch will also include the buckets with the document count of 0.

The date_histogram aggregation

As the date_range aggregation is a specialized form of the range aggregation, the date_histogram aggregation is an extension of the histogram aggregation that works on dates. So, again we will use our index with newspapers (it was called library2). An example of the query that uses the date_histogram aggregation looks as follows:

```
{  
  "aggs": {  
    "years": {  
      "date_histogram": {  
        "field" : "published",  
        "format" : "yyyy-MM-dd HH:mm",  
        "interval": "10d"  
      }  
    }  
  }  
}
```

We can spot one important difference to the interval property. It is now a string describing the time interval, which in our case is ten days. Of course, we can set it to anything we want—it uses the same suffixes that we discussed when talking about the formats in the date_range aggregation. It is worth mentioning that the number can be a float value; for example, 1.5m, which means every one and a half minutes. The format attribute is the same as in the date_range aggregation—thanks to this, Elasticsearch can add a human-readable date according to the defined format. Of course, the format attribute is not required, but it is useful.

In addition to this, similar to the other range aggregations, the keyed and min_doc_count attributes still work.

Time zones

Elasticsearch stores all the dates in the UTC time zone. You can define the time zone, which should be used for display purposes. There are two ways for date conversion; Elasticsearch can convert a date before assigning an element to the appropriate bucket or after the assignment is done. This leads to the situation where an element may be assigned to various buckets depending on the chosen way and the definition of the bucket. We have two attributes that define this behavior: pre_zone and post_zone. Also, there is a time_zone attribute that basically sets the pre_zone attribute value. There are three notations to set these attributes, which are as follows:

- We can set the hours offset; for example: pre_zone:-4 or time_zone:5

- We can use the time format; for example: `pre_zone: "-4:30"`
- We can use name of the time zone; for example: `time_zone: "Europe/Warsaw"`



Look at <http://joda-time.sourceforge.net/timezones.html> to see the available time zones.



The geo_distance aggregation

The next two aggregations are connected with maps and spatial search. We will talk about the geo types and queries later in this chapter, so feel free to skip these two topics now and return to them later.

Look at the following query:

```
{
  "aggs": {
    "neighborhood": {
      "geo_distance": {
        "field": "location",
        "origin": [-0.1275, 51.507222],
        "ranges": [
          { "to": 1200 },
          { "from": 1201 }
        ]
      }
    }
  }
}
```

You can see that this query is similar to the range aggregation. The preceding aggregation will calculate the number of cities that fall into two buckets: one bucket of cities within 1200 km, and the second bucket of cities further than 1200 km from the origin (in this case, the origin is London). The aggregation section of the response returned by Elasticsearch looks similar to the following:

```
"neighborhood": {
  "buckets": [
    {
      "key": "*-1200.0",
      "from": 0,
      "to": 1200,
```

```
        "doc_count": 1
    },
{
    "key": "1201.0-*",
    "from": 1201,
    "doc_count": 4
}
]
}
```

Of course, the keyed and key attributes work in the geo_distance aggregation as well.

Now, let's modify the preceding query to show the other possibilities of the geo_distance aggregation as follows:

```
{
  "aggs": {
    "neighborhood": {
      "geo_distance": {
        "field": "location",
        "origin": { "lon": -0.1275, "lat": 51.507222 },
        "unit": "m",
        "distance_type" : "plane",
        "ranges": [
          { "to": 1200 },
          { "from": 1201 }
        ]
      }
    }
  }
}
```

We have highlighted three things in the preceding query. The first change is about how we define the point of origin. We can specify the location in various forms, which is described more precisely later in the chapter about geo type.

The second change is the unit attribute. The possible values are km (the default), mi, in, yd, m, cm, and mm that define the units of the numbers used in ranges (kilometers, miles, inches, yards, meters, centimeters, and millimeters, respectively).

The last attribute—distance_type—specifies how Elasticsearch calculates the distance. The possible values are (from the fastest but least accurate to the slowest but the most accurate) plane, sloppy_arc (the default), and arc.

The geohash_grid aggregation

Now you know how to aggregate based on the distance from a given point. The second option is to organize areas as a grid and assign every location to an appropriate cell. For this purpose, the ideal solution is **Geohash** (<http://en.wikipedia.org/wiki/Geohash>), which encodes the location into a string.

The longer the string, the more accurate the description of a particular location will be. For example, one letter is sufficient to declare a box with about 5,000 x 5,000 km and five letters are enough to have the accuracy for about a 5 x 5 km square.

Let's look at the following query:

```
{
  "aggs": {
    "neighborhood": {
      "geohash_grid": {
        "field": "location",
        "precision": 5
      }
    }
  }
}
```

We define the geohash_grid aggregation with buckets that have a precision of the mentioned square of 5 x 5 km (the precision attribute describes the number of letters used in the geohash string object). The table with resolutions versus the length of geohash can be found at <http://www.elasticsearch.org/guide/en/elasticsearch/reference/master/search-aggregations-bucket-geohashgrid-aggregation.html>.

Of course, more accuracy usually means more pressure on the system because of the number of buckets. By default, Elasticsearch will not generate more than 10,000 buckets. You can increase this parameter using the size attribute, but in fact, you should decrease it when possible.

Nesting aggregations

This is a powerful feature that allows us to build complex queries. Let's start expanding an example with the nested aggregation. In the example we used for nested aggregation, we only had the possibility of working with the nested documents. But, look at the following example to know what happens when we add the nested aggregation:

```
{
  "aggs": {
    "variations": {
```

```
    "nested": {
      "path": "variation"
    },
    "aggs": {
      "sizes": {
        "terms": {
          "field": "variation.size"
        }
      }
    }
  }
}
```

As you can see, we've added another aggregation that was nested inside the top-level aggregation. The aggregation that has been nested is called `sizes`. The aggregation part of the result for the preceding query will look as follows:

```
"variations": {
  "doc_count": 2,
  "sizes": {
    "buckets": [
      {
        "key": "XL",
        "doc_count": 1
      },
      {
        "key": "XXL",
        "doc_count": 1
      }
    ]
  }
}
```

Perfect! Elasticsearch took the result from the parent aggregation and analyzed it using the `terms` aggregation. The aggregations can be nested even further—in theory, we can nest aggregations indefinitely. We can also have more aggregations on the same level.

Let's look at the following example:

```
{  
  "aggs": {  
    "years": {  
      "range": {  
        "field": "year",  
        "ranges": [  
          { "to" : 1850 },  
          { "from": 1851, "to": 1900 },  
          { "from": 1901, "to": 1950 },  
          { "from": 1951, "to": 2000 },  
          { "from": 2001 }  
        ]  
      },  
      "aggs": {  
        "statistics": {  
          "stats": {}  
        }  
      }  
    }  
  }  
}
```

You will probably see that the preceding example is similar to the one we used when discussing the range aggregation. However, now we added an additional aggregation, which adds statistics to every bucket. The output for one of these buckets will look as follows:

```
{  
  "from": 1851,  
  "to": 1900,  
  "doc_count": 1,  
  "statistics": {  
    "count": 1,  
    "min": 1886,  
    "max": 1886,  
    "avg": 1886,  
    "sum": 1886  
  }  
}
```

Note that in the `stats` aggregation, we omitted the information about the field that is used to calculate the statistics. Elasticsearch is smart enough to get this information from the context—in this case, the parent aggregation.

Bucket ordering and nested aggregations

Let's recall the example of the `terms` aggregation and ordering. We said that sorting is available on bucket keys or document count. This is only partially true. Elasticsearch can also use values from nested aggregations! Let's start with the following query example:

```
{  
    "aggs": {  
        "availability": {  
            "terms": {  
                "field": "copies",  
                "order": { "numbers.avg": "desc" }  
            },  
            "aggs": {  
                "numbers": { "stats" : {} }  
            }  
        }  
    }  
}
```

In the preceding example, the order in the availability aggregation is based on the average value from the `numbers` aggregation. In this case, the `numbers.avg` notation is required because `stats` is a multivalued aggregation. If it was the `sum` aggregation, the name of the aggregation would be sufficient.

Global and subsets

All of our examples have one thing in common—the aggregations take into consideration the data from the whole index. The aggregation framework allows us to operate on the results filtered to the documents returned by the query or to do the opposite—ignore the query completely. You can also mix both the approaches. Let's analyze the following example:

```
{  
    "query": {  
        "filtered": {  
            "query": {  
                "bool": {  
                    "must": [
```

```

        "match_all": {}
    },
    "filter": {
        "term": {
            "available": "true"
        }
    }
},
"aggs": {
    "with_global": {
        "global": {},
        "aggs": {
            "copies": {
                "value_count": {
                    "field": "copies"
                }
            }
        }
    },
    "without_global": {
        "value_count": {
            "field": "copies"
        }
    }
}
}

```

The first part is a query. In this case, we want to return all the books that are currently available. In the next part, we can see aggregations. They are named `with_global` and `without_global`. Both these aggregations are similar; they use the `value_count` aggregation on the `copies` field. The difference is that the `with_global` aggregation is nested in the `global` aggregation. This is something new—the `global` aggregation creates one bucket holding all the documents in the current search scope (this means all the indices and types we've used for searching), but ignores the defined queries. In other words, `global` aggregates all the documents, while `without_global` will make the aggregation work only on the documents returned by the query.

The `aggregations` section of the response to the preceding query looks as follows:

```

"aggregations": {
    "without_global": { .

```

```
        "value": 2
    },
    "with_global": {
        "doc_count": 4,
        "copies": {
            "value": 4
        }
    }
}
```

In our index, we have two documents that match the query (books that are available now). The `without_global` aggregation did an aggregation on these documents, which gave a value equal to both the documents. The `with_global` aggregation ignores the search operation and operates on each document in the index, which means on all the four books.

Now, let's look at how to have a few aggregations and how one of these aggregations operates on a subset of a document. To do this, we can use a filter with aggregation, which will create one bucket containing the documents narrowed down for a given filter. Let's look at the following example:

```
{
    "aggs": {
        "with_filter": {
            "filter": {
                "term": {
                    "available": "true"
                }
            },
            "aggs": {
                "copies": {
                    "value_count": {
                        "field": "copies"
                    }
                }
            }
        }
    }
},
```

```

    "without_filter": {
      "value_count": {
        "field": "copies"
      }
    }
  }
}

```

We have no query to narrow down the number of documents that are passed to the aggregation, but we've included a filter that will narrow down the number of documents on which the aggregation will be calculated. The effect is the same as we've previously shown.

Inclusions and exclusions

The `terms` aggregation has one additional possibility of narrowing the number of aggregations—the `include/exclude` feature can be applied to string values. Let's look at the following query:

```

{
  "aggs": {
    "availability": {
      "terms": {
        "field": "characters",
        "exclude": "al.*",
        "include": "a.*"
      }
    }
  }
}

```

The preceding query operates on a regular expression. It excludes all the terms starting with `al` from the aggregation calculation, but includes all the terms that start with `a`. The effect of such a query is that only the terms starting with the letter `a` will be counted, excluding the ones that have the `l` letter as the second letter in the word. The regular expressions are defined according to the JAVA API (<http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>) and Elasticsearch also allows you to define the `flags` attribute as defined in this specification.

Faceting

Elasticsearch is a full-text search engine that aims to provide search results on the basis of our queries. However, sometimes we would like to get more—for example, we would like to get aggregated data that is calculated on the result set we get, such as the number of documents with a price between 100 and 200 dollars or the most common tags in the result documents. In the *Aggregations* section of this chapter, we talked about the aggregations framework. In addition to this, Elasticsearch provides a faceting module that is responsible for providing the functionality we've mentioned. In this chapter, we will discuss different faceting methods provided by Elasticsearch.



Note that faceting offers a subset of functionality provided by the aggregation module. Because of this, Elasticsearch creators would like all the users to migrate from faceting to the mentioned aggregation module. Faceting is not deprecated and you can use it, but beware that sometime in the future, it may be removed from Elasticsearch.



The document structure

For the purpose of discussing faceting, we'll use a very simple index structure for our documents. It will contain the identifier of the document, document date, a multivalued field that can hold words describing our document (the `tags` field), and a field holding numeric information (the `total` field). Our mappings could look as follows:

```
{  
  "mappings" : {  
    "doc" : {  
      "properties" : {  
        "id" : { "type" : "long", "store" : "yes" },  
        "date" : { "type" : "date", "store" : "no" },  
        "tags" : { "type" : "string", "store" : "no", "index" :  
          "not_analyzed" },  
        "total" : { "type" : "long", "store" : "no" }  
      }  
    }  
  }  
}
```



Keep in mind that when dealing with the string fields, you should avoid doing faceting on the analyzed fields. Such results may not be human-readable, especially when using stemming or any other heavy processing analyzers or filters.

Returned results

Before we get into how to run queries with faceting, let's take a look at what to expect from Elasticsearch in the result from a faceting request. In most of the cases, you'll only be interested in the data specific to the faceting type. However, in most faceting types, in addition to information specific to a given faceting type, you'll get the following information also:

- `_type`: This defines the faceting type used. This will be provided for each faceting type.
- `missing`: This defines the number of documents that didn't have enough data (for example, the missing field) to calculate faceting.
- `total`: This defines the number of tokens present in the facet calculation.
- `other`: This defines the number of facet values (for example, terms used in the `terms` faceting) that are not included in the returned counts.

In addition to this information, you'll get an array of calculated facets, such as `count`, for your terms, queries, or spatial distances. For example, the following code snippet shows how the usual faceting results look:

```
{
  (...)

  "facets" : {
    "tags" : {
      "_type" : "terms",
      "missing" : 54715,
      "total" : 151266,
      "other" : 143140,
      "terms" : [ {
        "term" : "test",
        "count" : 151266
      }
    }
  }
}
```

```
        "count" : 1119
    },
    {
        "term" : "personal",
        "count" : 1063
    },
    ...
]
```

As you can see in the results, faceting was run against the `tags` field. We've got a total number of 151266 tokens processed by the faceting module and the 143140 tokens that were not included in the results. We also have 54715 documents that didn't have the value in the `tags` field. The `test` term appeared in 1119 documents, and the `personal` term appeared in 1063 documents. This is what you can expect from the faceting response.

Using queries for facetting calculations

Query is one of the simplest faceting types, which allows us to get the number of documents that match the query in the faceting results. The query itself can be expressed using the Elasticsearch query language, which we have already discussed. Of course, we can include multiple queries to get multiple counts in the faceting results. For example, facetting that will return the number of documents for a simple `term` query can look like the following code:

```
{
    "query" : { "match_all" : {} },
    "facets" : {
        "my_query_facet" : {
            "query" : {
                "term" : { "tags" : "personal" }
            }
        }
    }
}
```

As you can see, we've included the `query` type facetting with a simple `term` query. An example response for the preceding query could look as follows:

```
{
  (...)

  "facets" : {
    "my_query_facet" : {
      "_type" : "query",
      "count" : 1081
    }
  }
}
```

As you can see in the response, we've got the facetting type and the count of the documents that matched the facet query, and of course, the main query results that we omitted in the preceding response.

Using filters for facetting calculations

In addition to using queries, Elasticsearch allows us to use filters for facetting calculations. It is very similar to query facetting, but instead of queries, filters are used. The filter itself can be expressed using the Elasticsearch query DSL, and of course, multiple filter facets can be used in a single request. For example, the facetting that will return the number of documents for a simple `term` filter can look as follows:

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "my_filter_facet" : {
      "filter" : {
        "term" : { "tags" : "personal" }
      }
    }
  }
}
```

As you can see, we've included the `filter` type facetting with a simple `term` filter. When talking about performance, the filter facets are faster than the query facets or the filter facets that wrap queries.

An example response for the preceding query will look as follows:

```
{  
  (...)  
  "facets" : {  
    "my_filter_facet" : {  
      "_type" : "filter",  
      "count" : 1081  
    }  
  }  
}
```

As you can see in the response, we've got the facetting type and the count of the documents that matched the facet filter and the main query.

Terms facetting

Terms facetting allows us to specify a field that Elasticsearch will use and will return the top-most frequent terms. For example, if we want to calculate the most frequent terms for the `tags` field, we can run the following query:

```
{  
  "query" : { "match_all" : {} },  
  "facets" : {  
    "tags_facet_result" : {  
      "terms" : {  
        "field" : "tags"  
      }  
    }  
  }  
}
```

The following facetting response will be returned by Elasticsearch for the preceding query:

```
{  
  (...)  
  "facets" : {  
    "tags_facet_result" : {  
      "_type" : "terms",  
      "missing" : 54716,  
      "total" : 151266,  
      "terms" : [  
        {"term": "tag1", "count": 100},  
        {"term": "tag2", "count": 80},  
        {"term": "tag3", "count": 70},  
        {"term": "tag4", "count": 60},  
        {"term": "tag5", "count": 50},  
        {"term": "tag6", "count": 40},  
        {"term": "tag7", "count": 30},  
        {"term": "tag8", "count": 20},  
        {"term": "tag9", "count": 15},  
        {"term": "tag10", "count": 10},  
        {"term": "tag11", "count": 8},  
        {"term": "tag12", "count": 6},  
        {"term": "tag13", "count": 5},  
        {"term": "tag14", "count": 4},  
        {"term": "tag15", "count": 3},  
        {"term": "tag16", "count": 2},  
        {"term": "tag17", "count": 1},  
        {"term": "tag18", "count": 1},  
        {"term": "tag19", "count": 1},  
        {"term": "tag20", "count": 1}  
      ]  
    }  
  }  
}
```

```
"other" : 143140,
"terms" : [ {
    "term" : "test",
    "count" : 1119
}, {
    "term" : "personal",
    "count" : 1063
}, {
    "term" : "feel",
    "count" : 982
}, {
    "term" : "hot",
    "count" : 923
},
(...)

}
}
}
```

As you can see, our `terms` faceting results were returned in the `tags_facet_result` section and we've got the information that was already described.

There are a few additional parameters that we can use for the `terms` faceting, which are as follows:

- `size`: This parameter specifies how many of the top-most frequent terms should be returned at the maximum. The documents with the subsequent terms will be included in the count of the `other` field in the result.
- `shard_size`: This parameter specifies how many results per shard will be fetched by the node running the query. It allows you to increase the terms facetting accuracy in situations where the number of unique terms for a field is greater than the `size` parameter value. In general, the higher the `size` parameter, the more accurate are the results, but the more expensive is the calculation and more data is returned to the client. In order to avoid returning a long results list, we can set the `shard_size` value to a value higher than the value of the `size` parameter. This will tell Elasticsearch to use it to calculate the terms facets but still return a maximum of the `size` top terms. Please remember that the `shard_size` parameter cannot be set to a value lower than the `size` parameter.

- `order`: This parameter specifies the order of the facets. The possible values are `count` (by default this is ordered by frequency, starting from the most frequent), `term` (in ascending alphabetical order), `reverse_count` (ordered by frequency, starting from the less frequent), and `reverse_term` (in descending alphabetical order).
- `all_terms`: This parameter, when set to `true`, will return all the terms in the result, even those that don't match any of the documents. It can be demanding in terms of performance, especially on the fields with a large number of terms.
- `exclude`: This specifies the array of terms that should be excluded from the facet calculation.
- `regex`: This parameter specifies the regex expression that will control which terms should be included in the calculation.
- `script`: This parameter specifies the script that will be used to process the terms used in the facet calculation.
- `fields`: This parameter specifies the array that allows us to specify multiple fields for facet calculation (should be used instead of the `field` property). Elasticsearch will return aggregation across multiple fields. This property can also include a special value called `_index`. If such a value is present, the calculated counts will be returned per index, so we are able to distinguish the faceting calculations coming from multiple indices (if our query is run against multiple indices).
- `_script_field`: This defines the script that will provide the actual term for the calculation. For example, a `_source` field based terms may be used.

Ranges based facetting

Ranges based facetting allows us to get the number of documents for a defined set of ranges and in addition to this, allows us to get data aggregated for the specified field. For example, if we want to get the number of documents that have the `total` field values that fall into the ranges (lower bound inclusive and upper exclusive) to 90, from 90 to 180, and from 180, we will send the following query:

```
{  
  "query" : { "match_all" : {} },  
  "facets" : {  
    "ranges_facet_result" : {  
      "range" : {  
        "field" : "total",  
        "ranges" : [  
          { "to" : 90 },  
          { "from" : 90, "to" : 180 },  
          { "from" : 180 }  
        ]  
      }  
    }  
  }  
}
```

```
        { "from" : 90, "to" : 180 },
        { "from" : 180 }
    ]
}
}
}
}
```

As you can see in the preceding query, we've defined the name of the field by using the `field` property and the array of ranges using the `ranges` property. Each range can be defined by using the `to` or `from` properties or by using both at the same time.

The response for the preceding query can look like the following output:

```
{
(...)

"facets" : {
    "ranges_facet_result" : {
        "_type" : "range",
        "ranges" : [ {
            "to" : 90.0,
            "count" : 18210,
            "min" : 0.0,
            "max" : 89.0,
            "total_count" : 18210,
            "total" : 39848.0,
            "mean" : 2.1882482152663374
        }, {
            "from" : 90.0,
            "to" : 180.0,
            "count" : 159,
            "min" : 90.0,
            "max" : 178.0,
            "total_count" : 159,
            "total" : 19897.0,
            "mean" : 125.13836477987421
        }, {
            "from" : 180.0,
            "count" : 274,
```

```
        "min" : 182.0,
        "max" : 57676.0,
        "total_count" : 274,
        "total" : 585961.0,
        "mean" : 2138.543795620438
    } ]
}
}
}
```

As you can see, because we've defined three ranges in our query for the range faceting, we've got three ranges in response. For each range, the following statistics were returned:

- `from`: This defines the left boundary of the range (if present in the query)
- `to`: This defines the right boundary of the range (if present in the query)
- `min`: This defines the minimal field value for the field used for facetting in the given range
- `max`: This defines the maximum field value for the field used for facetting in the given range
- `count`: This defines the number of documents with the value of the defined field that falls into the specified range
- `total_count`: This defines the total number of values in the defined field that fall into the specified range (should be the same as `count` for single valued fields and can be different for fields with multiple values)
- `total`: This defines the sum of all the values in the defined field that fall into the specified range
- `mean`: This defines the mean value calculated for the values in the given field used for a range facetting calculation that fall into the specified range

Choosing different fields for an aggregated data calculation

If we would like to calculate the aggregated data statistics for a different field than the one for which we calculate the ranges, we can use two properties: `key_field` and `key_value` (or `key_script` and `value_script` which allow script usage). The `key_field` property specifies which field value should be used to check whether the value falls into a given range, and the `value_field` property specifies which field value should be used for the aggregation calculation.

Numerical and date histogram faceting

A histogram facetting allows you to build a histogram of the values across the intervals of the field value (numerical- and date-based fields). For example, if we want to see how many documents fall into the intervals of 1000 in our `total` field, we will run the following query:

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "total_histogram" : {
      "histogram" : {
        "field" : "total",
        "interval" : 1000
      }
    }
  }
}
```

As you can see, we've used the `histogram` facet type and in addition to the `field` property, we've included the `interval` property, which defines the interval we want to use. The example of the response for the preceding query can look like the following output:

```
{
  (...)

  "facets" : {
    "total_histogram" : {
      "_type" : "histogram",
      "entries" : [ {
        "key" : 0,
        "count" : 18565
      }, {
        "key" : 1000,
        "count" : 33
      }, {
        "key" : 2000,
        "count" : 14
      }, {
        "key" : 3000,
        "count" : 5
      }
    }
  }
}
```

```
        },
        (...),
        [
    }
}
}
```

You can see that we have 18565 documents for the first bracket of 0 to 1000, 33 documents for the second bracket of 1000 to 2000, and so on.

The date_histogram facet

In addition to the histogram facets type that can be used on numerical fields, Elasticsearch allows us to use the date_histogram faceting type, which can be used on the date-based fields. The date_histogram facet type allows us to use constants such as year, month, week, day, hour, or minute as the value of the interval property. For example, one can send the following query:

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "date_histogram_test" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "day"
      }
    }
  }
}
```

[ In both the numerical and date_histogram faceting, we can use the key_field, key_value, key_script, and value_script properties that we discussed when talking about the terms facetting earlier in this chapter.]

Computing numerical field statistical data

The statistical facetting allows us to compute the statistical data for a numeric field type. In return, we get the count, total, sum of squares, average, minimum, maximum, variance, and standard deviation statistics. For example, if we want to compute the statistics for our total field, we will run the following query:

```
{
  "query" : { "match_all" : {} },
```

```
"facets" : {  
    "statistical_test" : {  
        "statistical" : {  
            "field" : "total"  
        }  
    }  
}
```

And, in the results, we will get the following output:

```
{  
    (...)  
    "facets" : {  
        "statistical_test" : {  
            "_type" : "statistical",  
            "count" : 18643,  
            "total" : 645706.0,  
            "min" : 0.0,  
            "max" : 57676.0,  
            "mean" : 34.63530547658639,  
            "sum_of_squares" : 1.2490405256E10,  
            "variance" : 668778.6853747752,  
            "std_deviation" : 817.7889002516329  
        }  
    }  
}
```

The following are the statistics returned in the preceding output:

- `_type`: This defines the facetting type
- `count`: This defines the number of documents with the value in the defined field
- `total`: This defines the sum of all the values in the defined field
- `min`: This defines the minimal field value
- `max`: This defines the maximum field value
- `mean`: This defines the mean value calculated for the values in the specified field

- `sum_of_squares`: This defines the sum of squares calculated for the values in the specified field
- `variance`: This defines the variance value calculated for the values in the specified field
- `std_deviation`: This defines the standard deviation value calculated for the values in the specified field



Note that we are also allowed to use the `script` and `fields` properties in the `statistical` faceting just like in the `terms` faceting.



Computing statistical data for terms

In addition to the `terms` and `statistical` facetting, Elasticsearch allows us to use the `terms_stats` facetting. It combines both the `statistical` and `terms` faceting types as it provides us with the ability to compute statistics on a field for the values that we get from another field. For example, if we want the facetting for the `total` field but want to divide those values on the basis of the `tags` field, we will run the following query:

```
{  
  "query" : { "match_all" : {} },  
  "facets" : {  
    "total_tags_terms_stats" : {  
      "terms_stats" : {  
        "key_field" : "tags",  
        "value_field" : "total"  
      }  
    }  
  }  
}
```

We've specified the `key_field` property, which holds the name of the field that provides the terms, and the `value_field` property, which holds the name of the field with numerical data values. The following is a portion of the results we get from Elasticsearch:

```
{  
  (...)
```

```

"facets" : {
    "total_tags_terms_stats" : {
        "_type" : "terms_stats",
        "missing" : 54715,
        "terms" : [ {
            "term" : "personal",
            "count" : 1063,
            "total_count" : 254,
            "min" : 0.0,
            "max" : 322.0,
            "total" : 707.0,
            "mean" : 2.783464566929134
        }, {
            "term" : "me",
            "count" : 715,
            "total_count" : 218,
            "min" : 0.0,
            "max" : 138.0,
            "total" : 710.0,
            "mean" : 3.256880733944954
        }
    }
}
}

```

As you can see, the faceting results were divided on a per term basis. Note that the same set of statistics was returned for each term as the ones that were returned for the ranges facetting (to know what these values mean, refer to the *Ranges based facetting* section of the *Faceting* topic in this chapter). This is because we've used a numerical field (the `total` field) to calculate the facet values for each field.

Geographical faceting

The last faceting calculation type we would like to discuss is `geo_distance` faceting. It allows us to get information about the numbers of documents that fall into distance ranges from a given location. For example, let's assume that we have a `location` field in our documents in the index that stores geographical points. Now imagine that we want to get information about the document's distance from a given point, for example, from `10.0,10.0`. Let's assume that we want to know how many documents fall into the bracket of 10 kilometers from this point, how many fall into the bracket of 10 to 100 kilometers, and how many fall into the bracket of more than 100 kilometers. In order to do this, we will run the following query (you'll learn how to define the `location` field in the `Geo` section of this chapter):

```
{  
    "query" : { "match_all" : {} },  
    "facets" : {  
        "spatial_test" : {  
            "geo_distance" : {  
                "location" : {  
                    "lat" : 10.0,  
                    "lon" : 10.0  
                },  
                "ranges" : [  
                    { "to" : 10 },  
                    { "from" : 10, "to" : 100 },  
                    { "from" : 100 }  
                ]  
            }  
        }  
    }  
}
```

In the preceding query, we've defined the latitude (the `lat` property) and the longitude (the `lon` property) of the point from which we want to calculate the distance. One thing to notice is the name of the object that we pass in the `lat` and `lon` properties. The name of the object needs to be the same as the field holding the location information. The second thing is the `ranges` array that specifies the brackets—each range can be defined using the `to` or `from` properties or using both at the same time.

In addition to the preceding properties, we are also allowed to set the `unit` property (by default, `km` for distance in kilometers and `mi` for distance in miles) and the `distance_type` property (by default, `arc` for better precision and `plane` for faster execution).

Filtering faceting results

The filters that you include in your queries don't narrow down the faceting results, so the calculation is done on the documents that match your query. However, you may include the filters you want in your faceting definition. Basically, any filter we discussed in the *Filtering your results* section of *Chapter 3, Searching Your Data*, can be used with faceting – what you just need to do is include an additional section under the facet name.

For example, if we want our query to match all the documents and have facets calculated for the multivalued `tags` field but only for the documents that have the `fashion` term in the `tags` field, we can run the following query:

```
{
  "query" : { "match_all" : {} },
  "facets" : {
    "tags" : {
      "terms" : { "field" : "tags" },
      "facet_filter" : {
        "term" : { "tags" : "fashion" }
      }
    }
  }
}
```

As you can see, there is an additional `facet_filter` section on the same level as the type of facet calculation (which is `terms` in the preceding query). You just need to remember that the `facet_filter` section is constructed with the same logic as any filter described in *Chapter 2, Indexing Your Data*.

Memory considerations

Faceting can be memory demanding, especially with the large amounts of data in the indices and many distinct values. The demand for memory is high because Elasticsearch needs to load the data into the field data cache in order to calculate the faceting values. With the introduction of the doc values, which we talked about in the *Mappings configuration* section of *Chapter 2, Indexing Your Data*, Elasticsearch is able to use this data structure for all the operations that use the field data cache, such as faceting and sorting. In case of large amounts of data, it is a good idea to use doc values. The older methods also work, such as lowering the cardinality of your fields by using less precise dates, not-analyzed string fields, or types such as `short`, `integer`, or `float` instead of `long` and `double` when possible. If this doesn't help, you may need to give Elasticsearch more heap memory or even add more servers and divide your index to more shards.

Using suggesters

Starting from Elasticsearch 0.90, we've got the ability to use the so-called **suggesters**. We can define a suggester as a functionality that allows us to correct a user's spelling mistakes and build an autocomplete functionality, keeping the performance in mind. This section will introduce the world of suggesters to you; however, it is not a comprehensive guide. Describing all the details about suggesters will be very broad and is out of the scope of this book. If you want to learn more about suggesters, please refer to the official Elasticsearch documentation (<http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/search-suggesters.html>) or to our book, *Mastering ElasticSearch*, Packt Publishing.

Available suggester types

Elasticsearch gives us three types of suggesters that we can use, which are as follows:

- **term**: This defines the suggester that returns corrections for each word passed to it. It is useful for suggestions that are not phrases, such as single term queries.
- **phrase**: This defines the suggester that works on phrases, returning a proper phrase.
- **completion**: This defines the suggester designed to provide fast and efficient autocomplete results.

We will discuss each suggester separately. In addition to this, we can also use the `_suggest` REST endpoint.

Including suggestions

Now, let's try getting suggestions along with the query results. For example, let's use a `match_all` query and try getting a suggestion for a `serlock holnes` phrase, which has two incorrectly spelled terms. To do this, we will run the following command:

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{  
  "query" : {  
    "match_all" : {}  
  },  
  "suggest" : {  
    "first_suggestion" : {  
      "text" : "serlock holnes",  
      "term" : {
```

```
        "field" : "_all"
    }
}
}
}'
```

If we want to get multiple suggestions for the same text, we can embed our suggestions in the suggest object and place the text property as the suggest object option. For example, if we want to get suggestions for the `serlock holnes` text for the `title` and `_all` fields, we can run the following command:

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "match_all" : {}
  },
  "suggest" : {
    "text" : "serlock holnes",
    "first_suggestion" : {
      "term" : {
        "field" : "_all"
      }
    },
    "second_suggestion" : {
      "term" : {
        "field" : "title"
      }
    }
  }
}'
```

The suggester response

Now let's look at the response of the first query we've sent. As you can guess, the response will include both the query results and the suggestions:

```
{
  "took" : 1,
  "timed_out" : false,
  ...
```

```
"hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [
        ...
    ],
    "suggest" : {
        "first_suggestion" : [ {
            "text" : "serlock",
            "offset" : 0,
            "length" : 7,
            "options" : [ {
                "text" : "sherlock",
                "score" : 0.85714287,
                "freq" : 1
            } ]
        }, {
            "text" : "holnes",
            "offset" : 8,
            "length" : 6,
            "options" : [ {
                "text" : "holmes",
                "score" : 0.83333333,
                "freq" : 1
            } ]
        } ]
    }
}
```

We can see that we've got both search results and the suggestions (we've omitted the query response to make the example more readable) in the response.

The `term` suggester returned a list of possible suggestions for each term that are present in the `text` parameter. For each term, the `term` suggester will return an array of possible suggestions. Looking at the data returned for the `serlock` term, we can see the original word (the `text` parameter), its offset in the original `text` parameter (the `offset` parameter), and its length (the `length` parameter).

The `options` array contains suggestions for the given word and will be empty if Elasticsearch doesn't find any suggestions. Each entry in this array is a suggestion and is described by the following properties:

- `text`: This property defines the text of the suggestion.
- `score`: This property defines the suggestion score; the higher the score, the better the suggestion.
- `freq`: This property defines the frequency of the suggestion. Frequency represents how many times the word appears in the documents in the index against which we are running the suggestion query.

The term suggester

The `term` suggester works on the basis of the string edit distance. This means that the suggestion with fewer characters that need to be changed, added, or removed to make the suggestion look as the original word is the best one. For example, let's take the word `worl` and `work`. To change the `worl` term to `work`, we need to change the 1 letter to `k`, so it means a distance of 1. The text provided to the suggester is, of course, analyzed and then the terms are chosen to be suggested.

The term suggester configuration options

The common and mostly used `term` suggester options can be used for all the suggester implementations that are based on the `term` suggester. Currently, these are the phrase suggesters and of course, the base term suggesters.

The available options are as follows:

- `text`: This option defines the text for which we want to get the suggestions. This parameter is required for the suggester to work.
- `field`: This is another required parameter that we need to provide. The `field` parameter allows us to set the field for which the suggestions should be generated.

- **analyzer:** This defines the name of the analyzer, which should be used to analyze the text provided in the `text` parameter. If it is not set, Elasticsearch will use the analyzer used for the field provided by the `field` parameter.
- **size:** This option defaults to 5 and specifies the maximum number of suggestions that are allowed to be returned by each term provided in the `text` parameter.
- **sort:** This option allows us to specify how suggestions will be sorted in the result returned by Elasticsearch. By default, this option is set to `score` and tells Elasticsearch that the suggestions should be sorted by the suggestion score first, by the suggestion document frequency next, and finally by the term. The second possible value is `frequency`, which means that the results are first sorted by the document frequency, then by `score`, and finally by the term.

Additional term suggester options

In addition to the previously mentioned common `term` suggester options, Elasticsearch allows us to use additional ones that will only make sense to the `term` suggester itself. Some of these options are as follows:

- **lowercase_terms:** This option when set to `true` will tell Elasticsearch to lowercase all the terms that are produced from the `text` field after analysis.
- **max_edits:** This option defaults to 2 and specifies the maximum edit distance that the suggestion can have to be returned as a term suggestion. Elasticsearch allows us to set this value to 1 or 2.
- **prefix_len:** This option, by default, is set to 1. If we are struggling with suggester performance, increasing this value will improve the overall performance because a lower number of suggestions will need to be processed.
- **min_word_len:** This option defaults to 4 and specifies the minimum number of characters that a suggestion must have in order to be returned on the suggestions list.
- **shard_size:** This option defaults to the value specified by the `size` parameter and allows us to set the maximum number of suggestions that should be read from each shard. Setting this property to values higher than the `size` parameter can result in a more accurate document frequency at the cost of suggester performance degradation.

The phrase suggester

The `term` suggester provides a great way to correct a user's spelling mistakes on a per term basis, but it is not great for phrases. That's why the phrase suggester was introduced. It is built on top of the `term` suggester but adds an additional phrase calculation logic to it.

Let's start with the example of how to use the phrase suggester. This time we will omit the query section in our query. We can do this by running the following command:

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "suggest" : {
    "text" : "sherlock holnes",
    "our_suggestion" : {
      "phrase" : { "field" : "_all" }
    }
  }
}'
```

As you can see in the preceding command, it is almost the same as what we sent when using the `term` suggester. However, instead of specifying the `term` suggester type, we specified the `phrase` type. The response to the preceding command will be as follows:

```
{
  "took" : 1,
  ...
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [
      ...
    ]
  },
  "suggest" : {
    "our_suggestion" : [ {
      "text" : "sherlock holnes",
      "offset" : 0,
      "length" : 11
    } ]
  }
}
```

```
    "length" : 15,  
    "options" : [ {  
        "text" : "sherlock holmes",  
        "score" : 0.12227806  
    } ]  
} ]  
}  
}
```

As you can see, the response is very similar to the one returned by the `term` suggester, but instead of a single word being returned, it is already combined and returned as a phrase.

Configuration

Because the phrase suggester is based on the `term` suggester, it can also use some of the configuration options provided by the `term` suggester. The options are `text`, `size`, `analyzer`, and `shard_size`. In addition to the mentioned properties, the phrase suggester exposes additional options, which are as follows:

- `max_errors`: This option specifies the maximum number (or percentage) of terms that can be erroneous in order to correct them. The value of this property can either be an integer number such as `1` or a float value between `0` and `1`, which will be treated as a percentage value. By default, it is set to `1`, which means that at most, a single term can be misspelled in a given correction.
- `separator`: This option defaults to the whitespace character and specifies the separator that will be used to divide terms in the resulting bigram field.

The completion suggester

The completion suggester allows us to create the autocomplete functionality in a very performance effective way. This is because you can store complicated structures in the index instead of calculating them during query time.

To use a prefix-based suggester, we need to properly index our data with a dedicated field type called `completion`. To illustrate how to use this suggester, let's assume that we want to create an autocomplete feature that allows us to show the authors of the book. In addition to the author's name, we want to return the identifiers of the books that she/he has written. We start with creating the `authors` index by running the following command:

```
curl -XPOST 'localhost:9200/authors' -d '{  
    "mappings" : {
```

```
"author" : {
    "properties" : {
        "name" : { "type" : "string" },
        "ac" : {
            "type" : "completion",
            "index_analyzer" : "simple",
            "search_analyzer" : "simple",
            "payloads" : true
        }
    }
}
}'
```

Our index will contain a single type called `author`. Each document will have two fields—the `name` and the `ac` fields, which are the fields that will be used for autocomplete. We defined the `ac` field using the `completion` type. In addition to this, we used the `simple` analyzer for both index and query time. The last thing is the payload—the additional optional information that we will return along with the suggestion; in our case, it will be an array of book identifiers.

Indexing data

To index the data, we need to provide some additional information in addition to the ones we usually provide during indexing. Let's look at the following commands that index two documents describing the authors:

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
    "name" : "Fyodor Dostoevsky",
    "ac" : {
        "input" : [ "fyodor", "dostoevsky" ],
        "output" : "Fyodor Dostoevsky",
        "payload" : { "books" : [ "123456", "123457" ] }
    }
}'
curl -XPOST 'localhost:9200/authors/author/2' -d '{
    "name" : "Joseph Conrad",
    "ac" : {
        "input" : [ "joseph", "conrad" ],
        "output" : "Joseph Conrad",
        "payload" : { "books" : [ "123458", "123459" ] }
    }
}'
```

```
        "output" : "Joseph Conrad",
        "payload" : { "books" : [ "121211" ] }
    }
}'
```

Notice the structure of the data for the ac field. We provide the `input`, `output`, and `payload` properties. The optional `payload` property is used to provide additional information that will be returned. The `input` property is used to provide input information that will be used to build the completion used by the suggester. It will be used for user input matching. The optional `output` property is used to tell suggester which data should be returned for the document.

We can also omit the additional parameters section and index data in a way that we are used to just like in the following example:

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
  "name" : "Fyodor Dostoevsky",
  "ac" : "Fyodor Dostoevsky"
}'
```

However, because the completion suggester uses FST under the hood, we wouldn't be able to find the preceding document if we start with the second part of the ac field. That's why we think that indexing the data in a way we showed first is more convenient because we can explicitly control what we want to match and what we want to show as an output.

Querying the indexed completion suggester data

If we would like to find the documents that have author names starting with fyo, we would run the following command:

```
curl -XGET 'localhost:9200/authors/_suggest?pretty' -d '{
  "authorsAutocomplete" : {
    "text" : "fyo",
    "completion" : {
      "field" : "ac"
    }
  }
}'
```

Before we look at the results, let's discuss the query. As you can see, we've run the command to the `_suggest` endpoint because we don't want to run a standard query—we are just interested in the autocomplete results. The query is quite simple; we set its name to `authorsAutocomplete`, we set the text we want to get the completion for (the `text` property), and we add the `completion` object with configuration in it. The result of the preceding command will look as follows:

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 3,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 1.0, "payload" : {"books": ["123456", "123457"]}
    } ]
  } ]
}
```

As you can see in the response, we've got the document we were looking for along with the payload information.

We can also use fuzzy searches, which allow us to tolerate spelling mistakes. We can do this by including an additional `fuzzy` section in our query. For example, to enable a fuzzy matching in the `completion` suggester and to set the maximum edit distance to 2 (which means that a maximum of two errors are allowed), we will send the following query:

```
curl -XGET 'localhost:9200/authors/_suggest?pretty' -d '{
  "authorsAutocomplete" : {
    "text" : "fio",
    "completion" : {
```

```
    "field" : "ac",
    "fuzzy" : {
        "edit_distance" : 2
    }
}
}'
```

Although we've made a spelling mistake, we will still get the same results as we got before.

Custom weights

By default, the term frequency will be used to determine the weight of the document returned by the prefix suggester. However, this may not be the best solution. In such cases, it is useful to define the weight of the suggestion by specifying the `weight` property for the field defined as completion. The `weight` property should be set to an integer value. The higher the `weight` property value, the more important the suggestion. For example, if we want to specify a weight for the first document in our example, we will run the following command:

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
    "name" : "Fyodor Dostoevsky",
    "ac" : {
        "input" : [ "fyodor", "dostoevsky" ],
        "output" : "Fyodor Dostoevsky",
        "payload" : { "books" : [ "123456", "123457" ] },
        "weight" : 30
    }
}'
```

Now, if we run our example query, the results will be as follows:

```
{
    ...
    "authorsAutocomplete" : [ {
        "text" : "fyo",
        "offset" : 0,
        "length" : 3,
        "options" : [ {
```

```
"text" : "Fyodor Dostoevsky",
      "score" : 30.0, "payload" : {"books": ["123456", "123457"]}
    } ]
  } ]
}
```

Look at how the score of the result has changed. In our initial example, it was 1.0 and now it is 30.0. This is because we set the weight parameter to 30 during indexing.

Percolator

Have you ever wondered what would happen if we reverse the traditional model of using queries to find documents? Does it make sense to find documents matching the queries? It is not a surprise that there is a whole range of solutions where this model is very useful. Whenever you operate on an unbounded stream of input data, where you search for the occurrences of particular events, you can use this approach. This can be used for the detection of failures in a monitoring system or for the 'Tell me when a product with the defined criteria will be available in this shop' functionality. In this section, we will look at how an Elasticsearch percolator works and how it can handle this last example.

The index

In all the examples regarding a percolator, we will use an index called `notifier`, which we will create by using the following command:

```
curl -XPOST 'localhost:9200/notifier' -d '{
  "mappings": {
    "book" : {
      "properties" : {
        "available" : {
          "type" : "boolean"
        }
      }
    }
  }
}'
```

We defined only a single field; the rest of the fields will use the schema-less nature of Elasticsearch—their type will be guessed.

Percolator preparation

A percolator looks like an additional document type in Elasticsearch. This means that we can store any documents and also search them like an ordinary type in any index. However, percolator allows us to inverse the logic—index queries and send document to Elasticsearch to see which indexed queries it matched. Let's get the library example from *Chapter 2, Indexing Your Data*, and try to index this query in the percolator. We assume that our users need to be informed when any book matching the defined criteria is available.

Look at the following `query1.json` file that contains an example query generated by the user:

```
{  
  "query" : {  
    "bool" : {  
      "must" : {  
        "term" : {  
          "title" : "crime"  
        }  
      },  
      "should" : {  
        "range" : {  
          "year" : {  
            "gt" : 1900,  
            "lt" : 2000  
          }  
        }  
      },  
      "must_not" : {  
        "term" : {  
          "otitle" : "nothing"  
        }  
      }  
    }  
  }  
}
```

In addition to this, our users are allowed to define filters using our hypothetical user interface. To illustrate such a functionality, we've taken a user query. The query written into the `query2.json` file should find all the books written before 2010 that are currently available in our library. Such a query will look as follows:

```
{
  "query" : {
    "filtered": {
      "query" : {
        "range" : {
          "year" : {
            "lt" : 2010
          }
        }
      },
      "filter" : {
        "term" : {
          "available" : true
        }
      }
    }
  }
}
```

Now, let's register both the queries in the percolator (note that we are registering queries and haven't indexed any documents). In order to do this, we will run the following commands:

```
curl -XPUT 'localhost:9200/notifier/.percolator/1' -d @query1.json
curl -XPUT 'localhost:9200/notifier/.percolator/old_books' -d
@query2.json
```

In the preceding examples, we used two completely different identifiers. We did that in order to show that we can use an identifier that best describes the query. It is up to us under which name we would like the query to be registered.

We are now ready to use our percolator. Our application will provide documents to the percolator and check whether Elasticsearch finds corresponding queries. This is exactly what a percolator allows us to do—to reverse the search logic. Instead of indexing documents and running queries against them, we store queries and send documents. In return, Elasticsearch will show us which queries match the current document.

Let's use an example document that will match both the stored queries—it'll have the required title, release date, and will mention whether it is currently available. The command to send such a document can be as follows:

```
curl -XGET 'localhost:9200/notifier/book/_percolate?pretty' -d '{
  "doc" : {
    "title": "Crime and Punishment",
    "otitle": "Преступлénie и наказánie",
    "author": "Fyodor Dostoevsky",
    "year": 1886,
    "characters": ["Raskolnikov", "Sofia Semyonovna Marmeladova"],
    "tags": [],
    "copies": 0,
    "available" : true
  }
}'
```

As we expected, the Elasticsearch response will include the identifiers of the matching queries. Such a response will look as follows:

```
"matches" : [ {
  "_index" : "notifier",
  "_id" : "1"
}, {
  "_index" : "notifier",
  "_id" : "old_books"
} ]
```

This works like a charm. Please note the endpoint used in this query—we used the `_percolate` endpoint. The index name corresponds to the index where queries were stored, and the type is equal to the type defined in the mapping.

The response format contains information about the index and the query identifier. This information is included for the cases when we search against multiple indices at once. When using a single index, adding an additional query parameter, `percolate_format=ids`, will change the response as follows:

```
"matches" : [ "3" ].
```

Getting deeper

Because the queries registered in a percolator are in fact documents, we can use a normal query sent to Elasticsearch in order to choose which queries stored in the `.percolator` index should be used in the percolation process. This may sound weird, but it really gives a lot of possibilities. In our library, we can have several groups of users. Let's assume that some of them have permissions to borrow very rare books, or we can have several branches in the city, and the user can declare where he or she would like to go and get the book from.

Let's see how such use cases can be implemented by using the percolator. To do this, we will need to update our mapping. We will do that by adding the `.percolator` type using the following command:

```
curl -XPOST 'localhost:9200/notifier/.percolator/_mapping' -d '{
  ".percolator" : {
    "properties" : {
      "branches" : {
        "type" : "string",
        "index" : "not_analyzed"
      }
    }
  }
}'
```

Now, in order to register a query, we will use the following command:

```
curl -XPUT 'localhost:9200/notifier/.percolator/3' -d '{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "branches" : ["brA", "brB", "brD"]
}'
```

In the preceding example, the user is interested in any book with the `crime` term in the `title` field (the `term` query is responsible for this). He or she wants to borrow this book from one of the three listed branches. When specifying the mappings, we defined that the `branches` field is a not-analyzed string field. We can now include a query along with the document we've sent previously. Let's look at how to do this.

Our book system just got the book, and it is ready to report the book and check whether the book is of interest to someone. To check this, we send the document that describes the book and add an additional query to such a request—the query that will limit the users to only the ones interested in the `brB` branch. Such a request can look as follows:

```
curl -XGET 'localhost:9200/notifier/book/_percolate?pretty' -d '{
  "doc" : {
    "title": "Crime and Punishment",
    "otitle": "Преступл\u043e и наказ\u0430ние",
    "author": "Fyodor Dostoevsky",
    "year": 1886,
    "characters": ["Raskolnikov", "Sofia Semyonovna Marmeladova"],
    "tags": [],
    "copies": 0,
    "available" : true
  },
  "size" : 10,
  "query" : {
    "term" : {
      "branches" : "brB"
    }
  }
}'
```

If everything was executed correctly, the response returned by Elasticsearch should look as follows (we indexed our query with 3 as an identifier):

```
"total" : 1,
"matches" : [ {
  "_index" : "notifier",
  "_id" : "3"
} ]
```

There is one additional thing to note—the `size` parameter. It allows us to limit the number of matches returned. It is required for additional security—you should know what you do because the number of matched queries can be large, and this can mean memory-related issues.

Of course, if we are allowed to use queries along with the documents sent for percolation, why can we not use other Elasticsearch functionalities? Of course, this is possible. For example, the following document is sent along with an aggregation:

```
{
  "doc": {
    "title": "Crime and Punishment",
    "available": true
  },
  "aggs" : {
    "test" : {
      "terms" : {
        "field" : "branches"
      }
    }
  }
}
```

We can have queries, filters, faceting, and aggregations. What about highlighting? Please look at the following example document:

```
{
  "doc": {
    "title": "Crime and Punishment",
    "year": 1886,
    "available": true
  },
  "size" : 10,
  "highlight": {
    "fields": {
      "title": {}
    }
  }
}
```

As you can see, it contains the highlighting section. A fragment of the response will look as follows:

```
{
  "_index": "notifier",
  "_id": "3",
  "highlight": {
```

```
    "title": [
        "<em>Crime</em> and Punishment"
    ]
}
```

Everything works, even the highlighting that highlighted the relevant part of the `title` field.



Note that there are some limitations when it comes to the query types supported by the percolator functionality. In the current implementation, the parent/child and nested queries are not available, so you can't use queries such as `has_child`, `top_children`, `has_parent`, and `nested`.



Getting the number of matching queries

Sometimes, you don't care about the matched queries and what you want is only the number of matched queries. In such cases, sending a document against the standard percolator endpoint is not efficient. Elasticsearch exposes the `_percolate/count` endpoint to handle such cases in an efficient way. An example of such a command will be as follows:

```
curl -XGET 'localhost:9200/notifier/book/_percolate/count?pretty' -d '{
  "doc" : { ... }
}'
```

Indexed documents percolation

There is also another possibility. What if we want to check which queries are matched by an already indexed document? Of course, we can do this. Let's look at the following command:

```
curl -XGET 'localhost:9200/library/book/1/_percolate?percolate_
index=notifier'
```

This command checks the document with the `1` identifier from our `library` index against the percolator index defined by the `percolate_index` parameter. Please remember that, by default, Elasticsearch will use the percolator in the same index as the document; that's why we've specified the `percolate_index` parameter.

Handling files

The next use case we would like to discuss is searching the contents of files. The most obvious method is to add logic to an application, which will be responsible for fetching files, extracting valuable information from them, building JSON objects, and finally, indexing them to Elasticsearch.

Of course, the aforementioned method is valid and you can proceed this way, but there is another way we would like to show you. We can send documents to Elasticsearch for content extraction and indexing. This will require us to install an additional plugin. Note that we will describe plugins in *Chapter 7, Elasticsearch Cluster in Detail*, so we'll skip the detailed description. For now, just run the following command to install the attachments plugin:

```
bin/plugin -install elasticsearch/elasticsearch-mapper-attachments/2.0.0.RC1
```

After restarting Elasticsearch, it will miraculously gain a new skill, which we will play with now. Let's begin with preparing a new index that has the following mappings:

```
{
  "mappings" : {
    "file" : {
      "properties" : {
        "note" : { "type" : "string", "store" : "yes" },
        "book" : {
          "type" : "attachment",
          "fields" : {
            "file" : { "store" : "yes", "index" : "analyzed" },
            "date" : { "store" : "yes" },
            "author" : { "store" : "yes" },
            "keywords" : { "store" : "yes" },
            "content_type" : { "store" : "yes" },
            "title" : { "store" : "yes" }
          }
        }
      }
    }
  }
}
```

As we can see, we have the `book` type, which we will use to store the contents of our file. In addition to this, we've defined some nested fields, which are as follows:

- `file`: This field defines the file contents
- `date`: This field defines the file creation date
- `author`: This field defines the author of the file
- `keywords`: This field defines the additional keywords connected with the document
- `content_type`: This field defines the mime type of the document
- `title`: This field defines the title of the document

These fields will be extracted from files, if available. In our example, we marked all the fields as stored – this allows us to see their values in the search results. In addition, we defined the `note` field. This is an ordinary field, which will not be used by the plugin but by us.

Now, we should prepare our document. Let's look at the following example document placed in the `index.json` file:

```
{  
  "book" : "UEsDBBQABgAIAAAAIQDpURCwjQEAAMIFAAATAAgCW0NvbnnR  
  lbnRfVHlwZXNdLnhtbCCiBAIooAA...",  
  "note" : "just a note"  
}
```

As you can see, we have some strange contents of the `book` field. This is the content of the file that is encoded with the Base64 algorithm (please note that this is only a small part of it – we omitted the rest of this field for clarity). Because the file contents can be binary and thus, cannot be easily included in the JSON structure, the authors of Elasticsearch require us to encode the file contents with the mentioned algorithm. On the Linux operating system, there is a simple command that we use to encode a document's contents into Base64; for example, we can use the following command:

```
base64 -i example.docx -o example.docx.base64
```

We assume that you have successfully created a proper Base64 version of our document. Now, we can index this document by running the following command:

```
curl -XPUT 'localhost:9200/media/file/1?pretty' -d @index.json
```

This was simple. In the background, Elasticsearch decoded the file, extracted its contents, and created proper entries in our index.

Now, let's create the query (we've placed it in the `query.json` file) that we will use to find our document, as follows:

```
{  
    "fields" : ["title", "author", "date", "keywords",  
    "content_type", "note"],  
    "query" : {  
        "term" : { "book" : "example" }  
    }  
}
```

If you have read the previous chapters carefully, the preceding query should be simple to understand. We asked for the `example` word in the `book` field. Our example document, which we encoded, contains the following text: This is an example document for 'Elasticsearch Server' book. So, the example query we've just made should match our document. Let's check this assumption by executing the following command:

```
curl -XGET 'localhost:9200/media/_search?pretty' -d @query.json
```

If everything goes well, we should get a response similar to the following one:

```
{  
    "took" : 2,  
    "timed_out" : false,  
    "_shards" : {  
        "total" : 5,  
        "successful" : 5,  
        "failed" : 0  
    },  
    "hits" : {  
        "total" : 1,  
        "max_score" : 0.095891505,  
        "hits" : [ {  
            "_index" : "media",  
            "_type" : "file",  
            "_id" : "1",  
            "_score" : 0.095891505,  
            "fields" : {  
                "title" : "This is an example document for 'Elasticsearch Server' book.",  
                "author" : "Elasticsearch Server",  
                "date" : "2014-07-25T12:57:14Z",  
                "keywords" : "elasticsearch, search, server",  
                "content_type" : "text/plain",  
                "note" : "This is an example document for 'Elasticsearch Server' book."  
            }  
        }]  
    }  
}
```

```
        "book.date" : [ "2014-02-08T09:34:00.000Z" ],
        "book.content_type" : [ "application/vnd.openxmlformats-
            officedocument.wordprocessingml.document" ],
        "note" : [ "just a note" ],
        "book.author" : [ "Rafał Ku , Marek Rogozi ski" ]
    }
}
}
}
```

Looking at the result, you can see the content type as `application/vnd.openxmlformats-officedocument.wordprocessingml.document`. You can guess that our document was created in Microsoft Office and probably had the `.docx` extension. We can also see additional fields extracted from the document such as authors or modification date. And again, everything works!

Adding additional information about the file

When we are indexing files, the obvious requirement is the possibility of the filename being returned in the result list. Of course, we can add the filename as another field in the document, but Elasticsearch allows us to store this information within the file object. We can just add the `_name` field to the document we send to Elasticsearch. For example, if we want the name of `example.docx` to be indexed as the name of the document, we can send a document such as the following:

```
{
  "book" :
    "UESDBBQABgAIAAAAIQDpURCwjQEAMIFAAATAAgCW0NvbnRlbnRfVHL
    wZXNdLnhtbCCiBAIooAA...",
  "_name" : "example.docx",
  "note" : "just a note"
}
```

By including the `_name` field, Elasticsearch will include the name in the result list. The filename will be available as a part of the `_source` field. However, if you use the `fields` property and want to have the name of the file returned in the results, you should add the `_source` field as one of the entries in this property.

And at the end, you can use the `content_type` field to store information about the mime type just as we used the `_name` field to store the filename.

Geo

The search servers such as Elasticsearch are usually looked at from the perspective of full-text searching. However, this is only a part of the whole view. Sometimes, a full-text search is not enough. Imagine searching for local services. For the end user, the most important thing is the accuracy of the results. By accuracy, we not only mean the proper results of the full-text search, but also the results being as near as they can in terms of location. In several cases, this is the same as the text search on geographical names such as cities or streets, but in other cases, we can find it very useful to be able to search on the basis of the geographical coordinates of our indexed documents. And, this is also a functionality that Elasticsearch is capable of handling.

Mappings preparation for spatial search

In order to discuss the spatial search functionality, let's prepare an index with a list of cities. This will be a very simple index with one type named `poi` (which stands for the point of interest), the name of the city, and its coordinates. The mappings are as follows:

```
{
  "mappings" : {
    "poi" : {
      "properties" : {
        "name" : { "type" : "string" },
        "location" : { "type" : "geo_point" }
      }
    }
  }
}
```

Assuming that we put this definition into the `mapping.json` file, we can create an index by running the following command:

```
curl -XPUT localhost:9200/map -d @mapping.json
```

The only new thing is the `geo_point` type, which is used for the `location` field. By using it, we can store the geographical position of our city.

Example data

Our example file with documents looks as follows:

```
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 1 } }
{ "name" : "New York", "location" : "40.664167, -73.938611" }
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 2 } }
{ "name" : "London", "location" : [-0.1275, 51.507222] }
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 3 } }
{ "name" : "Moscow", "location" : { "lat" : 55.75, "lon" : 37.616667 } }
}
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 4 } }
{ "name" : "Sydney", "location" : "-33.859972, 151.211111" }
{ "index" : { "_index" : "map", "_type" : "poi", "_id" : 5 } }
{ "name" : "Lisbon", "location" : "eycs0p8ukc7v" }
```

In order to perform a bulk request, we've added information about the index name, type, and unique identifiers of our documents; so, we can now easily import this data using the following command:

```
curl -XPOST http://localhost:9200/_bulk --data-binary @documents.json
```

One thing that we should take a closer look at is the `location` field. We can use various notations for coordination. We can provide the latitude and longitude values as a string, as a pair of numbers, or as an object. Please note that the string and array methods of providing the geographical location have a different order for the latitude and longitude parameters. The last record shows that there is also a possibility to give coordination as a geohash value (the notation is described in detail at <http://en.wikipedia.org/wiki/Geohash>).

Sample queries

Now, let's look at several examples of how to use coordinates and how to solve common requirements in modern applications that require geographical data searching along with full-text searching.

Distance-based sorting

Let's start with a very common requirement: sorting results by distance from the given point. In our example, we want to get all the cities and sort them by their distances from the capital of France—Paris. To do this, we will send the following query to Elasticsearch:

```
{
  "query" : {
```

```
    "match_all" : {},
},
"sort" : [
    "_geo_distance" : {
        "location" : "48.8567, 2.3508",
        "unit" : "km"
    }
]
}
```

If you remember the *Sorting data* section from *Chapter 3, Searching Your Data*, you'll notice that the format is slightly different. We are using the `_geo_distance` key to indicate sorting by distance. We must give the base location (the `location` attribute, which holds the information of the location of Paris in our case), and we need to specify the units that can be used in the results. The available values are `km` and `mi`, which stand for kilometers and miles, respectively. The result of such a query will be as follows:

```
{
    "took" : 102,
    "timed_out" : false,
    "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    },
    "hits" : {
        "total" : 5,
        "max_score" : null,
        "hits" : [ {
            "_index" : "map",
            "_type" : "poi",
            "_id" : "2",
            "_score" : null, "_source" : { "name" : "London", "location" : [-0.1275, 51.507222] },
            "sort" : [ 343.46748684411773 ]
        }, {
            "_index" : "map",
            "_type" : "poi",
            "_id" : "5",
            "_score" : null, "_source" : { "name" : "Lisbon", "location" : "eycs0p8ukc7v" },
            "sort" : [ 343.46748684411773 ]
        } ]
    }
}
```

```
        "sort" : [ 1453.6450747751787 ]
    }, {
        "_index" : "map",
        "_type" : "poi",
        "_id" : "3",
        "_score" : null, "_source" : { "name" : "Moscow", "location" :
            { "lat" : 55.75, "lon" : 37.616667 } },
        "sort" : [ 2486.2560754763977 ]
    }, {
        "_index" : "map",
        "_type" : "poi",
        "_id" : "1",
        "_score" : null, "_source" : { "name" : "New York", "location" :
            { "lat" : 40.664167, "lon" : -73.938611 } },
        "sort" : [ 5835.763890418129 ]
    }, {
        "_index" : "map",
        "_type" : "poi",
        "_id" : "4",
        "_score" : null, "_source" : { "name" : "Sydney", "location" :
            { "lat" : -33.859972, "lon" : 151.211111 } },
        "sort" : [ 16960.04911335322 ]
    }
}
}
```

As for the other examples with sorting, Elasticsearch shows information about the value used for sorting. Let's look at the highlighted record. As we can see, the distance between Paris and London is about 343 km, and you can see that the map agrees with Elasticsearch in this case.

Bounding box filtering

The next example that we want to show is narrowing down the results to a selected area that is bounded by a given rectangle. This is very handy if we want to show results on the map or when we allow a user to mark the map area for searching. You already read about filters in the *Filtering your results* section of *Chapter 2, Indexing Your Data*, but there we didn't mention the spatial filters. The following query shows how we can filter by using the bounding box:

```
{
    "filter" : {
```

```
"geo_bounding_box" : {  
    "location" : {  
        "top_left" : "52.4796, -1.903",  
        "bottom_right" : "48.8567, 2.3508"  
    }  
}
```

In the preceding example, we selected a map fragment between Birmingham and Paris by providing the top-left and bottom-right corner coordinates. These two corners are enough to specify any rectangle we want, and Elasticsearch will do the rest of the calculation for us. The following screenshot shows the specified rectangle on the map:



As we can see, the only city from our data that meets the criteria is London. So, let's check whether Elasticsearch knows this by running the preceding query. Let's look at the returned results, as follows:

```
{  
  "took" : 9,  
  "timed_out" : false,  
  "_shards" : {  
    "total" : 5,  
    "successful" : 5,  
    "failed" : 0  
  },  
  "hits" : {  
    "total" : 1,  
    "max_score" : 1.0,  
    "hits" : [ {  
      "_index" : "map",  
      "_type" : "poi",  
      "_id" : "2",  
      "_score" : 1.0, "_source" : { "name" : "London", "location" :  
        [-0.1275, 51.507222] }  
    } ]  
  }  
}
```

As you can see, again Elasticsearch agrees with the map.

Limiting the distance

The last example shows the next common requirement: limiting the results to the places that are located in the selected distance from the base point. For example, if we want to limit our results to all the cities within the 500km radius from Paris, we can use the following filter:

```
{  
  "filter" : {  
    "geo_distance" : {  
      "location" : "48.8567, 2.3508",
```

```

        "distance" : "500km"
    }
}
}

```

If everything goes well, Elasticsearch should only return a single record for the preceding query, and the record should be London; however, we will leave it for you as a reader to check.

Arbitrary geo shapes

Sometimes, using a single geographical point or a single rectangle is just not enough. In such cases, something more sophisticated is needed, and Elasticsearch addresses this by giving you the possibility to define shapes. In order to show you how we can leverage custom shape limiting in Elasticsearch, we need to modify our index and introduce the `geo_shape` type. Our new mapping looks as follows (we will use this to create an index called `map2`):

```

{
  "poi" : {
    "properties" : {
      "name" : { "type" : "string", "index": "not_analyzed" },
      "location" : { "type" : "geo_shape" }
    }
  }
}

```

Next, let's change our example data to match our new index structure, as follows:

```

{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 1 }}
{ "name" : "New York", "location" : { "type": "point", "coordinates": [-73.938611, 40.664167] }}
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 2 }}
{ "name" : "London", "location" : { "type": "point", "coordinates": [-0.1275, 51.507222] }}
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 3 }}
{ "name" : "Moscow", "location" : { "type": "point", "coordinates": [37.616667, 55.75] }}
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 4 }}
{ "name" : "Sydney", "location" : { "type": "point", "coordinates": [151.211111, -33.865143] }}
{ "index" : { "_index" : "map2", "_type" : "poi", "_id" : 5 }}
{ "name" : "Lisbon", "location" : { "type": "point", "coordinates": [-9.142685, 38.736946] }}

```

The structure of the field of the `geo_shape` type is different from `geo_point`. It is syntactically called **GeoJSON** (<http://en.wikipedia.org/wiki/GeoJSON>). It allows us to define various geographical types. Let's sum up the types that we can use during querying, at least the ones that we think are the most useful ones.

Point

A point is defined by the table when the first element is the longitude and the second is the latitude. An example of such a shape can be as follows:

```
{  
  "location" : {  
    "type": "point",  
    "coordinates": [-0.1275, 51.507222]  
  }  
}
```

Envelope

An envelope defines a box given by the coordinates of the upper-left and bottom-right corners of the box. An example of such a shape is as follows:

```
{  
  "type": "envelope",  
  "coordinates": [[ -0.087890625, 51.50874245880332 ], [  
    2.4169921875, 48.80686346108517 ]]  
}
```

Polygon

A polygon defines a list of points that are connected to create our polygon. The first and the last point in the array must be the same so that the shape is closed. An example of such a shape is as follows:

```
{  
  "type": "polygon",  
  "coordinates": [ [  
    [-5.756836, 49.991408],  
    [-7.250977, 55.124723],  
    [1.845703, 51.500194],  
    [-5.756836, 49.991408]  
  ]]  
}
```

If you look closer at the shape definition, you will find a supplementary level of tables. Thanks to this, you can define more than a single polygon. In such a case, the first polygon defines the base shape and the rest of the polygons are the shapes that will be excluded from the base shape.

Multipolygon

The multipolygon shape allows us to create a shape that consists of multiple polygons. An example of such a shape is as follows:

```
{
  "type": "multipolygon",
  "coordinates": [
    [
      [
        [-5.756836, 49.991408],
        [-7.250977, 55.124723],
        [1.845703, 51.500194],
        [-5.756836, 49.991408]
      ],
      [
        [-0.087890625, 51.50874245880332],
        [2.4169921875, 48.80686346108517],
        [3.88916015625, 51.01375465718826],
        [-0.087890625, 51.50874245880332]
      ]
    ]
  ]
}
```

The multipolygon shape contains multiple polygons and falls into the same rules as the polygon type. So, we can have multiple polygons and in addition to this, we can include multiple exclusion shapes.

An example usage

Now that we have our index with the geo_shape fields, we can check which cities are located in the UK. The query that will allow us to do this will look as follows:

```
{
  "filter": {
    "geo_shape": {
      "location": {
        "shape": {
          "type": "polygon",

```

```
    "coordinates": [
        [-5.756836, 49.991408], [-7.250977, 55.124723], [-3.955078, 59.352096], [1.845703, 51.500194], [-5.756836, 49.991408]
    ]
}
}
}
}
```

The polygon type defines the boundaries of the UK (in a very, very imprecise way), and Elasticsearch gives the response as follows:

```
"hits": [
{
    "_index": "map2",
    "_type": "poi",
    "_id": "2",
    "_score": 1,
    "_source": {
        "name": "London",
        "location": {
            "type": "point",
            "coordinates": [
                -0.1275,
                51.507222
            ]
        }
    }
}]
```

As far as we know, the response is correct.

Storing shapes in the index

Usually, the shape definitions are complex, and the defined areas don't change too often (for example, the UK boundaries). In such cases, it is convenient to define the shapes in the index and use them in queries. This is possible, and we will now discuss how to do it. As usual, we will start with the appropriate mapping, which is as follows:

```
{
  "country": {
    "properties": {
      "name": { "type": "string", "index": "not_analyzed" },
      "area": { "type": "geo_shape" }
    }
  }
}
```

This mapping is similar to the mapping used previously. We have only changed the field name. The example data that we will use looks as follows:

```
{"index": { "_index": "countries", "_type": "country", "_id": 1 }}
{"name": "UK", "area": { "type": "polygon", "coordinates": [[[[-5.756836, 49.991408], [-7.250977, 55.124723], [-3.955078, 59.352096], [1.845703, 51.500194], [-5.756836, 49.991408]]]]}}
{"index": { "_index": "countries", "_type": "country", "_id": 2 }}
{"name": "France", "area": { "type": "polygon", "coordinates": [[[3.1640625, 42.09822241118974], [-1.7578125, 43.32517767999296], [-4.21875, 48.22467264956519], [2.4609375, 50.90303283111257], [7.998046875, 48.980216985374994], [7.470703125, 44.08758502824516], [3.1640625, 42.09822241118974]]]]}}
 {"index": { "_index": "countries", "_type": "country", "_id": 3 }}
 {"name": "Spain", "area": { "type": "polygon", "coordinates": [[[3.33984375, 42.22851735620852], [-1.845703125, 43.32517767999296], [-9.404296875, 43.19716728250127], [-6.6796875, 41.57436130598913], [-7.3828125, 36.87962060502676], [-2.109375, 36.52729481454624], [3.33984375, 42.22851735620852]]]]}}
```

As you can see in the data, each document contains a `polygon` type. The polygons define the area of the given countries (again, it is far from being accurate). If you remember, the first point of a shape needs to be the same as the last one so that the shape is closed. Now, let's change our query to include the shapes from the index. Our new query will look as follows:

```
{  
  "filter": {  
    "geo_shape": {  
      "location": {  
        "indexed_shape": {  
          "index": "countries",  
          "type": "country",  
          "path": "area",  
          "id": "1"  
        }  
      }  
    }  
  }  
}
```

When comparing these two queries, we can note that the `shape` object changed to `indexed_shape`. We need to tell Elasticsearch where to look for this shape. We can do this by defining the index (the `index` property, which defaults to `shape`), type (the `type` property), and path (the `path` property, which defaults to `shape`). The one item lacking is an `id` property of the shape. In our case, this is `1`. However, if you want to index more shapes, we will advise you to index shapes with their name as their identifier.

The scroll API

Let's imagine that we have an index with several million documents. We already know how to build our query, when to use filters, and so on. But looking at the query logs, we see that a particular kind of query is significantly slower than the others. These queries may be using pagination. The `from` parameter indicates that the offsets have large values. From the application side, this can mean that users go through an enormous number of results. Often, this doesn't make sense—if a user doesn't find the desirable results on the first few pages, he/she gives up. Because this particular activity can mean something bad (possible data stealing), many applications limit the paging to dozens of pages. In our case, we assume that this is a different scenario, and we have to provide this functionality.

Problem definition

When Elasticsearch generates a response, it must determine the order of the documents that form the result. If we are on the first page, this is not a big problem. Elasticsearch just finds the set of documents and collects the first ones; let's say, 20 documents. But if we are on the tenth page, Elasticsearch has to take all the documents from pages one to ten and then discard the ones that are on pages one to nine. The problem is not Elasticsearch specific; a similar situation can be found in the database systems, for example—generally, in every system that uses the so-called priority queue.

Scrolling to the rescue

The solution is simple. Since Elasticsearch has to do some operations (determine the documents for previous pages) for each request, we can ask Elasticsearch to store this information for the subsequent queries. The drawback is that we cannot store this information forever due to limited resources. Elasticsearch assumes that we can declare how long we need this information to be available. Let's see how it works in practice.

First of all, we query Elasticsearch as we usually do. However, in addition to all the known parameters, we add one more: the parameter with the information that we want to use scrolling with and how long we suggest that Elasticsearch should keep the information about the results. We can do this by sending a query as follows:

```
curl 'localhost:9200/library/_search?pretty&scroll=5m' -d '{
  "query" : {
    "match_all" : { }
  }
}'
```

The content of this query is irrelevant. The important thing is how Elasticsearch modifies the response. Look at the following first few lines of the response returned by Elasticsearch:

```
{
  "_scroll_id" :
  "cXVlcnlUaGVuRmV0Y2g7NTsxMDI6dk1NmlkzTG1RTDJ2b25oTDNENmJzzzsxMD
  U6dk1NmlkzTG1RTDJ2b25oTDNENmJzzzsxMDQ6dk1NmlkzTG1RTDJ2b25oTDNEN
  mJzzzsxMDE6dk1NmlkzTG1RTDJ2b25oTDNENmJzzzsxMDM6dk1NmlkzTG1RTDJ
  2b25oTDNENmJzzzswoW==",
```

```
"took" : 9,  
"timed_out" : false,  
"_shards" : {  
    "total" : 5,  
    "successful" : 5,  
    "failed" : 0  
},  
"hits" : {  
    "total" : 1341211,  
    ...  
}
```

The new part is the `_scroll_id` section. This is a handle that we will use in the queries that follow. Elasticsearch has a special endpoint for this: the `_search/scroll` endpoint. Let's look at the following example:

```
curl -XGET  
'localhost:9200/_search/scroll?scroll=5m&pretty&scroll_id=  
cXVlcnlUaGVuRmV0Y2g7NTsxMjg6dk1N1kzTG1RTDJ2b25oTDNENmJzZzsxMjk6  
dk1N1kzTG1RTDJ2b25oTDNENmJzZzsxMzA6dk1N1kzTG1RTDJ2b25oTDNENmJzZ  
zsxMjc6dk1N1kzTG1RTDJ2b25oTDNENmJzZzsxMjY6dk1N1kzTG1RTDJ2b25oT  
DNENmJzZzsow=='
```

Now, every call to this endpoint with `scroll_id` returns the next page of results. Remember that this handle is only valid for the defined time of inactivity. After the time has passed, a query with the invalidated `scroll_id` returns an error response, which will be similar to the following one:

```
{  
    "_scroll_id" :  
        "cXVlcnlUaGVuRmV0Y2g7NTsxMjg6dk1N1kzTG1RTDJ2b25oTDNENmJzZzsxMj  
        k6dk1N1kzTG1RTDJ2b25oTDNENmJzZzsxMzA6dk1N1kzTG1RTDJ2b25oTDNEN  
        mJzZzsxMjc6dk1N1kzTG1RTDJ2b25oTDNENmJzZzsxMjY6dk1N1kzTG1RTDJ2  
        b25oTDNENmJzZzsow==",  
    "took" : 3,  
    "timed_out" : false,  
    "_shards" : {  
        "total" : 5,  
        "successful" : 0,  
        "failed" : 0  
    }  
}
```

```
"failed" : 5,
"failures" : [ {
  "status" : 500,
  "reason" : "SearchContextMissingException[No search context
found for id [128]]"
}, {
  "status" : 500,
  "reason" : "SearchContextMissingException[No search context
found for id [126]]"
}, {
  "status" : 500,
  "reason" : "SearchContextMissingException[No search context
found for id [127]]"
}, {
  "status" : 500,
  "reason" : "SearchContextMissingException[No search context
found for id [130]]"
}, {
  "status" : 500,
  "reason" : "SearchContextMissingException[No search context
found for id [129]]"
} ]
},
"hits" : {
  "total" : 0,
  "max_score" : 0.0,
  "hits" : [ ]
}
```

Of course, this solution is not ideal, and it is not well suited when there are many requests to random pages of various results or when the time between the requests is difficult to determine. However, you can use this successfully for use cases where you want to get larger result sets, such as transferring data between several systems.

The terms filter

One of the filters available in Elasticsearch that is very simple at first glance is the `terms` filter. In its simplest form, it allows you to filter documents to only those that match one of the given terms and is not analyzed. An example use of the `terms` filter is as follows:

```
{  
  "query" : {  
    "constant_score" : {  
      "filter" : {  
        "terms" : {  
          "title" : [ "crime", "punishment" ]  
        }  
      }  
    }  
  }  
}
```

The preceding query would result in documents that match the `crime` or `punishment` terms in the `title` field. The way the `terms` filter works is that it iterates over the provided terms and finds the documents that match these terms. Of course, the matched document identifiers are loaded into a structure called `bitset` and are cached. Sometimes, we may want to alter the default behavior. We can do this by providing the `execution` parameter with one of the following values:

- `plain`: This is the default method that iterates over all the terms provided, storing the results in a bitset and caching them.
- `fielddata`: This value generates term filters that use the `fielddata` cache to compare terms. This mode is very efficient when filtering on the fields that are already loaded into the `fielddata` cache—for example, the ones used for sorting, facetting, or warming using index warmers. This execution mode can be very effective when filtering on a large number of terms.
- `bool`: This value generates a `term` filter for each term and constructs a `bool` filter from the generated ones. The constructed `bool` filter is not cached because it can execute the term filters that were constructed and were already cached.
- `and`: This value is similar to the `bool` value, but Elasticsearch constructs the `and` filter instead of the `bool` filter.
- `or`: This value is similar to the `bool` value, but Elasticsearch constructs the `or` filter instead of the `bool` filter.

An example query with the `execution` parameter can look like the following:

```
{
  "query" : {
    "constant_score" : {
      "filter" : {
        "terms" : {
          "title" : [ "crime", "punishment" ],
          "execution" : "and"
        }
      }
    }
  }
}
```

Terms lookup

We are talking about the `terms` filter not because of its ability to filter documents but because of the terms lookup functionality added in Elasticsearch 0.90.6. Instead of passing the list of terms explicitly, the terms lookup mechanism can be used to load the terms from a provided source. To illustrate how it works, let's create a new index and index three documents by using the following commands:

```
curl -XPOST 'localhost:9200/books/book/1' -d '{
  "id" : 1,
  "name" : "Test book 1",
  "similar" : [ 2, 3 ]
}'

curl -XPOST 'localhost:9200/books/book/2' -d '{
  "id" : 2,
  "name" : "Test book 2",
  "similar" : [ 1 ]
}'

curl -XPOST 'localhost:9200/books/book/3' -d '{
  "id" : 3,
  "name" : "Test book 3",
  "similar" : [ 1, 3 ]
}'
```

Now, let's assume that we want to get all the books that are similar to the book with the identifier equal to 3. Of course, we can first get the third book, get the value for the `similar` field, and run another query. But let's do it using the terms lookup functionality; basically, we will let Elasticsearch retrieve the document and load the value of the `similar` field for us. To do this, we can run the following command:

```
curl -XGET 'localhost:9200/books/_search?pretty' -d '{  
    "query" : {  
        "filtered" : {  
            "query" : {  
                "match_all" : {}  
            },  
            "filter" : {  
                "terms" : {  
                    "id" : {  
                        "index" : "books",  
                        "type" : "book",  
                        "id" : "3",  
                        "path" : "similar"  
                    },  
                    "_cache_key" : "books_3_similar"  
                }  
            }  
        }  
    },  
    "fields" : [ "id", "name" ]  
}'
```

The response to the preceding command will be as follows:

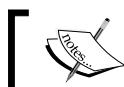
```
{  
    "took" : 2,  
    "timed_out" : false,  
    "_shards" : {  
        "total" : 5,  
        "successful" : 5,  
        "failed" : 0
```

```

},
"hits" : {
  "total" : 2,
  "max_score" : 1.0,
  "hits" : [ {
    "_index" : "books",
    "_type" : "book",
    "_id" : "1",
    "_score" : 1.0,
    "fields" : {
      "id" : 1,
      "name" : "Test book 1"
    }
  }, {
    "_index" : "books",
    "_type" : "book",
    "_id" : "3",
    "_score" : 1.0,
    "fields" : {
      "id" : 3,
      "name" : "Test book 3"
    }
  } ]
}
}

```

As you can see in the preceding response, we got exactly what we wanted: the books with the identifiers 1 and 3. Of course, the terms lookup mechanism is highly optimized—the cache information will be used if the information is present and so on. Also, the `_cache_key` property is used to specify the key under which the cached results for the terms lookup will be stored. It is advisable to set it in order to be able to easily clear the cache if needed. Of course the `_cache_key` property value should be different for different queries.



Note that the `_source` field needs to be stored for the terms lookup functionality to work.

The terms lookup query structure

Let's recall our `terms` lookup filter that we used in order to discuss the query to fully understand it:

```
"filter" : {  
    "terms" : {  
        "id" : {  
            "index" : "books",  
            "type" : "book",  
            "id" : "3",  
            "path" : "similar"  
        },  
        "_cache_key" : "books_3_similar"  
    }  
}
```

We used a simple filtered query, with the query matching all the documents and the `terms` filter. We are filtering the documents using the `id` field because of the name of the object that groups all the other properties in the filter. In addition to this, we've used the following properties:

- `index`: This specifies from which index we want the terms to be loaded. In our case, it's the `books` index.
- `type`: This specifies the type that we are interested in, which in our case, is the `book` type.
- `id`: This specifies the identifier of the documents we want the terms list to be fetched from. In our case, it is the document with the identifier `3`.
- `path`: This specifies the field name from which the terms should be loaded, which is the `similar` field in our query.

What's more is that we are allowed to use two more properties, which are as follows:

- `routing`: This specifies the routing value that should be used by Elasticsearch when loading the terms to the filter.
- `cache`: This specifies whether Elasticsearch should cache the filter built from the loaded documents. By default, it is set to `true`, which means that Elasticsearch will cache the filter.



Note that the `execution` property is not taken into account when using the `terms` lookup mechanism.

Terms lookup cache settings

Elasticsearch allows us to configure the cache used by the terms lookup mechanism. To control the mentioned cache, one can set the following properties in the `elasticsearch.yml` file:

- `indices.cache.filter.terms.size`: This defaults to 10mb and specifies the maximum amount of memory that Elasticsearch can use for the terms lookup cache. The default value should be enough for most cases; however, if you know that you'll load vast amount of data into it, you can increase it.
- `indices.cache.filter.terms.expire_after_access`: This specifies the maximum time after which an entry should expire after it is last accessed. By default, it is disabled.
- `indices.cache.filter.terms.expire_after_write`: This specifies the maximum time after which an entry should be expired after it is put into the cache. By default, it is disabled.

Summary

In this chapter, we learned more things about Elasticsearch data analysis capabilities. We used aggregations and faceting to bring meaning to the data we indexed. We also introduced the spellchecking and autocomplete functionalities to our application by using the Elasticsearch suggesters. We created the alerting functionality by using a percolator, and we indexed binary files by using the attachment functionality. We indexed and searched geospatial data and used the scroll API to efficiently fetch a large number of results. Finally, we used the terms lookup mechanism to speed up the querying process that fetches a list of terms.

In the next chapter, we'll focus on Elasticsearch clusters and how to handle them. We'll see what node discovery is, how it is used, and how to alter its configuration. We'll learn about the gateway and recovery modules, and we will alter their configuration. We will also see what the buffers in Elasticsearch are, where they are used, and how to configure them. We will prepare our cluster for a high indexing and querying throughput, and we will use index templates and dynamic mappings.

7

Elasticsearch Cluster in Detail

In the previous chapter, we learned more about Elasticsearch's data analysis capabilities. We used aggregations and faceting to add meaning to the data we indexed. We also introduced the spellcheck and autocomplete functionalities to our application by using Elasticsearch suggesters. We've created the alerting functionality by using a percolator, and we've indexed binary files by using the attachment capability. We've indexed and searched geospatial data, and we've used the scroll API to efficiently fetch a large number of results. Finally, we've used the terms lookup to speed up the queries that fetch a list of terms and use them.

By the end of this chapter, you will have learned the following topics:

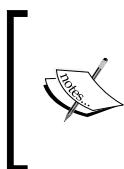
- Understanding a node's discovery mechanism, configuration, and tuning
- Controlling recovery and gateway modules
- Preparing Elasticsearch for high query and indexing use cases
- Using index templates and dynamic mappings

Node discovery

When you start your Elasticsearch node, one of the first things to occur is that the node starts looking for a master node that has the same cluster name and is visible. If a master is found, the node joins a cluster that is already formed. If no master is found, the node itself is selected as a master (of course, if the configuration allows such a behavior). The process of forming a cluster and finding nodes is called **discovery**. The module that is responsible for discovery has two main purposes—to elect a master and to discover new nodes within a cluster. In this section, we will discuss how we can configure and tune the discovery module.

Discovery types

By default, without installing additional plugins, Elasticsearch allows us to use the zen discovery, which provides us with multicast and unicast discovery. In computer networking terminology, multicast (<http://en.wikipedia.org/wiki/Multicast>) is the delivery of a message to a group of computers in a single transmission. On the other hand, we have unicast (<http://en.wikipedia.org/wiki/Unicast>), which is the transmission of a single message over the network to a single host at once.



When using the multicast discovery, Elasticsearch will try to find all the nodes that are able to receive and respond to the multicast message. If you use the unicast method, you'll need to provide at least some of the hosts that form your cluster and the node will try to connect to them.

When choosing between multicast and unicast, you should be aware whether your network can handle multicast messages. If it can, using multicast will be easier. If your network can't handle multicast, use the unicast type of discovery. The other reason for using the unicast discovery is security – you don't want any nodes to join your cluster by mistake. So, using unicast may be a good choice if you are going to run multiple clusters or your developer machines are in the same network.



If you are using the Linux operating system and want to check if your network supports multicast, please use the `ifconfig` command for your network interface (usually it will be `eth0`). If your network supports multicast, you'll see the `MULTICAST` property in the response from the preceding command.

The master node

As we have already seen, one of the main purposes of discovery is to choose a master node that will be used as a node that will look over the cluster. The master node is the one that checks all the other nodes to see if they are responsive (other nodes ping the master too). The master node will also accept the new nodes that want to join the cluster. If the master is somehow disconnected from the cluster, the remaining nodes will select a new master from among themselves. All these processes are done automatically on the basis of the configuration values we provide.

Configuring the master and data nodes

By default, Elasticsearch allows every node to be a master node and a data node. However, in certain situations, you may want to have worker nodes that will only hold the data and master nodes that will only be used to process requests and manage the cluster. One of these situations is when you have to handle massive amount of data, where data nodes should be as performant as possible. To set the node to only hold data, we need to instruct Elasticsearch that we don't want such a node to be a master node. In order to do this, we will add the following properties to the `elasticsearch.yml` configuration file:

```
node.master: false  
node.data: true
```

To set the node to not hold data and only be a master node, we need to instruct Elasticsearch that we don't want such a node to hold data. In order to do this, we add the following properties to the `elasticsearch.yml` configuration file:

```
node.master: true  
node.data: false
```

Please note that the `node.master` and `node.data` properties are set to `true` by default, but we tend to include them for the clarity of the configuration.

The master-election configuration

Imagine that you have a cluster built of 10 nodes. Everything is working fine until one day when your network fails and three of your nodes are disconnected from the cluster, but they still see each other. Because of the zen discovery and master-election process, the nodes that got disconnected elect a new master and you end up with two clusters with the same name and two master nodes. Such a situation is called a **split-brain**, and you must avoid it as much as possible. When a split-brain happens, you end up with two (or more) clusters that won't join each other until the network (or any other) problems are fixed.

In order to prevent split-brain situations, Elasticsearch provides a `discovery.zen.minimum_master_nodes` property. This property defines a minimum amount of the master-eligible nodes that should be connected to each other in order to form a cluster. So now, let's get back to our cluster; if we set the `discovery.zen.minimum_master_nodes` property to 50 percent of the total nodes available plus one (which is six in our case), we would end up with a single cluster. Why is that? Before the network failure, we would have 10 nodes, which is more than six nodes and these nodes would form a cluster. After the disconnection of the three nodes, we would still have the first cluster up and running. However, because only three nodes have been disconnected and three is less than six, the remaining three nodes wouldn't be allowed to elect a new master and they would have to wait for reconnection with the original cluster.

Setting the cluster name

If we don't set the `cluster.name` property in our `elasticsearch.yml` file, Elasticsearch will use the default value, `elasticsearch`. This is not always a good thing, and because of this, we suggest that you set the `cluster.name` property to some other value of your choice. Setting a different `cluster.name` property is also required if you want to run multiple clusters in a single network; otherwise, you would end up with nodes that belong to different clusters joining together.

Configuring multicast

Multicast is the default zen discovery method. Apart from the common settings, which we will discuss in a moment, there are four properties that we can control and they are as follows:

- `discovery.zen.ping.multicast.group`: The group address to be used for the multicast requests; it defaults to `224.2.2.4`.
- `discovery.zen.ping.multicast.port`: The port that is used for multicast communication; it defaults to `54328`.
- `discovery.zen.ping.multicast.ttl`: The time for which the multicast request will be considered valid; it defaults to 3 seconds.
- `discovery.zen.ping.multicast.address`: The address to which Elasticsearch should bind. It defaults to the `null` value, which means that Elasticsearch will try to bind to all the network interfaces visible by the operating system.

In order to disable multicast, one should add the `discovery.zen.ping.multicast.enabled` property to the `elasticsearch.yml` file and set its value to `false`.

Configuring unicast

Because of the way unicast works, we need to specify at least a single host that the unicast message should be sent to. To do this, we should add the `discovery.zen.ping.unicast.hosts` property to our `elasticsearch.yml` configuration file. Basically, we should specify all the hosts that form the cluster in the `discovery.zen.ping.unicast.hosts` property. We don't have to specify all the hosts; we just need to provide enough so that we are sure that at least one will work. For example, if we would like the 192.168.2.1, 192.168.2.2, and 192.168.2.3 hosts for our host, we should specify the preceding property in the following way:

```
discovery.zen.ping.unicast.hosts: 192.168.2.1:9300, 192.168.2.2:9300,  
192.168.2.3:9300
```

One can also define a range of ports that Elasticsearch can use; for example, to say that the ports from 9300 to 9399 can be used, we would specify the following command line:

```
discovery.zen.ping.unicast.hosts: 192.168.2.1:[9300-9399],  
192.168.2.2:[9300-9399], 192.168.2.3:[9300-9399]
```

Please note that the hosts are separated with the comma character and we've specified the port on which we expect the unicast messages.

 Always set the `discovery.zen.ping.multicast.enabled` property to `false` when using unicast.

Ping settings for nodes

In addition to the settings discussed previously, we can control or alter the default ping configuration. Ping is a signal sent between nodes to check whether they are running and responsive. The master node pings all the other nodes in the cluster, and each of the other nodes in the cluster pings the master node. The following properties can be set:

- `discovery.zen.fd.ping_interval`: This property defaults to `1s` (one second) and specifies how often nodes ping each other
- `discovery.zen.fd.ping_timeout`: This property defaults to `30s` (30 seconds) and defines how long a node will wait for a response to its ping message before considering the node as unresponsive
- `discovery.zen.fd.ping_retries`: This property defaults to `3` and specifies how many retries should be taken before considering a node as not working

If you experience some problems with your network or know that your nodes need more time to see the ping response, you can adjust the preceding values to the ones that are good for your deployment.

The gateway and recovery modules

Apart from our indices and the data indexed inside them, Elasticsearch needs to hold the metadata such as the type mappings and index-level settings. This information needs to be persisted somewhere so that it can be read during the cluster recovery. This is why Elasticsearch introduced the gateway module. You can think about it as a safe haven for your cluster data and metadata. Each time you start your cluster, all the required data is read from the gateway and when you make a change to your cluster, it is persisted using the gateway module.

The gateway

In order to set the type of gateway we want to use, we need to add the `gateway.type` property to the `elasticsearch.yml` configuration file and set it to a local value. Currently, Elasticsearch recommends that you use the local gateway type (`gateway.type` set to `local`), which is the default. There were additional gateway types in the past (such as the `fs`, `hdfs`, and `s3`), but they are deprecated and will be removed in the future versions. Because of this, we will skip discussing them.

The default local gateway type stores the indices and their metadata in the local file system. Compared to other gateways, the write operation to this gateway is not performed in an asynchronous way. So, whenever a write succeeds, you can be sure that the data was written into the gateway (so, basically it is indexed or stored in the transaction log).

Recovery control

In addition to choosing the gateway type, Elasticsearch allows us to configure when to start the initial recovery process. Recovery is a process of initializing all the shards and replicas, reading all the data from the transaction log, and applying the data on the shards – basically, it's a process needed to start Elasticsearch.

For example, let's imagine that we have a cluster that consists of 10 Elasticsearch nodes. We should inform Elasticsearch about the number of nodes by setting the `gateway.expected_nodes` property to this value; 10, in our case. We inform Elasticsearch about the amount of expected nodes that are eligible to hold the data and be selected as a master. Elasticsearch will start the recovery process immediately if the number of nodes in the cluster is equal to the `gateway.expected_nodes` property.

We would also like to start the recovery after eight nodes for the cluster. In order to do this, we should set the `gateway.recover_after_nodes` property to 8. We could set the value to any value we like. However, we should set it to a value that ensures that the newest version of the cluster state snapshot is available, which usually means that you should start recovery when most of your nodes are available.

However, there is one more thing – we would like the gateway recovery process to start 10 minutes after the cluster was formed, so we set the `gateway.recover_after_time` property to `10m`. This property tells the gateway module how long it should wait with the recovery after the number of nodes specified by the `gateway.recover_after_nodes` property has formed the cluster. We may want to do this because we know that our network is quite slow and we want the communication between nodes to be stable.

The preceding property values should be set in the `elasticsearch.yml` configuration file. If we would like to have the preceding value in the mentioned file, we would end up with the following section in the file:

```
gateway.recover_after_nodes: 8
gateway.recover_after_time: 10m
gateway.expected_nodes: 10
```

Additional gateway recovery options

In addition to the mentioned options, Elasticsearch allows us some additional degree of control. The additional options are as follows:

- `gateway.recover_after_master_nodes`: This property is similar to the `gateway_recover_after_nodes` property. However, instead of taking into consideration all the nodes, it allows us to specify how many nodes that is eligible to be the master should be present in the cluster before the recovery starts.
- `gateway.recover_after_data_nodes`: This property is also similar to the `gateway_recover_after_nodes` property, but it allows you to specify how many data nodes should be present in the cluster before the recovery starts.
- `gateway.expected_master_nodes`: This property is similar to the `gateway.expected_nodes` property, but instead of specifying the number of nodes that we expect in the cluster, it allows you to specify how many nodes you expect to be present are eligible to be the master.
- `gateway.expected_data_nodes`: This property is also similar to the `gateway.expected_nodes` property, but allows you to specify how many data nodes you expect to be present in the cluster.

Preparing Elasticsearch cluster for high query and indexing throughput

Until now, we mostly talked about the different functionalities of Elasticsearch, both in terms of handling queries as well as indexing data. However, preparing Elasticsearch for high query and indexing throughput is something that we would briefly like to talk about. We start this section by mentioning some functionalities of Elasticsearch that we didn't talk until now but are pretty important when it comes to tuning your cluster. We know that it is a very concentrated knowledge, but we will try to limit it to only those things that we think are important. After discussing the functionality, we will give you general advice on how to tune the discussed functionalities and what to pay attention to. We hope that by reading this section you will be able to see which things to look for when you are tuning your cluster.

The filter cache

The filter cache is responsible for caching the filters used in a query. You can retrieve information from the cache very fast. When properly configured, it will speed up querying efficiently, especially the ones that includes filters that were already executed.

Elasticsearch includes two types of filter caches: the node filter cache (the default one) and the index filter cache. The node filter cache is shared across all the indices allocated on a single node and can be configured to use a specific amount of memory or a percentage of the total memory given to Elasticsearch. To specify this value, we should include the node property named `indices.cache.filter.size` and set it to the desired size or percentage.

The second type of the filter cache is the per index one. In general, one should use the node-level filter cache because it is hard to predict the final size of the per-index filter cache. This is because you usually don't know how many indices will end up on a given node. We will omit further explanation of the per-index filter cache; more information about it can be found in the official documentation and the book, *Mastering ElasticSearch*, Rafał Kuć and Marek Rogoziński, Packt Publishing.

The field data cache and circuit breaker

The field data cache is a part of Elasticsearch that is used mainly when a query performs sorting or faceting on a field. Elasticsearch loads the data used for such fields to the memory, which allows a fast access to the values on a per document basis. Building the field data cache is expensive, so it is advisable to have enough memory so that the data in this cache is kept in the memory once loaded.



Instead of the field data cache, one can configure a field to use the doc values. Doc values were discussed in the *Mappings configuration* section in *Chapter 2, Indexing Your Data*.



The amount of memory field data cache that is allowed to use can be controlled using the `indices.fielddata.cache.size` property. We can set it to an absolute value (for example, 2GB) or to a percentage of the memory given to an Elasticsearch instance (for example, 40%). Please note that these values are per node properties and not per index. Discarding parts of the cache to make room for other entries will result in a poor query performance, so it is advisable to have enough physical memory. Also, remember that by default, the field data cache size is not bounded. So, if we are not careful, we can have our cluster exploded.

We can also control the expiry time for the field data cache; again, by default, the field data cache does not expire. We can control this by using the `indices.fielddata.cache.expire` property, setting its value to a maximum inactivity time. For example, setting this property to `10m` will result in the cache being invalidated after 10 minutes of inactivity. Remember that rebuilding the field data cache is very expensive and in general, you shouldn't set the expiration time.

The circuit breaker

The field data circuit breaker allows the memory estimation that a field will require to be loaded into the memory. By using it, we can prevent loading such fields into the memory by raising an exception. Elasticsearch has two properties to control the behavior of the circuit breaker. First, we have the `indices.fielddata.breaker.limit` property, which defaults to 80% and can be updated dynamically by using the cluster update settings API. This means that an exception will be raised as soon as our query results in the loading of values for a field that is estimated to take 80 percent or more of the heap memory available to the Elasticsearch process. The second property is `indices.fielddata.breaker.overhead`, which defaults to `1.03`. It defines a constant value that will be used to multiply the original estimate for a field.

The store

The store module in Elasticsearch is responsible for controlling how the index data is written. Our index can be stored completely in the memory or in a persistent disk storage. The pure RAM-based index will be blazingly fast but volatile, while the disk-based index will be slower but tolerant to failure.

By using the `index.store.type` property, we can specify which store type we want to use for the index. The available options are as follows:

- `simplefs`: This is a disk-based storage that accesses the index files by using the random access files. It doesn't offer good performance for concurrent access and thus, it is not advised to be used in production.
- `niofs`: This is the second one of the disk-based index storages that uses Java NIO classes to access the index files. It offers very good performance in highly concurrent environments, but it is not advised to be used on Windows-based deployments because of the Java implementation bugs.
- `mmapfs`: This is another disk-based storage that maps index files in the memory (please have a look at what mmap is at <http://en.wikipedia.org/wiki/Mmap>). This is the default storage for 64-bit systems and allows a more efficient reading of the index because of the same operating system-based cache being used for index files access. You need to be sure to have a good amount of virtual address space, but on 64-bits systems, you shouldn't have problems with this.
- `memory`: This stores the index in RAM memory. Please remember that you need to have enough physical memory to store all the documents or Elasticsearch will fail.

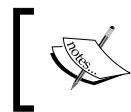
Index buffers and the refresh rate

When it comes to indices, Elasticsearch allows you to set the amount of memory that can be consumed for indexing purposes. The `indices.memory.index_buffer_size` property (defaults to 10%) allows us to control the total amount of memory (or a percentage of the maximum heap memory) that can be divided between the shards of all the indices on a given node. For example, setting this property to 20% will tell Elasticsearch to give 20 percent of the maximum heap size to index buffers.

In addition to this, we have `indices.memory.min_shard_index_buffer_size`, which defaults to 4mb and allows us to set the value of minimum indexing buffer per shard.

The index refresh rate

One last thing about the indices is the `index.refresh_interval` property specified per index. It defaults to 1s (one second) and specifies how often the index searcher object is refreshed, which basically means how often the data view is refreshed. The lower the refresh rate, the sooner the documents will be visible for search operations. However, it also means that Elasticsearch will need to put in more resources for refreshing the index view, which means that the indexing and searching operations will be slower.



For massive bulk indexing, for example, when reindexing your data, it is advisable to set the `index.refresh_interval` property to `-1` at the time of indexing.

The thread pool configuration

Elasticsearch uses several pools to allow control over how threads are handled and how far memory consumption is allowed for user requests.



The Java virtual machine allows an application to have multiple threads, concurrently running forks of application execution. For more information about Java threads, please refer to <http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>.

We are especially interested in the following types of thread pools exposed by Elasticsearch:

- `cache`: This is an unbounded thread pool that will create a thread for each incoming request.
- `fixed`: This is a thread pool that has a fixed size (specified by the `size` property) and allows you to specify a queue (specified by the `queue_size` property) that will be used to hold requests until there is a free thread that can execute a queue request. If Elasticsearch isn't able to put a new request in the queue (if the queue is full), the request will be rejected.

There are many thread pools, (we can specify the type we are configuring by specifying the `type` property); however, when it comes to performance, the most important are as follows:

- `index`: This thread pool is used to index and delete operations. Its `type` defaults to `fixed`, its `size` to the number of available processors, and the size of the queue to 300.
- `search`: This thread pool is used for search and count requests. Its `type` defaults to `fixed`, its `size` to the number of available processors multiplied by 3, and the size of the queue to 1000.
- `suggest`: This thread pool is used for suggest requests. Its `type` defaults to `fixed`, its `size` to the number of available processors, and the size of the queue to 1000.
- `get`: This thread pool is used for real-time GET requests. Its `type` defaults to `fixed`, its `size` to the number of available processors, and the size of the queue to 1000.

- `bulk`: As you can guess, this thread pool is used for bulk operations. Its `type` defaults to `fixed`, its `size` to the number of available processors, and the size of the queue to 50.
- `percolate`: This thread pool is used for percolation requests. Its `type` defaults to `fixed`, its `size` to the number of available processors, and the size of the queue to 1000.

For example, if we would like to configure the thread pool for indexing operations to be of the `fixed` type, have a size of 100, and a queue of 500, we would set the following in the `elasticsearch.yml` configuration file:

```
threadpool.index.type: fixed
threadpool.index.size: 100
threadpool.index.queue_size: 500
```

Remember that the thread-pool configuration can be updated using the cluster update API, as follows:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "threadpool.index.type" : "fixed",
    "threadpool.index.size" : 100,
    "threadpool.index.queue_size" : 500
  }
}'
```

Combining it all together – some general advice

Now that we know about the caches and buffers exposed by Elasticsearch, we can try combining this knowledge to configure the cluster for a high indexing and query throughput. In the next two sections, we will discuss what can be changed in the default configuration and what you should pay attention to when setting up your cluster.

Before we discuss all the things related to Elasticsearch specific configuration, we should remember that we have to give enough memory to Elasticsearch—physical memory. In general, we shouldn't give more than 50 to 60 percent of the total available memory to the JVM process running Elasticsearch. We do this because we want to leave some memory free for the operating system and for the operating system I/O cache.

However, we need to remember that the 50 to 60 percent is not always true. You can imagine having nodes with 256 GB of RAM and having an index with a total weight of 30 GB on that node. In such circumstances, even assigning more than 60 percent of the physical RAM to Elasticsearch would leave plenty of RAM for the operating system. It is also a good idea to set the `Xmx` and `Xms` arguments to the same values in order to avoid JVM heap size resizing.

One thing to remember when tuning your system is the performance tests that can be repeated under the same circumstances. Once you have made a change, you need to be able to see how it affects the overall performance. In addition to this, Elasticsearch scales, and because of this, it is sometimes a good thing to do a simple performance test on a single machine, see how it performs, and what we can get from it. Such observations may be a good starting point for further tuning.

Before we continue, note that we can't give you recipes for high indexing and querying because each deployment is different. Because of this, we will only discuss what you should pay attention to when tuning. However, if you are interested in such use cases, you can visit <http://blog.sematest.com> when one of the authors writes about performance.

Choosing the right store

Of course, apart from the physical memory, about which we've already talked, we should choose the right store implementation. In general, if you are running a 64-bit operating system, you should again go for `mmapfs`. If you are not running a 64-bit operating system, choose the `niofs` store for UNIX-based systems and `simplefs` for Windows-based ones. If you can allow yourself to have a volatile store, but want it to be very fast, you can look at the `memory` store; it will give you the best index access performance but requires enough memory to handle not only all the index files, but also to handle indexing and querying.

The index refresh rate

The second thing we should pay attention to is the index refresh rate. We know that the refresh rate specifies how fast documents will be visible for search operations. The equation is quite simple; the faster the refresh rate, the slower the queries will be and the lower the indexing throughput. If we can allow ourselves to have a slower refresh rate, such as 10s or 30s, it may be a good thing to set it. This puts less pressure on Elasticsearch as the internal objects will have to be reopened at a slower pace and thus, more resources will be available both for indexing and querying.

Tuning the thread pools

We really suggest tuning the default thread pools, especially for querying operations. After performance tests, you usually see when your cluster is overwhelmed with queries. This is the point when you should start rejecting the requests. We think that in most cases it is better to reject the request right away rather than put it in the queue and force the application to wait for very long periods of time to have that request processed. We would really like to give you a precise number, but that again highly depends on the deployment and general advice is rarely possible.

Tuning your merge process

The merge process is another thing that is highly dependent on your use case and also depends on several factors such as whether you are indexing, how much data you add, and how often you do that. In general, remember that queries against an index with multiple segments are slower than the ones with a smaller number of segments. But again, to have a smaller number of segments, you need to pay the price of merging more often.

We discussed segment merging in the *Introduction to segment merging* section of *Chapter 2, Indexing Your Data*. We also mentioned throttling, which allows us to limit the I/O operations.

Generally, if you want your queries to be faster, aim for fewer segments for your indices. If you want indexing to be faster, go for more segments for indices. If you want both of these things, you need to find a golden spot between these two so that the merging is not too often but also doesn't result in an extensive number of segments. Use concurrent merge scheduler and tune default throttling value so that your I/O subsystem is not overwhelmed by merging.

The field data cache and breaking the circuit

By default, the field data cache in Elasticsearch is unbound. This can be very dangerous, especially when you are using faceting and sorting on many fields. If the fields have high cardinality, you can run into even more trouble; by trouble, we mean you can run out of memory.

We have two different factors that we can tune to be sure that we don't run into out-of-memory errors. First, we can limit the size of the field data cache. The second is the circuit breaker, which we can easily configure to just throw an exception instead of loading too much data. Combining these two things will ensure that we don't run into memory issues.

However, we should also remember that Elasticsearch will evict data from the field data cache if its size is not enough to handle the faceting request or sorting. This will affect the query performance because loading the field data information is not very efficient. However, we think that it is better to have our queries slower than to have our cluster blown up because of the out-of-memory errors.

RAM buffer for indexing

Remember, the more the available RAM for indexing the buffer (the `indices.memory.index_buffer_size` property), the more documents Elasticsearch can hold in memory. But of course, we don't want to occupy 100 percent of the available memory with just Elasticsearch. By default, this is set to 10 percent, but if you really need a high indexing rate, you can increase the percentage. We've seen this property being set to 30 percent or some clusters that were focusing on data indexing and it really helped.

Tuning transaction logging

We haven't discussed this, but Elasticsearch has an internal module called **translog** (<http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/index-modules-translog.html>). It is a per-shard structure that serves the purpose of write-ahead logging (http://en.wikipedia.org/wiki/Write-ahead_logging). Basically, it allows Elasticsearch to expose the newest updates for GET operations, ensure data durability, and optimize the writing to Lucene indices.

By default, Elasticsearch keeps a maximum of 5000 operations in the transaction log with a maximum size of 200mb. However, if we can pay the price of the data not being available for search operations for longer periods of time but we want more indexing throughput, we can increase the defaults. By specifying the `index.translog.flush_threshold_ops` and `index.translog.flush_threshold_size` properties (both are set as per index and can be updated in real time using the Elasticsearch API), we can set the maximum number of operations allowed to be stored in the transaction log and its maximum size. We've seen deployments having this property value set to ten times the default value.

One thing to remember is that in case of a failure, shard initialization will be slower – of course, on the ones that had large transaction logs. This is because Elasticsearch needs to process all the information from the transaction log before the shard is ready for usage.

Things to keep in mind

Of course, the preceding mentioned factors are not everything that matters. You should monitor your Elasticsearch cluster and react accordingly to what you see. For example, if you see that the number of segments in your indices starts to grow and you don't want this, tune your merge policy. When you see merging taking too much I/O resources and affecting the overall performance, tune throttling. Just keep in mind that tuning won't be a one-time thing; your data will grow and so will your query number, and you'll have to adapt to that.

Templates and dynamic templates

In the *Mappings configuration* section of *Chapter 2, Indexing Your Data*, we read about mappings, how they are created, and how the type-determining mechanism works. Now we will get into more advanced topics; we will show you how to dynamically create mappings for new indices and how to apply some logic to the templates.

Templates

As we have seen earlier in the book, the index configuration and mappings in particular can be complicated beasts. It would be very nice if there was a possibility of defining one or more mappings once and using them in every newly created index without the need of sending them every time an index is created. Elasticsearch creators predicted this and implemented a feature called **index templates**. Each template defines a pattern, which is compared to a newly created index name. When both of them match, values defined in the template are copied to the index structure definition. When multiple templates match the name of the newly created index, all of them are applied and values from the later applied templates override the values defined in the previously applied templates. This is very convenient because we can define a few common settings in the more general templates and change them in the more specialized ones. In addition, there is an `order` parameter that lets us force the desired template ordering. You can think of templates as dynamic mappings that can be applied not to the types in documents, but to the indices.

An example of a template

Let's see a real example of a template. Imagine that we want to create many indices in which we don't want to store the source of the documents so that our indices will be smaller. We also don't need any replicas. We can create a template that matches our need by using the Elasticsearch REST API by sending the following command:

```
curl -XPUT http://localhost:9200/_template/main_template?pretty -d '{
```

```

"template" : "*",
"order" : 1,
"settings" : {
  "index.number_of_replicas" : 0
},
"mappings" : {
  "_default_" : {
    "_source" : {
      "enabled" : false
    }
  }
}
}'

```

From now on, all the created indices will have no replicas and no source stored. That's because the `template` parameter value is set to `*`, which matches all the names of the indices. Note the `_default_` type name in our example. This is a special type name, which indicates that the current rule should be applied to every document type. The second interesting thing is the `order` parameter. Let's define a second template by using the following command:

```

curl -XPUT http://localhost:9200/_template/ha_template?pretty -d '{
  "template" : "ha_*",
  "order" : 10,
  "settings" : {
    "index.number_of_replicas" : 5
  }
}'

```

After running the preceding command, all the new indices will behave as before except the ones with the names beginning with `ha_`. In case of these indices, both the templates are applied. First, the template with the lower `order` value is used and then the next template overwrites the replica's setting. So, indices whose names start with `ha_` will have five replicas and disabled sources stored.

Storing templates in files

Templates can also be stored in files. By default, files should be placed in the `config/templates` directory. For example, our `ha_template` template should be placed in the `config/templates/ha_template.json` file and have the following content:

```
{
  "ha_template" : {

```

```
"template" : "ha_",
"order" : 10,
"settings" : {
    "index.number_of_replicas" : 5
}
}
```

Note that the structure of JSON is a little bit different and has the template name as the main object key. The second important thing is that templates must be placed on every instance of Elasticsearch. Also, the templates defined in the files are not available with the REST API calls.

Dynamic templates

Sometimes, we want to have a possibility of defining type that is dependent on the field name and the type. This is where dynamic templates can help. Dynamic templates are similar to the usual mappings, but each template has its pattern defined, which is applied to a document's field name. If a field name matches the pattern, the template is used. Let's have a look at the following example:

```
{
  "mappings" : {
    "article" : {
      "dynamic_templates" : [
        {
          "template_test" : {
            "match" : "*",
            "mapping" : {
              "index" : "analyzed",
              "fields" : {
                "str": {"type": "{dynamic_type}",
                         "index": "not_analyzed" }
              }
            }
          }
        }
      ]
    }
  }
}
```

```
}
```

In the preceding example, we defined a mapping for the `article` type. In this mapping, we have only one dynamic template named `template_test`. This template is applied for every field in the input document because of the single asterisk pattern in the `match` property. Each field will be treated as a multifield, consisting of a field named as the original field (for example, `title`) and the second field with a name suffixed with `str` (for example, `title.str`). The first field will have its type determined by Elasticsearch (with the `{dynamic_type}` type), and the second field will be a string (because of the string type).

The matching pattern

We have two ways of defining the matching pattern; they are as follows:

- `match`: This template is used if the name of the field matches the pattern (this pattern type was used in our example)
- `unmatch`: This template is used if the name of the field doesn't match the pattern

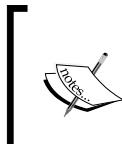
By default, the pattern is very simple and uses glob patterns. This can be changed by using `match_pattern=regexp`. After adding this property, we can use all the magic provided by regular expressions to match and unmatch patterns.

There are variations such as `path_match` and `path_unmatch` that can be used to match the names in nested documents.

Field definitions

When writing a target field definition, the following variables can be used:

- `{name}`: The name of the original field found in the input document
- `{dynamic_type}`: The type determined from the original document



Please note that Elasticsearch checks templates in the order of their definitions and the first matching template is applied. This means that the most generic templates (for example, with "match": "*") must be defined at the end.

Summary

In this chapter, we learned a few things about Elasticsearch such as the node discovery, what this module is responsible for, and how to tune it. We also looked at the recovery and gateway modules, how to set them up to match our cluster, and what configuration options they provide. We also discussed some of the Elasticsearch internals, and we used these to tune our cluster for high indexing and high querying use cases. And finally, we've used templates and dynamic mappings to help us manage our dynamic indices better.

In the next chapter, we'll focus on some of the Elasticsearch administration capabilities. We will learn how to back up our cluster data and how to monitor our cluster using the available API calls. We will discuss how to control shard allocation and how to move shards around the cluster, again using the Elasticsearch API. We'll learn what index warmers are and how they can help us, and we will use aliases. Finally, we'll learn how to install and manage Elasticsearch plugins and what we can do using the update settings API.

8

Administrating Your Cluster

In the previous chapter, we learned how node discovery works in Elasticsearch and how to tune it. We also learned about the recovery and gateway modules. We saw how to prepare our cluster for high indexing and querying use cases by tuning some of the Elasticsearch internals and what those internals are. Finally, we used index templates and dynamic mappings to easily control dynamic indices' structure. By the end of this chapter, you will learn the following aspects:

- Using Elasticsearch snapshotting functionality
- Monitoring our cluster using the Elasticsearch API
- Adjusting cluster rebalancing to match our needs
- Moving shards around by using the Elasticsearch API
- Warming up
- Using aliasing to ease the everyday work
- Installing the Elasticsearch plugins
- Using the Elasticsearch update settings API

The Elasticsearch time machine

A good piece of software is one that manages an exceptional situation such as hardware failure or human error. Even though a cluster of a few servers is less exposed to hardware problems, bad things can still happen. For example, let's imagine that you need to restore your indices. One possible solution is to reindex all your data from a primary data store as a SQL database. But what will you do if it takes too long or, even worse, the only data store is Elasticsearch? Before Elasticsearch 1.0, creating backups of indices was not easy. The procedure included shutdown of the cluster before copying the data files. Fortunately, now we can take snapshots. Let's see how this works.

Creating a snapshot repository

A snapshot keeps all the data related to the cluster from the time snapshot creation starts and it includes information about the cluster state and indices. Before we create snapshots, at least the first one, a snapshot repository must be created. Each repository is recognized by its name and should define the following aspects:

- **name:** This is a unique name of the repository; we will need it later.
- **type:** This is the type of the repository. The possible values are `fs` (repository on a shared filesystem) and `url` (read-only repository available via URL).
- **settings:** This is the additional information needed depending on the repository type.

Now, let's create a filesystem repository. Please note that every node in the cluster should be able to access this directory. To create a new filesystem repository, we can run a command shown as follows:

```
curl -XPUT localhost:9200/_snapshot/backup -d '{  
  "type": "fs",  
  "settings": {  
    "location": "/tmp/es_backup_folder/cluster1"  
  }  
'
```

The preceding command creates a repository named `backup`, which stores the backup files in the directory given by the `location` attribute. Elasticsearch responds with the following information:

```
{"acknowledged":true}
```

At the same time, `backup_folder` on the local filesystem is created – without any content yet.

As we said, the second repository type is `url`. It requires a `url` parameter instead of `location`, which points to the address where the repository resides, for example, the HTTP address. You can also store snapshots in Amazon S3 or HDFS using the additional plugins available (see <https://github.com/elasticsearch/elasticsearch-cloud-aws#s3-repository> and <https://github.com/elasticsearch/elasticsearch-hadoop/tree/master/repository-hdfs>).



Now that we have our first repository, we can see its definition using the following command:

```
curl -XGET localhost:9200/_snapshot/backup?pretty
```

We can also check all the repositories by running a command like the following:

```
curl -XGET localhost:9200/_snapshot/_all?pretty
```

If you want to delete a snapshot repository, the standard `DELETE` command helps:

```
curl -XDELETE localhost:9200/_snapshot/backup?pretty
```

Creating snapshots

By default, Elasticsearch takes all the indices and cluster settings (except the transient ones) when creating snapshots. You can create any number of snapshots, and each will hold information available right from the time when the snapshot was created. The snapshots are created in a smart way; only new information is copied. It means that Elasticsearch knows which segments are already stored in the repository and doesn't save them again.

To create a new snapshot, we need to choose a unique name and use the following command:

```
curl -XPUT 'localhost:9200/_snapshot/backup/bckp1'
```

The preceding command defines a new snapshot named `bckp1` (you can only have one snapshot with a given name; Elasticsearch will check its uniqueness), and data is stored in the previously defined backup repository. The command returns an immediate response, which looks as follows:

```
{"accepted":true}
```

The preceding response means that the process of snapshotting has started and continues in the background. If you would like the response to be returned only when the actual snapshot is created, we can add the `wait_for_completion` parameter as shown in the following example:

```
curl -XPUT 'localhost:9200/_snapshot/backup/bckp2?wait_for_completion=true&pretty'
```

The response to the preceding command shows the status of a created snapshot:

```
{  
  "snapshot" : {  
    "snapshot" : "bckp2",  
    "indices" : [ "art" ],  
    "state" : "SUCCESS",  
    "start_time" : "2014-02-22T13:04:40.770Z",  
    "start_time_in_millis" : 1393074280770,  
    "end_time" : "2014-02-22T13:04:40.781Z",  
    "end_time_in_millis" : 1393074280781,  
    "duration_in_millis" : 11,  
    "failures" : [ ],  
    "shards" : {  
      "total" : 5,  
      "failed" : 0,  
      "successful" : 5  
    }  
  }  
}
```

As we can see, Elasticsearch presents information about the time taken by the snapshotting process, its status, and the indices affected.

Additional parameters

The snapshot command also accepts the following additional parameters:

- `indices`: These are the names of the indices of which we want to take snapshots.
- `ignore_unavailable`: When this is set to `false` (the default is `true`), meaning that when the `indices` parameter points to the nonexistent index, the command will fail.
- `include_global_state`: When this is set to `true` (the default), the cluster state is also written in the snapshot (except for the transient settings).
- `partial`: The snapshot success depends on the availability of all the shards. If any of the shards are not available, the snapshot fails. Setting `partial` to `true` causes Elasticsearch to save only the available shards and omit the lost ones.

An example of using additional parameters can look as follows:

```
curl -XPUT 'localhost:9200/_snapshot/backup/bckp?wait_for_completion=true&pretty' -d '{ "indices": "b*", "include_global_state": "false" }'
```

Restoring a snapshot

Now that we have our snapshots done, we will also learn how to restore data from a given snapshot. As we said earlier, a snapshot can be addressed by its name. We can list all the snapshots by using the following command:

```
curl -XGET 'localhost:9200/_snapshot/backup/_all?pretty'
```

The repository we created earlier is called `backup`. To restore a snapshot named `bckp1` from our snapshot repository, run the following command:

```
curl -XPOST 'localhost:9200/_snapshot/backup/bckp1/_restore'
```

During the execution of this command, Elasticsearch takes indices defined in the snapshot and creates them with the data from the snapshot. However, if the index already exists and is not closed, the command will fail. In this case, you may find it convenient to only restore certain indices, for example:

```
curl -XPOST 'localhost:9200/_snapshot/backup/bck1/_restore?pretty' -d '{ "indices": "c*" }'
```

The preceding command restores only the indices that begin with the letter `c`. The other available parameters are as follows:

- `ignore_unavailable`: This is the same as in the snapshot creation.
- `include_global_state`: This is the same as in the snapshot creation.
- `rename_pattern`: This allows you to change the name of the index stored in the snapshot. Thanks to this, the restored index will have a different name. The value of this parameter is a regular expression that defines source index name. If a pattern matches the name of the index, the name substitution will occur. In the pattern, you should use groups limited by parentheses used in the `rename_replacement` parameter.
- `rename_replacement`: This along with `rename_pattern` defines the target index name. Using the dollar sign and number, you can recall the appropriate group from `rename_pattern`.

For example, due to `rename_pattern=products_(.*)`, only the indices with names that begin with `products_` will be restored. The rest of the index name will be used in replacement. Together with `rename_replacement=items_$1` causes that `products_cars` index will be restored to an index called `items_cars`.

Cleaning up – deleting old snapshots

Elasticsearch leaves snapshot repository management up to you. Currently, there is no automatic cleanup process. But don't worry; this is simple. For example, let's remove our previously taken snapshot:

```
curl -XDELETE 'localhost:9200/_snapshot/backup/bckp1?pretty'
```

And that's all. The command causes the snapshot named `bckp1` from the `backup` repository to be deleted.

Monitoring your cluster's state and health

During the normal life of an application, a very important aspect is monitoring. This allows the administrators of the system to detect possible problems, prevent them before they occur, or at least know what happens during a failure.

Elasticsearch provides very detailed information that allows you to check and monitor a node or the cluster as a whole. This includes statistics, information about servers, nodes, indices, and shards. Of course, we are also able to get information about the whole cluster state. Before we get into details about the mentioned API, please remember that the API is complex and we are only describing the basics. We will try to show you when to start, so you'll be able to know what to look for when you need very detailed information.

The cluster health API

One of the most basic APIs is the cluster health API that allows us to get information about the whole cluster state with a single HTTP command. For example, let's run the following command:

```
curl 'localhost:9200/_cluster/health?pretty'
```

A sample response returned by Elasticsearch for the preceding command looks as follows:

```
{  
  "cluster_name" : "es-book",
```

```
"status" : "green",
"timed_out" : false,
"number_of_nodes" : 1,
"number_of_data_nodes" : 1,
"active_primary_shards" : 4,
"active_shards" : 4,
"relocating_shards" : 0,
"initializing_shards" : 0,
"unassigned_shards" : 0
}
```

The most important information is the one about the status of the cluster. In our example, we see that the cluster is in the green status. This means that all the shards have been allocated properly and there were no errors.

Let's stop here and talk about the cluster and when, as a whole, it will be fully operational. The cluster is fully operational when Elasticsearch is able to allocate all the shards and replicas according to the configuration. When this happens, the cluster is in the green state. The yellow state means that we are ready to handle requests because the primary shards are allocated but some (or all) replicas are not. The last state, the red one, means that at least one primary shard was not allocated and because of this, the cluster is not ready yet. This means that the queries may return errors or incomplete results.

The preceding command can also be executed to check the health state of a certain index. For example, if we want to check the health of the library and map indices, we will run the following command:

```
curl 'localhost:9200/_cluster/health/library,map/?pretty'
```

Controlling information details

Elasticsearch allows us to specify a special `level` parameter that can take the value of `cluster` (default), `indices`, or `shards`. This allows us to control the details of the information returned by the health API. We've already seen the default behavior. When setting the `level` parameter to `indices`, apart from the cluster information, we will also get the per index health. Setting the mentioned parameter to `shards` tells Elasticsearch to return per shard information in addition to what we've seen in the example.

Additional parameters

In addition to the `level` parameter, we have a few additional parameters that can control the behavior of the health API.

The first of the mentioned parameters is `timeout`. It allows us to control the maximum time for which the command execution will wait. By default, it is set to `30s` and means that the health command will wait for a maximum of 30 seconds and will return the response by then.

The `wait_for_status` parameter allows us to tell Elasticsearch which health status the cluster should be at to return the command. It can take the values `green`, `yellow`, and `red`. For example, when set to `green`, the health API call returns the results until the green status or timeout is reached.

The `wait_for_nodes` parameter allows us to set the required number of nodes available to return the health command response (or until a defined timeout is reached). It can be set to an integer number like `3` or to a simple equation like `>=3` (means more than or equal to three nodes) or `<=3` (means less than or equal to three nodes).

The last parameter is `wait_for_relocating_shard`, which is not specified by default. It allows us to tell Elasticsearch how many relocating shards it should wait for (or until the timeout is reached). Setting this parameter to `0` means that Elasticsearch should wait for all the relocating shards.

An example usage of the health command with some of the mentioned parameters is as follows:

```
curl 'localhost:9200/_cluster/health?wait_for_status=green&wait_for_nodes=>=3&timeout=100s'
```

The indices stats API

Elasticsearch index is the place where our data lives, and it is a crucial part for most deployments. With the use of the indices stats API available using the `_stats` endpoint, we can get various information about the indices living inside our cluster. Of course, as with most of the APIs in Elasticsearch, we can give a command to get information about all the indices (using the pure `_stats` endpoint), about one particular index (for example, `library/_stats`), or several indices at the same time (for example, `library, map/_stats`). For example, to check the statistics for the `map` and `library` indices we've used in the book, we could run the following command:

```
curl localhost:9200/library,map/_stats?pretty
```

The response to the preceding command has more than 500 lines, so we only describe its structure and omit the response itself. Apart from the information about the response status and the response time, we can see three objects named `primaries`, `total`, and `indices`. The `indices` object contains information about the `library` and `map` indices. The `primaries` object contains information about the primary shards allocated on the current node, and the `total` object contains information about all the shards including replicas. All these objects can contain objects that describe a particular statistic, such as `docs`, `store`, `indexing`, `get`, `search`, `merges`, `refresh`, `flush`, `warmer`, `filter_cache`, `id_cache`, `fielddata`, `percolate`, `completion`, `segments`, and `translog`. Let's discuss the information stored in these objects.

Docs

The `docs` section of the response shows information about the indexed documents. For example, it could look as follows:

```
"docs" : {  
    "count" : 4,  
    "deleted" : 0  
}
```

The main information is the `count`, indicating the number of documents. When we delete documents from the index, Elasticsearch doesn't remove these documents immediately but only marks them as deleted. The documents are physically deleted in a segment merge process. The number of documents marked as deleted are presented as a `deleted` attribute and should be 0 right after the merge.

Store

The next set of statistics, `store`, provides information regarding storage. For example, such a section could look as follows:

```
"store" : {  
    "size_in_bytes" : 6003,  
    "throttle_time_in_millis" : 0  
}
```

The main information is about the index (or indices) size. We can also look at throttling statistics. This information is useful when the system has problems with the I/O performance and has configured limits on internal operation during segments merge.

Indexing, get, and search

The indexing, get, and search sections of the response provide information about data manipulation: indexing with delete operations, using real-time get, and searching. Let's look at the following example returned by Elasticsearch:

```
"indexing" : {
  "index_total" : 11501,
  "index_time_in_millis" : 4574,
  "index_current" : 0,
  "delete_total" : 0,
  "delete_time_in_millis" : 0,
  "delete_current" : 0
},
"get" : {
  "total" : 3,
  "time_in_millis" : 0,
  "exists_total" : 2,
  "exists_time_in_millis" : 0,
  "missing_total" : 1,
  "missing_time_in_millis" : 0,
  "current" : 0
},
"search" : {
  "query_total" : 0,
  "query_time_in_millis" : 0,
  "query_current" : 0,
  "fetch_total" : 0,
  "fetch_time_in_millis" : 0,
  "fetch_current" : 0
}
```

As we can see, all of these statistics have a similar structure. We can read the total time spent in the various request types (in milliseconds) and the number of requests that the total time allows you to calculate the average time of a single query. In the case of real time, the get requests valuable information is the number of fetches that were unsuccessful (missing documents).

Additional information

In addition, Elasticsearch provides the following information:

- `merges`: This section contains information about Lucene segment merges
- `refresh`: This section contains information about refresh operations
- `flush`: This section contains information about flushes
- `warmer`: This section contains information about warmers and how long they were executed
- `filter_cache`: These are the filter cache statistics
- `id_cache`: These are the identifiers cache statistics
- `fielddata`: These are the field data cache statistics
- `percolate`: This section contains information about the percolator usage
- `completion`: This section contains information about the completion suggester
- `segments`: This section contains information about Lucene segments
- `translog`: This section contains information about transaction logs count and size

The status API

Another way to obtain information about indices is the status API available by using the `_status` endpoint. The information that is returned describes the available shards and includes information on which shard is currently considered primary, which node it is assigned to, which node it is reallocated to (if it is), status of the shard (is it active or not), information about the transaction log and the merge process, and the refresh and flush statistics.

The nodes info API

The nodes info API provides us with the information about the nodes in the cluster. To get information from this API, we need to send the request to the `_nodes` REST endpoints.

This API can be used to fetch information about particular nodes or a single node using the following aspects:

- **Node name:** If we want to get information about the node named `Pulse`, we can run a command to the `_nodes/_nodes/Pulse` REST endpoint
- **Node identifier:** If we want to get information about the node with an identifier equal to `ny4hftjNQtuKMyEvpUdQWg`, we can run a command to the `_nodes/_nodes/ny4hftjNQtuKMyEvpUdQWg` REST endpoint
- **IP address:** If we want to get information about the node with an IP address equal to `192.168.1.103`, we can run a command to the `_nodes/_nodes/192.168.1.103` REST endpoint
- **Parameters from the Elasticsearch configuration:** If we want to get information about all the nodes with the `node.rack` property set to `2`, we can run a command to the `/_nodes/rack:2` REST endpoint

This API also allows you to get information about several nodes at once by using the following:

- Patterns, for example, `_nodes/_nodes/192.168.1.*` or `_nodes/_nodes/P*`
- Nodes enumeration, for example, `_nodes/_nodes/Pulse,Slab`
- Both patterns and enumerations, for example, `/_nodes/_nodes/P*,S*`

By default, the request to the nodes API will return the basic information about a node, such as name, identifier, and its address. But by adding additional parameters, we can obtain other information. The available parameters are as follows:

- `settings`: This parameter is used to get Elasticsearch configuration
- `os`: This parameter is used to get information about the server, such as processor, RAM, and swap space
- `process`: This parameter is used to get the process identifier and the available file descriptors
- `jvm`: This parameter is used to get information about the **Java virtual machine (JVM)**, such as the memory limit
- `thread_pool`: This parameter is used to get the configuration of the thread pools for various operations
- `network`: This parameter is used to get name and addresses of the network interface
- `transport`: This parameter is used to get the listen addresses for transport

- `http`: This parameter is used to get the listen addresses for HTTP
- `plugins`: This parameter is used to get information about the installed plugins

An example usage of the earlier described API can be illustrated by the following command:

```
curl 'localhost:9200/_nodes/Pulse/os,jvm,plugins?pretty'
```

The preceding command will return information related to the operating system, Java virtual machine, and plugins in addition to the basic information. Of course, all the information will be about the nodes named `Pulse`.

The nodes stats API

The nodes stats API is similar to the nodes info API described previously. The main difference is that the previous API provides information about the environment, and the one we are currently discussing tells us what happened with the cluster during its work. To use the nodes stats API, one needs to send a command to the `/_nodes/stats` REST endpoint. However, similar to the nodes info API, we can also retrieve information about specific nodes (for example, `_nodes/Pulse/stats`).

By default, Elasticsearch returns all the available statistics, but we can limit it to the ones we are interested in. The available options are as follows:

- `indices`: This provides information about the indices, including size, document count, indexing-related statistics, search and get time, caches, segment merges, and so on
- `os`: This provides operating system related information, such as free disk space, memory, and swap usage
- `process`: This provides information about the memory, CPU, and file handler usage related to the Elasticsearch process
- `jvm`: This provides information about the Java virtual machine memory and garbage collector statistics
- `network`: This provides information about the TCP-level information
- `transport`: This provides information about data sent and received by the transport module
- `http`: This provides information about HTTP connections
- `fs`: This provides information about the available disk space and I/O operations statistics

- `thread_pool`: This provides information about the state of the threads assigned to various operations
- `breaker`: This provides information about the field data cache circuit breaker

An example usage of the previously described API can be illustrated by the following command:

```
curl 'localhost:9200/_nodes/Pulse/stats/os,jvm,breaker?pretty'
```

The cluster state API

Another API provided by Elasticsearch is the cluster state API. As the name suggests, it allows us to get information about the whole cluster. (We can also limit the returned information to a local node by adding the `local=true` parameter to the request.) The basic command used to get all the information retuned by the discussed API looks as follows:

```
curl 'localhost:9200/_cluster/state?pretty'
```

However, we can also limit the provided information to the given metrics (separated by commas and specified after the `_cluster/state` part of the REST call) and to the given indices (again separated by commas and specified after the `_cluster/state/metrics` part of the REST call). An example call that would only return node related information about the `map` and `library` indices could look as follows:

```
curl 'localhost:9200/_cluster/state/nodes/map,library?pretty'
```

The following metrics can be used:

- `version`: This returns information about the cluster state version.
- `master_node`: This returns information about the elected master node.
- `nodes`: This returns information on nodes.
- `routing_table`: This returns routing-related information.
- `metadata`: This returns metadata-related information. When specifying the retrieval of the metadata metric, we can also include an additional parameter, `index_templates=true`, which will result in the defined index templates being included.
- `blocks`: This returns the blocks part of the response.

The pending tasks API

One of the APIs introduced in Elasticsearch 1.0 is the pending tasks API, which allows us to check which tasks are waiting to be executed. To retrieve this information, we need to send a request to the `/_cluster/pending_tasks` REST endpoint. In the response, we will see an array of tasks with information about them, such as task priority and time in queue.

The indices segments API

The last API we wanted to mention is the Lucene segments API available by using the `/_segments` endpoint. We can run it for the whole cluster and for individual indices too. This API provides information about shards, their placement, and the segments connected with the physical index managed by the Lucene library.

The cat API

Of course, we may say that all the information we need to diagnose and observe the cluster can be retrieved by using the provided API. However, the response returned by the API is in JSON; great, but not especially convenient to use at least for a human being. That's why Elasticsearch allows us to use a friendlier API: the cat API.

To use the cat API, one needs to send a request to the `_cat` REST endpoint followed by one of the options, which are as follows:

- `aliases`: This returns information about aliases (we'll learn about aliases in the *Index aliasing and using it to simplify your everyday work* section of this chapter)
- `allocation`: This returns information about the allocated shards and disk usage
- `count`: This returns information about the document count for all the indices or an individual one
- `health`: This returns information about cluster health
- `indices`: This returns information about all the indices or an individual one
- `master`: This returns information about the elected master node
- `nodes`: This returns information about the cluster topology

- `pending_tasks`: This returns information about the tasks that are waiting to be executed
- `recovery`: This provides a view of the recovery process
- `thread_pool`: This provides cluster wide statistics regarding thread pools
- `shards`: This returns information about shards

This may be a bit confusing, so let's have a look at an example command that would return information about shards. The command that would allow us to get this information is given as follows:

```
curl -XGET 'localhost:9200/_cat/shards?v'
```

 Note that we've included the `v` parameter to the request. This means that we want the information to be more verbose, for example, including the header. In addition to the `v` parameter, we can also use the `help` parameter, which will return the header's description for a given command and the `h` parameter, which accepts a comma-separated list of the columns we want to include in the response.

And the response to the preceding command will look as follows:

```
index shard prirep state  docs  store ip           node
map      0     p      STARTED    4  5.9kb 192.168.1.40 es_node_1
library  0     p      STARTED    9 11.8kb 192.168.56.1 es_node_2
```

We can see that we have two indices, each with a single shard. We can also see the ID of the shard, that is, whether it is a primary shard, its state, number of documents, its size, node IP address, and the node name.

Limits returned information

Some of the cat API commands allow us to limit the information they return. For example, the `aliases` call allows us to get information about a specific alias by appending the alias name just like in the following command:

```
curl -XGET 'localhost:9200/_cat/aliases/current_index'
```

Let's summarize the commands that allow information limiting:

- `aliases`: This limits the information to a specific alias by appending the alias in the request
- `count`: This limits the information to a specific index by appending the index name we are interested in to the request

- `indices`: This limits the information to a specific index just like in the `count` command
- `shards`: This limits the information to a specific index by appending the index name we are interested in

Controlling cluster rebalancing

By default, Elasticsearch tries to keep the shards and their replicas evenly balanced across the cluster. Such behavior is good in most cases, but there are times when we would want to control this behavior. In this section, we will look at how to avoid cluster rebalance and how to control this process' behavior in depth.

Imagine a situation where you know that your network can handle a very high amount of traffic, or the opposite; your network is used extensively and you want to avoid too much stress on it. The other example is that you may want to decrease the pressure that is put on your I/O subsystem after a full-cluster restart, and you want to have less shards and replicas being initialized at the same time. These are only two examples where rebalance control may be handy.

Rebalancing

Rebalancing is the process of moving shards between the different nodes in your cluster. As we have already mentioned, it is fine in most situations, but sometimes you may want to completely avoid this. For example, if we define how our shards are placed and we want to keep it that way, we want to avoid rebalancing. However, by default, Elasticsearch will try to rebalance the cluster whenever the cluster state changes and Elasticsearch thinks rebalance is needed.

Cluster being ready

We already know that our indices can be built of shards and replicas. Primary shards or just shards are the ones that are used when the new documents are indexed, there is an update or delete, or just in case of any index change. We also have replicas that get the data from the primary shards.

You can think of the cluster as being ready to be used when all the primary shards are assigned to their nodes in your cluster – as soon as the yellow health state is achieved. However, Elasticsearch may still initialize other shards: the replicas. However, you can use your cluster, and be sure that you can search your whole data set and you can send index change commands. Then, those will be processed properly.

The cluster rebalance settings

Elasticsearch lets us control the rebalance process with the use of a few properties that can be set in the `elasticsearch.yml` file or by using the Elasticsearch REST API (described in the *The update settings API* section of this chapter).

Controlling when rebalancing will start

The `cluster.routing.allocation.allow_rebalance` property allows us to specify when rebalancing will be started. This property can take the following values:

- `always`: This value indicates that rebalancing will start as soon as it's needed
- `indices_primaries_active`: This value indicates that rebalancing will start when all the primary shards are initialized
- `indices_all_active`: This is the default value, which means that rebalancing will start when all the shards and replicas are initialized

Controlling the number of shards being moved between nodes concurrently

The `cluster.routing.allocation.cluster_concurrent_rebalance` property allows us to specify how many shards can be moved between nodes at once in the whole cluster. If you have a cluster that is built of many nodes, you can increase this value. This value defaults to 2.

Controlling the number of shards initialized concurrently on a single node

The `cluster.routing.allocation.node_concurrent_recoveries` property lets us set the number of shards that Elasticsearch is allowed to initialize on a single node at once. Please note that the shard recovery process is very I/O intensive, so you'll probably want to avoid too many shards being recovered concurrently. This value defaults to the same value as the previous one: 2.

Controlling the number of primary shards initialized concurrently on a single node

The `cluster.routing.allocation.node_initial_primaries_recoveries` property lets us control how many primary shards are allowed to be concurrently initialized on a node.

Controlling types of shards allocation

By using the `cluster.routing.allocation.enable` property, we can control what kind of shards are allowed to be allocated. The mentioned property can take the following values:

- `all`: This is the default value, which tells Elasticsearch that all types of shards are allowed to be allocated
- `primaries`: This tells Elasticsearch that it should only allocate primary shards and leave the replicas that are not allocated
- `new_primaries`: This tells Elasticsearch that only the newly created primary shards can be allocated
- `none`: This disables shard allocation completely

Controlling the number of concurrent streams on a single node

The `indices.recovery.concurrent_streams` property allows us to control how many streams are allowed to be opened on a node at once, in order to recover a shard from the target shards. It defaults to 3. If you know that your network and nodes can handle more, you can increase the value.

Controlling the shard and replica allocation

Indices that live inside your Elasticsearch cluster can be built of many shards, and each shard can have many replicas. With the ability to have multiple shards of a single index, we can deal with indices that are too large to fit on a single machine. The reasons may be different, from memory and CPU related to storage ones. With the ability to have multiple replicas of each shard, we can handle a higher query load by spreading the replicas over multiple servers. We can say that by using shards and replicas, we can scale out Elasticsearch. However, Elasticsearch has to figure out where in the cluster it should place the shards and replicas. It needs to figure out which server/nodes each shard or replica should be placed on.

Explicitly controlling allocation

Imagine that we want to have our indices to be placed on different cluster nodes. For example, we want one index named `shop` to be placed on some nodes and the second index called `users` to be placed on other nodes. Finally, the last index called `promotions` to be placed on all the nodes that `users` and `shop` indices were placed on. We may want to do this because of performance reasons. We know that some of the servers on which we installed Elasticsearch are more powerful than the others. With the default Elasticsearch behavior, we can't be sure where the shards and replicas will be placed, but luckily, Elasticsearch allows us to control that.

Specifying node parameters

So let's divide our cluster into two zones. We say zones, but it can be any name you want; we just like zone. We will assume that we have four nodes. We want our more powerful nodes numbered 1 and 2 to be placed in a zone called `zone_one`, and the nodes numbered 3 and 4, which are smaller in terms of resources, to be placed in a zone called `zone_two`.

Configuration

To achieve our described indices distribution, we add the following property to the `elasticsearch.yml` configuration file on node 1 and 2 (the nodes that are more powerful):

```
node.zone: zone_one
```

Of course, we will add a similar property to the `elasticsearch.yml` configuration file on node 3 and 4 (the less powerful nodes):

```
node.zone: zone_two
```

Index creation

Now let's create our indices. First, let's create the `shop` index. Place this index on the more powerful nodes. We can do this by running the following commands:

```
curl -XPUT 'http://localhost:9200/shop' -d '{
  "settings" : {
    "index" : {
      "routing.allocation.include.zone" : "zone_one"
    }
  }
}'
```

The preceding command will result in the `shop` index being created and the `index.routing.allocation.include.zone` property being specified to it. We set this property to the `zone_one` value, which means that we want to place the `shop` index on the nodes that have the `node.zone` property set to `zone_one`.

We perform similar steps for the `users` index:

```
curl -XPUT 'http://localhost:9200/users' -d '{  
  "settings" : {  
    "index" : {  
      "routing.allocation.include.zone" : "zone_two"  
    }  
  }  
}'
```

However, this time we've specified that we want the `users` index to be placed on the nodes with the `node.zone` property set to `zone_two`.

Finally, the `promotions` index should be placed in all the preceding nodes, so we will use the following command to create and configure this index:

```
curl -XPOST 'http://localhost:9200/promotions'  
curl -XPUT 'http://localhost:9200/promotions/_settings' -d '{  
  "index.routing.allocation.include.zone" : "zone_one,zone_two"  
}'
```

This time we've used a different set of commands. The first one creates the index, and the second one updates the `index.routing.allocation.include.zone` property. We did this just to illustrate that it can be done in such a way.

Excluding nodes from allocation

In the same manner as how we specified on which nodes the index should be placed, we can also exclude nodes from index allocation. Referring to the previously shown example, if we would like the index called `pictures` to not be placed on nodes with the `node.zone` property set to `zone_one`, we would run the following command:

```
curl -XPUT 'localhost:9200/pictures/_settings' -d '{  
  "index.routing.allocation.exclude.zone" : "zone_one"  
}'
```

Notice that instead of the `index.routing.allocation.include.zone` property, we've used the `index.routing.allocation.exclude.zone` property.

Requiring node attributes

In addition to inclusion and exclusion rules, we can also specify the rules that must match for a shard to be allocated to a given node. The difference is that when using the `index.routing.allocation.include` property, the index will be placed on any node that matches at least one of the provided property values. By using the `index.routing.allocation.require` property, Elasticsearch will place the index on a node that has all the defined values. For example, let's assume that we've set the following settings for the pictures index:

```
curl -XPUT 'localhost:9200/pictures/_settings' -d '{  
  "index.routing.allocation.require.size" : "big_node",  
  "index.routing.allocation.require.zone" : "zone_one"  
}'
```

After running the preceding command, Elasticsearch would only place the shards of the pictures index on a node with the `node.size` property set to `big_node` and the `node.zone` property set to `big_node`.

Using IP addresses for shard allocation

Instead of adding a special parameter to the nodes' configuration, we can use IP addresses to specify which nodes we want to include or exclude from the shards and replicas allocation. To do this instead of using the zone part of the `index.routing.allocation.include.zone` or `index.routing.allocation.exclude.zone` properties, we use `_ip`. For example, if we would like our `shop` index to be placed only on the nodes with IP address `10.1.2.10` and `10.1.2.11`, we will run the following command:

```
curl -XPUT 'localhost:9200/shop/_settings' -d '{  
  "index.routing.allocation.include._ip" : "10.1.2.10,10.1.2.11"  
}'
```

Disk-based shard allocation

In addition to the already described allocation filtering methods, Elasticsearch 1.0 brings one additional method: the disk-based one. It allows us to set allocation rules based on the node's disk usage, so we won't run out of disk space or something similar.

Enabling disk-based shard allocation

The disk-based shard allocation is disabled by default. We can enable it by specifying the `cluster.routing.allocation.disk.threshold_enabled` property and setting it to true. We can do this in the `elasticsearch.yml` file or by dynamically using the cluster settings API (you can read about it in the *The update settings API* section of this chapter):

```
curl -XPUT localhost:9200/_cluster/settings -d '{
  "transient" : {
    "cluster.routing.allocation.disk.threshold_enabled" : true
  }
}'
```

Configuring disk-based shard allocation

There are three properties that control the behavior of disk-based shard allocation. All of them can be updated dynamically or set in the `elasticsearch.yml` configuration file.

The first of these is `cluster.info.update.interval`, which is by default set to 30 seconds and defines how often Elasticsearch updates information about disk usage on the nodes.

The second property is `cluster.routing.allocation.disk.watermark.low`, which is by default set to 0.70. This means that Elasticsearch will not allocate new shards to a node that uses more than 70 percent of its disk space.

The third property is `cluster.routing.allocation.disk.watermark.high`, which controls when Elasticsearch will start relocating shards from a given node. It defaults to 0.85 and means that Elasticsearch will start reallocating shards when the disk usage on a given node is equal to or more than 85 percent.

Both the `cluster.routing.allocation.disk.watermark.low` and `cluster.routing.allocation.disk.watermark.high` properties can be set to a percentage value (such as 0.60, meaning 60 percent) and to an absolute value (such as 600mb, meaning 600 megabytes).

Cluster wide allocation

Instead of specifying allocation inclusion and exclusion at the index level (which we did until now), we can do that for all the indices in our cluster. For example, if we would like to place all new indices on the nodes with the IP addresses 10.1.2.10 and 10.1.2.11, we will run the following command:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "cluster.routing.allocation.include._ip" : "10.1.2.10,10.1.2.11"
  }
}'
```

Notice that the command was sent to the `_cluster/settings` REST endpoint instead of the `INDEX_NAME/_settings` endpoint. Of course, we can use both include and exclude and require rules just as we did on the index level.

Please note that the transient and persistent cluster properties were discussed in the *Controlling cluster rebalancing* section earlier in this chapter.

Number of shards and replicas per node

In addition to specifying shards and replica allocation, we are also allowed to specify the maximum number of shards that can be placed on a single node for a single index. For example, if we would like our `shop` index to have only a single shard per node, we will run the following command:

```
curl -XPUT 'localhost:9200/shop/_settings' -d '{
  "index.routing.allocation.total_shards_per_node" : 1
}'
```

This property can be placed in the `elasticsearch.yml` file or can be updated on live indices using the preceding command. Please remember that your cluster can stay in the red state if Elasticsearch is not able to allocate all the primary shards.

Moving shards and replicas manually

The last thing we want to discuss is the ability to manually move shards between nodes. This may be useful, for example, if you want to bring a single node down, but before doing this, you want to move all the shards from that node. Elasticsearch exposes the `_cluster/reroute` REST endpoint, which allows us to control this. The following operations are available:

- Moving a shard from node to node

- Canceling shard allocation
- Forcing shard allocation

Now let's look closer at all of the preceding operations.

Moving shards

Let's say we have two nodes called `es_node_one` and `es_node_two`. In addition to this, we have two shards of the `shop` index placed by Elasticsearch on the first node. Now, we would like to move the second shard to the second node. In order to do this, we can run the following command:

```
curl -XPOST 'localhost:9200/_cluster/reroute' -d '{
  "commands" : [ {
    "move" : {
      "index" : "shop",
      "shard" : 1,
      "from_node" : "es_node_one",
      "to_node" : "es_node_two"
    }
  } ]
}'
```

We've specified the `move` command, which allows us to move shards (and replicas) of the index specified by the `index` property. The `shard` property is the number of shards we want to move. And finally, the `from_node` property specifies the name of the node we want to move the shard from, and the `to_node` property specifies the name of the node we want the shard to be placed on.

Canceling shard allocation

If we would like to cancel an ongoing allocation process, we can run the `cancel` command and specify the index, node, and shard we want to cancel the allocation for. For example, consider the following command:

```
curl -XPOST 'localhost:9200/_cluster/reroute' -d '{
  "commands" : [ {
    "cancel" : {
      "index" : "shop",
      "shard" : 0,
    }
  } ]
}'
```

```
"node" : "es_node_one"
}
} ]
}'
```

The preceding command will cancel the allocation of the shard 0 of the `shop` index on the `es_node_one` node.

Forcing shard allocation

In addition to canceling and moving shards and replicas, we can also allocate an unallocated shard to a specific node. For example, if we have an unallocated shard numbered 0 for the `users` index and we would like to allocate it to `es_node_two` by Elasticsearch, we will run the following command:

```
curl -XPOST 'localhost:9200/_cluster/reroute' -d '{
  "commands" : [ {
    "allocate" : {
      "index" : "users",
      "shard" : 0,
      "node" : "es_node_two"
    }
  ]
}'
```

Multiple commands per HTTP request

We can, of course, include multiple commands in a single HTTP request. For example, consider the following command:

```
curl -XPOST 'localhost:9200/_cluster/reroute' -d '{
  "commands" : [
    {"move" : {"index" : "shop", "shard" : 1, "from_node" : "es_node_one",
    "to_node" : "es_node_two"}},
    {"cancel" : {"index" : "shop", "shard" : 0, "node" : "es_node_one"}}
  ]
}'
```

Warming up

Sometimes, there may be a need to prepare Elasticsearch in order to handle your queries. Maybe it's because you heavily rely on the field data cache and you want it to be loaded before your production queries arrive or maybe you want to warm up your operating system's I/O cache. Whatever the reason, Elasticsearch allows us to define the warming queries for our types and indices.

Defining a new warming query

A warming query is nothing more than the usual query stored in a special index called `_warmer` in Elasticsearch. Let's assume that we have the following query that we want to use for warming up:

```
{  
  "query" : {  
    "match_all" : {}  
  },  
  "facets" : {  
    "warming_facet" : {  
      "terms" : {  
        "field" : "tags"  
      }  
    }  
  }  
}
```

To store the preceding query as a warming query for our `library` index, we will run the following command:

```
curl -XPUT 'localhost:9200/library/_warmer/tags_warming_query' -d '{  
  "query" : {  
    "match_all" : {}  
  },  
  "facets" : {  
    "warming_facet" : {  
      "terms" : {  
        "field" : "tags"  
      }  
    }  
  }  
}'
```

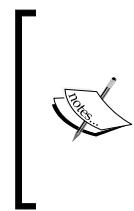
```
        }
    }
}
}'
```

The preceding command will register our query as a warming query with the name `tags_warming_query`. You can have multiple warming queries for your index, but each of these queries needs to have a unique name.

We can not only define warming queries for the whole index, but also for the specific type in it. For example, to store our previously shown query as the warming query only for the `book` type in the `library` index, run the preceding command not to the `/library/_warmer` URI but to `/library/book/_warmer`. So, the entire command will be as follows:

```
curl -XPUT 'localhost:9200/library/book/_warmer/tags_warming_query' -d '{
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "warming_facet" : {
      "terms" : {
        "field" : "tags"
      }
    }
  }
}'
```

After adding a warming query, before Elasticsearch allows a new segment to be searched on, it will be warmed up by running the defined warming queries on that segment. It allows Elasticsearch and the operating system to cache data and thus, speed up searching.



Just as we read in the *Full-text searching* section of *Chapter 1, Getting Started with the Elasticsearch Cluster*, Lucene divides the index into parts called segments, which once written can't be changed. Every new commit operation creates a new segment (which is eventually merged if the number of segments is too high), which Lucene uses for searching.

Retrieving the defined warming queries

In order to get a specific warming query for our index, we just need to know its name. For example, if we want to get the warming query named `tags_warming_query` for our library index, we will run the following command:

```
curl -XGET 'localhost:9200/library/_warmer/tags_warming_query?pretty=true'
```

The result returned by Elasticsearch will be as follows (note that we've used the `pretty=true` parameter to make the response easier to read):

```
{
  "library" : {
    "warmers" : {
      "tags_warming_query" : {
        "types" : [ ],
        "source" : {
          "query" : {
            "match_all" : { }
          },
          "facets" : {
            "warming_facet" : {
              "terms" : {
                "field" : "tags"
              }
            }
          }
        }
      }
    }
  }
}
```

We can also get all the warming queries for the index and type by using the following command:

```
curl -XGET 'localhost:9200/library/_warmer'
```

And finally, we can also get all the warming queries that start with the given prefix. For example, if we want to get all the warming queries for the `library` index that start with the `tags` prefix, we will run the following command:

```
curl -XGET 'localhost:9200/library/_warmer/tags*'
```

Deleting a warming query

Deleting a warming query is very similar to getting one; we just need to use the `DELETE` HTTP method. To delete a specific warming query from our index, we just need to know its name. For example, if we want to delete the warming query named `tags_warming_query` for our `library` index, we will run the following command:

```
curl -XDELETE 'localhost:9200/library/_warmer/tags_warming_query'
```

We can also delete all the warming queries for the index by using the following command:

```
curl -XDELETE 'localhost:9200/library/_warmer/_all'
```

And finally, we can also remove all the warming queries that start with the given prefix. For example, if we want to remove all the warming queries for the `library` index that start with the `tags` prefix, we will run the following command:

```
curl -XDELETE 'localhost:9200/library/_warmer/tags*'
```

Disabling the warming up functionality

To disable the warming queries totally, but to save them in the `_warmer` index, you should set the `index.warmer.enabled` configuration property to `false` (setting this property to `true` will result in enabling the warming up functionality). This setting can be either put into the `elasticsearch.yml` file or just set using the REST API on a live cluster.

For example, if we want to disable the warming up functionality for the `library` index, we will run the following command:

```
curl -XPUT 'http://localhost:9200/library/_settings' -d '{
  "index.warmer.enabled" : false
}'
```

Choosing queries

You may ask which queries should be used as the warming queries; typically, you'll want to choose the ones that are expensive to execute and the ones that require caches to be populated. So you'll probably want to choose the queries that include faceting and sorting based on the fields in your index. In addition to this, parent-child queries and the ones that include common filters may also be the ones to consider. You may also choose other queries by looking at the logs, finding where your performance is not as great as you want it to be. Such queries may also be perfect candidates for warming up.

For example, let's say that we have the following logging configuration set in the `elasticsearch.yml` file:

```
index.search.slowlog.threshold.query.warn: 10s
index.search.slowlog.threshold.query.info: 5s
index.search.slowlog.threshold.query.debug: 2s
index.search.slowlog.threshold.query.trace: 1s
```

And, we have the following logging level set in the `logging.yml` configuration file:

```
logger:
  index.search.slowlog: TRACE, index_search_slow_log_file
```

Notice that the `index.search.slowlog.threshold.query.trace` property is set to `1s`, and the `index.search.slowlog` logging level is set to `TRACE`. This means that whenever a query is executed for longer than one second (on a shard, not in total), it will be logged into the slow logfile (the name of which is specified by the `index_search_slow_log_file` configuration section of the `logging.yml` configuration file). For example, the following can be found in a slow logfile:

```
[2013-01-24 13:33:05,518] [TRACE] [index.search.slowlog.query] [Local
test] [library] [1] took[1400.7ms], took_millis[1400], search_
type[QUERY_THEN_FETCH], total_shards[32], source[{"query": {"match_
all": {}}}], extra_source[]
```

As you can see, in the preceding log line, we have the query time, search type, and the query source itself, which shows us the executed query.

Of course, the values can be different in your configuration, but the slow log can be a valuable source of the queries that have been running too long and may need to have some warm up defined—maybe these are parent-child queries and need some identifiers fetched to perform better, or maybe you are using a filter that is expensive when you execute it for the first time?



There is one thing you should remember: don't overload your Elasticsearch cluster with too many warming queries because you may end up spending too much time warming up instead of processing your production queries.

Index aliasing and using it to simplify your everyday work

When working with multiple indices in Elasticsearch, you can sometimes lose track of them. Imagine a situation where you store logs in your indices. Usually, the amount of log messages is quite large, and therefore, it is a good solution to have the data divided somehow. A logical division of such data is obtained by creating a single index for a single day of logs (if you are interested in an open source solution used to manage logs, look at the Logstash at <http://logstash.net>). But after some time, if we keep all the indices, we will start to have a problem in taking care of all that. An application needs to take care of all the information, such as which index to send data to, which to query, and so on. With the help of aliases, we can change this to work with a single name just as we would use a single index, but we will work with multiple indices.

An alias

What is an index alias? It's an additional name for one or more indices that allow us to query indices with the use of that name. A single alias can have multiple indices as well as the other way around; a single index can be a part of multiple aliases.

However, please remember that you can't use an alias that has multiple indices for indexing or for real-time GET operations. Elasticsearch will throw an exception if you do that. We can still use an alias that links to only a single index for indexing, though. This is because Elasticsearch doesn't know in which index the data should be indexed or from which index the document should be fetched.

Creating an alias

To create an index alias, we need to run the HTTP POST method to the `_aliases` REST endpoint with a defined action. For example, the following request will create a new alias called `week12` that will include the indices named `day10`, `day11`, and `day12`:

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
    { "add" : { "index" : "day10", "alias" : "week12" } },
    { "add" : { "index" : "day11", "alias" : "week12" } },
    { "add" : { "index" : "day12", "alias" : "week12" } }
  ]
}'
```

If the alias `week12` isn't present in our Elasticsearch cluster, the preceding command will create it. If it is present, the command will just add the specified indices to it.

We would run a search across the three indices as follows:

```
curl -XGET 'localhost:9200/day10,day11,day12/_search?q=test'
```

If everything goes well, we can instead run it as follows:

```
curl -XGET 'localhost:9200/week12/_search?q=test'
```

Isn't this better?

Modifying aliases

Of course, you can also remove indices from an alias. We can do this similar to how we add indices to an alias, but instead of the `add` command, we use the `remove` one. For example, to remove the index named `day9` from the `week12` index, we will run the following command:

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
    { "remove" : { "index" : "day9", "alias" : "week12" } }
  ]
}'
```

Combining commands

The `add` and `remove` commands can be sent as a single request. For example, if you would like to combine all the previously sent commands into a single request, we will have to send the following command:

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
    { "add" : { "index" : "day10", "alias" : "week12" } },
    { "add" : { "index" : "day11", "alias" : "week12" } },
    { "add" : { "index" : "day12", "alias" : "week12" } },
    { "remove" : { "index" : "day9", "alias" : "week12" } }
  ]
}'
```

Retrieving all aliases

In addition to adding or removing indices to or from aliases, we and our applications that use Elasticsearch may need to retrieve all the aliases available in the cluster or all the aliases that an index is connected to. To retrieve these aliases, we send a request using the HTTP GET command. For example, the following command gets all the aliases for the day10 index, and the second one will get all the aliases available:

```
curl -XGET 'localhost:9200/day10/_aliases'  
curl -XGET 'localhost:9200/_aliases'
```

The response from the second command is as follows:

```
{  
  "day10" : {  
    "aliases" : {  
      "week12" : { }  
    }  
  },  
  "day11" : {  
    "aliases" : {  
      "week12" : { }  
    }  
  },  
  "day12" : {  
    "aliases" : {  
      "week12" : { }  
    }  
  }  
}
```

Removing aliases

You can also remove an alias using the `_alias` endpoint. For example, sending the following command will remove the `client` alias from the `data` index:

```
curl -XDELETE localhost:9200/data/_alias/client
```

Filtering aliases

Aliases can be used in a way similar to how views are used in SQL databases. You can use a full Query DSL (discussed in detail in *Chapter 2, Indexing Your Data*) and have your query applied to all count, search, delete by query, and so on.

Let's look at an example. Imagine that we want to have aliases that return data for a certain client, so we can use it in our application. Let's say that the client identifier we are interested in is stored in the `clientId` field, and we are interested in client 12345. So, let's create the alias named `client` with our data index, which will apply a filter for `clientId` automatically:

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
    {
      "add" : {
        "index" : "data",
        "alias" : "client",
        "filter" : { "term" : { "clientId" : "12345" } }
      }
    }
  ]
}'
```

So, when using the defined alias, you will always get your request filtered by a term query that ensures that all the documents have the 12345 value in the `clientId` field.

Aliases and routing

Similar to aliases that use filtering, we can add routing values to the aliases. Imagine that we are using routing on the basis of the user identifier, and we want to use the same routing values with our aliases. So, for the alias named `client`, we will use the routing value of 12345, 12346, 12347 for querying, and only 12345 for indexing. To do this, we will create an alias using the following command:

```
curl -XPOST 'localhost:9200/_aliases' -d '{
  "actions" : [
    {
      "add" : {
        "index" : "data",
        "alias" : "client",
        "search_routing" : "12345,12346,12347",
        "index_routing" : "12345"
      }
    }
  ]
}'
```

This way, when we index our data by using the `client` alias, the values specified by the `index_routing` property will be used. At the time of query, the ones specified by the `search_routing` property will be used.

There is one more thing. Please look at the following query sent to the preceding defined alias:

```
curl -XGET 'localhost:9200/client/_search?q=test&routing=99999,12345'
```

The value used as a routing value will be `12345`. This is because Elasticsearch will take the common values of the `search_routing` attribute and the query routing parameter, which in our case is `12345`.

Elasticsearch plugins

At various places in this book, we have used different Elasticsearch plugins. You probably remember the additional programming languages used in scripts and support for the attachments described in the *Handling files* section of *Chapter 6, Beyond Full-text Searching*. In this section, we will look at how plugins work and how to install them.

The basics

Elasticsearch plugins are located in their own subdirectory in the `plugins` directory. If you have downloaded a new plugin from a site, you can just create a new directory with the plugin name and unpack that plugin archive to this directory. There is also a more convenient way to install plugins: by using the plugin script. We have used it several times in this book, so this is the time to describe this tool.

Elasticsearch has two main types of plugins. These two types can be categorized based on their content: Java plugins and site plugins. Elasticsearch treats the site plugins as a file set that should be served by the built-in HTTP server under the `/_plugin/plugin_name/` URL (for example, `/_plugin/bigdesk/`). In addition, every plugin without Java content is automatically treated as a site plugin. That's all. From Elasticsearch's point of view, a site plugin doesn't change anything in Elasticsearch's behavior.

Java plugins usually contain the `.jar` files that are scanned for the `es-plugin.properties` file. This file contains information about the main class that should be used by Elasticsearch as an entry point to configure plugins and allow them to extend the Elasticsearch functionality. The Java plugins can contain the site part that will be used by the built-in HTTP server (just like with the site plugins). This part of the plugin needs to be placed in the `_site` directory.

Installing plugins

By default, plugins are downloaded from the `download.elasticsearch.org` website. If the plugin is not available in this location, Maven Central (`http://search.maven.org/`), Maven Sonatype (`https://repository.sonatype.org/`), and GitHub (`https://github.com/`) repositories are checked. The plugin tool assumes that the given plugin address contains the organization name followed by the plugin name and version number. Let's look at the following command example:

```
bin/plugin -install elasticsearch/elasticsearch-lang-javascript/2.0.0.RC1
```

The preceding command results in the installation of a plugin that allows us to use additional scripting language: JavaScript. We choose Version 2.0.0.RC1 of this plugin. We can also omit the version number; in such cases, Elasticsearch will try to find a version equal to the Elasticsearch version or the latest master version of the plugin.

Just so we know what to expect, this is an example result of running the preceding command:

If you write your own plugin and you have no access to the earlier-mentioned sites, there is no problem. The plugin tool also provides the `-url` option that allows us to set any location for the plugins including the local filesystem (using the `file://` prefix). For example, the following command will result in the installation of a plugin archived on the local filesystem at `/tmp/elasticsearch-lang-javascript-2.0.0.RC1.zip`:

```
bin/plugin -install lang-javascript -url file:///tmp/elasticsearch-lang-javascript-2.0.0.RC1.zip
```

Removing plugins

Removing a plugin is as simple as removing its directory. You can also do this by using the plugin tool. For example, to remove the `river-mongodb` plugin, we can run a command as follows:

```
bin/plugin -remove river-mongodb
```



You need to restart the Elasticsearch node for the plugin installation or removal to take effect.



The update settings API

Elasticsearch lets us tune it by specifying various parameters in the `elasticsearch.yml` file. But you should treat this file as the set of default values that can be changed in the runtime using the Elasticsearch REST API.

In order to set one of the properties, we need to use the HTTP `PUT` method and send a proper request to the `_cluster/settings` URI. However, we have two options: transient and permanent property settings.

The first one, transient, will set the property only until the first restart. In order to do this, we will send the following command:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "PROPERTY_NAME" : "PROPERTY_VALUE"
  }
}'
```

As you can see, in the preceding command, we used the object named `transient` and we added our property definition there. This means that the property will be valid only until the restart. If we want our property settings to persist between restarts, instead of using the object named `transient`, we will use the one named `persistent`.

In every moment, you can fetch these settings using the following command:

```
curl -XGET localhost:9200/_cluster/settings
```

Summary

In this chapter, we learned how to create backups of our cluster; we created a backups repository, we created backups, and we managed them. In addition to this, we learned how to monitor our cluster using the Elasticsearch API, what the cat API is, and why it is more convenient for usage from a human perspective. We also controlled shard allocation, learned how to move a shard around the cluster, and controlled cluster rebalancing. We used the warmers functionality to prepare the cluster for production queries, and we saw how aliasing can help to manage the data in our cluster. Finally, we looked at what Elasticsearch plugins are and how to use the update settings API that Elasticsearch provides.

We have reached the end of the book. We hope that it was a nice reading experience and that you found the book interesting. We really hope that you have learned something from this book, and now, you will find it easier to use Elasticsearch every day. As the authors of this book and Elasticsearch users, we tried to bring you, our readers, the best reading experience we could. Of course, Elasticsearch is more than what we have described in the book—especially when it comes to monitoring and administration capabilities and API. However, the number of pages is limited, and if we describe everything in great detail, we would end up with a book that is one thousand pages long. Also, we are afraid we wouldn't be able to write about everything in enough detail. We need to remember that Elasticsearch is not only user friendly, but also provides a large amount of configuration options, querying possibilities, and so on. Due to this, we had to choose which functionality had to be described in greater detail, which had to be only mentioned, and which had to be totally skipped. We hope that the our choices regarding the topics were right

We would also like to say that it is worth remembering that Elasticsearch is constantly evolving. When writing this book, we went through a few stable versions finally making it to the release of 1.0.0 and 1.0.1. Even back then, we knew that new features and improvements are will come. Be sure to check www.elasticsearch.org periodically for the release notes for new versions of Elasticsearch if you want to be up to date with the new features being added. We will also be writing about new features that we think are worth mentioning on www.elasticsearchserverbook.com. So if you are interested, do visit this site from time to time.

Index

Symbols

`_all field` 75, 76
`_cache parameter` 146
`--data-binary parameter` 71, 160
`_doc object` 196
`-d parameter` 160
`{dynamic_type}` 341
`_fields object` 197
`_id field` 73, 74
`_index field` 77
`_local search operation` 105
`{name}` 341
`_name field` 300
`_name property` 144
`_only_node:node_id search operation` 105
`_prefer_node:node_id search operation` 106
`?pretty parameter` 71
`_primary_first search operation` 105
`_primary search operation` 105
`_script_field parameter` 268
`_shards:1,2 search operation` 106
`_shards parameter` 106
`_size field` 77
`_source field`

- about 76, 319
- exclusion 76
- inclusion 76

`_source object` 197
`_timestamp field` 78
`_ttl field` 79, 80
`_type field` 74, 75
`_uid field` 73, 74

A

add command 375
additional gateway recovery options

- `gateway.expected_data_nodes`
 - property 329
- `gateway.expected_master_nodes`
 - property 329
- `gateway.recover_after_data_nodes`
 - property 329
- `gateway.recover_after_master_nodes`
 - property 329

additional parameters

- `about` 349
- `ignore_unavailable parameter` 346
- `include_global_state parameter` 346
- `indices parameter` 346
- `partial parameter` 346

after_effect property 65

aggregations

- `about` 233
- Bucketing aggregations 240
- bucket ordering 258
- global aggregation 258-261
- Metric aggregations 236
- nested aggregations 255-257
- query structure 234-236

aliases

- and routing 377, 378
- filtering 376
- removing 376
- retrieving 376

aliases option 357

allocation
node, excluding from 363
allocation option 357
allow_leading_wildcard parameter 116
all_terms parameter 268
analysis process 173, 174
Analyze API
 URL 61
analyzer attribute 54
analyzer field 62
analyzer option 282
analyzer parameter 111-126
analyzer property 37, 58
analyzers
 defining 59-62
 keyword analyzer 59
 language analyzer 59
 pattern analyzer 59
 simple analyzer 58
 snowball analyzer 59
 standard analyzer 58
 stop analyzer 58
 URL 204
 using 58
 whitespace analyzer 58
analyzeWildcard parameter 117
analyzeWildcard property 40
and value 316
Apache Lucene
 URL 8
 using 149
Apache Lucene scoring
 about 193
 document matching 194
 scoring formula 194, 195
Apache Lucene TF/IDF scoring formula
 URL 11
Apache Solr synonyms
 equivalent synonyms 226
 expanding synonyms 226
 Explicit synonyms 225
Apache Tika
 URL 203
array 175
auto_generate_phrase_queries parameter 117

automatic identifier creation 27
automatic index creation
 altering 46
avg aggregation 236
avg value 182

B

basic_model property 65
basic queries, Elasticsearch
 common terms query 110, 111
 dismax query 128
 fuzzy_like_this_field query 122
 fuzzy_like_this query 121, 122
 fuzzy query 122, 123
 identifiers query 119
 match_all query 110
 match query 112
 more_like_this_field query 126
 more_like_this query 125
 multi_match query 115
 prefix query 120
 query_string query 116
 range query 127
 regular expression query 129
 simple_query_string query 118
 term query 108, 109
 terms query 109
 wildcard query 124
binary field, core type 56
binary packages
 URL 18
bitset 142
bloom_default 67
bloom filter
 URL 67
bloom_pulsing 67
Boolean, core type 55
Boolean match query
 about 112, 113
 analyzer parameter 113
 cutoff_frequency parameter 113
 fuzziness parameter 113
 max_expansions parameter 113
 operator parameter 112
 prefix_length parameter 113
 zero_terms_query parameter 113

bool filter 142
bool query 130, 185
bool value 316
boost
 about 209
 adding, to queries 209-211
boost attribute 53, 120
boost_factor function 215
boost_factor parameter 215
boosting query 131, 213
boost_mode parameter 214-216
boost parameter 111-215
boost property 109
boost_terms parameter 126
Bounding box filtering 304-306
breaker option 356
bucket 240, 236
Bucketing aggregation
 date_histogram aggregation 252
 date_range aggregation 245-247
 geo_distance aggregation 253, 254
 geohash_grid aggregation 255
 histogram aggregation 251
 IPv4 range aggregation 248, 249
 missing aggregation 249
 nested aggregation 250
 range aggregation 242, 243
 terms aggregation 240, 241
bulk indexing
 data, preparing for 69, 70
bulk thread pool 334

C

cache property 320
cache thread pool 333
calculations
 faceting, filters used 265, 266
 faceting, queries used 264, 265
cancel command 367
cat API 357, 358
cat API, options
 aliases option 357
 allocation option 357
 count option 357
 health option 357
 indices option 357

 master option 357
 nodes option 357
 pending_tasks option 358
 recovery option 358
 shards option 358
 thread_pool option 358
child document
 about 184
 data, querying 185
child mappings 183
CIDR notation
 URL 248
circuit
 breaking 336, 337
circuit breaker 331
cluster 14
cluster health API
 about 348
 additional parameters 349
 information details, controlling 349
cluster name
 multicast, configuring 326
 unicast, configuring 327
cluster.name property 326
cluster rebalancing
 concurrent streams on single node,
 controlling 361
 controlling 359
 primary shards initialized on single node,
 controlling 360
 settings 360
 shards allocation types, controlling 361
 shards initialized on single node,
 controlling 360
 shards moved between nodes,
 controlling 360
cluster.routing.allocation.allow_rebalance
 property 360
cluster state API 356
cluster wide allocation 366
commands
 combining 375
common attributes 52, 53
common terms query
 about 110, 111
 analyzer parameter 111
 boost parameter 111

cutoff_frequency parameter 111
disable_coord parameter 112
high_freq_operator parameter 111
low_freq_operator parameter 111
minimum_should_match parameter 111
query parameter 111
completion object 287
completion section 353
completion suggester
about 278, 284, 285
custom weights 288
data, indexing 285, 286
indexed completion suggester data,
 querying 286, 287
compound queries
about 129
bool query 130
boosting query 131
constant_score query 132
indices query 133
concepts, Elasticsearch
cluster 14
gateway 15
node 14
replica 14
shards 14
concurrent streams
on single node, controlling 361
constant_score_boolean rewrite method 168
constant_score_filter rewrite method 168
constant_score query 132, 212
content_type field 300
Coord 194
copy_to attribute 53
core types
binary field 56
boolean 55
common attributes 52, 53
date 56
number 55
string 53, 54
count option 357
count type 104
CRUD (create-retrieve-update-delete) 24
custom_boost_factor query
 replacing 219, 220

custom_filters_score query
 replacing 221
custom_score query
 replacing 220
custom value 106
custom weights 288
cutoff_frequency parameter 111, 113

D

data
about 174, 175
indexing 70-72
preparing, for bulk indexing 69, 70
sorting 161
storing, in Elasticsearch 25
data concepts, Elasticsearch
document 12, 13
document type 13
Index 12
mapping 13
data indexing 183
data node
 configuring 325
data, querying
 in child documents 185
 in parent documents 187
data, sorting
collation 166
dynamic criteria 165
national characters 166
data structure 172
data type
determining 47-49
field type guess, disabling 49
date, core type 56
date_histogram aggregation
about 252
time zones 252
date_histogram facet 272
date histogram faceting 271
date_range aggregation 245-247
DEB package
used, for installing Elasticsearch 18
decay functions 217
decay parameter 219
default analyzers 63

default_field parameter 116
default filters and tokenizers
 URL 60
default indexing 84
default_operator parameter 116
default_operator property 37
default postings format 66
default sorting 161, 162
DELETE HTTP method 372
deprecated query
 about 219
 custom_boost_factor query,
 replacing 219, 220
 custom_filters_score query, replacing 221
 custom_score query, replacing 220
DFR similarity
 configuring 65
dfs_query_and_fetch type 104
dfs_query_then_fetch type 104
different languages
 handling 202
directory structure 18, 19
direct postings format 67
disable_coord parameter 112, 130
discovery 323
 discovery types 324
 discovery.zen.fd.ping_interval property 327
 discovery.zen.fd.ping_retries property 327
 discovery.zen.fd.ping_timeout property 327
 discovery.zen.minimum_master_nodes
 property 326
 discovery.zen.ping.multicast.enabled
 property 326, 327
 discovery.zen.ping.unicast.hosts
 property 327
Disjunction Max. *See* **DisMax**
disk-based shard allocation
 configuring 365
 enabling 365
DisMax 117
dismax query 128
Distance-based sorting 302-304
Divergence from randomness model 65
doc_count property 241
docs section 351
document 12, 13

Document boost 194
document language
 detecting 203
document matching 194
documents
 deleting 30
 retrieving 27, 28
 updating 28-30
document structure 262, 263
document type 13
doc values
 configuring 68, 69
doc_values_format property 69
dynamic criteria 165
dynamic_date_formats property 48
dynamic mapping
 turning off 178
dynamic property 178
dynamic templates
 about 340, 341
 field definition 341
 matching pattern 341

E

Elasticsearch
 circuit breaker 331
 concepts 14
 configuring 19, 20
 data concepts 12, 13
 data, querying 206
 data, storing in 25
 dedicated filters 141
 different languages, handling 202
 document language, detecting 203
 field data cache 330
 filter cache 330
 index buffers 332
 indexing 15, 16
 index refresh rate 332
 installing, DEB package used 18
 installing, RPM package used 18
 mappings 204-206
 mappings, sending to 177
 multiple languages, handling 203
 MVEL 198

objects, used in scripts 196, 197
other languages, using 198
querying 91, 92
running 20-22
running, as system service on Linux 23
running, as system service on Windows 24
sample document 204
script library, using 199
searching 15, 16
shutting down 22
store module 331
thread pool configuration 333, 334

Elasticsearch API
cat API 357, 358
cluster health API 348
cluster state API 356
indices segments API 357
indices stats API 350, 351
nodes info API 353-355
nodes stats API 355, 356
pending tasks API 357
status API 353
used, for monitoring cluster 348

Elasticsearch index
about 43
automatic index creation, altering 46
creating 45
newly created index, setting 46, 47
replicas, creating 44, 45
shards, creating 44, 45

Elasticsearch, installing
directory structure 18, 19
Java, installing 17
on Linux, from binary packages 18
URL 17

Elasticsearch plugins
about 378
installing 379
removing 380
URL 379

Elasticsearch, querying
example data 92-94
fields, choosing 99
paging 95
result size 95
score, limiting 97
script fields, using 101, 102

URI request query, using 94, 95
version value, returning 96

Elasticsearch query response 33, 34

Elasticsearch RESTful API 25

Elasticsearch service wrapper
URL 23

Elasticsearch snapshotting
old snapshots, deleting 348
snapshot repository, creating 344, 345
snapshots, creating 345, 346
snapshots, restoring 347
using 343

enabled property 77

enable_position_increments parameter 117

envelope shape 308

equivalent synonyms 226

example data 92-94

exclude parameter 268

exclusions 261

execution property 320

existence parameter 138

exists filter 137

exists property 28

expanding synonyms 226

expand property 226

explain parameter 38, 160

Explicit synonyms 225, 226

extended_stats aggregation 238, 240

F

faceting
date histogram faceting 271
document structure 262, 263
geographical faceting 276
memory demanding 277
numerical faceting 271
numerical field statistical data, computing 272-274
Range based faceting 268-270
returned results 263, 264
statistical data, computing for terms 274, 275
terms facetting 266-268

faceting results
filtering 277

field analysis 227, 229

field boost 194
field boosting
 defining, in input data 222, 223
 defining, in mapping 223
field data cache 330, 336, 337
fielddata section 353
fielddata value 316
field definition 341
field option 281
field parameter 201
fields
 about 52
 choosing 99
 configuring 149
 modifying 190, 191
 selecting, for sorting 162-164
fields parameter 39, 118, 121, 125, 268
fields property 122, 300
field type guess
 disabling 49
file-based synonyms 225
files
 handling 297-300
 information, adding 300
 templates, storing 339, 340
filter cache 330
filter_cache section 353
filters
 caching 146
 combining 141
 named filters 143-145
 used, for facetting calculations 265, 266
 using 134, 136
filter types
 exists filter 137
 identifiers filter 139, 140
 limit filter 139
 missing filter 138
 range filter 136, 137
 script filter 138
 type filter 139
 URL 60
final mappings 176, 177
fixed thread pool 333
flush section 353
format attribute 56, 247
format parameter 247
freq property 281
from_node property 367
from parameter 40
from property 95
fs option 355
FST (Finite State Transducers) 67
full-text searching
 about 8
 input data analysis 10
 Lucene architecture 8, 9
 Lucene glossary 8, 9
 query relevance 11
 scoring relevance 11
function_score query
 about 213
 structure 214-219
 URL 219
fuzziness parameter 113
fuzzy_like_this_field query 122
fuzzy_like_this query
 about 121, 122
 analyzer parameter 122
 boost parameter 122
 fields parameter 121
 ignore_tf parameter 121
 like_text parameter 121
 max_query_terms parameter 121
 min_similarity parameter 121
 prefix_length parameter 121
fuzzy_max_expansions parameter 117
fuzzy_min_sim parameter 117
fuzzy_prefix_length parameter 117
fuzzy query
 about 122, 123
 boost parameter 123
 max_expansions parameter 124
 min_similarity parameter 123
 prefix_length parameter 124
 value parameter 123

G

gateway 15, 328
gateway.expected_data_nodes property 329
gateway.expected_master_nodes
 property 329
gateway.expected_nodes property 328

gateway.recover_after_data_nodes
 property 329
gateway.recover_after_master_nodes
 property 329
gateway.recover_after_nodes property 329
gateway.recover_after_time property 329
gateway.type property 328
gather phase 17, 103
Geo
 example data 302
 mappings, for spatial search 301
 sample queries 302
geo_distance aggregation 253, 254
geographical faceting 276
Geohash
 URL 255, 302
geohash_grid aggregation 255
GeoJSON
 URL 308
geo shapes
 about 307
 envelope 308
 example usage 309, 310
 multipolygon shape 309
 point 308
 polygon 308
get section 352
get thread pool 333
GitHub
 URL 379
global aggregation 258-261
global settings 151, 152
gte parameter 127
gt parameter 127

H

health option 357
high_freq_operator parameter 111
highlighted fragments
 controlling 151
highlighting
 about 147-149
 Apache Lucene, using 149
 field, configuring 149
 global settings 151, 152

highlighted fragments, controlling 151
HTML tags, configuring 150, 151
local settings 151, 152
matching requirement 152-155
postings highlighter 155-158
histogram aggregation 251
HTML tags
 configuring 150, 151
http.max_content_length property 70
http option 355
http parameter 355
HTTP PUT command 189

|

IB similarity
 configuring 66
id_cache section 353
identified language
 queries, using with 206
Identifier fields
 _id field 73, 74
 _uid field 73, 74
identifiers filter 139, 140
identifiers query 119
id property 320
ifconfig command 324
ignore_above attribute 54
ignore_conflicts parameter 191
ignore_malformed attribute 55, 56
ignore_tf parameter 121
ignore_unavailable parameter 346
include_global_state parameter 346
include_in_all attribute 53
include_in_all property 75
inclusions 261
index
 about 12
 creating 362, 363
 shapes, storing 311, 312
index alias
 about 374
 creating 374, 375
 modifying 375
index_analyzer attribute 54
index attribute 53
index buffers 332

indexed completion suggester data
 querying 286, 287

indexed documents percolation 296

indexing 10, 11

indexing, Elasticsearch 15, 16

indexing section 352

index.mapper.dynamic property 178

index_name attribute 52

index_name property 56

index_options attribute 54, 155

index_options property 156

index property 320

index.refresh_interval property 332

index refresh rate 332, 335

index.routing.allocation.include.zone
 property 363

index.routing.allocation.require
 property 364

index_routing property 378

index.search.slowlog.threshold.query.trace
 property 373

index structure 183

index structure mapping
 about 50, 51
 analyzers, using 58
 core types 52
 fields 52
 IP address type 57
 multifields 57
 token_count type 58
 type definitions 51

index structure, modifying
 fields, modifying 190, 191
 mappings 189
 new field, adding 189, 190

index templates 338

index thread pool 333

index-time boosting
 about 222
 field boosting, defining in input
 data 222, 223
 field boosting, defining in mapping 223

index-time synonyms
 using 227

indices analyze API
 URL 35

indices.cache.filter.terms.expire_after_access
 property 321

indices.cache.filter.terms.expire_after_write
 property 321

indices.cache.filter.terms.size property 321

indices.fielddata.breaker.limit property 331

indices.fielddata.cache.expire property 331

indices.fielddata.cache.size property 331

indices option 355, 357

indices parameter 346

indices query 133

indices segments API 357

indices stats API
 about 350, 351
 docs section 351
 get section 352
 indexing section 352
 search section 352
 store section 351

indices.store.throttle.type property 83

Information-based model 65

information details
 controlling 349

input data
 field boosting, defining 222, 223

input data analysis
 about 10
 indexing 10, 11
 querying 10, 11

input property 286

install command 24

Inverse document frequency 194

inverted index 8

IP addresses
 used, for shard allocation 364

IP address type 57

IPv4 range aggregation 248

J

Java
 installing 17

JAVA API
 URL 261

JavaScript Object Notation. See **JSON**

Java threads
 URL 333

Java Virtual Machine (JVM) 20

Joda Time library

URL 247

JSON

URL 21, 51

jvm option 355

jvm parameter 354

K

key attribute 244

keyed attribute 243

key_field property 274

key property 241

keyword analyzer 59

kill command 22

L

lambda property 66

Lang property 196

language analyzer 59

URL 59

Language detection

URL 203

Length norm 194

lenient parameter 117

like_text parameter 121, 125

limit filter 139

Linux

Elasticsearch, installing from binary

packages 18

Elasticsearch, running as system service 23

local settings 151, 152

local=true parameter 356

location attribute 344

Logstash

URL 374

lowercase_expanded_terms property 40

lowercase_expand_terms parameter 117

lowercase_terms option 282

low_freq_operator parameter 111

lte parameter 127

lt parameter 127

Lucene architecture 8, 9

Lucene glossary 8, 9

Lucene Javadocs

URL 195

Lucene query syntax

about 41

URL 41

M

mappings

about 13, 175, 176, 189, 204, 206

creating 47

data type, determining 47-49

dynamic mapping 178

field boosting, defining 223

final mappings 176, 177

for spatial search 301

index structure mapping 50, 51

postings format 66, 67

sending, to Elasticsearch 177

similarity models 63

synonym, using 224

mappings, creating

array 175

data 174, 175

objects 175

master-election process

configuring 325

master node

configuring 325

master option 357

match_all query 110

matching pattern 341

match_phrase query

about 114

analyzer parameter 114

slop parameter 114

match_phrase_prefix query 114

match query

about 112

Boolean match query 112, 113

match_phrase_prefix query 114

match_phrase query 114

match template 341

Maven Central

URL 379

Maven Sonatype

URL 379

max aggregation 236

max_boost parameter 215

max_doc_freq parameter 126
max_edits option 282
max_errors option 284
max_expansions parameter 113, 114, 124
max_query_terms parameter 121, 125
max value 182
max_word_len parameter 126
memory 332
memory postings format 67
merge factor 82
merge policy 81
merge scheduler 82
merges section 353
Metric aggregations
 avg aggregation 236
 extended_stats aggregation 238-240
 max aggregation 236
 min aggregation 236
 stats aggregation 238-240
 sum aggregation 236
 value_count aggregation 238
metrics 356
Mike McCandless
 URL 67
min aggregation 236
min_doc_freq parameter 125
minimum_match property 109
minimum_should_match
 parameter 111, 117, 130
min_similarity parameter 121, 123
min_term_freq parameter 125
min_word_len option 282
min_word_len parameter 126
missing aggregation 249
missing fields
 behavior, specifying for 164, 165
missing filter 138
missing parameter 165
mmapfs 332
more_like_this_field query 126
more_like_this query
 about 125
 analyzer parameter 126
 boost parameter 126
 boost_terms parameter 126
 fields parameter 125
like_text parameter 125
max_doc_freq parameter 126
max_query_terms parameter 125
max_word_len parameter 126
min_doc_freq parameter 125
min_term_freq parameter 125
min_word_len parameter 126
percent_terms_to_match parameter 125
stop_words parameter 125
move command 367
multicast
 configuring 326
 URL 324
MULTICAST property 324
multifields 57
multi_match query
 about 115
 example 140
 tie_breaker parameter 115
 use_dis_max parameter 115
multiple commands
 per HTTP request 368
multiple languages
 handling 203
multipolygon shape 309
MVEL
 about 198
 URL 198
MVFLEX Expression Language. *See MVEL*

N

named filters 143-145
native code
 factory implementation 199
 implementing 200, 201
native script
 installing 201
 running 201
nested aggregation 250, 255-257
nested objects
 using 178-182
 working, URL 179
nested query 182
network option 355
network parameter 354

new document
automatic identifier creation 27
creating 25, 26
new field
adding 189, 190
newly created index
setting 46, 47
newScript() method 200
niofs 332
node
about 14
cluster name, setting 326
discovery types 324
excluding, from allocation 363
master node 324
ping settings 327
node attributes
requiring 364
node parameters
specifying 362
nodes info API 353-355
node.size property 364
nodes option 357
nodes stats API 355, 356
node.zone property 363
no_match_query property 133
none value 182
normalization property 65
norms.enabled attribute 54
norms.loading attribute 54
null_value attribute 53
number, core type 55
number of matching queries
obtaining 296
numerical faceting 271
numerical field statistical data
computing 272-274

O

objects 175
offset parameter 219
Okapi BM25 model 65
old snapshots
deleting 348
omit_norms attribute 54

OpenJDK
URL 17
operator parameter 112
optimistic locking
URL 31
order attribute 241
order parameter 268, 338
or value 316
os option 355
os parameter 354

P

paging 95
parameters
passing, to script fields 102
Params object 196
paramYear variable 102
parent-child relationship
data indexing 183
index structure 183
performance considerations 188
querying 184
used, as filters 188
using 182
parent document
about 184
data, querying 187
parent mappings 183
parent_type property 187
partial fields 100
partial parameter 346
path property 320
pattern analyzer
about 59
URL 59
payload property 286
pending tasks API 357
pending_tasks option 358
percent_terms_to_match parameter 125
percolate_index parameter 296
percolate section 353
percolate thread pool 334
percolator
about 289
index, using 289

number of matching queries, obtaining 296
 preparing 290-292
performance considerations 188
phrase_slop parameter 117
phrase suggester
 about 278, 283, 284
 configuring 284
plain method 316
plugins parameter 355
point 219
point shape 308
polygon shape 308
position_offset_gap attribute 54
post_filter parameter 134
postings format
 about 66, 67
 bloom_default 67
 bloom_pulsing 67
 configuring 67
 default postings format 66
 direct postings format 67
 memory postings format 67
 pulsing postings format 67
postings highlighter 155-158
precision_step attribute 55-57
prefix_length parameter 113, 121, 124
prefix_len option 282
prefix query 120
pretty parameter 22
pretty=true parameter 371
primary shard 14
primary shards
 initialized on single mode, controlling 360
process option 355
process parameter 354
pulsing postings format 67

Q

queries
 boost, adding to 209-211
 choosing 372, 373
 combining 208, 209
 used, for faceting calculations 264, 265
 validate API, using 158-160
 validating 158
 with identified language 206

with unknown languages 207
query 229-231
query analysis 35, 36
query_and_fetch type 104
query boosts
 scores, influencing with 209
query DSL 91
querying 10, 11
querying process
 execution preferences, searching 105, 106
 query logic 103, 104
 Search shards API 106-108
 search types 104, 105
query logic 103, 104
Query norm 194
query parameter 111, 116
query property 133
query relevance 11
query rewrite
 about 166
 properties 168, 169
query_string query
 about 116
 allow_leading_wildcard parameter 116
 analyzer parameter 116
 analyzeWildcard parameter 117
 auto_generate_phrase_queries
 parameter 117
 boost parameter 117
 default_field parameter 116
 default_operator parameter 116
 enable_position_increments parameter 117
 fuzzy_max_expansions parameter 117
 fuzzy_min_sim parameter 117
 fuzzy_prefix_length parameter 117
 lenient parameter 117
 lowercase_expand_terms parameter 117
 minimum_should_match parameter 117
 phrase_slop parameter 117
 query parameter 116
 running, against multiple fields 118
query structure 234-236
query_then_fetch type 104
query-time synonyms
 using 227
queue_size property 333

R

RAM buffer
for indexing 337
random_score function 216
range aggregation 242, 243
range attribute 160
Range based facetting 268-270
range filter 136, 137
range query
about 127
gte parameter 127
gt parameter 127
lte parameter 127
lt parameter 127
recovery 328
recovery control 328, 329
recovery option 358
refresh section 353
regex parameter 268
regular expression query 129
regular expression syntax
URL 129
relevance 195, 196
rename_replacement parameter 347
replica 14
replica allocation
configuration 362
controlling 362
disk-based shard allocation 364
index, creating 362, 363
IP addresses, using for shard allocation 364
node attributes, requiring 364
node parameters, specifying 362
nodes, excluding from allocation 363
replicas
creating 44, 45
replica shards 14
replicas per node 366
require_field_match property 152
REST API
documents, deleting 30
documents, retrieving 27
documents, updating 28-30
Elasticsearch RESTful API 25
new document, creating 25, 26
versioning 30, 31

results

filtering 134
result size 95
returned information
limiting 358
returned results 263, 264
rewrite method 120
rewrite parameter 168
rewrite process
example 166, 168
rewrite property 167, 169
right store
choosing 335
routing
about 86, 87
and aliases 377, 378
default indexing 84
routing fields 89, 90
routing parameters 88
routing property 320
RPM package
used, for installing Elasticsearch 18
run() method 201

S

sample data 32
sample queries
Bounding box filtering 304-306
Distance-based sorting 302-304
distance, limiting 306, 307
scan type 105
scatter phase 16, 103
score
influencing, with query boosts 209
limiting 97
modifying 212
score_mode parameter 215
score_mode property 182
score, modifying
boosting query 213
constant_score query 212
deprecated query 219
function_score query 213
score parameter 186
score property 281

score property calculation
 factors 194

scoring_boolean rewrite method 168

scoring formula 194, 195

scoring relevance 11

script fields
 parameters, passing to 102
 using 101, 102

script filter 138

script parameter 165, 201, 268

script property 196, 237

scripts
 using 237, 238

script_score function 216

scroll API
 about 312
 drawback 313
 problem definition 313
 solution 313-315

search_analyzer attribute 54

searching 85, 86

searching, Elasticsearch 15-17

search_routing attribute 378

search_routing property 378

search section 352

Search shards API 106, 108

search thread pool 333

search_type=count parameter 234

search types
 about 104, 105
 count type 104
 dfs_query_and_fetch type 104
 dfs_query_then_fetch type 104
 query_and_fetch type 104
 query_then_fetch type 104
 scan type 105

segment merging
 about 80, 81
 merge factor 82
 merge policy 81
 merge scheduler 82
 need for 81
 throttling 83
 tuning 336

segments merge 9

segments section 353

separator option 284

settings API
 updating 380

settings parameter 354

shapes
 storing, in index 311, 312

shard allocation
 canceling 367
 forcing 368
 IP addresses, used for 364

shard property 367

shards
 about 14
 creating 44, 45
 initialized on single mode, controlling 360
 moved between nodes, controlling 360
 moving 367

shards allocation types
 controlling 361

shard_size option 282

shard_size parameter 267

shards option 358

similarity models
 Divergence from randomness model 65
 Information-based model 65
 Okapi BM25 model 65
 per-field similarity, setting 64

similarity property 65

simple analyzer 58

simplefs 332

simple_query_string query 118

size attribute 241

size option 282

size parameter 40, 139, 267, 294

size property 96, 333

slop parameter 114

snapshot
 creating 345, 346
 restoring 347, 348

snapshot repository
 creating 344, 345

snowball analyzer 59

URL 59

sorting
 fields, selecting for 162-164

sort option 282

sort parameter 39
split-brain 325
standard analyzer
 about 58
 URL 58
statistical data
 computing, for terms 274, 275
stats aggregation 238, 240
status API 353
Stemming 59
stop analyzer 58
 URL 58
stop words
 URL 110
stop_words parameter 125
store attribute 53
store module 331
store property 177.
store section 352
store type
 memory 332
 mmapfs 332
 nios 332
 simplefs 332
string, core type 53, 54
suggester response 279-281
suggesters
 URL 278
 using 278
suggester types
 completion suggester 278
 phrase suggester 278
 term suggester 278
suggestions
 including 278, 279
suggest thread pool 333
sum aggregation 236
synonym
 used, in mappings 224
synonym filter
 file-based synonyms 225
 synonym, used in mappings 224
 using 224
synonym rules
 Apache Solr synonyms, using 225
 defining 225
 WordNet synonyms, using 227
synonyms_path property 225
synonyms property 224

T

template parameter 339
templates
 example 338, 339
 storing, in files 339, 340
Term frequency 194
term query 108, 109
terms aggregation 240, 241
terms facetting 266-268
terms filter
 about 316
 terms lookup 317-319
terms lookup
 about 317-319
 cache settings 321
 query structure 320
terms query 109
term suggester
 about 278, 281
 configuration options 281
term suggester, configuration options
 analyzer option 282
 field option 281
 lowercase_terms option 282
 max_edits option 282
 min_word_len option 282
 prefix_len option 282
 shard_size option 282
 size option 282
 sort option 282
 text option 281
term_vector attribute 53
term_vector property 149
text option 281
text property 281
thread_pool option 356, 358
thread_pool parameter 354
thread pools
 bulk thread pool 334
 cache thread pool 333
 configuring 333

fixed thread pool 333
get thread pool 333
index thread pool 333
percolate thread pool 334
search thread pool 333
suggest thread pool 333
tuning 336
throttling 83
tie_breaker parameter 115, 128
tie parameter 128
timeout parameter 40
time zones 252
 URL 253
token_count type 58
token stream 10
to_node property 367
top children query 186
top_terms_boost_N rewrite method 169
top_terms_N rewrite method 169
total value 182
track_scores=true property 39
translog
 about 337
 URL 337
translog section 353
transport option 355
transport parameter 354
tree-like structures
 analysis process 173, 174
 data structure 172
 indexing 171
type definitions 51
type filter 139
type parameter 140
type property 119, 176, 185, 320

U

unicast
 configuring 327
 URL 324
unknown languages
 queries, using with 207
unmatch template 341
URI query string parameters 37-40

URI request
 about 33
 Elasticsearch query response 33, 34
 query analysis 35, 36
 URI query string parameters 37-40
URI request query
 Lucene query syntax 41
 sample data 32
 using 94, 95
use_dis_max parameter 115
User Datagram Protocol (UDP) 72

V

validate API
 using 158-160
valid attribute 160
value_count aggregation 238
value_field property 274
value parameter 123
value property 109
versioning
 example 31
 from external system 31, 32
version property 96
version_type=external parameter 31
version value
 returning 96

W

wait_for_completion parameter 345
wait_for_nodes parameter 350
wait_for_status parameter 350
warmer section 353
warming query
 defining 369, 370
 deleting 372
 retrieving 371
warming up functionality
 disabling 372
weight parameter 289
weight property 288
whitespace analyzer 58
wildcard query 124

Windows

Elasticsearch, running as system service 24

WordNet

URL 227

WordNet synonyms

using 227

write-ahead logging

URL 337

Z

zero_terms_query parameter 113



Thank you for buying **Elasticsearch Server** *Second Edition*

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.

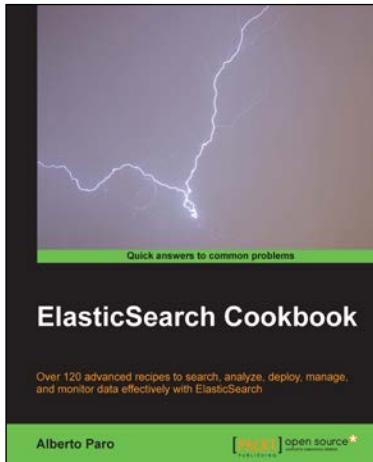
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. 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.

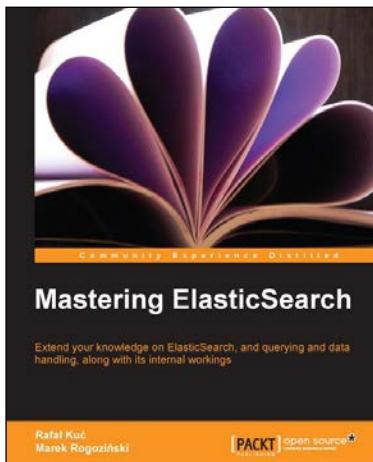


ElasticSearch Cookbook

ISBN: 978-1-78216-662-7 Paperback: 422 pages

Over 120 advanced recipes to search, analyze, deploy, manage, and monitor data effectively with ElasticSearch

1. Write native plugins to extend the capabilities of ElasticSearch to boost your business.
2. Integrate the power of ElasticSearch in your Java applications using the native API or Python applications, with the ElasticSearch community client.
3. Step-by-step instructions to help you easily understand ElasticSearch's capabilities, that act as a good reference for everyday activities.



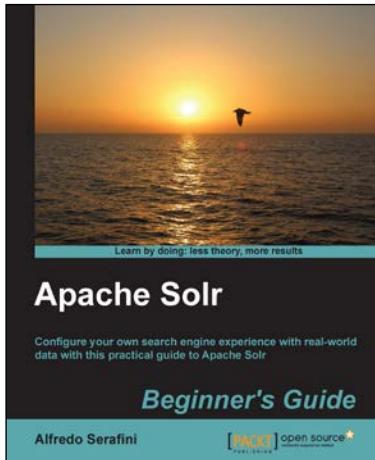
Mastering ElasticSearch

ISBN: 978-1-78328-143-5 Paperback: 386 pages

Extend your knowledge on ElasticSearch, and querying and data handling, along with its internal workings

1. Learn about Apache Lucene and ElasticSearch design and architecture to fully understand how this great search engine works.
2. Design, configure, and distribute your index, coupled with a deep understanding of the workings behind it.
3. Learn about the advanced features in an easy to read book with detailed examples that will help you understand and use the sophisticated features of ElasticSearch.

Please check www.PacktPub.com for information on our titles



Apache Solr Beginner's Guide

ISBN: 978-1-78216-252-0 Paperback: 324 pages

Configure your own search engine experience with real-world data with this practical guide to Apache Solr

1. Learn to use Solr in real-world contexts, even if you are not a programmer, using simple configuration examples.
2. Define simple configurations for searching data in several ways in your specific context, from suggestions to advanced faceted navigation.
3. Teaches you in an easy-to-follow style, full of examples, illustrations, and tips to suit the demands of beginners.



Instant Lucene.NET

ISBN: 978-1-78216-594-1 Paperback: 66 pages

Learn how to index and search through unstructured data using Lucene.NET

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Learn how to execute searches for document indexes.
3. Understand scoring and influencing search results.
4. Easily maintain your index.

Please check www.PacktPub.com for information on our titles