



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

Mastering Elasticsearch

Extend your knowledge on Elasticsearch, and querying and data handling, along with its internal workings

Rafał Kuć
Marek Rogoziński

[PACKT] open source*
PUBLISHING community experience distilled

Mastering Elasticsearch

Extend your knowledge on Elasticsearch, and querying and data handling, along with its internal workings

Rafał Kuć

Marek Rogoziński

[PACKT] open source 
PUBLISHING
community experience distilled
BIRMINGHAM - MUMBAI

Mastering Elasticsearch

Copyright © 2013 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: October 2013

Production Reference: 1211013

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

ISBN 978-1-78328-143-5

www.packtpub.com

Cover Image by Prashant Timappa Shetty (sparkling.spectrum.123@gmail.com)

Credits

Authors

Rafał Kuć

Marek Rogoziński

Reviewers

Ravindra Bharathi

Surendra Mohan

Marcelo Ochoa

Acquisition Editor

James Jones

Lead Technical Editor

Arun Nadar

Technical Editors

Iram Malik

Krishnaveni Nair

Shruti Rawool

Project Coordinator

Shiksha Chaturvedi

Proofreader

Mario Cecere

Indexer

Priya Subramani

Graphics

Ronak Dhruv

Production Coordinator

Kyle Albuquerque

Cover Work

Kyle Albuquerque

About the Authors

Rafał Kuć is a born team leader and a Software Developer. Working as a Consultant and a Software Engineer at Sematext Group, Inc., he concentrates on open source technologies such as Apache Lucene, Solr, ElasticSearch, and Hadoop stack. He has more than 11 years of experience in various software branches – from banking software to e-commerce products. He is mainly focused on Java, but open to every tool and programming language that will make the achievement of his goal easier and faster. He is also one of the founders of the `solr.pl` site, where he tries to share his knowledge and help people to resolve their problems with Solr and Lucene. He is also a speaker for 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 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 an author of *Solr 3.1 Cookbook*, the update to it – *Solr 4.0 Cookbook*, and is a co-author of *ElasticSearch Server* all published by *Packt Publishing*.

The book you are holding in your hands was something that I wanted to write after finishing the *ElasticSearch Server* book and I got the opportunity. I wanted not to jump from topic to topic, but concentrate on a few of them and write about what I know and share the knowledge. Again, just like the *ElasticSearch Server* book, I couldn't include all topics I wanted, and some small details that are more or less important, depending on the use case, had to be left aside. Nevertheless, I hope that by reading this book you'll be able to easily get into all the details about ElasticSearch and underlying Apache Lucene, and I also hope that it will let you get the desired knowledge easier and faster.

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

I would also like to thank all the people I'm working with at Sematext, especially Otis, who took 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 be written and open source search would have been less powerful.

Once again, thank you.

Marek Rogoziński is a Software Architect and a Consultant with more than 10 years of experience. His specialization involves solutions based on open source search engines such as Solr and ElasticSearch and software stack for big data analytics including Hadoop, Hbase, and Twitter Storm.

He is also a co-founder of the `solr.pl` site which publishes information and tutorials about Solr and Lucene library and is the co-author of the *ElasticSearch Server* book published by Packt Publishing.

He currently holds a position of Chief Technology Officer in a company building products based on the processing and analysis of large streams of input data.

Just like the previous book, writing *Mastering Elasticsearch* was a difficult task. To tell the truth, it was much harder not only because of more advanced topics covered in this book, but also because of the constantly introduced changes in the Elasticsearch codebase. The development of it is not going to slow down and literally speaking, every day brings something new. Please remember that this book should be treated as a continuation of the previous book. This means, we have tried to omit all the topics that we had covered before, and we wanted to add everything that was omitted. You can see if you have succeeded yourself. Now it's time to thank everyone.

Thanks to all the people who have created Elasticsearch, Lucene, and all of those libraries and modules published around these projects.

I would also like to thank the team working on this book. First of all, to the ones who worked on the extermination of all my errors, typos, and ambiguities.

Last but not the least, thanks to all the friends, who withstood me during this time.

About the Reviewers

Ravindra Bharathi has worked in the software industry for over a decade in various domains such as education, Digital Media Marketing/ Advertising, Enterprise Search, and Energy Management Systems. He has a keen interest in search-based applications that involve data visualization, mashups, and dashboards. He blogs at <http://ravindrabharathi.blogspot.com>.

I wish to thank my wife, Vidya, for her support in all my endeavors.

Surendra Mohan is currently serving as a Drupal Consultant cum Drupal Architect at a well-known Software Consulting Ltd. organization in India. Prior to joining this organization, he served a few Indian MNCs and a couple of startups in varied roles such as Programmer, Technical Lead, Project Lead, Project Manager, Solution Architect, and Service Delivery Manager. He has around nine years of work experience in web technologies covering media and entertainment, real estate, travel and tours, publishing, e-learning, enterprise architecture, and so on. He is also a well-known speaker who delivers talks on Drupal, Open Source, PHP, Moodle, and so on, along with organizing and delivering TechTalks in Drupal meetups and Drupal Camps in Mumbai, India.

He also reviewed other technical books such as *Drupal 7 Multi Site Configuration*, by Matt Butcher, *Drupal Search Engine Optimization*, by Ric Shreves, *Building e-commerce Sites with Drupal Commerce Cookbook*, by Richard Carter. In addition to technical reviewing activities, he is also writing a book on Apache Solr which is scheduled to be published by the end of October, 2013.

I would like to thank my family and friends who supported and encouraged me in completing my reviews on time with good quality.

Marcelo Ochoa works at the System Laboratory of Facultad de Ciencias Exactas of the Universidad Nacional del Centro de la Provincia de Buenos Aires and is the CTO at `Scotas.com`, a company specialized in Near Real Time Search solutions using Apache Solr and Oracle. He divides his time between University jobs and external projects related to Oracle and big data technologies. He has worked in several Oracle-related projects such as translation of Oracle manuals and multimedia CBTs. His background is in database, network, web, and Java technologies. In the XML world he is known as the *developer of DB Generator for the Apache Cocoon project*, the open source projects DBPrism and DBPrism CMS, the Lucene-Oracle integration by using Oracle JVM Directory implementation and in the `Restlet.org` project the Oracle XDB Restlet Adapter, an alternative to write native REST web services inside the database-resident JVM.

Since 2006, he is a part of the Oracle ACE program; Oracle ACEs are known for their strong credentials as Oracle community enthusiasts and advocates, with candidates nominated by ACEs in the Oracle Technology and Applications communities.

He is the author of *Chapter 17, 360-Degree Programming the Oracle Database* of the book, *Oracle Database Programming Using Java and Web Services*, by Kuassi Mensah, at Digital Press and *Chapter 21, DB Prism: A Framework to Generate Dynamic XML from a Database* of the book *Professional XML Databases*, by Kevin Williams, at Wrox Press.

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: Introduction to ElasticSearch | 7 |
| Introducing Apache Lucene | 8 |
| Getting familiar with Lucene | 8 |
| Overall architecture | 8 |
| Analyzing your data | 10 |
| Indexing and querying | 11 |
| Lucene query language | 11 |
| Understanding the basics | 12 |
| Querying fields | 13 |
| Term modifiers | 13 |
| Handling special characters | 14 |
| Introducing ElasticSearch | 15 |
| Basic concepts | 15 |
| Index | 15 |
| Document | 15 |
| Mapping | 16 |
| Type | 16 |
| Node | 16 |
| Cluster | 16 |
| Shard | 17 |
| Replica | 17 |
| Gateway | 17 |
| Key concepts behind ElasticSearch architecture | 17 |
| Working of ElasticSearch | 18 |
| The bootstrap process | 18 |
| Failure detection | 19 |
| Communicating with ElasticSearch | 20 |
| Summary | 23 |

| | |
|--|-----------|
| Chapter 2: Power User Query DSL | 25 |
| Default Apache Lucene scoring explained | 26 |
| When a document is matched | 26 |
| The TF/IDF scoring formula | 27 |
| The Lucene conceptual formula | 27 |
| The Lucene practical formula | 28 |
| The Elasticsearch point of view | 29 |
| Query rewrite explained | 29 |
| Prefix query as an example | 29 |
| Getting back to Apache Lucene | 32 |
| Query rewrite properties | 33 |
| Rescore | 35 |
| Understanding rescore | 35 |
| Example Data | 35 |
| Query | 36 |
| Structure of the rescore query | 36 |
| Rescore parameters | 38 |
| To sum up | 39 |
| Bulk Operations | 39 |
| MultiGet | 39 |
| MultiSearch | 41 |
| Sorting data | 43 |
| Sorting with multivalued fields | 44 |
| Sorting with multivalued geo fields | 45 |
| Sorting with nested objects | 47 |
| Update API | 48 |
| Simple field update | 49 |
| Conditional modifications using scripting | 50 |
| Creating and deleting documents using the Update API | 50 |
| Using filters to optimize your queries | 51 |
| Filters and caching | 52 |
| Not all filters are cached by default | 53 |
| Changing Elasticsearch caching behavior | 54 |
| Why bother naming the key for the cache? | 55 |
| When to change the Elasticsearch filter caching behavior | 55 |
| The terms lookup filter | 55 |
| How does it work? | 58 |
| Performance considerations | 59 |
| Loading terms from inner objects | 59 |
| Terms lookup filter cache settings | 60 |
| Filter and scopes in Elasticsearch faceting mechanism | 60 |
| Example data | 61 |

| | |
|---|-----------|
| Faceting and filtering | 61 |
| Filter as a part of the query | 63 |
| The Facet filter | 65 |
| Global scope | 67 |
| Summary | 69 |
| Chapter 3: Low-level Index Control | 71 |
| Altering Apache Lucene scoring | 71 |
| Available similarity models | 72 |
| Setting per-field similarity | 73 |
| Similarity model configuration | 74 |
| Choosing the default similarity model | 75 |
| Configuring the chosen similarity models | 76 |
| Configuring TF/IDF similarity | 76 |
| Configuring Okapi BM25 similarity | 77 |
| Configuring DFR similarity | 77 |
| Configuring IB similarity | 78 |
| Using codecs | 78 |
| Simple use cases | 78 |
| Let's see how it works | 79 |
| Available posting formats | 81 |
| Configuring the codec behavior | 82 |
| Default codec properties | 83 |
| Direct codec properties | 83 |
| Memory codec properties | 83 |
| Pulsing codec properties | 83 |
| Bloom filter-based codec properties | 84 |
| NRT, flush, refresh, and transaction log | 85 |
| Updating index and committing changes | 86 |
| Changing the default refresh time | 86 |
| The transaction log | 87 |
| The transaction log configuration | 88 |
| Near Real Time GET | 89 |
| Looking deeper into data handling | 90 |
| Input is not always analyzed | 90 |
| Example usage | 94 |
| Changing the analyzer during indexing | 95 |
| Changing the analyzer during searching | 96 |
| The pitfall and default analysis | 97 |
| Segment merging under control | 97 |
| Choosing the right merge policy | 98 |
| The tiered merge policy | 99 |
| The log byte size merge policy | 99 |
| The log doc merge policy | 100 |

| | |
|---|------------|
| Merge policies configuration | 100 |
| The tiered merge policy | 100 |
| The log byte size merge policy | 101 |
| The log doc merge policy | 102 |
| Scheduling | 103 |
| The concurrent merge scheduler | 103 |
| The serial merge scheduler | 104 |
| Setting the desired merge scheduler | 104 |
| Summary | 104 |
| Chapter 4: Index Distribution Architecture | 105 |
| Choosing the right amount of shards and replicas | 106 |
| Sharding and over allocation | 106 |
| A positive example of over allocation | 108 |
| Multiple shards versus multiple indices | 108 |
| Replicas | 108 |
| Routing explained | 109 |
| Shards and data | 109 |
| Let's test routing | 110 |
| Indexing with routing | 112 |
| Indexing with routing | 114 |
| Querying | 115 |
| Aliases | 117 |
| Multiple routing values | 118 |
| Altering the default shard allocation behavior | 119 |
| Introducing ShardAllocator | 119 |
| The even_shard ShardAllocator | 119 |
| The balanced ShardAllocator | 120 |
| The custom ShardAllocator | 121 |
| Deciders | 121 |
| SameShardAllocationDecider | 121 |
| ShardsLimitAllocationDecider | 122 |
| FilterAllocationDecider | 122 |
| ReplicaAfterPrimaryActiveAllocationDecider | 122 |
| ClusterRebalanceAllocationDecider | 122 |
| ConcurrentRebalanceAllocationDecider | 123 |
| DisableAllocationDecider | 123 |
| AwarenessAllocationDecider | 123 |
| ThrottlingAllocationDecider | 124 |
| RebalanceOnlyWhenActiveAllocationDecider | 124 |
| DiskThresholdDecider | 124 |
| Adjusting shard allocation | 125 |
| Allocation awareness | 126 |
| Forcing allocation awareness | 128 |

| | |
|---|------------|
| Filtering | 128 |
| But what those properties mean? | 129 |
| Runtime allocation updating | 130 |
| Index-level updates | 130 |
| Cluster-level updates | 130 |
| Defining total shards allowed per node | 131 |
| Inclusion | 132 |
| Requirements | 133 |
| Exclusion | 134 |
| Additional shard allocation properties | 135 |
| Query execution preference | 136 |
| Introducing the preference parameter | 137 |
| Using our knowledge | 139 |
| Assumptions | 139 |
| Data volume and queries specification | 140 |
| Configuration | 142 |
| Node-level configuration | 143 |
| Indices configuration | 143 |
| The directories layout | 143 |
| Gateway configuration | 143 |
| Recovery | 144 |
| Discovery | 144 |
| Logging slow queries | 145 |
| Logging garbage collector work | 145 |
| Memory setup | 146 |
| One more thing | 146 |
| Changes are coming | 147 |
| Reindexing | 147 |
| Routing | 148 |
| Multiple Indices | 148 |
| Summary | 149 |
| Chapter 5: Elasticsearch Administration | 151 |
| Choosing the right directory implementation – the store module | 151 |
| Store type | 152 |
| The simple file system store | 152 |
| The new IO filesystem store | 152 |
| The MMap filesystem store | 153 |
| The memory store | 153 |
| The default store type | 154 |
| Discovery configuration | 155 |
| Zen discovery | 155 |
| Multicast | 156 |
| Unicast | 157 |
| Minimum master nodes | 157 |
| Zen discovery fault detection | 158 |

| | |
|---|------------|
| Amazon EC2 discovery | 158 |
| EC2 plugin's installation | 159 |
| Gateway and recovery configuration | 161 |
| Gateway recovery process | 161 |
| Configuration properties | 162 |
| Expectations on nodes | 163 |
| Local gateway | 163 |
| Backing up the local gateway | 164 |
| Recovery configuration | 164 |
| Cluster-level recovery configuration | 165 |
| Index-level recovery settings | 166 |
| Segments statistics | 166 |
| Introducing the segments API | 167 |
| The response | 167 |
| Visualizing segments information | 170 |
| Understanding Elasticsearch caching | 170 |
| The filter cache | 171 |
| Filter cache types | 171 |
| Index-level filter cache configuration | 172 |
| Node-level filter cache configuration | 173 |
| The field data cache | 173 |
| Index-level field data cache configuration | 174 |
| Node-level field data cache configuration | 174 |
| Filtering | 175 |
| Clearing the caches | 180 |
| Index, indices, and all caches clearing | 181 |
| Clearing specific caches | 181 |
| Clearing fields-related caches | 182 |
| Summary | 182 |
| Chapter 6: Fighting with Fire | 183 |
| Knowing the garbage collector | 184 |
| Java memory | 184 |
| The life cycle of Java object and garbage collections | 185 |
| Dealing with garbage collection problems | 186 |
| Turning on logging of garbage collection work | 186 |
| Using JStat | 187 |
| Creating memory dumps | 189 |
| More information on garbage collector work | 189 |
| Adjusting garbage collector work in Elasticsearch | 190 |
| Avoiding swapping on Unix-like systems | 191 |
| When it is too much for I/O – throttling explained | 193 |
| Controlling I/O throttling | 193 |
| Configuration | 193 |
| Throttling type | 193 |
| Maximum throughput per second | 194 |

| | |
|--|------------|
| Node throttling defaults | 194 |
| Configuration example | 194 |
| Speeding up queries using warmers | 196 |
| Reason for using warmers | 196 |
| Manipulating warmers | 197 |
| Using the PUT Warmer API | 197 |
| Adding warmers during index creation | 198 |
| Adding warmers to templates | 199 |
| Retrieving warmers | 199 |
| Deleting warmers | 200 |
| Disabling warmers | 200 |
| Testing the warmers | 201 |
| Querying without warmers present | 202 |
| Querying with warmer present | 203 |
| Very hot threads | 204 |
| Hot Threads API usage clarification | 205 |
| Hot Threads API response | 206 |
| Real-life scenarios | 207 |
| Slower and slower performance | 207 |
| Heterogeneous environment and load imbalance | 210 |
| My server is under fire | 212 |
| Summary | 213 |
| Chapter 7: Improving the User Search Experience | 215 |
| Correcting user spelling mistakes | 216 |
| Test data | 216 |
| Getting into technical details | 217 |
| Suggesters | 218 |
| Using the _suggest REST endpoint | 218 |
| Including suggestions requests in a query | 221 |
| The term suggester | 224 |
| The phrase suggester | 227 |
| Completion suggester | 237 |
| The logic behind completion suggester | 238 |
| Using completion suggester | 238 |
| Improving query relevance | 243 |
| The data | 244 |
| The quest for improving relevance | 246 |
| The standard query | 247 |
| The Multi match query | 248 |
| Phrases comes into play | 250 |
| Let's throw the garbage away | 254 |
| And now we boost | 256 |
| Making a misspelling-proof search | 257 |
| Drill downs with faceting | 260 |
| Summary | 264 |

| | |
|---|------------|
| Chapter 8: Elasticsearch Java APIs | 265 |
| Introducing the Elasticsearch Java API | 266 |
| The code | 267 |
| Connecting to your cluster | 268 |
| Becoming the Elasticsearch node | 268 |
| Using the transport connection method | 270 |
| Choosing the right connection method | 271 |
| Anatomy of the API | 272 |
| CRUD operations | 274 |
| Fetching documents | 274 |
| Handling errors | 276 |
| Indexing documents | 276 |
| Updating documents | 279 |
| Deleting documents | 282 |
| Querying Elasticsearch | 284 |
| Preparing a query | 284 |
| Building queries | 285 |
| Using the match all documents query | 287 |
| The match query | 287 |
| Using the geo shape query | 288 |
| Paging | 289 |
| Sorting | 290 |
| Filtering | 290 |
| Faceting | 292 |
| Highlighting | 292 |
| Suggestions | 293 |
| Counting | 294 |
| Scrolling | 295 |
| Performing multiple actions | 295 |
| Bulk | 296 |
| The delete by query | 296 |
| Multi GET | 296 |
| Multi Search | 297 |
| Percolator | 297 |
| ElasticSearch 1.0 and higher | 298 |
| The explain API | 299 |
| Building JSON queries and documents | 300 |
| The administration API | 302 |
| The cluster administration API | 302 |
| The cluster and indices health API | 302 |
| The cluster state API | 303 |
| The update settings API | 303 |
| The reroute API | 303 |

| | |
|--|------------|
| The nodes information API | 304 |
| The node statistics API | 304 |
| The nodes hot threads API | 305 |
| The nodes shutdown API | 305 |
| The search shards API | 305 |
| The Indices administration API | 306 |
| The index existence API | 306 |
| The Type existence API | 306 |
| The indices stats API | 306 |
| Index status | 307 |
| Segments information API | 307 |
| Creating an index API | 307 |
| Deleting an index | 308 |
| Closing an index | 308 |
| Opening an index | 308 |
| The Refresh API | 308 |
| The Flush API | 309 |
| The Optimize API | 309 |
| The put mapping API | 309 |
| The delete mapping API | 310 |
| The gateway snapshot API | 310 |
| The aliases API | 310 |
| The get aliases API | 311 |
| The aliases exists API | 311 |
| The clear cache API | 311 |
| The update settings API | 312 |
| The analyze API | 312 |
| The put template API | 312 |
| The delete template API | 313 |
| The validate query API | 313 |
| The put warmer API | 314 |
| The delete warmer API | 314 |
| Summary | 314 |
| Chapter 9: Developing Elasticsearch Plugins | 315 |
| Creating the Apache Maven project structure | 316 |
| Understanding the basics | 316 |
| Structure of the Maven Java project | 317 |
| The idea of POM | 317 |
| Running the build process | 319 |
| Introducing the assembly Maven plugin | 319 |
| Creating a custom river plugin | 322 |
| Implementation details | 322 |
| Implementing the URLChecker class | 324 |
| Implementing the JSONRiver class | 327 |
| Implementing the JSONRiverModule class | 329 |
| Implementing the JSONRiverPlugin class | 329 |
| Informing Elasticsearch about the JSONRiver plugin class | 330 |

| | |
|---|------------|
| Testing our river | 331 |
| Building our river | 331 |
| Installing our river | 331 |
| Initializing our river | 332 |
| Checking if our JSON river works | 333 |
| Creating custom analysis plugin | 333 |
| Implementation details | 334 |
| Implementing TokenFilter | 335 |
| Implementing the TokenFilter factory | 336 |
| Implementing custom analyzer | 337 |
| Implementing analyzer provider | 338 |
| Implementing analysis binder | 339 |
| Implementing analyzer indices component | 340 |
| Implementing analyzer module | 342 |
| Implementing analyzer plugin | 342 |
| Informing Elasticsearch about our custom analyzer | 343 |
| Testing our custom analysis plugin | 343 |
| Building our custom analysis plugin | 344 |
| Installing the custom analysis plugin | 344 |
| Checking if our analysis plugin works | 345 |
| Summary | 346 |
| Index | 347 |

Preface

Welcome to the world of Elasticsearch and to the *Mastering Elasticsearch* book. While reading the book you'll be taken through different topics, all connected to Elasticsearch. We will start with the introduction to Apache Lucene and Elasticsearch, because even if you are familiar with it, it is crucial to have the background in order to fully understand what is going on when you form a cluster, send a document for indexing, or make a query.

You will learn how Apache Lucene scoring works, how to influence it, and how to tell Elasticsearch to choose different scoring algorithms. The book will show you what query rewriting is and why it happens. Apart from that, you'll see how to change your queries to leverage Elasticsearch caching capabilities and make maximum use of it.

After that we will focus on index control. We will learn the way to change how index fields are written, by using different posting formats. We will discuss segments merging, why it is important, and how to adjust it when there is a need. We'll take a deeper look at shard allocation mechanism and routing, and finally we'll learn what to do when data and query number grows.

The book can't omit garbage collector description – how it works and where to start and when you need to tune its behavior. In addition to that, it covers functionalities that allow us to troubleshoot Elasticsearch, such as describing how segments merging works, how to see what Elasticsearch does beneath its high-level interface, and how to limit the I/O operations. But the book doesn't only pay attention to low-level aspects of Elasticsearch; it includes user search experience improvements tips, such as dealing with spelling mistakes, highly effective autocomplete feature, and a tutorial on how you can deal with query related improvements.

In addition to this, the book you are holding will guide you through Elasticsearch Java API, showing how to use it, not only when it comes to CRUD operations but also when it comes to cluster and indices maintenance and manipulation. Finally, we will take a deep look at Elasticsearch extensions by developing a custom river plugin for data indexing and a custom analysis plugin for data analysis during query and index time.

What this book covers

Chapter 1, Introduction to Elasticsearch, will guide you through how Apache Lucene works and will reintroduce you to the world of Elasticsearch describing the basic concepts and showing how Elasticsearch works internally.

Chapter 2, Power User Query DSL, describes how Apache Lucene scoring works, why Elasticsearch rewrites queries, and how query rescore mechanism works. In addition to that, it explains the batch APIs available in Elasticsearch and shows how to use filters to optimize your queries.

Chapter 3, Low-level Index Control, describes how to alter Apache Lucene scoring and how to alter fields' structure by using different posting formats. It also covers NRT searching and indexing, transaction log usage, allows you to understand segments merging, and tune it for your use case.

Chapter 4, Index Distribution Architecture, covers techniques for choosing the right number of shards and replicas, how routing works, and describes deeply how shard allocation works and how to alter its behavior. In this chapter, we also discuss how to configure your Elasticsearch cluster in the beginning and what to do when the data and query number increases.

Chapter 5, Elasticsearch Administration, describes how to choose the right directory implementation for your use case, what are the Discovery, Gateway, and Recovery modules, how to configure them, and why you should bother. We also describe how to look at the segments' information provided by Elasticsearch and how to tune and use Elasticsearch caching mechanism.

Chapter 6, Fighting with Fire, covers how JVM garbage collector works, why it is so important, and how to start tuning it. It also describes how to control the amount of I/O operations Elasticsearch is using, what warmers are and how to use them, and how to diagnose problems with Elasticsearch.

Chapter 7, Improving the User Search Experience, introduces you to the world of suggesters, which allows us to correct user query spelling mistakes and build efficient autocomplete mechanisms. In addition to that you'll see, on real-life example, how to improve query relevance by using different queries and Elasticsearch functionalities.

Chapter 8, Elasticsearch Java APIs, covers Elasticsearch Java API, from basics such as connecting to Elasticsearch, through indexing documents both one by one and in batches and retrieving them afterwards. It also describes different methods exposed by Elasticsearch Java API that allow us to control the cluster.

Chapter 9, Developing Elasticsearch plugins, covers Elasticsearch plugins development by showing and deeply describing how to write your own river and language plugin.

What you need for this book

This book was written using Elasticsearch server 0.90.x; all the examples and functions should work with it. In addition to that, you'll need a command that allows sending 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 is understood by the tool of your choice.

In addition to that, to run examples in *Chapter 8, Elasticsearch Java APIs* and *Chapter 9, Developing Elasticsearch Plugins*, you will need a **JDK (Java Development Kit)** installed and an editor that will allow you to develop your code (or Java IDE such as Eclipse). In both the mentioned chapters we are also using Apache Maven to build the code.

Who this book is for

This book was written for Elasticsearch users and enthusiasts who are already familiar with the basics concepts of this great search server and want to extend their knowledge when it comes to Elasticsearch itself, but it also deals with topics such as how Apache Lucene or JVM garbage collector works. In addition to that, readers who want to see how to improve their query relevancy, how to use Elasticsearch Java API, and how to extend Elasticsearch with their own plugin, may find this book interesting and useful.

If you are new to Elasticsearch and you are not familiar with basic concepts such as querying and data indexing, you may find it hard to use this book as most of the chapters assume that you have this knowledge already. In such cases, we suggest looking at our previous book about Elasticsearch – the *ElasticSearch Server* book from *Packt Publishing*.

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: "What we would like to do is, use the BM25 similarity model for the `name` field and the `contents` field."

A block of code is set 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" }
      }
    }
  }
}
```

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

```
{
  "settings" : {
    "index" : {
      "similarity" : {
        "default" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  ...
}
```

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

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test" }'
```

New terms and **important words** are shown in bold.

 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.

Dear reader, refer to <http://www.elasticsearchserverbook.com> from time to time, where you'll be able to find the newest errata dedicated to the book and additional articles extending it.

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

Introduction to ElasticSearch

We hope that by reading this book you want to extend and build on basic ElasticSearch knowledge. We have assumed that you already know how to index data to ElasticSearch using single requests as well as bulk indexing. You should also know how to send queries to get the documents you are interested in, how to narrow down the results of your queries by using filtering, and how to calculate statistics for your data with the use of the faceting/aggregation mechanism. However, before getting to the exciting functionality that ElasticSearch offers, we think that we should start with a quick tour of Apache Lucene, the full text search library that ElasticSearch uses to build and search its indices, as well as the basic concepts that ElasticSearch is built on. In order to move forward and extend our learning, we need to ensure we don't forget the basics. It is easy to do. We also need to make sure that we understand Lucene correctly as *Mastering ElasticSearch* requires this understanding. By the end of this chapter we will have covered:

- What Apache Lucene is
- What overall Lucene architecture looks like
- How the analysis process is done
- What Apache Lucene query language is and how to use it
- What are the basic concepts of ElasticSearch
- How ElasticSearch communicates internally

Introducing Apache Lucene

In order to fully understand how Elasticsearch works, especially when it comes to indexing and query processing, it is crucial to understand how Apache Lucene library works. Under the hood, Elasticsearch uses Lucene to handle document indexing. The same library is also used to perform search against the indexed documents. In the next few pages we will try to show you the basics of Apache Lucene, just in case you've never used it.

Getting familiar with Lucene

You may wonder why Elasticsearch creators decided to use Apache Lucene instead of developing their own functionality. We don't know for sure, because we were not the ones that made the decision, but we assume that it was because Lucene is mature, highly performing, scalable, light, and yet, very powerful. Its core comes as a single file of Java library with no dependencies, and allows you to index documents and search them with its out of the box full text search capabilities. Of course there are extensions to Apache Lucene that allows different languages handling, enables spellchecking, highlighting, and much more; but if you don't need those features, you can download a single file and use it in your application.

Overall architecture

Although I would like to jump straight to Apache Lucene architecture, there are some things we need to know first in order to fully understand it, and those are:

- **Document:** It is a main data carrier used during indexing and search, containing one or more fields, which contain the data we put and get from Lucene
- **Field:** It is a section of the document which is built of two parts, the name and the value
- **Term:** It is a unit of search representing a word from text
- **Token:** It is an occurrence of a term from the text of the field. It consists of term text, start and end offset, 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, not the other way around like the relational database does. You can think of inverted index as a data structure where data is term oriented rather than document oriented. Let's see how a simple inverted index can look. For example, let's assume that we have the documents with only title field to be indexed and they look like this:

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

So the index (in a very simple way) could be visualized as follows:

| Term | Count | Docs |
|---------------|-------|---------|
| 4 | 1 | <3> |
| Apache | 1 | <3> |
| Cookbook | 1 | <3> |
| ElasticSearch | 2 | <1> <2> |
| Mastering | 1 | <1> |
| Server | 1 | <1> |
| Solr | 1 | <3> |

As you can see, each term points to the number of documents it is present in. This allows for very efficient and fast search, such as the term-based queries. In addition to that each term has a number connected to it: the count, telling Lucene how often it occurs.

Of course, the actual index created by Lucene is much more complicated and advanced, because **term vectors** (a small inverted index for a single field, which allows getting all tokens for that particular field) can be stored, original values of the fields can be stored, markers about deleted documents can be written, and so on. But all you need to know is how the data is organized, not what is exactly stored.

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

However, multiple segments can be merged together in a process called **segments merge**. After forcing segments merge, or after Lucene decides it is time for merging to be performed, segments are merged together by Lucene to create larger ones. This can be I/O demanding, however it is needed to clean up some information, because during that time some information that is not needed anymore is deleted; for example, the deleted documents. In addition to this, searching with the use of one larger segment is faster than searching against multiple smaller ones holding the same data. However, once again, remember that segments merge is I/O demanding operation and you shouldn't force merging, just configure your merge policy carefully.



If you want to know what files are building the segments and what information is stored inside them, please take a look at Apache Lucene documentation available at http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/codecs/lucene45/package-summary.html.

Analyzing your data

Of course the question arises, how the data 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.

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

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

Apart from tokenizer, Lucene analyzer is built of zero or more filters that are used to process tokens in the token stream. For example, it can remove tokens from the stream, change them, or even produce new ones. There are numerous filters and you can easily create new ones. Some examples of filters are:

- **Lowercase filter**: It makes all the tokens lowercased
- **ASCII folding filter**: It removes non ASCII parts from tokens

- **Synonyms filter:** It 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 adding multiple filters one after another.

The last thing is the character mappings, which is used before tokenizer and is responsible for processing text before any analysis is done. One of the examples of character mapper is HTML tags removal process.

Indexing and querying

We may wonder how that all affects indexing and querying when using Lucene and all the software that is built on top of it. During indexing, Lucene will use analyzer of your choice to process contents of your document; of course different analyzer can be used for different fields, so the `title` field of your document can be analyzed differently compared to the `description` field.

During query time, if you use one of the provided query parsers, your query will be analyzed. However, you can also choose the other path and not analyze your queries. This is crucial to remember, because some of the Elasticsearch queries are being analyzed and some are not. For example, the prefix query is not analyzed and the match query is analyzed.

What you should remember about indexing and querying analysis is that the index should be matched by 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 be sure that the term in the query are also lowercased and stemmed, or your queries will return no results at all.

Lucene query language

Some of the query types provided by Elasticsearch support Apache Lucene query parser syntax. Because of that, let's go deeper into Lucene query language and describe it.

Understanding the basics

A query is divided by Apache Lucene into terms and operators. A term, in Lucene, can be a single word or a phrase (group of words surrounded by double quote characters). If the query is set to be analyzed, the defined analyzer will be used on each of the terms that form the query.

A query can also contain Boolean operators that connect terms to each other forming clauses. The list of Boolean operators is as follows:

- **AND:** It means that the given two terms (left and right operand) need to match in order for the clause to be matched. For example, we would run a query, such as `apache AND lucene`, to match documents with both `apache` and `lucene` terms in a document.
- **OR:** It means that any of the given terms may match in order for the clause to be matched. For example, we would run a query, such as `apache OR lucene`, to match documents with `apache` or `lucene` (or both) terms in a document.
- **NOT:** It means that in order for the document to be considered a match, the term appearing after the **NOT** operator must not match. For example, we would run a query `lucene NOT elasticsearch` to match documents that contain `lucene` term, but not `elasticsearch` term in the document.

In addition to that, we may use the following operators:

- **+:** It means that the given term needs to be matched in order for the document to be considered as a match. For example, in order to find documents that match `lucene` term and may match `apache` term, we would run a query, such as `+lucene apache`.
- **-:** It means that the given term can't be matched in order for the document to be considered a match. For example, in order to find document with `lucene` term, but not `elasticsearch` term we would run a query, such as `+lucene -elasticsearch`.

When not specifying any of the previous operators, the default **OR** operator will be used.

In addition to all these, there is one more thing; you can use parenthesis to group clauses together. For example, with something like this:

```
elasticsearch AND (mastering OR book)
```

Querying fields

Of course, just like in ElasticSearch, in Lucene all your data is stored in fields that build the document. In order to run a query against a field, you need to provide the field name, add the colon character, and provide the clause that should be run against that field. For example, if you would like to match documents with the term `elasticsearch` in the `title` field, you would run a query like this:

```
title:elasticsearch
```

You can also group multiple clauses to a single field. For example, if you would like your query to match all the documents having the `elasticsearch` term and the `mastering book` phrase in the `title` field, you could run a query like this:

```
title:(+elasticsearch +"mastering book")
```

Of course, the previous query can also be expressed in the following way:

```
+title:elasticsearch +title:"mastering book"
```

Term modifiers

In addition to the standard field query with a simple term or clause, Lucene allows us to modify the terms we pass in the query with modifiers. The most common modifiers, which you are surely familiar with, are wildcards. There are two wildcards supported by Lucene the `?` and `*`. The first one will match any character and the second one will match multiple characters.



Please note by default these wildcard characters can't be used as the first character in a term because of the performance reasons.

In addition to that, Lucene supports fuzzy and proximity searches with the use of `~` character and an integer following it. When used with a single word term, it means that we want to search for terms that are similar to the one we've modified (so, called fuzzy search). The integer after the `~` character specifies the maximum number of edits that can be done to consider the term similar. For example, if we ran a query, such as `writer~2`, both the terms `writer` and `writers` would be considered a match.

When the `~` character is used on a phrase, the integer number we provide is telling Lucene how much distance between words is acceptable. For example, let's take the following query:

```
title:"mastering elasticsearch"
```

It would match the document with the `title` field containing `mastering elasticsearch`, but not `mastering book elasticsearch`. However, if we ran a query, such as `title:"mastering elasticsearch"~2`, it would result in both example documents matched.

In addition to that we can use boosting in order to increase our term importance by using the `^` character and providing a float number. The boost lower than one would result in decreasing the importance, boost higher than one will result in increasing the importance, and the default boost value is 1. Please refer to the *Default Apache Lucene scoring explained* section in *Chapter 2, Power User Query DSL*, for further reference what boosting is and how it is taken into consideration during document scoring.

In addition to all these, we can use square and curly brackets to allow range searching. For example, if we would like to run a range search on a numeric field we could run the following query:

```
price:[10.00 TO 15.00]
```

The above query would result in all documents with the `price` field between `10.00` and `15.00` inclusive.

In case of string based fields, we also can run a range query, for example:

```
name:[Adam TO Adria]
```

The previous query would result in all documents containing all the terms between `Adam` and `Adria` in the `name` field including them.

If you would like your range bound or bounds to be exclusive, use curly brackets instead of the square ones. For example, in order to find documents with the `price` field between `10.00` inclusive and `15.00` exclusive, we would run the following query:

```
price:[10.00 TO 15.00}
```

Handling special characters

In case you want to search for one of the special characters (which are `+`, `-`, `&&`, `|`, `!`, `(`, `)`, `{`, `}`, `[`, `]`, `^`, `"`, `~`, `*`, `?`, `:`, `\`, `/`), you need to escape it with the use of the backslash (`\`) character. For example, to search for `abc"efg` term you need to do something like this:

```
abc\"efg
```

Introducing ElasticSearch

If you hold this book in your hands, you are probably familiar with ElasticSearch, at least the core concepts and basic usage. However, in order to fully understand how this search engine works, let's discuss it briefly.

As you probably know ElasticSearch is production-ready software for building search-oriented applications. It was originally started by Shay Banon and published in February 2010. After that it has rapidly gained popularity just within a few years, and became an important alternative to other open source and commercial solutions. It is one of the most downloaded open source projects, hitting more than 200,000 downloads a month.

Basic concepts

Let's go through the basic concepts of ElasticSearch and its features.

Index

ElasticSearch stores its data in one or more indices. Using analogies from the SQL world, index is something similar to a database. It is used to store the documents and read them from it. As we already mentioned, under the hood, ElasticSearch uses Apache Lucene library to write and read the data from the index. What one should remember about is that a single ElasticSearch index may be built of more than a single Apache Lucene index, by using shards and replicas.

Document

Document is the main entity in the ElasticSearch world (and also in Lucene world). At the end, all use cases of using ElasticSearch can be brought to a point where it is all about searching for documents. Document consists of fields and each field has a name and one or many values (in this case, field is called multi-valued). Each document may have a different set of fields; there is no schema or imposed structure. It should look familiar (these are the same rules as for Lucene documents). In fact, ElasticSearch documents are stored as Lucene documents. From the client point of view, document is a JSON object (see more about JSON format at <http://en.wikipedia.org/wiki/JSON>).

Mapping

As you already read in the *Introducing Apache Lucene* section, all documents are analyzed before being stored. We can configure how the input text is divided into tokens, which tokens should be filtered out, or what additional processing, such as removing HTML tags, is needed. In addition, various features are offered by Elasticsearch, such as sorting needs information about fields contents. This is where mapping comes to play: it holds all of these information. Besides the fact that Elasticsearch can automatically discover field type by looking at its value, sometimes (in fact usually always) we will want to configure the mappings ourselves to avoid unpleasant surprises.

Type

Each document in Elasticsearch has its type defined. This allows us to store various document types in one index and have different mappings for different document types.

Node

The single instance of the Elasticsearch server is called a node. A single node Elasticsearch deployment can be sufficient for many simple use cases, but when you have to think about fault tolerance or you have lots of data that cannot fit in a single server, you should think about multi-node Elasticsearch cluster.

Cluster

Cluster is a set of Elasticsearch nodes that work together to handle the load bigger than single instance can handle (both in terms of handling queries and documents). This is also the solution which allows us to have uninterrupted work of application even if several machines (nodes) are not available due to outage or administration tasks, such as upgrade. The Elasticsearch provides clustering almost seamlessly. In our opinion, this is one of the major advantages over competition; setting up a cluster in Elasticsearch world is really easy.

Shard

As we said previously, clustering allows us to store information volumes that exceed abilities of a single server. To achieve this requirement, Elasticsearch spread data to several physical Lucene indices. Those Lucene indices are called **shards** and the process of this spreading is called sharding. Elasticsearch can do this automatically and all parts of the index (shards) are visible to the user as one-big index. Note that besides this automation, it is crucial to tune this mechanism for particular use case because the number of shard index is built or is configured during index creation and cannot be changed later, at least currently.

Replica

Sharing allows us to push more data into Elasticsearch that is possible for a single node to handle. Replicas can help where load increases and a single node is not able to handle all the requests. The idea is simple: create additional copy of a shard, which can be used for queries just as original, primary shard. Note that we get safety for free. If the server with the shard is gone, Elasticsearch can use replica and no data is lost. Replicas can be added and removed at any time, so you can adjust their numbers when needed.

Gateway

During its work, Elasticsearch collects various information about cluster state, indices settings, and so on. This data is persisted in the gateway.

Key concepts behind Elasticsearch architecture

ElasticSearch was built with few concepts in mind. The development team wanted to make it easy to use and scalable, and these core features are visible in every corner of Elasticsearch. From the architectural perspective, the main features are:

- Reasonable default values that allow the user to start using Elasticsearch just after installing it, without any additional tuning. This includes built-in discovery (for example, field types) and auto configuration.
- Working in distributed mode by default. Nodes assume that there are or will be a part of the cluster, and during setup nodes try to automatically join the cluster.

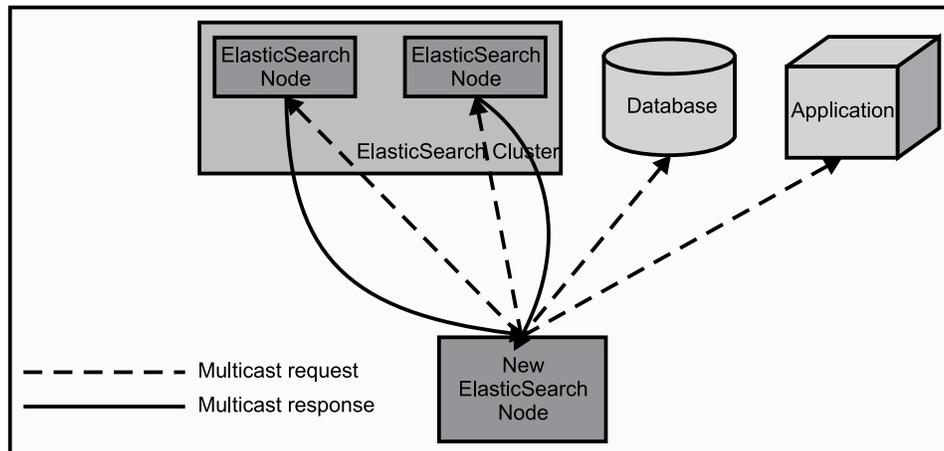
- Peer-to-peer architecture without **single point of failure (SPOF)**. Nodes automatically connect to other machines in the cluster for data interchange and mutual monitoring. This covers automatic replication of shards.
- Easily scalable both in terms of capacity and the number of data by adding new nodes to cluster.
- Elasticsearch does not impose restriction on data organization in the index. This allows users to adjust to existing data model. As we noted in type description, Elasticsearch supports multiple data types in a single index and adjustment to business model includes handling relation between documents (although, this functionality is rather limited).
- **Near Real Time (NRT)** searching and versioning. Because of distributed nature of Elasticsearch, there is no possibility to avoid delays and temporary differences between data located on the different nodes. Elasticsearch tries to reduce these issues and provide additional mechanisms as versioning.

Working of Elasticsearch

Let's now discuss briefly how Elasticsearch works.

The bootstrap process

When the Elasticsearch node starts, it uses multicast (or unicast, if configured) to find the other nodes in the same cluster (the key here is the cluster name defined in the configuration) and connect to them. You can see the process illustrated in the following figure:



In the cluster, one of the nodes is elected as the master node. This node is responsible for managing the cluster state and process of assigning shards to nodes in reaction of changes in cluster topology.



Note that a master node in ElasticSearch has no importance from the user perspective, which is different from other systems available (such as the databases). In practice you do not need to know which node is a master node; all operations can be sent to any node, and internally ElasticSearch will do all the magic. If necessary, any node can send subqueries parallel to other nodes and merge responses to return the full response to the user. All of this is done without accessing master node (nodes operate in peer-to-peer architecture).

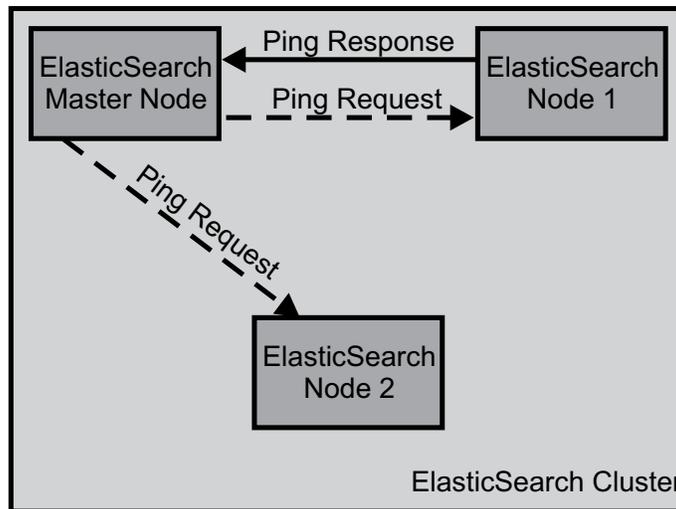
The master node reads the cluster state and if necessary, goes into recovery process. During this state, it checks which shards are available and decides which shards will be the primary shards. After this the whole cluster enters into yellow state.

This means that a cluster is able to run queries but full throughput and all possibilities are not achieved yet (it basically means that all primary shard are allocated, but replicas are not). The next thing to do is find duplicated shards and treat them as replicas. When a shard has too few replicas, the master node decides where to put missing shards and additional replica are created based on a primary shard. If everything went well, the cluster enters into a green state (which means that all primary shard and replicas are allocated).

Failure detection

During normal cluster work, the master node monitors all the available nodes and checks if they are working. If any of them are not available for configured amount of time, the node is treated as broken and process of handling failure starts. This may mean rebalancing of the cluster – shards, which were present on the broken node are gone and for each such shard other nodes have to take responsibility. In other words, for every lost primary shard, a new primary shard should be elected from the remaining replicas of this shard. The whole process of placing new shards and replicas can (and usually should) be configured to match our needs. More information about it can be found in *Chapter 4, Index Distribution Architecture*.

Just to illustrate how it works, let's take an example of three nodes cluster, there will be a single master node and two data nodes. The master node will send the ping requests to other nodes and wait for the response. If the response won't come (actually how many ping requests may fail depends on the configuration), such a node will be removed from the cluster.



Communicating with ElasticSearch

We talked about how ElasticSearch is built, but after all, the most important part for us is how to feed it with data and how to build your queries. In order to do that ElasticSearch exposes a sophisticated API. The primary API is REST based (see http://en.wikipedia.org/wiki/Representational_state_transfer) and is easy to integrate with practically any system that can send HTTP requests.

ElasticSearch assumes that data is sent in the URL, or as the request body as JSON document (<http://en.wikipedia.org/wiki/JSON>). If you use Java or language based on JVM, you should look at Java API, which in addition to everything that is offered by the REST API has built-in cluster discovery.

It is worth mentioning that Java API is also used internally by the ElasticSearch itself to do all the node to node communication. You will find more about Java API in *Chapter 8, ElasticSearch Java APIs*, but for now let's briefly look on the possibilities and functionality exposed by this API. Note that we treat this as a little reminder (this book assumes that you have used these elements already). If not, we strongly suggest reading about this, for example, our *ElasticSearch Server* book covers all this information.

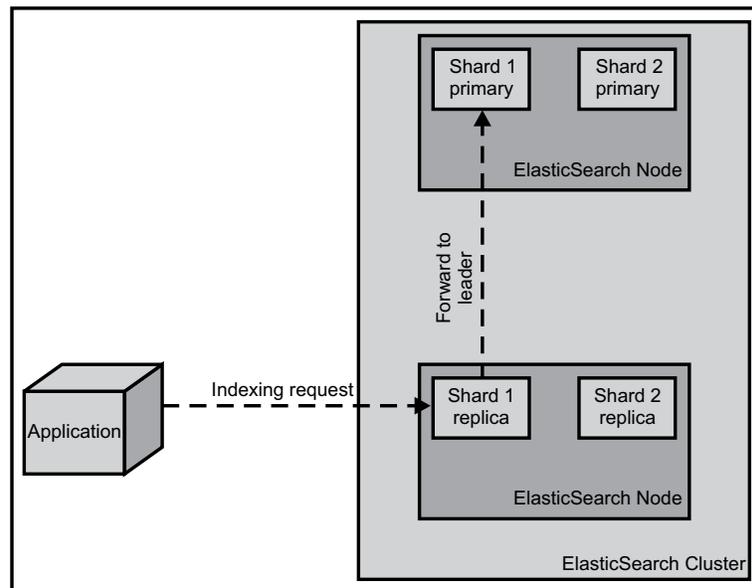
Indexing data

ElasticSearch has four ways of indexing data. The easiest way is using the index API, which allows you to send one document to a particular index. For example, by using the curl tool (see <http://curl.haxx.se/>), we can create a new document by using the following command:

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

The second and third way allows us to send many documents using the bulk API and the UDP bulk API. The difference between methods is the connection type. Common bulk command sends documents by HTTP protocol and UDP bulk sends these using connectionless datagram protocol. This is faster but not so reliable. The last method uses plugins, called rivers. The river runs on the ElasticSearch node and is able to fetch data from the external systems.

One thing to remember is that the indexing only takes place on the primary shard, not on the replica. If the indexing request will be sent to a node, which doesn't have the correct shard or contains replica, it will be forwarded to the primary shard.

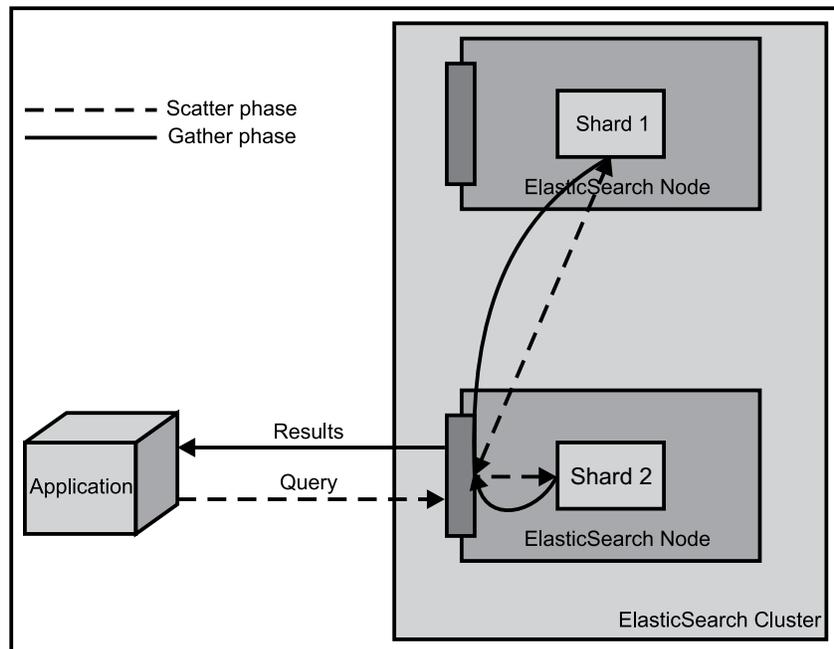


Querying data

Query API is a big part of Elasticsearch API. Using the Query DSL (JSON based language for building complex queries), we can:

- Use various query types including: simple term query, phrase, range, boolean, fuzzy, span, wildcard, spatial, and other query
- Build complex queries with the use of simple queries combined together
- Filter documents, throwing away ones, which does not match selected criteria without influencing the scoring
- Find documents similar to given document
- Find suggestions and corrections of a given phrase
- Build dynamic navigation and calculate statistics using faceting
- Use prospective search and find queries matching given document

When talking about querying, the important thing is that query is not a simple, single stage process. In general, the process can be divided into two phases, the scatter phase and the gather phase. The scatter phase is about querying all the relevant shards of your index. The gather phase is about gathering the results from the relevant shards, combining them, sorting, processing, and returning to the client.





You can control the scatter and gather phases by specifying the search type to one of the six values currently exposed by Elasticsearch. We've talked about query scope in our previous book *ElasticSearch Server*, by Packt Publishing.

Index configuration

We already talked about automatic index configuration and ability to guess document field types and structure. Of course, Elasticsearch gives us the possibility to alter this behavior. We may, for example, configure our own document structure with the use of mappings, set the number of shards and replicas index will be built of, configure the analysis process, and so on.

Administration and monitoring

The administration and monitoring part of API allows us to change the cluster settings, for example, to tune the discovery mechanism or change index placement strategy. You can find various information about cluster state or statistics regarding each node and index. The API for the cluster monitoring is very comprehensive and example usage will be discussed in *Chapter 5, ElasticSearch Administration*.

Summary

In this chapter we've looked at the general architecture of Apache Lucene, how it works, how the analysis process is done, and how to use Apache Lucene query language. In addition to that we've discussed the basic concepts of Elasticsearch, its architecture, and internal communication.

In the next chapter you'll learn about the default scoring formula Apache Lucene uses, what the query rewrite process is, and how it works. In addition to that we'll discuss some of the Elasticsearch functionality, such as query rescore, multi near real-time get, and bulk search operations. We'll also see how to use the update API to partially update our documents, how to sort our data, and how to use filtering to improve performance of our queries. Finally, we'll see how we can leverage the use of filters and scopes in the faceting mechanism.

2

Power User Query DSL

In the previous chapter, we looked at what Apache Lucene is, how its architecture looks, and how the analysis process is handled. In addition to that we've seen what Lucene query language is and how to use it. We also discussed Elasticsearch, its architecture, and core concepts. In this chapter, we will dive deep into Elasticsearch focusing on the Query DSL. We will first go through how Lucene scoring formula works before turning to advanced queries. By the end of this chapter we will have covered:

- How default Apache Lucene scoring formula works
- What query rewrite is
- How does query rescore work
- How to send multiple near real-time get operations in a single request
- How to send multiple queries in a single request
- How to sort our data including nested documents and multivalued fields
- How to update our documents that are already indexed
- How to use filters to optimize our queries
- How to use filters and scopes in Elasticsearch faceting mechanism

Default Apache Lucene scoring explained

One important thing when talking about query relevance is how the score of the document is calculated for a query. What is the score? The **score** is a parameter that describes how well the document matched the query. In this section, we'll look at the default Apache Lucene scoring mechanism: the **TF/IDF (term frequency/inverse document frequency)** algorithm and how it affects the returned document. Knowing how this works is valuable when designing complicated queries and choosing which queries parts should be more relevant than others.

When a document is matched

When a document is returned by Lucene it means that it matched the query we sent. In this case, the document is given a score. The higher the score value, the more relevant the document is, at least at the Apache Lucene level and from the scoring formula point of view. Naturally, the score calculated for the same document on two different queries will be different and comparing scores between queries usually doesn't make much sense. One should remember that not only should we avoid comparing the scores of individual documents returned by different queries, but we should also avoid comparing the maximum score calculated for different queries. This is because the score depends on multiple factors, not only the boosts and query structure, but also on how many terms were matched, in which fields, and the type of matching that was used on query normalization, and so on. In extreme cases, a similar query may result in totally different scores for a document, only because we've used a custom score query or the number of matched terms increased dramatically.

For now, let's get back to the scoring. In order to calculate the score property for a document, multiple factors are taken into account:

- **Document boost:** It is the boost value given for a document during indexing.
- **Field boost:** It is the boost value given for a field during querying.
- **Coord:** It 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:** It is a term based factor telling the scoring formula how rare the given term is. The lower the inverse document frequency is, the rarer the term is. The scoring formula uses this factor to boost documents that contain rare terms.

- **Length norm:** It is a field based factor for normalization based on the number of terms a given field contains (calculated during indexing and stored in the index). The longer the field, the lesser boost this factor will give, which means that Apache Lucene scoring formula will favor documents with fields containing lower terms.
- **Term frequency:** It is a term based factor describing how many times given term occurs in a document. The higher the term frequency the higher the score of the document will be.
- **Query norm:** It is a query based normalization factor that is calculated as sum of a squared weight of each of the query terms. Query norm is used to allow score comparison between queries, which we said is not always easy and possible.

The TF/IDF scoring formula

Now let's look at how the scoring formula looks. Keep in mind, that in order to adjust your query relevance, you don't need to understand that, but it is very important to at least know how it works.

The Lucene conceptual formula

The conceptual version of the TF/IDF formula looks like:

$$score(q, d) = coord(q, d) * queryBoost(q) * \frac{V(q) * V(d)}{|V(q)|} * lengthNorm(d) * docBoost(d)$$

The previous presented formula is a representation of Boolean model of Information Retrieval combined with Vector Space Model of Information Retrieval. Let's not discuss it and let's just jump into the practical formula, which is implemented by Apache Lucene and is actually used.



The information about Boolean model and Vector Space Model of Information Retrieval are far beyond the scope of this book. If you would like to read more about it, start with http://en.wikipedia.org/wiki/Standard_Boolean_model and http://en.wikipedia.org/wiki/Vector_Space_Model.

The Lucene practical formula

Now let's look at the practical formula Apache Lucene uses:

$$\text{score}(q, d) = \text{coord}(q, d) * \text{queryNorm}(q) * \sum_{t \text{ in } q} (\text{tf}(t \text{ in } d) * \text{idf}(t)^2 * \text{boost}(t) * \text{norm}(t, d))$$

As you may be able to see, the score factor for the document is a function of query q and document d . There are two factors that are not dependent directly on query terms, the `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 previously.

Sounds a bit complicated, right? Don't worry, you don't need to remember all of that. What you should be aware of is what matters when it comes to document score. Basically there are a few rules which come from the previous equations:

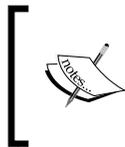
- The more rare the term matched is, the higher the score the document will have
- The smaller the document fields are (contain less terms), the higher the score the document will have
- The higher the boost (both given during indexing and querying), the higher the score the document will have

As we can see, Lucene will give the highest score for the documents that have many uncommon query terms matched in the document contents, have shorter fields (less terms indexed), and will also favor rarer terms instead of the common ones.

 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_5_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

The Elasticsearch point of view

On top of all this is Elasticsearch which leverages Apache Lucene and thankfully allows us to change the default scoring algorithm (more about this can be found in the *Altering Apache Lucene scoring* section, *Chapter 3, Low-level Index Control*). But remember, Elasticsearch is more than just Lucene, because we are not bound to only rely on Apache Lucene scoring. We have different types of queries where we can strictly control how the score of the documents is calculated (such as the `custom_boost_factor` query, `constant_score` query, and `custom_score` query), we are allowed to use scripting to alter score of the documents, we can use the `rescore` functionality introduced in Elasticsearch 0.90 to recalculate the score of the returned documents, by another query run against top N documents, and so on.



For more information about the queries from Apache Lucene point of view, please refer to Javadocs, for example, the one available at http://lucene.apache.org/core/4_5_0/queries/org/apache/lucene/queries/package-summary.html.

Query rewrite explained

If you ever used queries, such as the prefix query and the wildcard query, basically any query that is said to be multiterm, you've probably heard about query rewriting. Elasticsearch (actually Apache Lucene to be perfectly clear) does that because of the performance reasons. The rewrite process is about changing the original, expensive query to a set of queries that are far less expensive from Lucene's point of view.

Prefix query as an example

The best way to illustrate how the rewrite process is done internally is to look at an example and see what terms are used instead of the original query term. Let's say, we have the following data in our index:

```
curl -XPUT 'localhost:9200/clients/client/1' -d
'{
  "id": "1", "name": "Joe"
}'
curl -XPUT 'localhost:9200/clients/client/2' -d
'{
  "id": "2", "name": "Jane"
}'
```

```
curl -XPUT 'localhost:9200/clients/client/3' -d
'{
  "id": "3", "name": "Jack"
}'
curl -XPUT 'localhost:9200/clients/client/4' -d
'{
  "id": "4", "name": "Rob"
}'
curl -XPUT 'localhost:9200/clients/client/5' -d
'{
  "id": "5", "name": "Jannet"
}'
```

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.

What we would like is to find all the documents that start with the j letter. Simple as that, we run the following query against our `clients` index:

```
curl -XGET 'localhost:9200/clients/_search?pretty' -d '{
  "query" : {
    "prefix" : {
      "name" : "j",
      "rewrite" : "constant_score_boolean"
    }
  }
}'
```

We've used a simple `prefix` query; we've said that we would like to find all the documents with the j letter in the `name` field. We've 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 later part of this section.

As the response to the previous query, we've got the following:

```
{
  ...
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "clients",
```

```

    "_type" : "client",
    "_id" : "5",
    "_score" : 1.0, "_source" : {"id":"5", "name":"Jannet"}
  }, {
    "_index" : "clients",
    "_type" : "client",
    "_id" : "1",
    "_score" : 1.0, "_source" : {"id":"1", "name":"Joe"}
  }, {
    "_index" : "clients",
    "_type" : "client",
    "_id" : "2",
    "_score" : 1.0, "_source" : {"id":"2", "name":"Jane"}
  }, {
    "_index" : "clients",
    "_type" : "client",
    "_id" : "3",
    "_score" : 1.0, "_source" : {"id":"3", "name":"Jack"}
  } ]
}
}

```

As you can see, in response we've got the three documents that have the contents of the name field starting with the desired character. We didn't specify the mappings explicitly, so Elasticsearch has guessed the name field mapping and have set it to string-based and analyzed. You can check that by running the following command:

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

ElasticSearch's response will be similar to the following code:

```

{
  "client" : {
    "properties" : {
      "id" : {
        "type" : "string"
      },
      "name" : {
        "type" : "string"
      }
    }
  }
}

```

Getting back to Apache Lucene

Now let's take a step back and look at Apache Lucene again. If you recall what Lucene inverted index is built of, you can tell that it contains a term, count, and document pointer (if you don't recall, please refer to the *Introduction to Apache Lucene* section in *Chapter 1, Introduction*). So, let's see how the simplified view of the index may look for the previous data we've put to the `clients` index:

| Term | Count | Docs |
|------|-------|---------|
| jack | 1 | <3> |
| jane | 2 | <2> <5> |
| joe | 1 | <1> |
| rob | 1 | <4> |

What you see in the column with the term text is quite important. If we look at Elasticsearch and Apache Lucene internals, you can see that our prefix query was rewritten to the following Lucene query:

```
ConstantScore(name:jack name:jane name:joe)
```

This means that our prefix query was rewritten to a constant score query, that consists of Boolean query, which is built of three terms queries. What Apache Lucene did was enumerating the terms from the index and constructing a new query using this information. Of course, if we would be allowed to compare a query that is not rewritten to a rewritten one, we would see a performance increase with the rewritten query, especially on indices with a larger amount of distinct terms.

If we would like to construct the rewritten query manually, we could run a query something similar to the following code (we've put the contents of this query in `constant_score_query.json` file):

```
{
  "query" : {
    "constant_score" : {
      "query" : {
        "bool" : {
          "should" : [
            {
              "term" : {
                "name" : "jack"
              }
            }
          ]
        }
      }
    }
  },
}
```

```
{
  "term" : {
    "name" : "jane"
  }
},
{
  "term" : {
    "name" : "joe"
  }
}
]
}
}
```

Now let's look at the possibilities of configuring the behavior of query rewriting.

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, for example, like the following code:

```
{
  "query" : {
    "prefix" : {
      "name" : "j",
      "rewrite" : "constant_score_boolean"
    }
  }
}
```

Now, let's look at what 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 Boolean query. This query rewrite method may be CPU intensive (because the score for each term is calculated and kept), and for queries that have many terms may exceed the Boolean query limit, which is set to 1024. Also, this query keeps the computed score. The default Boolean query limit can be changed by setting the `index.query.bool.max_clause_count` property in the `elasticsearch.yml` file. However, please remember that the more Boolean queries are produced, the lower the query performance may be.
- `constant_score_boolean`: This rewrite method is similar to the `scoring_boolean` rewrite method described previously, but less CPU demanding because scoring is not computed and instead of that, each term receives a score equal to the query boost, which is one by default, and can be set using the `boost` property. Similar to the `scoring_boolean` rewrite method, this method can also hit the maximum Boolean clauses limit.
- `constant_score_filter`: As Apache Lucene Javadocs state, this rewrite method rewrites the query by creating a private filter by visiting each term in a sequence and marking all 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 not small.
- `top_terms_N`: A rewrite method that translates each generated term into a Boolean should be a 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 Boolean clauses limit.
- `top_terms_boost_N`: It is a rewrite method similar to the `top_terms_N` one, but the scores are only computed as the boost, not the query.



When the `rewrite` property is set to `constant_score_auto` value 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.

Let's go through one more example. If we would like our example query to use the `top_terms_N` with `N` equal to 10, our query would look like this:

```
{
  "query" : {
    "prefix" : {
      "name" : "j",
      "rewrite" : "top_terms_10"
    }
  }
}
```

Before we finish the query rewrite section of this chapter, we should ask ourselves one last question: when to use which rewrite types? The answer to this question greatly depends on your use case, but to summarize, if you can live with lower precision (but higher performance), you can go for the top `N` rewrite method. If you need high precision (but lower performance), choose the Boolean approach.

Rescore

Sometimes it is handy to change the ordering of documents already returned by the query. The reasons for such behavior can vary. One of the reasons may be performance, for example, calculating target ordering is very costly in terms of performance and we would like to do this on the subset of documents returned by the original query. You can imagine that rescoring gives us many great opportunities for business use cases. Now let's look at this functionality and how it can be useful for us.

Understanding rescore

Rescore in the Elasticsearch is the process of recalculation of the score for a defined number of documents returned by the query. This means that Elasticsearch takes first `N` documents for given query and calculates their score using provided rescore definition.

Example Data

Our example data is stored in the `documents.json` file (provided with the book) and can be indexed with the following command:

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

Query

Let's start with a simple query that looks like this:

```
{
  "fields" : ["title", "available"],
  "query" : {
    "match_all" : {}
  }
}
```

It returns all the documents from the index. Every document returned by the query will have the score equal to 1.0, because of the `match_all` query type. This is enough to show how `rescore` affects our result set. One more thing about the query is that we've specified which fields we want in the results for each document: `title` and `available`.

Structure of the rescore query

The example query with `rescore` looks like this:

```
{
  "fields" : ["title", "available"],
  "query" : {
    "match_all" : {}
  },
  "rescore" : {
    "query" : {
      "rescore_query" : {
        "custom_score" : {
          "query" : {
            "match_all" : {}
          },
          "script" : "doc['year'].value"
        }
      }
    }
  }
}
```

In the previous example, in the `rescore` object you can see a `query` object. When this book was written `query` was the only option, but in the future versions we may expect other ways to affect the resulting score. In our case, we use a simple query that returns all the documents and every document has score equal to value of `year` field (please, don't even ask about the business sense of this query!).

If we save this query in the `query.json` file and send it using the command `curl localhost:9200/library/book/_search?pretty -d @query.json`, we should see the following documents (we omit the structure of the response):

```
"_score" : 1962.0,
"title" : "Catch-22",
"available" : false
"_score" : 1937.0,
"title" : "The Complete Sherlock Holmes",
"available" : false
"_score" : 1930.0,
"title" : "All Quiet on the Western Front",
"available" : true
"_score" : 1887.0,
"title" : "Crime and Punishment",
"available" : true
```

As we can see, Elasticsearch found all the documents from the original query. Now look at the score of the documents. Elasticsearch took the first N documents and applied the second query to them. In the result, the score of those documents is the sum of the score from the first and the second query.

As you know, scripts execution can be demanding when it comes to performance; that's why we've used it as the second query. If our initial, `match_all`, query (used in the previous example) would return thousands of results, calculating script based scoring for all those can affect query performance. Rescore gave us the possibility to only calculate such scoring on the top N documents and thus reduce the performance impact.

Now let's see how to tune this behavior and what parameters are available.

Rescore parameters

In the query under the `rescore` object, we may use the following parameters:

- `window_size` (defaults to sum of the `from` and `size` parameters): It gives the information connected with the `N` documents mentioned previously. The `window_size` parameter is the number of documents used for rescoring on every shard.
- `query_weight` (defaults to one): The resulting score of the original query will be multiplied by this value before adding the score generated by `rescore`.
- `rescore_query_weight` (defaults to one): The resulting score of the `rescore` will be multiplied by this value before adding the score generated by the original query.
- `rescore_mode` (defaults to `total`): Introduced in ElasticSearch 0.90.3 (before 0.90.3 ElasticSearch behaved like this parameter would be set to `total`), it defines how the score of the rescored documents are calculated. The possible values are `total`, `max`, `min`, `avg`, and `multiply`. When setting this parameter to `total`, the final score of the document will be equal to the sum of original query score and the `rescore` query score. When setting this parameter to `max`, the maximum of original query score and `rescore` query score will be given to the document. Similar to `max`, when setting the `rescore_mode` to `min`, the minimum value of the original query score and `rescore` query score will be given to the document. You can guess that when choosing `avg`, the average of both query scores will be given to the document, and when setting this parameter to `multiply`, those scores will be multiplied.

For example, the target score for the document when using `rescore_mode` equal to `total` is equal to:

```
original_query_score * query_weight + rescore_query_score *
rescore_query_weight
```



Please remember that the `rescore_mode` parameter is not available until ElasticSearch 0.90.3. In the versions prior to ElasticSearch 0.90.3, the `rescore` mechanism acts as the default value of `total` would be used.

To sum up

Sometimes we want to show results, where the ordering of the first documents on the page is affected by some additional rules. Unfortunately, this cannot be achieved by the `rescore` functionality. The first idea points to the `window_size` parameter, but this parameter, in fact, is not connected with the first documents on the result list, but with number of results returned on every shard. In addition, `window_size` cannot be less than the page size. (If it is less, Elasticsearch silently uses page size). Also, one very important thing, rescoring cannot be combined with sorting, because sorting is done before the changes to the documents score are done by rescoring, and thus sorting won't take the newly calculated score into consideration.

The previously mentioned limitations and the lack of possibility of using several different rescorings (for example, one `rescore` definition for first three positions on the result list and the second for the following five) limits the usefulness of this functionality, and should be remembered before using this functionality.

Bulk Operations

In several examples present in this book, you are holding the present sample data in bulk indexing format, which allows us to effectively send data to Elasticsearch. However, Elasticsearch exposes batch functionality for fetching the data and also for searching. It is worth mentioning that similar to bulk indexing, these solutions allows us to group several request together, where each request may have its own target index and type. Let's look at the possibilities of those.

MultiGet

The MultiGet operation is available via the `_mget` endpoint and allows fetching several documents using a single request. Similar to the RealTime Get functionality documents are fetched in real-time fashion, Elasticsearch will return documents that were sent to indexing regardless of whether they are already available for searching or are still waiting to be visible for queries. Let's look at the example command:

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "ids" : [1,3]
}'
```

It fetches two documents with given identifiers from the index and type defined in the URL. In the previous example, we've also set the list of fields that we would like to be returned (with the use of the field `request` parameter). Elasticsearch returns the list of documents in the following form:

```
{
  "docs" : [ {
    "_index" : "library",
    "_type" : "book",
    "_id" : "1",
    "_version" : 1,
    "exists" : true,
    "fields" : {
      "title" : "All Quiet on the Western Front"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "3",
    "_version" : 1,
    "exists" : true,
    "fields" : {
      "title" : "The Complete Sherlock Holmes"
    }
  } ]
}
```

Our example can also be rewritten into more compact form:

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "docs" : [{ "_id" : 1}, { "_id" : 3}]
}'
```

This form is handier for fetching documents from various indices and types, or when we want to return various sets of fields. In this case, the information contained in the address is treated as default value. For example, let's look at the following query:

```
curl localhost:9200/library/book/_mget?fields=title -d '{
  "docs" : [
    { "_index": "library_backup", "_id" : 1, "fields": ["otitle"]},
    { "_id" : 3}
  ]
}'
```

This query returns two documents with identifiers equal to 1 and 3, but the first of them is fetched from the `library_backup` index and second one is fetched from the `library` index (because the `library` index is defined in the URL and thus is taken as the default value). In addition to that, in case of the first document, we limit the returned fields to one named `otitle`.



With the release of Elasticsearch 1.0, the MultiGet API will allow us to specify the version of the document that we want to operate on. If the version of the document won't match the one provided in the request, Elasticsearch will not perform the MultiGet operation. The additional parameters are `version`, which allows us to pass the version we are interested in; the second parameter is called `version_type` and supports two values: `internal` and `external`.

MultiSearch

Similar to MultiGet, the MultiSearch functionality allows us to group several search requests into one package. However, the grouping is slightly different, more similar to how bulk indexing looks. Elasticsearch parses the input as lines, where every pair of lines contains information about the target index along with additional parameters and a query itself. Look at the following, simple example:

```
curl localhost:9200/library/books/_msearch?pretty --data-binary '
{ "type" : "book" }
{ "filter" : { "term" : { "year" : 1936} }}
{ "search_type": "count" }
{ "query" : { "match_all" : {} }}
{ "index" : "library-backup", "type" : "book" }
{ "sort" : ["year"] }
'
```

As you can see, the request was sent to `_msearch` endpoint. The index and type given in the path is optional and is used as the default value for odd lines, which define target index and type for the query. These lines can contain information about type of the search (`search_type`) and information about routing or hints for query execution (`preference`). Because each of these parameters is not obligatory, in the special case, line can contain empty object (`{}`) or even an empty line. The even lines of the request are responsible for carrying the actual queries. Now let's look at the result for the previous request:

```
{
  "responses" : [ {
    "took" : 2,
    "timed_out" : false,
    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : {
      "total" : 1,
      "max_score" : 1.0,
      "hits" : [ {
        ...
      } ]
    }
  },
  ...
  {
    "took" : 2,
    "timed_out" : false,
    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : {
      "total" : 4,
      "max_score" : null,
      "hits" : [ {
        ...
      } ]
    }
  }
]
```

The returned JSON contains results response object with array of particular responses from queries from batch. As we said earlier, MultiSearch allows us to group totally independent queries together, so documents returned by each query (omitted in example) may have different structure according to the data placed in the particular index.



Note that like in bulk indexing, request does not allow any additional indentation. Every line has clear defined purpose: control information or query. So make sure that every line is followed by appropriate new line character and the tool used for sending query does not change anything in data send. This is why, in the curl command, we use `--data-binary` instead of `-d`, which does not preserve new line characters.

Sorting data

When you send your query to Elasticsearch, the returned documents list is by default sorted by calculated document score (we already talked about it in the *Default Apache Lucene scoring explained* section in this chapter). This is usually exactly what we want: the first document from the results is the one that the query wanted to capture. However, there are times when we want to change this ordering. It is very easy in our example, because we've used single term string data. Let's look at the following example:

```
{
  "query" : {
    "terms" : {
      "title" : [ "crime", "front", "punishment" ],
      "minimum_match" : 1
    }
  },
  "sort" : [
    { "section" : "desc" }
  ]
}
```

The previous query returns all documents with at least one of the mentioned terms in the `title` field and sorts those results on the basis of the `section` field.

We can also define sorting behavior for documents that doesn't have the value in the `section` field by adding the `missing` property to the `sort` section. For example, the `sort` section of the previous query could look like this:

```
{ "section" : { "order" : "asc", "missing" : "_last" } }
```

Sorting with multivalued fields

With versions prior to 0.90, Elasticsearch had problems with sorting on the field that had multiple values in their contents. Attempts to sort on this field resulted in an error similar to the following one: [Can't sort on string types with more than one value per doc, or more than one token per field]. In fact, sorting on such fields doesn't make much sense as Elasticsearch doesn't know which value to choose from the ones that were indexed. However, with the introduction of Elasticsearch 0.90, we are allowed to sort on a multivalued field. For example, let's say that our data contains the `release_dates` field that can have multiple release dates for a movie (for example, in different countries). If we are using Elasticsearch 0.90, we could send the following query:

```
{
  "query" : {
    "match_all" : {}
  },
  "sort" : [
    { "release_dates" : { "order" : "asc", "mode" : "min" } }
  ]
}
```

Note that in our case, the `query` section is redundant and assumed by default, so in the next example we will omit it.

In this case, Elasticsearch will choose the minimal value from the `release_dates` field for every document and sort on the basis of that value. The `mode` parameter can be set to the following values:

- `min`: It is the default value for ascending sorting. Elasticsearch takes the lowest value from available tokens for each document.
- `max`: It is the default value for descending sorting. Elasticsearch takes the greatest value from tokens for each document.
- `avg`: Elasticsearch takes an average from tokens in the field for each document.
- `sum`: Elasticsearch takes sum of all the values from the field for each document.

Note that the last two can be only used with numeric fields. However, the current implementation accepts `avg` and `sum` for text fields, but the results are not as you would expect and it is not recommended for usage.

Sorting with multivalued geo fields

ElasticSearch version 0.90.0RC2 introduces the possibility of sorting one field with multiple coordinates. This feature works exactly the same as previously mentioned sorting, of course from the user's perspective. Let's look at this a little bit closer by using a real example. We will try to find the nearest branch in a given country. For example, let's assume that we have the following mappings:

```
{
  "mappings": {
    "poi": {
      "properties": {
        "country": { "type": "string" },
        "loc": { "type": "geo_point" }
      }
    }
  }
}
```

And now a single sample data record that looks like this:

```
{ "country": "UK", "loc": ["51.511214,-0.119824", "53.479251,
-2.247926", "53.962301,-1.081884"] }
```

Our query is a very simple one, which looks like this:

```
{
  "sort": [{
    "_geo_distance": {
      "loc": "51.511214,-0.119824",
      "unit": "km",
      "mode": "min"
    }
  }]
}
```

As you can see, we have a single document with multiple geographical points in it. Let's now try to run our query against that document and see the results:

```
{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : null,
    "hits" : [ {
      "_index" : "map",
      "_type" : "poi",
      "_id" : "1",
      "_score" : null, "_source" : {
        "country": "UK", "loc": ["51.511214,-0.119824",
          "53.479251,-2.247926", "53.962301,-1.081884"] }
      ,
      "sort" : [ 0.0 ]
    } ]
  }
}
```

As you can see, the query gives us a sort section like this one: `"sort" : [0.0]`. This is because our geographical point in the query is exactly the same as the one defined in the documents. However, if you would change the `mode` property of the query to `max`, the results would be different and the highlighted part would look like this:

```
"sort" : [ 280.4459406165739 ]
```



ElasticSearch 0.90.1 introduced the possibility of using the `avg` value as the value of the `mode` property for geographical distance sorting.

Sorting with nested objects

One last thing about sorting and Elasticsearch 0.90 and newer is that we are now allowed to sort using fields defined in the nested objects. Using fields from nested documents for sorting works both for documents with explicit nested mappings (when using `type="nested"` in the mappings), as well as, when using the object type. However, there are some slight differences one needs to remember.

Assume that the index contains the following data:

```
{
  "country": "PL", "cities": { "name": "Cracow", "votes": {
    "users": "A" }}
}
{
  "country": "EN", "cities": { "name": "York", "votes": [{"users":
    "B"}, {"users": "C" }]}
}
{
  "country": "FR", "cities": { "name": "Paris", "votes": {
    "users": "D" } }
}
```

As you can see, there are repeatedly nested objects and some data may contain multiple values (for example, multiple votes).

Let's look at the following query:

```
{
  "sort": [{"cities.votes.users": { "order": "desc", "mode":
    "min" }}]
}
```

The previous query will result in documents that are sorted in the descending order by the lowest value taken from the table users. However, if we used a subdocument that was indexed as an object type, we could simplify the query to the following one:

```
{
  "sort": [{"users": { order: "desc", mode: "min" }}]
}
```

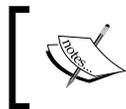
We can simplify the query because when using the object type, the whole object structure is stored as a single Lucene document. In case of nested type, Elasticsearch requires more accurate field information, because those documents are actually separate Lucene documents. Sometimes it is more convenient to just use notation with the `nested_path` property and write a query like this:

```
{
  "sort": [{ "users": { "nested_path": "cities.votes", "order":
    "desc", "mode": "min" }}]
}
```

Please note that we can also use the `nested_filter` parameter, which works only with the nested documents (the ones explicitly marked as nested). Thanks to this, we can provide a filter that will exclude documents from sorting, but not from the query results.

Update API

When you index a new document, the underlying Lucene library analyzes every field and generates token stream, which after additional filtering hits the inverted index. During this process, some input information is thrown away as unnecessary. This information may be, for example, position of particular words (if term vectors are not stored), some words (when using stop words or change words into its synonyms), or inflections (when using stemming). This is why there is no possibility of updating a Lucene document in the index and every time, when we want to change a document, we have to send all the fields to the index. Elasticsearch bypasses this problem by using `_source` pseudo field for storing and retrieving the original field values of the document. When we want to update a document, Elasticsearch takes the value of the written in the `_source` field, makes the desired changes and sends the new document to the index. Of course, the `_source` field must be enabled for this feature to work. The important limitation is that the update command can only update one particular document, and update by query is still officially unsupported.



If you are not familiar with how Apache Lucene analysis works or any of the mentioned terms, please refer to the *Introduction to Apache Lucene* section in *Chapter 1, Introduction to ElasticSearch*.

From the API point of view, the update is executed by sending a request to the endpoint, which address is built by adding `_update` to the address of the particular document, for example, `/library/book/1/_update`. Now it's time to describe what this functionality offers.

As an example, for the rest of this section we will use the document indexed by the following command:

```
curl -XPUT localhost:9200/library/book/1 -d '{
  "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
}'
```

Simple field update

The first use case is to change a single field value of chosen document. For example, let's look at the following command:

```
curl -XPOST localhost:9200/library/book/1/_update -d '{
  "doc" : {
    "title" : "The Complete Sherlock Holmes Book",
    "year" : 1935
  }
}'
```

In the previous example, we've changed two fields in the document, the `title` and the `year` fields. Elasticsearch responds with a reply similar to the one we see when we send the document for indexing:

```
{"ok":true,"_index":"library","_type":"book","_id":"1","_version":
2}
```

Let's now fetch the document and see if the fields were updated by running the following command:

```
curl -XGET localhost:9200/library/book/1?pretty
```

The response to the above command is as follows:

```
{
  "_index" : "library",
  "_type" : "book",
  "_id" : "1",
  "_version" : 2,
  "exists" : true, "_source" : {"title":"The Complete Sherlock
  Holmes Book","author":"Arthur Conan
  Doyle","year":1935,"characters":["Sherlock Holmes","Dr.
  Watson","G.
  Lestrade"],"tags":[],"copies":0,"available":false,
  "section":12}
}
```

As we can see in the `_source` field, the `title` and the `year` fields were updated. Let's go to the next example which uses scripting.

Conditional modifications using scripting

Sometimes it is convenient to add some additional logic when modifying a document and that's why Elasticsearch allows us to use scripting along with the update API. For example, we can send a request like the following one:

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "if(ctx._source.year == start_date) ctx._source.year
    = new_date; else ctx._source.year = alt_date;",
  "params" : {
    "start_date" : 1935,
    "new_date" : 1936,
    "alt_date" : 1934
  }
}'
```

As you can see, the `script` field defines what to do with the current document. This can be any script. We can also refer to the `ctx` variable holding document source. As usually, we can define additional variables, which can be used in script. Using `ctx._source`, we can modify current fields or create new ones (ElasticSearch will create a new field, when you refer to the field, which does not exist). This is exactly what happened in the example previously in `ctx._source.year = new_date`. We can also remove fields using the `remove()` method, for example:

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx._source.remove(\"year\");"
}'
```

Creating and deleting documents using the Update API

Update API is not only about the modification of a single field, but it can also be used to manipulate whole documents. The `upsert` feature gives us power to create a new document when document addresses by the URL we've used do not exist. Let's look at the following command:

```
curl localhost:9200/library/book/1/_update -d '{
  "doc" : {
    "year" : 1900
  },
  "upsert" : {
    "title" : "Unknown Book"
  }
}'
```

It sets the `year` field for the existing document (one in the `library` index, with `book` type and identifier of `1`). However, if the document is non-existing, it would have been created with the `title` field given in the `upsert` section. Just for the record, this example can also be rewritten using scripting:

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx._source.year = 1900",
  "upsert" : {
    "title" : "Unknown Book"
  }
}'
```

The last interesting feature allows us to conditionally remove the whole document. This can be achieved by setting `ctx.op` value to `delete`. For example, the following command will remove the document from the index:

```
curl localhost:9200/library/book/1/_update -d '{
  "script" : "ctx.op = \"delete\""
}'
```

Of course, we may implement more advanced logic using scripts and delete the document only when a certain condition is met.

Using filters to optimize your queries

ElasticSearch allows us to make different kinds of queries which you are probably familiar with. However, queries is not the only thing we are allowed to use when it comes to choosing which documents should be matched and which ones should be returned. Most of the queries exposed by ElasticSearch query DSL have their counterpart and can be used by wrapping them into the following query types:

- `constant_score`
- `filtered`
- `custom_filters_score`

So the question can arise "why bother using filtering, when we can just use queries?". We will try to answer that question right now.

Filters and caching

First of all, filters are very good candidates for caching and as you may have expected, Elasticsearch provides a special cache, the filter cache for storing results of filters. What's more, cached filters don't require too much memory (it only carries the information about which documents match the filter) and can be easily reused by consecutive queries run against the same filter to greatly increase query performance. Imagine that you are running a simple query like this one:

```
{
  "query" : {
    "bool" : {
      "must" : [
        {
          "term" : { "name" : "joe" }
        },
        {
          "term" : { "year" : 1981 }
        }
      ]
    }
  }
}
```

It returns the documents that have the `joe` value in the `name` field and the `1981` value in the `year` field. It's a simple query, but for example, it can be used to get soccer players with the given name, who were born in the specified year.

In the current form, the query will be cached with both those conditions bound together; so if we search for the same name, but with different birth year, Elasticsearch won't be able to use any of the information it got during the query. So now, let's think about how to optimize the query. There can be many names, so it is not a perfect candidate for caching, but the year is (we can't have too many distinct values for the `year` field, right?). So we introduce a different query, one that will combine a simple query with a filter:

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : { "year" : 1981 }
      }
    }
  }
}
```

We've used the `filtered` query to include both query and filter elements. After the first run of the previous query, our filter will be cached by Elasticsearch and reused whenever the same filter will be used in a different query. That way, Elasticsearch doesn't have to load the information about it multiple times.

Not all filters are cached by default

Caching is great, but in fact, not all filters are cached by Elasticsearch by default. This is because some of the filters in Elasticsearch work using the field data cache, a special type of cache, which is also used during sorting using field values and when calculating faceting results. The following filters are not cached using the filter cache by default:

- `numeric_range`
- `script`
- `geo_bbox`
- `geo_distance`
- `geo_distance_range`
- `geo_polygon`
- `geo_shape`
- `and`
- `or`
- `not`

Although the last three of the mentioned filter types do not use the field cache, they manipulate other filters and thus, are not cached. This is because the filters they manipulate are already cached if needed.

Changing Elasticsearch caching behavior

If we wish, Elasticsearch allows us to turn on and off the caching mechanism for filters with the use of `_cache` and `_cache_key` properties. Let's get back to our example and specify that we want to store our term filter cache result under the key, named `year_1981_cache`:

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : {
          "year" : 1981,
          "_cache_key" : "year_1981_cache"
        }
      }
    }
  }
}
```

And now, let's disable caching of the term filter for the same query:

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "term" : { "name" : "joe" }
      },
      "filter" : {
        "term" : {
          "year" : 1981,
          "_cache" : false
        }
      }
    }
  }
}
```

Why bother naming the key for the cache?

The question that we may now ask is if we should bother using the `_cache_key` property at all, can't Elasticsearch do it by itself? Of course it can and it will handle caching when needed, but sometimes we may want to have a bit more control over what is happening. For example, we know that we run our query very rarely and we want to periodically clear the cache of our previous query. If we didn't specify the `_cache_key`, we would be forced to clear the whole filter cache; but we did specify it, so we can just run the following command:

```
curl -XPOST 'localhost:9200/users /_cache/clear?filter_keys=year_1981_cache'
```

When to change the Elasticsearch filter caching behavior

Of course, there are times when you should know better about what you want to achieve than Elasticsearch can predict. For example, you may want to limit your queries to a few locations using the `geo_distance` filter and you use this filter alongside many queries with the same parameters of your `script` filter, which will be used many times along with the same script. In this scenario, it may be worthwhile turning on caching for those filters. Every time you should ask yourself a question "will I use that filter many times or not?" Putting data into the cache will consume resources and you should aim to avoid consuming resources when it is not necessary to do so.

The terms lookup filter

Caching and standard queries is not everything. With the release of Elasticsearch 0.90, we've got a neat filter which can help us when we need to pass multiple terms to a query that are fetched from Elasticsearch itself (something such as the SQL `IN` operator).

Let's take a simple example. Let's say we have an online book shop and we store the information about which books were bought by our users, the clients of our shop. The `books` index looks very simple (we've stored it in the `books.json` file):

```
{
  "mappings" : {
    "book" : {
      "properties" : {
        "id" : { "type" : "string", "store" : "yes", "index" :
          "not_analyzed" },

```

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

In the previous code, there's nothing unusual; there's just the identifier of the book and its title.

Now, let's look at the `clients.json` file, which stores the mappings describing the index structure for the `clients` index:

```
{
  "mappings" : {
    "client" : {
      "properties" : {
        "id" : { "type" : "string", "store" : "yes", "index" :
                "not_analyzed" },
        "name" : { "type" : "string", "store" : "yes", "index" :
                  "analyzed" },
        "books" : { "type" : "string", "store" : "yes", "index" :
                   "not_analyzed" }
      }
    }
  }
}
```

We have the client's identifier, name, and the array of book identifiers he/she bought. In addition to that, let's index some example data:

```
curl -XPUT 'localhost:9200/clients/client/1' -d '{
  "id":"1", "name":"Joe Doe", "books":["1","3"]
}'
curl -XPUT 'localhost:9200/clients/client/2' -d '{
  "id":"2", "name":"Jane Doe", "books":["3"]
}'
curl -XPUT 'localhost:9200/books/book/1' -d '{
  "id":"1", "title":"Test book one"
}'
curl -XPUT 'localhost:9200/books/book/2' -d '{
  "id":"2", "title":"Test book two"
}'
curl -XPUT 'localhost:9200/books/book/3' -d '{
  "id":"3", "title":"Test book three"
}'
```

Now imagine that we want to show all the books bought by a given user, for example, for user with the identifier 1. Of course, we could run a `get` request like this `curl -XGET 'localhost:9200/clients/client/1'` to return the document representing our client and just take the value of the `books` field and run another query like this:

```
curl -XGET 'localhost:9200/books/_search' -d '{
  "query" : {
    "ids" : {
      "type" : "book",
      "values" : [ "1", "3" ]
    }
  }
}'
```

However, ElasticSearch 0.90 introduced the term, `lookup filter`, which allows us to run the two previous queries in a single filtered query, which could look like this:

```
curl -XGET 'localhost:9200/books/_search' -d '{
  "query" : {
    "filtered" : {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "terms" : {
          "id" : {
            "index" : "clients",
            "type" : "client",
            "id" : "1",
            "path" : "books"
          },
          "_cache_key" : "terms_lookup_client_1_books"
        }
      }
    }
  }
}'
```

Please note the parameter `_cache_key` value. As you can see, it is set to `terms_lookup_client_1_books`, which contains the identifier of the client. Please be aware of this, because if you'll use the same `_cache_key` value for different queries, you'll probably get results that are wrong and unexpected. This is because ElasticSearch will cache results for one query under the specified key and then reuse them for the different query.

Now, let's look at the response to the previous query:

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
        one"}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "3",
      "_score" : 1.0, "_source" : {"id":"3", "title":"Test book
        three"}
    } ]
  }
}
```

This is exactly what we were looking for. Awesome!

How does it work?

Let's look at the query we've sent to ElasticSearch. As you can see, it is a simple filtered query, with the query matching all the documents and the `terms` filter. But this time, the `terms` filter is used in a slightly different manner – instead of specifying the terms we are interested in explicitly, we will let ElasticSearch load them for us from another index.

As you can see, we are interested in running the filter against the `id` field, because of the name of the object grouping all other properties. The next things are the new properties: `index`, `type`, `id`, and `path`. The `index` property specifies which index we want the terms to be loaded from (in our case it is the `clients` index). The `type` property tells ElasticSearch which document type we are interested in (in our case it is `client`). The `id` property specifies the document identifier from the index and `type` we've just specified by using the `index` and `type` properties. Finally, the `path` property tells ElasticSearch from which field the terms should be loaded, which in our case is the `books` field in the `clients` index.

So to sum up, what Elasticsearch will do is loading the terms from the `books` field of the `clients` index, from a document with ID `1` and type `client`. The loaded values will be then used in the terms filter to filter the query to only those documents from the `books` index (because we've run our query against that index), that have the given values in the `id` field (because of the terms filter name, which is `id`).



Please note that the `_source` field needs to be stored in order for terms lookup functionality to work.

Performance considerations

The previous query execution is already optimized by Elasticsearch internals, by using caching mechanism. The terms will be loaded in the filter cache under the key provided in the query. In addition to that, once the terms (which in our case are books identifiers) are loaded into the cache, they won't be loaded during consecutive executions, which means that Elasticsearch will be able to execute such query faster.

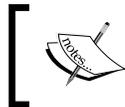
If your data that will be used in the terms lookup is not large, it is recommended for your index (in our case `clients` one) to have only a single shard and for that single shard to have replicas on all the nodes that the `books` index is present at. This is recommended because Elasticsearch will prefer to execute the terms lookup query locally to avoid unnecessary network traffic, network latency, and thus, improve performance.

Loading terms from inner objects

Please change that to: If our `clients` data had the `books` property as an array of inner objects instead of array of values, we would have to specify the `id` property of our query to include information about object nesting. So we would change the `"id": "books"` to `"id": "books.book"`.

Terms lookup filter cache settings

As we've mentioned, in order to provide the terms lookup functionalities, ElasticSearch introduced a new type of cache, which uses a fast **LRU (Least Recently Used)** cache to handle terms caching.



If you want to learn more about what LRU cache is and how it works, please look at the following URL: http://en.wikipedia.org/wiki/Page_replacement_algorithm#Least_recently_used

In order to configure this cache, one can set the following properties in the `elasticsearch.yml` file:

- `indices.cache.filter.terms.size`: It defaults to 10mb and specifies the maximum amount of memory ElasticSearch can use for the terms lookup cache. The default value should be enough for most cases, but if you know you'll load vast amount of data into it, you may want to increase it.
- `indices.cache.filter.terms.expire_after_access`: It 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`: It specifies the maximum time after which an entry should be expired after it is put into cache. By default, it is disabled.

Filter and scopes in ElasticSearch faceting mechanism

When using ElasticSearch faceting mechanism there are a few things one needs to remember. First of all remember that the faceting results will only be calculated for your query results. If you include filters outside of the `query` object, inside the `filter` object, such filters will not be used to limit the documents on which faceting is calculated. The other thing to remember about is the scope, which can help you with extending the documents on which faceting calculation is done. So let's get into examples.

Example data

Let's begin with recalling how queries, filters, and facets work together. To do that we will index a few documents to the `books` index by using the following commands:

```
curl -XPUT 'localhost:9200/books/book/1' -d '{
  "id":"1", "title":"Test book 1", "category":"book",
  "price":29.99
}'
curl -XPUT 'localhost:9200/books/book/2' -d '{
  "id":"2", "title":"Test book 2", "category":"book",
  "price":39.99
}'
curl -XPUT 'localhost:9200/books/book/3' -d '{
  "id":"3", "title":"Test comic 1", "category":"comic",
  "price":11.99
}'
curl -XPUT 'localhost:9200/books/book/4' -d '{
  "id":"4", "title":"Test comic 2", "category":"comic",
  "price":15.99
}'
```

Faceting and filtering

Let's try to check how the faceting will work when using queries and filters. To do that we will run a simple query, that would return all the documents from the `books` index. We will also include a filter that will narrow down the results of the query to only the `book` category and we will include a simple range faceting for the `price` field, to see how many documents have a price lower than 30 and how many higher than 30. The whole query would look like this (stored in the `query_with_filter.json` file):

```
{
  "query" : {
    "match_all" : {}
  },
  "filter" : {
    "term" : { "category" : "book" }
  },
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
```

```
        { "from" : 30 }
      ]
    }
  }
}
```

After running the previous query, we would get the following result:

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
1", "category":"book", "price":29.99}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
2", "category":"book", "price":39.99}
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
      "ranges" : [ {
        "to" : 30.0,
        "count" : 3,
        "min" : 11.99,
        "max" : 29.99,
        "total_count" : 3,
        "total" : 57.97,
        "mean" : 19.323333333333334
      }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
      } ]
    }
  }
}
```

Although, the results of the query were limited to only the documents with the `book` value in the `category` field, our faceting was not. In fact, the faceting was run against all the documents from the `books` index (because of the `match_all` query). So now we know for sure that Elasticsearch faceting mechanism doesn't take filter into account when doing calculations. What about filters that are part of the query, such as in the `filtered` query type, for example? Let's check that out.

Filter as a part of the query

Now, let's try the same example as mentioned previously, but using the `filtered` query type. So, again we want to get all the books from the index that are filtered to the `book` category and let's get a simple range faceting for the `price` field to see how many documents have a price lower than 30 and how many have higher than 30. To achieve this, we run the following query (stored in the `filtered_query.json` file):

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "term" : {
          "category" : "book"
        }
      }
    }
  },
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      }
    }
  }
}
```

The results of the previous query would be as follows:

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
      1", "category":"book", "price":29.99}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
      2", "category":"book", "price":39.99}
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
      "ranges" : [ {
        "to" : 30.0,
        "count" : 1,
        "min" : 29.99,
        "max" : 29.99,
        "total_count" : 1,
        "total" : 29.99,
        "mean" : 29.99
      }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
      } ]
    }
  }
}
```

As you can see, our faceting result was limited to the same set of results our query returned and this is exactly what we were expecting, because now the filter is part of the query! In our case, the faceting results consist of two ranges and each range contains a single document.

The Facet filter

Now imagine that we would like to calculate the faceting for only those books that have the term `2` in the `title` field. We could introduce a second filter to our query, but that would narrow down our query results and we don't want that. What we will do is introduce the facet filter.

We use the `facet_filter` filter on the same level as we provide the type of the facets. It allows us to narrow down the documents we calculate faceting, by using a filter, just like the ones used during querying. For example, if we were to include `facet_filter` to filter our range faceting calculation to only the documents that contain the `2` term in the `title` field, we would change our query facets section to the following one (the whole query was included in the file, named `filtered_query_facet_filter.json`):

```
{
  ...
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      },
      "facet_filter" : {
        "term" : {
          "title" : "2"
        }
      }
    }
  }
}
```

As you can see, we've introduced a new, simple `term` filter. The results returned by the modified query look like this:

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"id":"1", "title":"Test book
        1", "category":"book", "price":29.99}
    }, {
      "_index" : "books",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"id":"2", "title":"Test book
        2", "category":"book", "price":39.99}
    } ]
  },
  "facets" : {
    "price" : {
      "_type" : "range",
      "ranges" : [ {
        "to" : 30.0,
        "count" : 0,
        "total_count" : 0,
        "total" : 0.0,
        "mean" : 0.0
      }, {
        "from" : 30.0,
        "count" : 1,
        "min" : 39.99,
        "max" : 39.99,
        "total_count" : 1,
        "total" : 39.99,
        "mean" : 39.99
      } ]
    }
  }
}
```

Looking at the first result, you may be able to see the difference. By using the facet filter in the query, we were able to limit the faceting calculation to only one document, but our query still returns two documents.

Global scope

Now, what if we would like to run a query for all the documents that have the `2` term in their name, but we would like to show the same range faceting for all the documents in our index? Luckily, we are not forced to run a second query, we can use the global faceting scope by adding the `global` property with the `true` value to the faceting type we are interested in getting global results.

For example, let's modify the first query we've used. In this section, we will not include filtering, but instead a term query. In addition to that we'll add the `global` property, so the query looks like this (we've put it in the `query_global_scope.json` file):

```
{
  "query" : {
    "term" : { "category" : "book" }
  },
  "facets" : {
    "price" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 30 },
          { "from" : 30 }
        ]
      },
      "global" : true
    }
  }
}
```

And now, let's look at the results of such query:

```
{
  ...
  "hits" : {
    "total" : 2,
    "max_score" : 0.30685282,
    "hits" : [ {
      "_index" : "books",
      "_type" : "book",
      "_id" : "1",
      "_score" : 0.30685282, "_source" : { "id":"1", "title":"Test
        book 1", "category":"book", "price":29.99}
    }, {
```

```
    "_index" : "books",
    "_type" : "book",
    "_id" : "2",
    "_score" : 0.30685282, "_source" : {"id":"2",
    "title":"Test book 2", "category":"book", "price":39.99}
  } ]
},
"facets" : {
  "price" : {
    "_type" : "range",
    "ranges" : [ {
      "to" : 30.0,
      "count" : 3,
      "min" : 11.99,
      "max" : 29.99,
      "total_count" : 3,
      "total" : 57.97,
      "mean" : 19.323333333333334
    }, {
      "from" : 30.0,
      "count" : 1,
      "min" : 39.99,
      "max" : 39.99,
      "total_count" : 1,
      "total" : 39.99,
      "mean" : 39.99
    } ]
  }
}
```

Even though our query was limited to only two documents, our facets were calculated for all the documents in the index, thanks to the `global` property included in the query.

One of the possible use cases for the `global` property is showing navigation built with the use of faceting. Imagine a situation where you would like to always show a top-level navigation after the user makes his query, for example, use the terms faceting to enumerate all the top-level categories on the e-commerce website. The `global` scope can come in handy in such cases.

Summary

In this chapter we've looked at how Apache Lucene works, what query rewrite is, and how we can affect the score of our documents with query rescore. In addition to that we've looked at the possibilities of sending multiple queries and real-time get requests with a single HTTP request and how to sort our data using multivalued fields and nested documents. We've used the update API and we learned how we can optimize our queries using filters. Finally, we've used filters and scopes to narrow down or expand the list of documents we calculate faceting on.

In the next chapter, we will look at how to choose different scoring formula and adjust the indexing using postings formats. We'll look at multilingual data handling, configuring transaction log, and dive even deeper into how Elasticsearch caches work.

3

Low-level Index Control

In the previous chapter we've looked at how Apache Lucene works when it comes to scoring documents, what query rewrite is, and how to affect the score of the returned documents with the new feature introduced in Elasticsearch 0.90—the query rescore. We also discussed how to send multiple queries and multiple real-time GET requests with a single HTTP request and how to sort out data using multivalued fields and nested documents. In addition to all this, we've used the update API and we've learned how we can optimize our queries using filters. Finally, we've used filters and scopes to narrow down or expand the list of documents we calculate faceting on. By the end of this chapter we will have covered the following topics:

- How to use different scoring formulae and what they can bring
- How to use different posting formats and what they can bring
- How to handle Near Real Time searching, real-time GET, and what searcher reopening means
- Looking deeper into multilingual data handling
- Configuring transaction log to our needs and see how it affects our deployments
- Segments merging, different merge policies, and merge scheduling

Altering Apache Lucene scoring

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. Lucene API was changed to allow easier modification and extension of the scoring formula. However, that was not the only change that was made to Lucene when it comes to documents score calculation. Lucene 4.0 was shipped with additional similarity models, which basically allows us to use different scoring formula for our documents. In this section we will take a deeper look at what Lucene 4.0 brings and how those features were incorporated into Elasticsearch.

Available similarity models

As already mentioned, apart from the original and default similarity models available before Apache Lucene 4.0, the TF/IDF model was available. We've already discussed it in detail in the *Default Apache Lucene scoring explained* section in *Chapter 2, Power User Query DSL*.

The three new similarity models are as follows:

- **Okapi BM25:** It is a similarity model 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 name, `BM25`. The Okapi BM25 similarity model is said to be performing best when dealing with short text documents, where term repetitions are especially hurtful to the overall document score.
- **Divergence from randomness:** It is a similarity model based on the probabilistic model of the same name. In order to use this similarity in ElasticSearch, you need to use the name, `DFR`. It is said that the Divergence from randomness similarity model performs well on text similar to natural language.
- **Information based:** It is the last of the newly introduced similarity models, which is very similar to the model used by Divergence from randomness. In order to use this similarity in ElasticSearch, you need to use the name `IB`. Similar to the DFR similarity model, it is said that the information based model performs well on data similar to natural language text.



All the mentioned similarity models require mathematical knowledge to fully understand the deep explanation of those models and are far beyond the scope of this book. However, if you would like to explore those models and increase your knowledge about them, please go to http://en.wikipedia.org/wiki/Okapi_BM25 for Okapi BM25 similarity and to http://terrier.org/docs/v3.5/dfr_description.html for divergence from randomness similarity.

Setting per-field similarity

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

```
{
  "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" }
      }
    }
  }
}
```

What we would like to do is, 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 (stored in the `posts_similarity.json` file) would appear as shown in the following code:

```
{
  "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.



In case of the Divergence from randomness and Information based similarity model, we need to configure some additional properties to specify the behavior of those similarities. How to do that is covered in the next part of the current section.

Similarity model configuration

As we now know how to set the desired similarity for each field in our index, it's time to see how to configure those if we need it, which is actually pretty easy. What we need to do is, use the index settings section to provide additional similarity section, for example, as shown in the following code (this example is stored in the `posts_custom_similarity.json` file):

```
{
  "settings" : {
    "index" : {
      "similarity" : {
        "mastering_similarity" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
          "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
          "analyzed", "similarity" : "mastering_similarity" },
        "contents" : { "type" : "string", "store" : "no", "index"
          : "analyzed" }
      }
    }
  }
}
```

You can have more than one similarity configuration, but let's focus on the preceding example. We've defined a new similarity model named `mastering_similarity`, which is based on the default similarity, which is the TF/IDF one. We've configured the `discount_overlaps` property to `false` for that similarity and we've used it as the similarity for the `name` field. We'll talk about what properties can be used for different similarities later in this section. Now let's see how to change the default similarity model Elasticsearch will use.

Choosing the default similarity model

In order to change the similarity model used by default, we need to provide a configuration of a similarity model that will be called `default`. For example, if we would like to use our `mastering_similarity` model as the default one, we would have to change the preceding configuration to the following one (the whole example is stored in the `posts_default_similarity.json` file):

```
{
  "settings" : {
    "index" : {
      "similarity" : {
        "default" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  ...
}
```

Because of the fact that the `query_norm` and `coord` factors (explained in the *Default Apache Lucene scoring explained* section in *Chapter 2, Power User Query DSL*) are used by all similarity models globally and are taken from the `default` configuration similarity, Elasticsearch allows us to change that, when needed. In order to do that, we need to define another similarity called as `base`. It is defined exactly in the same manner as we've shown in the preceding code, but instead of setting its name to `default`, we would set it to `base`, as shown in the following code (the whole example is stored in the `posts_base_similarity.json` file):

```
{
  "settings" : {
    "index" : {
      "similarity" : {
        "base" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  ...
}
```

If the `base` similarity is present in the index configuration, Elasticsearch will use it to calculate the `query_norm` and `coord` factors when calculating the score using other similarity models.

Configuring the chosen similarity models

Each of the newly introduced similarity models can be configured to match our needs. Elasticsearch allows us to use the `default` and `BM25` similarities without any configuration, because they are pre-configured for us. In case of `DFR` and `IB` we need to provide the configuration in order to use them. Let's now see what properties each of the similarity models implementation provides.

Configuring TF/IDF similarity

In the case of TF/IDF similarity, we are allowed to set only a single parameter – the `discount_overlaps` property, whose value defaults to `true`. By default, the tokens that have their position increment set to 0 (that are placed at the same position as the one before them) will not be taken into consideration when calculating the score. If we want them to be taken into consideration, we need to configure the similarity with the `discount_overlaps` property set to `false`.

Configuring Okapi BM25 similarity

In the case of Okapi BM25 similarity, we have the following parameters that we can configure:

- The `k1` parameter (controls **saturation**, which is a non-linear term in frequency normalization) as a float value
- The `b` parameter (controls how the document length affects the term frequency values) as a float value
- The `discount_overlaps` property, which is exactly the same as in TF/IDF similarity

Configuring DFR similarity

In the case of DFR similarity, we have the following parameters that we can configure:

- The `basic_model` parameter (which can take the value: `be`, `d`, `g`, `if`, `in`, and `ine`)
- The `after_effect` parameter (with values of `no`, `b`, and `l`)
- The `normalization` parameter (which can be `no`, `h1`, `h2`, `h3`, or `z`)

If we choose normalization other than `no`, we need to set the normalization factor. Depending upon the chosen normalization, 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. The following code snippet is an example of how the similarity configuration could look:

```
"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 that we can configure:

- The `distribution` property (which can take the value `ll` or `spl`)
- The `lambda` property (which can take the value `df` or `tfidf`)

In addition to this, we can choose the normalization factor which is the same as for the DFR similarity, so we'll omit describing it for the second time. The following code snippet shows how the example IB similarity configuration could look:

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

Using codecs

One of the most significant changes introduced by Apache Lucene 4.0 was the ability to alter how index files are written. Back in the days prior to Lucene 4.0, if we wanted to change the way the index was written, we had to patch Lucene. It is no longer the case with the introduction of flexible indexing, when one can alter the way postings (posting format) are written.

Simple use cases

A question may arise, is this really useful? and it is a proper one, why one may need to alter the way Lucene index is written? One of the reasons is performance. Some fields may require special treatment, like the primary keys which are unique and with the help of some techniques they can be searched very fast as compared to standard numeric or text fields that have many unique values. You can also use it for debugging what is actually going to be written in the Lucene index by using `SimpleTextCodec` (on the Apache Lucene level because, `ElasticSearch` doesn't expose this codec).

Let's see how it works

Let's assume that we have the following mappings for our `posts` index (stored in the `posts.json` file):

```
{
  "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" }
      }
    }
  }
}
```

The codecs are defined per field. In order to configure our field to use a codec, we need to add a property called `postings_format` along with the value of the desired codec, for example, `pulsing`. So, after introduction of the mentioned codec our mappings file would appear as shown in the following code snippet (stored in the `posts_codec.json` file):

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

If we would now run the following command:

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

to check if the codec was taken into consideration by ElasticSearch, we would get the following response:

```
{
  "posts" : {
    "post" : {
      "properties" : {
        "contents" : {
          "type" : "string"
        },
        "id" : {
          "type" : "long",
          "store" : true,
          "postings_format" : "pulsing",
          "precision_step" : 2147483647
        },
        "name" : {
          "type" : "string",
          "store" : true
        }
      }
    }
  }
}
```

As we can see, the `id` field is configured to use the pulsing `postings_format` property and that's what we wanted.



Please remember that codecs were introduced in Apache Lucene 4.0 and because of this, the described functionality can't be used in ElasticSearch older than 0.90.

Available posting formats

The following posting formats were introduced and can be used:

- `default`: It is used when no explicit format is defined. 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`: It is a codec that encodes the post listing into the terms array for high cardinality fields, which results in one less seek, Lucene needs to do when retrieving a document. Using this codec for high cardinality fields can speed up queries on such fields.
- `direct`: It is a codec that during reads loads terms into arrays, which are held in the memory uncompressed. This codec may give you performance boost on commonly used fields, but should be used with caution, as it is very memory intensive, because the terms and postings arrays need to be stored in the memory.



Since all the terms are held in the byte array you can have upto 2.1 GB of memory used for this per segment.

- `memory`: As its name suggests, this codec writes all the data to disk, but reads the terms and post listings into the memory, using a structure called FST (Finite State Transducers). More information about this structure can be found in a great post by Mike McCandless at <http://blog.mikemccandless.com/2010/12/using-finite-state-transducers-in.html>). Because of storing the data in memory, this codec may result in performance boost for commonly used terms.
- `bloom_default`: It is an extension of the `default` codec that adds the functionality of a **bloom filter** that is written to the disk. When reading, the bloom filter is read and held into memory to allow very fast checking if a given value exists. This codec is very useful for high cardinality fields such as primary key. More information about bloom filter is available at http://en.wikipedia.org/wiki/Bloom_filter, which uses the bloom filter in addition to what the default codec does.
- `bloom_pulsing`: It is an extension of the `pulsing` codec, which uses the bloom filter in addition to what the `pulsing` codec does.

Configuring the codec behavior

All the postings format comes with the default configuration that is sufficient for most use cases, but there are times when you may want to configure the behavior to match your deployment needs. In such cases, Elasticsearch allows us to index settings API with the defined `codec` section. For example, if we would like to configure the default codec and name it as `custom_default`, we would define the following mappings (stored in the `posts_codec_custom.json` file):

```
{
  "settings" : {
    "index" : {
      "codec" : {
        "postings_format" : {
          "custom_default" : {
            "type" : "default",
            "min_block_size" : "20",
            "max_block_size" : "60"
          }
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
          "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
          "analyzed", "postings_format" : "custom_default" },
        "contents" : { "type" : "string", "store" : "no", "index"
          : "analyzed" }
      }
    }
  }
}
```

As you can see, we've changed the `min_block_size` and the `max_block_size` properties of the default codec and we named the newly configured codec as `custom_default`. After that, we've used it as the postings format for the name field.

Default codec properties

When using the `default` codec we are allowed to configure the following properties:

- `min_block_size`: It specifies the minimum block size Lucene term dictionary uses to encode blocks. It defaults to 25.
- `max_block_size`: It specifies the maximum block size Lucene term dictionary uses to encode blocks. It defaults to 48.

Direct codec properties

The direct codec allows us to configure the following properties:

- `min_skip_count`: It specifies the minimum number of terms with a shared prefix to allow writing of a skip pointer. It defaults to 8.
- `low_freq_cutoff`: The codec will use a single array object to hold postings and positions that have document frequency lower than this value. It defaults to 32.

Memory codec properties

By using the `memory` codec we are allowed to alter the following properties:

- `pack_fst`: It is a Boolean option that defaults to `false` and specifies if the memory structure that holds the postings should be packed into the FST. Packing into FST will reduce the memory needed to hold the data.
- `acceptable_overhead_ratio`: It is a compression ratio of the internal structure specified as a float value which defaults to 0.2. When using the 0 value, there will be no additional memory overhead but the returned implementation may be slow. When using the 0.5 value, there can be a 50 percent memory overhead, but the implementation will be fast. Values higher than 1 are also possible, but may result in high memory overhead.

Pulsing codec properties

When using the `pulsing` codec we are allowed to use the same properties as with the `default` codec and in addition to them one more property, which is described as follows:

- `freq_cut_off`: It defaults to 1. The document frequency at which the postings list will be written into the term dictionary. The documents with the frequency equal to or less than the value of `freq_cut_off` will be processed.

Bloom filter-based codec properties

If we want to configure a bloom filter based codec, we can use the `bloom_filter` type and set the following properties:

- `delegate`: It specifies the name of the codec we want to wrap, with the bloom filter.
- `ffp`: It is a value between 0 and 1.0 which specifies the desired false positive probability. We are allowed to set multiple probabilities depending on the amount of documents per Lucene segment. For example, the default value of `10k=0.01,1m=0.03` specifies that the `fpp` value of 0.01 will be used when the number of documents per segment is larger than 10.000 and the value of 0.03 will be used when the number of documents per segment is larger than one million.

For example, we could configure our custom bloom filter based codec to wrap a direct posting format as shown in the following code (stored in `posts_bloom_custom.json` file):

```
{
  "settings" : {
    "index" : {
      "codec" : {
        "postings_format" : {
          "custom_bloom" : {
            "type" : "bloom_filter",
            "delegate" : "direct",
            "ffp" : "10k=0.03,1m=0.05"
          }
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes",
          "precision_step" : "0" },
        "name" : { "type" : "string", "store" : "yes", "index" :
          "analyzed", "postings_format" : "custom_bloom" },
        "contents" : { "type" : "string", "store" : "no", "index" :
          "analyzed" }
      }
    }
  }
}
```

NRT, flush, refresh, and transaction log

In an ideal search solution, when new data is indexed it is instantly available for searching. At the first glance it is exactly how Elasticsearch works even in multiserver environments. But this is not the truth (or at least not all the truth) and we will show you why it is like this. Let's index an example document to the newly created index by using the following command:

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test" }'
```

Now, we will replace this document and immediately we will try to find it. In order to do this, we'll use the following command chain:

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test2" }' ; curl localhost:9200/test/test/_search?pretty
```

The preceding command will probably result in the response, which is very similar to the following response:

```
{ "ok": true, "_index": "test", "_type": "test", "_id": "1", "_version": 2 } {
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "test",
      "_type" : "test",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title": "test" }
    } ]
  }
}
```

The first line starts with a response to the indexing command—the first command. As you can see everything is correct, so the second, search query should return the document with the `title` field `test2`, however, as you can see it returned the first document. What happened?

But before we give you the answer to the previous question, we should take a step backward and discuss about how underlying Apache Lucene library makes the newly indexed documents available for searching.

Updating index and committing changes

As we already know from the *Introduction to Apache Lucene* section in *Chapter 1, Introduction to Elasticsearch*, during the indexing process new documents are written into segments. The segments are independent indices, which means that queries that are run in parallel to indexing, from time to time should add newly created segments to the set of those segments that are used for searching. Apache Lucene does that by creating subsequent (because of write-once nature of the index) `segments_N` files, which list segments in the index. This process is called committing. Lucene can do this in a secure way – we are sure that all changes or none of them hits the index. If a failure happens, we can be sure that the index will be in consistent state.

Let's return to our example. The first operation adds the document to the index, but doesn't run the `commit` command to Lucene. This is exactly how it works.

However, a commit is not enough for the data to be available for searching.

Lucene library use an abstraction class called `Searcher` to access index.

After a commit operation, the `Searcher` object should be reopened in order to be able to see the newly created segments. This whole process is called refresh. For performance reasons `ElasticSearch` tries to postpone costly refreshes and by default refresh is not performed after indexing a single document (or a batch of them), but the `Searcher` is refreshed every second. This happens quite often, but sometimes applications require the refresh operation to be performed more often than once every second. When this happens you can consider using another technology or requirements should be verified. If required, there is possibility to force refresh by using `ElasticSearch` API. For example, in our example we can add the following command:

```
curl -XGET localhost:9200/test/_refresh
```

If we add the preceding command before the search, `ElasticSearch` would respond as we had expected.

Changing the default refresh time

The time between automatic `Searcher` refresh can be changed by using the `index.refresh_interval` parameter either in the `ElasticSearch` configuration file or by using the update settings API. For example:

```
curl -XPUT localhost:9200/test/_settings -d '{
  "index" : {
    "refresh_interval" : "5m"
  }
}'
```

The preceding command will change the automatic refresh to be done every 5 minutes. Please remember that the data that are indexed between refreshes won't be visible by queries.



As we said, the refresh operation is costly when it comes to resources. The longer the period of refresh is, the faster your indexing will be. If you are planning for very high indexing procedure when you don't need your data to be visible until the indexing ends, you can consider disabling the refresh operation by setting the `index.refresh_interval` parameter to `-1` and setting it back to its original value after the indexing is done.

The transaction log

Apache Lucene can guarantee index consistency and *all or nothing* indexing, which is great. But this fact cannot ensure us that there will be no data loss when failure happens while writing data to the index (for example, when there isn't enough space on the device, the device is faulty or there aren't enough file handlers available to create new index files). Another problem is that frequent commit is costly in terms of performance (as you recall, a single commit will trigger a new segment creation and this can trigger the segments to merge). Elasticsearch solves those issues by implementing transaction log. Transaction log holds all uncommitted transactions and from time to time, Elasticsearch creates a new log for subsequent changes. When something goes wrong, transaction log can be replayed to make sure that none of the changes were lost. All of these tasks are happening automatically, so, the user may not be aware of the fact that commit was triggered at a particular moment. In Elasticsearch, the moment when the information from transaction log is synchronized with the storage (which is Apache Lucene index) and transaction log is cleared is called **flushing**.



Please note the difference between flush and refresh operations. In most of the cases refresh is exactly what you want. It is all about making new data available for searching. From the opposite side, the flush operation is used to make sure that all the data is correctly stored in the index and transaction log can be cleared.

In addition to automatic flushing, it can be forced manually using the flush API. For example, we can run a command to flush all the data stored in the transaction log for all indices, by running the following command:

```
curl -XGET localhost:9200/_flush
```

Or we can run the `flush` command for the particular index, which in our case is the one called `library`:

```
curl -XGET localhost:9200/library/_flush
```

```
curl -XGET localhost:9200/library/_refresh
```

In the second example we used it together with the `refresh`, which after flushing the data opens a new searcher.

The transaction log configuration

If the default behavior of the transaction log is not enough Elasticsearch allows us to configure its behavior when it comes to the transaction log handling. The following parameters can be set in the `elasticsearch.yml` file as well as using index settings update API to control transaction log behavior:

- `index.translog.flush_threshold_period`: It defaults to 30 minutes (30m). It controls the time, after which flush will be forced automatically even if no new data was being written to it. In some cases this can cause a lot of I/O operation, so sometimes it's better to do flush more often with less data being stored in it.
- `index.translog.flush_threshold_ops`: It specifies the maximum number of operations after which the flush operation will be performed. It defaults to 5000.
- `index.translog.flush_threshold_size`: It specifies the maximum size of the transaction log. If the size of the transaction log is equal to or greater than the parameter, the flush operation will be performed. It defaults to 200 MB.
- `index.translog.disable_flush`: This option disables automatic flush. By default flushing is enabled, but sometimes it is handy to disable it temporarily, for example, during import of large amount of documents.



All of the mentioned parameters are specified for an index of our choice, but they are defining the behavior of the transaction log for each of the index shards.

Of course, in addition to setting the preceding parameters in the `elasticsearch.yml` file, they can also be set by using Settings Update API. For example:

```
curl -XPUT localhost:9200/test/_settings -d '{
  "index" : {
    "translog.disable_flush" : true
  }
}'
```

The preceding command was run before the import of a large amount of data, which gave us a performance boost for indexing. However, one should remember to turn on flushing when the import is done.

Near Real Time GET

Transaction log gives us one more feature for free that is, real-time GET operation, which provides the possibility of returning the previous version of the document including non-committed versions. The real-time GET operation fetches data from the index, but first it checks if a newer version of that document is available in the transaction log. If there is no flushed document, data from the index is ignored and a newer version of the document is returned – the one from the transaction log. In order to see how it works, you can replace the search operation in our example with the following command:

```
curl -XGET localhost:9200/test/test/1?pretty
```

ElasticSearch should return the result similar to the following:

```
{
  "_index" : "test",
  "_type" : "test",
  "_id" : "1",
  "_version" : 2,
  "exists" : true, "_source" : { "title": "test2" }
}
```

If you look at the result, you would see that again, the result was just as we expected and no trick with refresh was required to obtain the newest version of the document.

Looking deeper into data handling

When starting to work with Elasticsearch, you can be overwhelmed with the different ways of searching and the different query types it provides. Each of these query types behaves differently and we do not say only about obvious differences, for example, like the one you would see when comparing range search and prefix search. It is crucial to know about these differences to understand how the queries work, especially when doing a little more than just using the default Elasticsearch instance, for example, for handling multilingual information.

Input is not always analyzed

Before we start discussing queries analysis, let's create the index by using the following command:

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "properties" : {
        "title" : { "type" : "string", "analyzer" : "snowball" }
      }
    }
  }
}'
```

As you can see, the index is pretty simple. The document contains only one field, processed by snowball analyzer. Now, let's index a simple document. We do it by running the following command:

```
curl -XPUT localhost:9200/test/test/1 -d '{
  "title" : "the quick brown fox jumps over the lazy dog"
}'
```

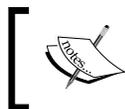
We have our big index, so we may bomb it with queries. Look closely at the following two commands:

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "term" : {
      "title" : "jumps"
    }
  }
}'
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "match" : {
      "title" : "jumps"
    }
  }
}'
```

The first query will not return our document, but the second query will, surprise! You probably already know (or suspect) what the reason for such behavior is and that it is connected to analyzing. Let's compare what we, in fact, have in the index and what we are searching for. To do that, we will use the Analyze API by running the following command:

```
curl 'localhost:9200/test/_analyze?text=the+quick+brown+fox+jumps+over+the+lazy+dog&pretty&analyzer=snowball'
```

The `_analyze` endpoint allows us to see what Elasticsearch does with the input that is given in the `text` parameter. It also gives us the possibility to define which analyzer should be used (the `analyzer` parameter).



Other features of the analyze API are available at <http://www.elasticsearch.org/guide/reference/api/admin-indices-analyze/>.

The response returned by Elasticsearch for the preceding request will look similar to the following:

```
{
  "tokens" : [ {
    "token" : "quick",
    "start_offset" : 4,
    "end_offset" : 9,
    "type" : "<ALPHANUM>",
    "position" : 2
  }, {
    "token" : "brown",
    "start_offset" : 10,
    "end_offset" : 15,
    "type" : "<ALPHANUM>",
    "position" : 3
  }, {
    "token" : "fox",
    "start_offset" : 16,
    "end_offset" : 19,
    "type" : "<ALPHANUM>",
    "position" : 4
  }, {
    "token" : "jump",
    "start_offset" : 20,
    "end_offset" : 25,
    "type" : "<ALPHANUM>",
    "position" : 5
  }, {
    "token" : "over",
    "start_offset" : 26,
    "end_offset" : 30,
    "type" : "<ALPHANUM>",
    "position" : 6
  }, {
    "token" : "lazi",
    "start_offset" : 35,
    "end_offset" : 39,
    "type" : "<ALPHANUM>",
    "position" : 8
  }, {
    "token" : "dog",
    "start_offset" : 40,
    "end_offset" : 43,
    "type" : "<ALPHANUM>",
    "position" : 9
  } ]
}
```

You can see how Elasticsearch changed the input into a stream of tokens. As you recall from the *Introduction to Apache Lucene* section in *Chapter 1, Introduction to Elasticsearch* every token has information about its position in the original text, about its type (this is not interesting from our perspective, but may be used by filters), and a term, word, which is stored in the index and used for comparison when searching. Our original text, *the quick brown fox jumps over the lazy dog* was converted into the following words (terms): *quick, brown, fox, jump, over, lazy* (this is interesting), and *dog*. So, we'll summarize what the snowball analyzer did:

- skipped non-significant words (*the*)
- converted words into their base forms (*jump*)
- sometimes messed up the conversion (*lazi*)

The third thing is not as bad as it looks, as long as all forms of the same word are converted into the same form. If such a thing happens, the goal of stemming will be achieved – Elasticsearch will match words from the query with the words stored in the index, independently of its form. But now let's return to our queries. The term query just searches for a given term (*jumps* in our case) but there is no such term in the index (there is *jump*). In the case of the match query the given text is first passed on to the analyzer, which converts *jumps* into *jump*, and after that the converted form is being used in the query.

Now let's look at the second example:

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "prefix" : {
      "title" : "lazy"
    }
  }
}'
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "match_phrase_prefix" : {
      "title" : "lazy"
    }
  }
}'
```

In the preceding case both queries are similar, but again, the first query returns nothing (because `lazy` is not equal to `lazi` in the index) and the second query, which is analyzed, will return our document.

Example usage

All of this is interesting and you should remember the fact that some of the queries are being analyzed and some are not. However, the most interesting part is, how we can do all of this consciously to improve search-based applications.

Let's imagine searching the content of the books. It is possible that sometimes our users search by the name of the character, place name, probably by the quote fragment. We don't have any natural language analysis functionality in our application so we don't know the meaning of the phrase entered by the user. However, with some degree of probability we can assume that the most interesting result will be the one that exactly matches the phrase entered by the user. It is also very probable that the second scale of importance, will be the documents that have exactly the same words in the same form as the user input, and those documents – the ones with words with the same meaning or with a different language form.

In order to use another example let's use a command, which creates a simple index with only a single field defined:

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "properties" : {
        "lang" : { "type" : "string" },
        "title" : {
          "type" : "multi_field",
          "fields" : {
            "i18n" : { "type" : "string", "index" : "analyzed",
              analyzer : "english" },
            "org" : { "type" : "string", "index" : "analyzed",
              "analyzer" : "standard"}
          }
        }
      }
    }
  }
}'
```

We have the single field, but it is analyzed in two different ways because of the `multi_field`: with the `standard` analyzer (field `title.org`), and with the `english` analyzer (field `title.i18n`) which will try to change the input to its base form. If we index an example document with the following command:

```
curl -XPUT localhost:9200/test/test/1 -d '{ "title" : "The quick brown fox jumps over the lazy dog." }'
```

We will have the `jumps` term indexed in the `title.org` field and the `jump` term indexed in the `title.i18n` field. Now let's run the following query:

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "multi_match" : {
      "query" : "jumps",
      "fields" : ["title.org^1000", "title.i18n"]
    }
  }
}'
```

Our document will be given a higher score for perfect match, thanks to boosting and matching the `jumps` term in the `field.org` field. The score is also given for hit in `field.i18n`, but the impact of this field on the overall score is much smaller, because we didn't specify the boost and thus the default value of 1 is used.

Changing the analyzer during indexing

The next thing worth mentioning when it comes to handling multilingual data is the possibility of dynamically changing the analyzer during indexing. Let's modify the previous mapping by adding the `_analyzer` part to it:

```
curl -XPUT localhost:9200/test -d '{
  "mappings" : {
    "test" : {
      "_analyzer" : {
        "path" : "lang"
      },
      "properties" : {
        "lang" : { "type" : "string" },

```

```
    "title" : {
      "type" : "multi_field",
      "fields" : {
        "i18n" : { "type" : "string", "index" : "analyzed"},
        "org" : { "type" : "string", "index" : "analyzed",
                  "analyzer" : "standard"}
      }
    }
  }
}
}'
```

The change we just did, allows Elasticsearch to determine the analyzer basing on the contents of the document that is being processed. The `path` parameter is the name of the document field, which contains the name of the analyzer. The second change is removal of the analyzer definition from the `field.i18n` field definition. Now our indexing command will look like this:

```
curl -XPUT localhost:9200/test/test/1 -d '{ "title" : "The quick brown fox jumps over the lazy dog.", "lang" : "english" }'
```

In the preceding example, Elasticsearch will take the value of the `lang` field and will use it as the analyzer for that document. It can be useful when you want to have different analysis for different documents (for example, some documents should have stop words removed and some shouldn't).

Changing the analyzer during searching

Changing the analyzer is also possible in the query time, by specifying the `analyzer` property. For example, let's look at the following query:

```
curl localhost:9200/test/_search?pretty -d '{
  "query" : {
    "multi_match" : {
      "query" : "jumps",
      "fields" : ["title.org^1000", "title.i18n"],
      "analyzer": "english"
    }
  }
}'
```

Thanks to the highlighted code fragment. Elasticsearch will choose the analyzer that we've explicitly mentioned.

The pitfall and default analysis

Combining the mechanism of replacing analyzer per document on index-time and query-time level is a very powerful feature, but it can also introduce hard-to-spot errors. One of them can be a situation where the analyzer is not defined. In such cases, Elasticsearch will choose the so-called default analyzer, but sometimes this is not what you can expect, because default analyzer, for example, can be redefined by plugins. In such cases, it is worth defining what the default Elasticsearch analysis should look like. To do this, we just define analyzer as usual, but instead of giving it a custom name we use the `default` name.



As an alternative you can define the `default_index` analyzer and the `default_search` analyzer, which will be used as a default analyzer respectively on index-time and search-time analysis.

Segment merging under control

As you already know (we've discussed it throughout *Chapter 1, Introduction to Elasticsearch*) every Elasticsearch index is built out of one or more shards and can have zero or more replicas. You also know that each of the shards and replicas are actual Apache Lucene indices, which are built of multiple segments (at least one segment). If you recall, the segments are written once and read many times, apart from the information about the deleted documents which are held in one of the files and can be changed. After some time, when certain conditions are met, the contents of some segments can be copied to a bigger segment and the original segments are discarded and thus deleted from the disk. Such an operation is called **segment merging**.

You may ask yourself, why bother about segment merging? There are a few reasons. First of all, the more segments the index is built of, the slower the search will be and the more memory Lucene needs to use. In addition to this, the segments are immutable, so the information is not deleted from it. If you happen to delete many documents from your index, until the merge happens, those documents are only marked as deleted, not deleted physically. So, when segment merging happens the documents, which are marked as deleted, are not written into the new segment and in this way, they are removed, which decreases the final segment size.



Many small changes can result in a large number of small segments, which can lead to problems with large number of opened files. We should always be prepared to handle such situations, for example, by having the appropriate opened files limit set.

So, just to quickly summarize, segments merging takes place and from the user's point of view will result in two effects:

- It will reduce the number of segments to allow faster searching when a few segments are merged into a single one
- It will reduce the size of the index because of removing the deleted documents when the merge is finalized

However, you have to remember that segment merging comes with a price; the price of I/O (input/output) operations, which on slower systems can affect performance. Because of this, Elasticsearch allows us to choose the merge policy and the store level throttling. We will discuss the merge policy in the following section and we will get back to throttling in the *When it is all too much for the I/O* section in *Chapter 6, Fighting with Fire*.

Choosing the right merge policy

Although segments merging is Apache Lucene's duty, Elasticsearch allows us to configure which merge policy we would like to use. There are three policies that we are currently allowed to use:

- `tiered` (the default one)
- `log_byte_size`
- `log_doc`

Each of the preceding mentioned policies have their own parameters, which define their behavior and which default values we can override (please look at the section dedicated to the policy of your choice to see what are those parameters).

In order to tell Elasticsearch, which merge policy we want to use, we should set `index.merge.policy.type` to the desired type, shown as follows:

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

 Once the index is created with the specified merge policy type it can't be changed. However, all the properties defining merge policy behavior can be changed using the index update API.

Let's now look at the different merge policies and what functionality they provide. After this, we will discuss all the configuration options provided by the policies.

The tiered merge policy

This is the default merge policy that Elasticsearch uses. It merges the segments of approximately similar size, taking into account the maximum number of segments allowed per tier. It is also possible to differentiate the number of segments that are allowed to be merged at once from how many segments are allowed to be present per tier. During indexing, this merge policy will compute how many segments are allowed to be present in the index, which is called **budget**. If the number of segments the index is built of is higher than the computed budget, the tiered policy will first sort the segments by decreasing order of their size (taking into account the deleted documents). After that it will find the merge that has the lowest cost. The merge cost is calculated in a way that merges reclaiming more deletes and having a smaller size is favored.

If the merge produces a segment that is larger than the value specified by the `index.merge.policy.max_merged_segment` property, the policy will merge fewer segments to keep the segment size under the budget. This means, that for indices that have large shards, the default value of the `index.merge.policy.max_merged_segment` property may be too low and will result in the creation of many segments, slowing down your queries. Depending on the volume of your data you should monitor your segments and adjust the merge policy setting to match your needs.

The log byte size merge policy

This is a merge policy, which over time will produce an index that will be built of a logarithmic size of indices. There will be a few large segments, then there will be a few merge factor smaller segments and so on. You can imagine that there will be a few segments of the same level of size, when the number of segments will be lower than the merge factor. When an extra segment is encountered and all the segments within that level are merged. The number of segments an index will contain is proportional to the logarithm of the next size in bytes. This merge policy is generally able to keep the low number of segments in your index while minimizing the cost of segments merging.

The log doc merge policy

It 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. This merge policy will perform well when the documents are similar in terms of size or if you want segments of similar size in terms of the number of documents.

Merge policies configuration

We now know how merge policies work, but we lack the knowledge about the configuration options. So now, let's discuss each of the merge policies and see what options are exposed to us. Please remember that the default values will usually be OK for most of the deployments and they should be changed only when needed.

The tiered merge policy

When using the `tiered` merge policy the following options can be altered:

- `index.merge.policy.expunge_deletes_allowed`: It defaults to 10 and it specifies the percentage of deleted documents in a segment in order for it to be considered to be merged when running `expungeDeletes`.
- `index.merge.policy.floor_segment`: It is a property that enables us to prevent frequent flushing of very small segments. Segments smaller than the size defined by this property are treated by the merge mechanism, as they would have the size equal to the value of this property. It defaults to 2 MB.
- `index.merge.policy.max_merge_at_once`: It specifies the maximum number of segments that will be merged at the same time during indexing. By default it is set to 10. Setting the value of this property to higher values can result in multiple segments being merged at once, which will need more I/O resources.
- `index.merge.policy.max_merge_at_once_explicit`: It specifies the maximum number of segments that will be merged at the same time during optimize operation or `expungeDeletes`. By default it is set to 30. This setting will not affect the maximum number of segments that will be merged during indexing.
- `index.merge.policy.max_merged_segment`: It defaults to 5 GB and it specifies the maximum size of a single segment that will be produced during segment merging when indexing. This setting is an approximate value, because the merged segment size is calculated by summing the size of segments that are going to be merged minus the size of the deleted documents in those segments.

- `index.merge.policy.segments_per_tier`: It specifies the allowed number of segments per tier. Smaller values of this property result in less segments, which means, more merging and lower indexing performance. It defaults to 10 and should be set to a value higher than or equal to the `index.merge.policy.max_merge_at_once` or you'll be facing too many merges and performance issues.
- `index.reclaim_deletes_weight`: It defaults to 2.0 and specifies how many merges that reclaim deletes are favored. When setting this value to 0.0 the deletes reclaim will not affect merge selection. The higher the value, the more favored will be the merge that will reclaim deletes.
- `index.compound_format`: It is a Boolean value that specifies whether the index should be stored in a compound format or not. It defaults to `false`. If set to `true`, Lucene will store all the files that build the index in a single file. This is sometimes useful for systems running constantly out of file handlers, but will decrease the searching and indexing performance.
- `index.merge.async`: It is a Boolean value specifying if the merge should be done asynchronously. It defaults to `true`.
- `index.merge.async_interval`: When the `index.merge.async` value is set to `true` (so the merging is done asynchronously), this property specifies the interval between merges. The default value of this property is 1s. Please note that the value of this property needs to be kept low, for merging to actually happen and the index segments reduction will take place.

The log byte size merge policy

When using the `log_byte_size` merge policy the following options can be configured:

- `merge_factor`: It specifies how often segments are merged during indexing. With a smaller `merge_factor` value, the searches are faster, less memory is used, but that comes with the cost of slower indexing. With larger `merge_factor` values, it is the opposite – the indexing is faster (because of less merging being done), but the searches are slower and more memory is used. By default, the `merge_factor` is given the value of 10. It is advised to use larger values of `merge_factor` for batch indexing and lower values of this parameter for normal index maintenance.
- `min_merge_size`: It defines the size (total size of the segment files in bytes) of the smallest segment possible. If a segment is lower in size than the number specified by this property, it will be merged if the `merge_factor` property allows us to do that. This property defaults to 1.6 MB and is very useful to avoid having many very small segments. However, one should remember that setting this property to a large value will increase the merging cost.

- `max_merge_size`: It defines the maximum size (total size of the segment files in bytes) of the segment that can be merged with other segments. By default it is not set, so there is no limit on the maximum size a segment can be in order to be merged.
- `maxMergeDocs`: It defines the maximum number of documents a segment can have in order to be merged with other segments. By default it is not set, so there is no limit on the maximum number of documents a segment can have.
- `calibrate_size_by_deletes`: It is a Boolean value, which is set to `true` and specifies if the size of deleted documents should be taken into consideration when calculating segment size.
- `index.compound_format`: It is a Boolean value that specifies if the index should be stored in a compound format. It defaults to `false`. Please refer to `tiered merge policy` for the explanation of what this parameter does.

The mentioned properties we just saw, should be prefixed with the `index.merge.policy` prefix. So if we would like to set the `min_merge_docs` property, we should use the `index.merge.policy.min_merge_docs` property.

In addition to this, the `log_byte_size` merge policy accepts the `index.merge.async` property and the `index.merge.async_interval` property just like `tiered merge policy` does.

The log doc merge policy

When using the `log_doc` merge policy the following options can be configured:

- `merge_factor`: It is same as the property that is present in the `log_byte_size` merge policy, so please refer to that policy for explanation.
- `min_merge_docs`: It defines the minimum number of documents for the smallest segment. If a segment contains a lower document count than the number specified by this property it will be merged if the `merge_factor` property allows this. This property defaults to `1000` and is very useful to avoid having many very small segments. However, one should remember that setting this property to a large value will increase the merging cost.
- `max_merge_docs`: It defines the maximum number of documents a segment can have in order to be merged with other segments. By default it is not set, so there is no limit on the maximum number of documents a segment can have.

- `calibrate_size_by_deletes`: It is a Boolean value which defaults to `true` and specifies if the size of deleted documents should be taken into consideration when calculating the segment size.
- `index.compound_format`: It is a Boolean value that specifies if the index should be stored in a compound format. It defaults to `false`. Please refer to `tiered` merge policy for the explanation of what this parameter does.

Similar to the previous merge policy, the previously mentioned properties should be prefixed with the `index.merge.policy` prefix. So if we would like to set the `min_merge_docs` property, we should use the `index.merge.policy.min_merge_docs` property.

In addition to this, the `log_doc` merge policy accepts the `index.merge.async` property and the `index.merge.async_interval` property, just like `tiered` merge policy does.

Scheduling

In addition to having a control over how merge policy is behaving, Elasticsearch allows us to define the execution of merge policy once a merge is needed. There are two merge schedulers available with the default being the `ConcurrentMergeScheduler`.

The concurrent merge scheduler

This is a merge scheduler that will use multiple threads in order to perform segments merging. This scheduler will create a new thread until the maximum number of threads is reached. If the maximum number of threads is reached and a new thread is needed (because segments merge needs to be performed), all the indexing will be paused until at least one merge is completed.

In order to control the maximum threads allowed, we can alter the `index.merge.scheduler.max_thread_count` property. By default, it is set to the value calculated by the following equation:

```
maximum_value(1, minimum_value(3, available_processors / 2))
```

So, if our system has eight processors available, the maximum number of threads that concurrent merge scheduler is allowed to use will be equal to 4.

The serial merge scheduler

A simple merge scheduler that uses the same thread for merging. It results in a merge that stops all the other document processing that was happening in the same thread, which in this case means stopping of indexing.

Setting the desired merge scheduler

In order to set the desired merge scheduler, one should set the `index.merge.scheduler.type` property to the value of `concurrent` or `serial`. For example, in order to use the concurrent merge scheduler, one should set the following property:

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

In order to use the serial merge scheduler, one should set the following property:

```
index.merge.scheduler.type: serial
```



When talking about merge policy and merge schedulers it would be nice to visualize it. If one needs to see how the merges are done in the underlying Apache Lucene library, we suggest visiting Mike McCandless' blog post at <http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>.

In addition to this, there is a plugin allowing us to see what is happening to the segments called SegmentSpy. Please refer to the following URL for more information:

<https://github.com/polyfractal/elasticsearch-segmentspy>

Summary

In this chapter, we've learned how to use different scoring formulae and what they bring to the table. We've also seen how to use different posting formats and how we benefit from using them. In addition to this, we now know how to handle Near Real Time searching and real-time GET requests and what searcher reopening means for Elasticsearch. We've discussed multilingual data handling and we've configured transaction log to our needs. Finally, we've learned about segments merging, merge policies, and scheduling.

In the next chapter, we'll look closely at what Elasticsearch offers us when it comes to shard control. We'll see how to choose the right amount of shards and replicas for our index, we'll manipulate shard placement and we will see when to create more shards than we actually need. We'll discuss how the shard allocator works. Finally, we'll use all the knowledge we've got so far to create fault tolerant and scalable clusters.

4

Index Distribution Architecture

In the previous chapter, we've learned how to use different scoring formulas and how we can benefit from using them. We've also seen how to use different posting formats to change how the data is indexed. In addition to that, we now know how to handle near real-time searching and real-time get and what searcher reopening means for Elasticsearch. We've discussed multilingual data handling and we've configured the transaction log to our needs. Finally, we've learned about segments merging, merge policies, and scheduling. By the end of this chapter, you will have learned:

- How to choose the right amount of shards and replicas for our cluster
- What routing is and what it means for Elasticsearch
- How ShardAllocator works and how we can configure it
- How to adjust the shard allocation mechanism to our needs
- How to choose on which shards the given operation should be executed
- How to combine our knowledge to configure a real-life example cluster
- How to react when the data and queries number increases

Choosing the right amount of shards and replicas

In the beginning, when you started using Elasticsearch, you probably began by creating the index, importing your data to it and after that you started sending queries. We are pretty sure all worked well at least in the beginning when the amount of data and the number of queries per second were not high. In the background, Elasticsearch created some shards and probably replicas as well (if you are using the default configuration for example) and you didn't pay much attention to this part of the deployment.

When your application grows, you have to index more and more data and handle more and more queries per second. This is the point where everything changes. Problems start to appear (you can read about how we can handle the application's growth in the *Using our knowledge* section of this chapter). It's now time to think how you should plan your index and its configuration to rise with your application. In this chapter, we will give some guidelines on how to handle that. Unfortunately there is no exact recipe, each application has different characteristics and requirements, on which not only the index structure depends, but also the configuration. For example, these factors can be ones like the size of the document or whole index, query types, and desired throughput.

Sharding and over allocation

You already know from the *Introducing Elasticsearch* section in *Chapter 1, Introduction to Elasticsearch* what sharding is, but let's recall it. Sharding is the splitting of an index to a set of smaller indices, which allows us to spread them among multiple nodes in the same cluster. While querying, the result is a sum of all the results that were returned by each shard of an index (although it's not really a sum because a single shard may hold all the data we are interested). By default, Elasticsearch creates five shards for every index even in a single-node environment. This redundancy is called over allocation: it seems to be totally needless at this point and only leads to more complexity when indexing (spreading document to shards) and handling queries (querying shards and merging the results). Happily, this complexity is handled automatically, but why does Elasticsearch do this?

Let's say that we have an index that is built only of a single shard. This means that if our application grows above the capacity of a single machine, we will face a problem. In the current version of Elasticsearch there is no possibility of splitting the index to multiple, smaller parts: we need to say how many shards the index should be built of when we create that index. What we can do is prepare a new index with more shards and re-index the data. However, such an operation requires additional time and server resources such as CPU time, RAM, and mass storage, and of course we may not have time and mentioned resources. From the other side, while using over allocation, we can just add a new server with Elasticsearch installed and Elasticsearch will rebalance the cluster and move parts of the index to the new machine without additional cost of re-indexing. The default configuration (which means five shards and one replica) chosen by the authors of the Elasticsearch is the balance between possibilities of growing and overhead resulting from the need to merge results from a different shard.

The default shard number of 5 is chosen for standard use cases. So now, the question arises: When should we start with more shards or contrary, try to keep the number of shards as low as possible?

The first answer is obvious. If you have a limited and strongly defined data set you can use only a single shard. If you do not, however, the rule of thumb dictates that the optimal number of shards is dependent on the target number of nodes. So, if you plan to use 10 nodes in the future, you need to configure the index to have 10 shards. One important thing to remember is that: for high availability and query throughput we should also configure replicas, and it also takes room on the nodes just like the normal shard. If you have one additional copy of each shard (`number_of_replicas` equal to one) you end with 20 shards: 10 with main data and 10 with its replicas. To sum up, our simple formula can be presented as follows:

$$\text{Max number of nodes} = \text{Number of shards} * (\text{number of replicas} + 1)$$

In other words, if you have planned to use 10 shards and you like to have 2 replicas, the maximum number of nodes for this setup will be 30.

A positive example of over allocation

If you carefully read the previous part of this chapter you will have a strong conviction that you should use minimal number of shards. But sometimes having more shards is handy, because a shard is in fact an Apache Lucene index and more shards means that every operation executed on a single, smaller Lucene index (especially indexing) will be faster. Sometimes this is a good enough reason to use many shards. Of course, there is the possible cost of splitting a query into multiple requests to every shards and merge response from it. This can be avoided for particular types of applications where the queries are always filtered by the concrete parameter. This is the case with multitenant systems, where every query is run in context of the defined user. The idea is simple, we can index data of this user in a single shard and use only that shard during querying. This is in place when routing should be used (we will discuss it in detail in the *Routing explained* section in this chapter).

Multiple shards versus multiple indices

You may wonder, if a shard is de facto of a small Lucene index, what about "true" Elasticsearch indices? What is the difference between them? Technically, it is the same, but some additional features work either with indices or with shards. For sharding, there is a possibility of targeting queries to a particular shard using routing or query execution preference. For indices, a more universal mechanism is stitched in addressing, queries can be sent to several indices using the `/index1,index2,.../` notation. While querying, we can also use the aliasing feature and make the indices visible as one index, just as with sharding. More differences can be spotted in the shard and index balancing logic, although less automation with indexes can be partially hidden by the manual force of deploying indices on particular nodes.

Replicas

While sharding lets us store more data than which fits on a single node, replicas are there to handle increasing throughput and for data security. When a node with the primary shard is lost, Elasticsearch can promote one of the available replicas to be a new primary shard. By default, Elasticsearch creates one replica. However, differently to sharding, the number of replicas can be changed any time using the settings API. This is very convenient while building the applications, our query throughput can grow together with the number of users using it and while using replicas we can handle the increasing number of parallel queries.

The drawback of using more replicas is obvious: the cost of additional space used by additional copies of each shard, and of course the cost of data copy between the primary shard and all the replicas. While choosing the number of shards you should also consider how many replicas need to be present. If you select too many replicas, you can end up using disk space and Elasticsearch resources, when in fact they won't be used. On the other hand, choosing to have none of the replicas may result in the data being lost if something bad happens to the primary shard.

Routing explained

In the *Choosing the right amount of shards and replicas* section in this chapter, we've mentioned routing as a solution for limiting the query to be executed only on a single shard to allow us to increase the query throughput. Now it's time to look closer at this functionality.

Shards and data

Usually it is not important how Elasticsearch divides data into shards and which shard holds the particular document. While querying, the query will be sent to all of them so the only crucial thing is to use the algorithm, which spreads our data evenly. The situation complicates slightly when we want to remove or add a newer version of the document. Elasticsearch must be able to know where the document resides. In practice, this is not a problem. It is enough to use the sharding algorithm, which for the same document identifier will always generate the same value. If we have such an algorithm, Elasticsearch will know which shard to point to while dealing with a document. But don't you think it would be handy to be able to use a more intelligent way of determining in which shard the document should be stored in? For example, we would like to store every book of a particular type only on a particular shard and during searching for that kind of book we could avoid searching on many shards and merging results from them. This is exactly what the routing does. It allows us to provide information to Elasticsearch, which will be used to determine which shard should be used for document storage and for querying. The same routing value will always result in the same shard. It's basically like saying: "Search for documents on the shard where you've put the documents by using the provided routing value".

Let's test routing

To show you an example that will illustrate how Elasticsearch allocates shards and which documents are placed on a particular shard, we will use an additional plugin. It will help us to visualize what Elasticsearch did with our data. Let's install the Paramedic plugin using the following command:

```
bin/plugin -install karmi/elasticsearch-paramedic
```

After Elasticsearch restarts, we can point our browser to `http://localhost:9200/_plugin/paramedic/index.html` and we will be able to see a page with various statistics and information about indices. For our example, the most interested information is the cluster color that indicates the cluster's state and the list of shards and replicas next to every index.

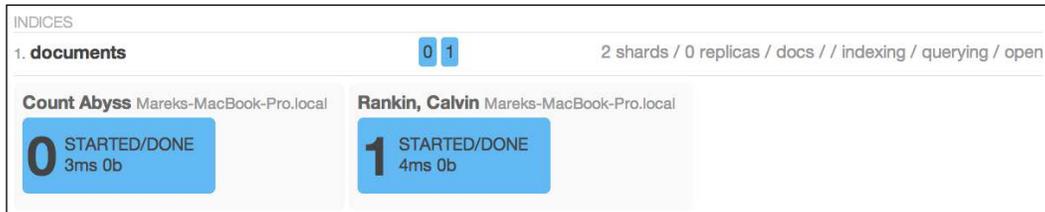
Let's start two Elasticsearch nodes and create an index by running the following command:

```
curl -XPUT localhost:9200/documents -d '{
  settings: {
    number_of_replicas: 0,
    number_of_shards: 2
  }
}'
```

We've created an index without replicas and only two shards. This means that the largest cluster can have only two nodes, and each following node cannot be filled with data unless we increase the number of replicas (you can read about this in the *Choosing the right amount of shards and replicas* section of this chapter). The next operation is to index some documents. In order to do that we will use the following set of commands:

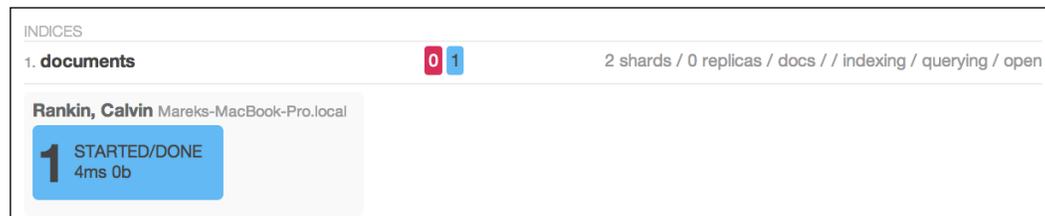
```
curl -XPUT localhost:9200/documents/doc/1 -d '{ "title" : "Document No. 1" }'
curl -XPUT localhost:9200/documents/doc/2 -d '{ "title" : "Document No. 2" }'
curl -XPUT localhost:9200/documents/doc/3 -d '{ "title" : "Document No. 3" }'
curl -XPUT localhost:9200/documents/doc/3 -d '{ "title" : "Document No. 4" }'
```

After that Paramedic shows us two primary shards, as given in the following screenshot:



In the information given about nodes, we can also find the information that we are currently interested in. Each of the nodes in the cluster holds exactly two documents which leads us to the conclusion that the sharding algorithm perfectly did its work and we have an index that is built of shards, and that has evenly redistributed documents.

Now let us do some disaster and shutdown the second node. Now using Paramedic, we should see something similar to the following screenshot:



The first information we see is that the cluster is now in red state. This means that at least one primary shard is missing, which tells us that some of the data is not available and some parts of the index are not available. Nevertheless, Elasticsearch allows us to execute queries, it is our decision what an application should: inform the user about the possibility of the incomplete results or block querying attempts. Look at the result of the query matching all the documents:

```
{
  "took" : 30,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 1,
    "failures" : [ {
      "index" : "documents",
```

```
        "shard" : 1,
        "status" : 500,
        "reason" : "No active shards"
    } ]
},
"hits" : {
  "total" : 2,
  "max_score" : 1.0,
  "hits" : [ {
    "_index" : "documents",
    "_type" : "doc",
    "_id" : "1",
    "_score" : 1.0, "_source" : { "title" : "Document No. 1" }
  }, {
    "_index" : "documents",
    "_type" : "doc",
    "_id" : "3",
    "_score" : 1.0, "_source" : { "title" : "Document No. 3" }
  } ]
}
}
```

As you can see, Elasticsearch returned the information about failures. We saw that shard 1 is not available. In the returned result set, we can only see the documents with identifiers of 1 and 3. Other documents have been lost, at least until the primary shard is back online. If you start the second node, after a while (depending on the network and gateway module settings) the cluster should return to green state and all the documents should be available. Now, we will try to do the same using routing, and we will try to observe the difference in the Elasticsearch's behavior.

Indexing with routing

With routing, we can control the target shard that Elasticsearch will choose to send the documents to. The value of the routing parameter is irrelevant, you can use whatever value you choose. The important thing is that the same value of the routing parameter should be used to place different documents together in the same shard.

There are a few possibilities, with which we can provide the routing information to Elasticsearch. The simplest way is to add a `routing` URI parameter when indexing a document. For example, as follows:

```
curl -XPUT localhost:9200/documents/doc/1?routing=A -d '{ "title" :
"Document" }'
```

Another option is to place a `_routing` field inside the document as follows:

```
curl -XPUT localhost:9200/documents/doc/1 -d '{ "title" : "Document No. 1", "_routing" : "A" }'
```

However, this will work only when the `_routing` field is defined in the mappings, for example:

```
"mappings": {
  "doc": {
    "_routing": {
      "required": true,
      "path": "_routing"
    },
    "properties": {
      "title" : {"type": "string" }
    }
  }
}
```

Let's stop here for a while. In this example, we have used the `_routing` field. It is worth mentioning that the `path` parameter can point to any not-analyzed field from the document. This is a very powerful feature and one of the main advantages of the routing feature. For example, if we extend our document with the `library_id` field indicated library where the book is available, this is logical that all queries based on the library can be more effective when we set up routing based on the `library_id` field.

Now, let's get back to the routing value definition possibilities. The last way is used during bulk indexing. In this case, routing is given in the header for each document. For example:

```
curl -XPUT localhost:9200/_bulk --data-binary '
{ "index" : { "_index" : "documents", "_type" : "doc", "_routing" : "A"
}}
{ "title" : "Document No. 1" }
'
```

Now that we know how it works, let's return to our example.

Indexing with routing

Now, we will do the same as in the previous example, but using routing. The first thing is deleting the old documents. If we will not do this, and add documents with the same identifier, routing may cause the same document to be placed in the other shard. Therefore, we run the following command to delete all the documents from our index:

```
curl -XDELETE localhost:9200/documents/_query?q=:*:*
```

After that, we index our data again, but this time we add routing information, so the command used to index our documents looks as follows:

```
curl -XPUT localhost:9200/documents/doc/1?routing=A -d '{ "title" :
"Document No. 1" }'
curl -XPUT localhost:9200/documents/doc/2?routing=B -d '{ "title" :
"Document No. 2" }'
curl -XPUT localhost:9200/documents/doc/3?routing=A -d '{ "title" :
"Document No. 3" }'
curl -XPUT localhost:9200/documents/doc/4?routing=A -d '{ "title" :
"Document No. 4" }'
```

The routing parameter indicates to Elasticsearch in which shard the document should be placed. Of course, it doesn't tell us that a document with a different routing value will be placed in a different shard. But in our case and with small number of documents, it is true. You can verify it on the Paramedic page, one node has only one document (the one with the routing value of B) and the second node has three documents (the one with the routing value of A). If we kill one node, Paramedic will again show the red cluster, state, and query for all the documents and will return the following response:

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 1,
    "failures" : [ {
      "index" : "documents",
      "shard" : 1,
      "status" : 500,
      "reason" : "No active shards"
    } ]
  },
}
```

```

"hits" : {
  "total" : 3,
  "max_score" : 1.0,
  "hits" : [ {
    "_index" : "documents",
    "_type" : "doc",
    "_id" : "1",
    "_score" : 1.0, "_source" : { "title" : "Document No. 1" }
  }, {
    "_index" : "documents",
    "_type" : "doc",
    "_id" : "3",
    "_score" : 1.0, "_source" : { "title" : "Document No. 3" }
  }, {
    "_index" : "documents",
    "_type" : "doc",
    "_id" : "4",
    "_score" : 1.0, "_source" : { "title" : "Document No. 4" }
  } ]
}
}

```

In this case, a document with the identifier 2 is missing. We lost a node with the documents that had the routing value of B. If we were less lucky, we would have lost three documents!

Querying

Routing allows us to build queries more effectively when we are able to use it. Why send a query to all the nodes if we want to get data from a particular subset of the whole index? For example, these are indexed with routing, A. It is as simple as running the following query:

```
curl -XGET 'localhost:9200/documents/_search?pretty&q=*:*&routing=A'
```

We've just added a routing parameter with the value we are interested in. Elasticsearch replied with the following result:

```

{
  "took" : 0,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,

```

```
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title" : "Document No. 1" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "3",
      "_score" : 1.0, "_source" : { "title" : "Document No. 3" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "4",
      "_score" : 1.0, "_source" : { "title" : "Document No. 4" }
    } ]
  }
}
```

Everything works similar to a charm. But look closer! We forgot to start the node that holds the shard with the documents that were indexed with the routing value of B. Even though we didn't have a full index view, the reply from Elasticsearch doesn't contain information about shard failures. This is a proof that queries with routing hits only the chosen shard and ignores the rest. If we run the same query with `routing=B` we will get an exception similar to the following one:

```
{
  "error" : "SearchPhaseExecutionException[Failed to execute phase
[query_fetch], total failure; shardFailures {[_na_] [documents] [1]: No
active shards}]",
  "status" : 500
}
```

The routing is a very powerful mechanism for optimizing a cluster. It lets us to deploy documents in a way dependent on the application logic, which allows us to build faster queries using fewer resources.

There is one important thing we would like to repeat, however. Routing ensures that during indexing, documents with the same routing value are indexed in the same shard. However, you need to remember that a given shard may have many documents with different routing values. Routing allows limiting the number of nodes used during query, but cannot replace filtering! This means that a query with routing and without routing should have the same set of filters.

Aliases

At the end, it is worth mentioning about one feature, which simplifies working with routing. If you work as a search engine specialist you probably want to hide some configuration details to programmers to allow them to work faster and not care about search details. In an ideal world, they should not worry about routing, shards, and replicas. Aliases allow us to use shards with routing as ordinary indices. For example, let's create an alias using the following command:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '{
  "actions" : [
    {
      "add" : {
        "index" : "documents",
        "alias" : "documentsA",
        "routing" : "A"
      }
    }
  ]
}'
```

In the preceding example we've created a virtual index (an alias) named `documentsA`, which presents information from the `documents` index. However, in addition to that, searching will be limited to the shard which stores information according to the `A` routing value. Thanks to this approach, you can give information about the `documentsA` alias to developers and they may use it for querying and indexing like any other index.

Multiple routing values

ElasticSearch gives us the possibility of searching with several routing values in a single query. Depending on which shard documents with given routing values are placed, it could mean searching on one or more shards. Let's look at the following query:

```
curl -XGET 'localhost:9200/documents/_search?routing=A,B'
```

After executing it, ElasticSearch will send the search request to all the shards in our index, because the routing value of A covers one of two shards of our index and routing value of B covers the second shard of our index.

Of course, multiple routing values are supported in aliases. The following example shows the usage of these features:

```
curl -XPOST 'http://localhost:9200/_aliases' -d '{
  "actions" : [
    {
      "add" : {
        "index" : "documents",
        "alias" : "documentsA",
        "search_routing" : "A,B",
        "index_routing" : "A"
      }
    }
  ]
}'
```

The preceding example shows two additional configuration parameters which we didn't talk until now, we can define different values of routing for searching and indexing. In the preceding case, we've defined that during querying (the `search_routing` parameter) two values of routing (A and B) will be applied. While indexing (the `index_routing` parameter) only one value (A) will be used. Note that indexing doesn't support multiple routing values and you should also remember proper filtering (you can also add it in your alias).

Altering the default shard allocation behavior

In the previous chapters, we've learned many things about sharding and the features connected with it. We've also discussed how shard allocation works (the *Adjusting shard allocation* section in this chapter). However, we didn't talk about anything else than the default behavior. Elasticsearch gives us more possibilities of building advanced systems with specific sharding placement strategies. In this section, we will take a deeper look on what else we can do when it comes to shard allocation.

Introducing ShardAllocator

`ShardAllocator` is the main class responsible for making a decision about the shard's placement. As you recall, shards need to be allocated when Elasticsearch change data allocation on the nodes, for example due to changes in the cluster topology (when nodes are added or removed) or by forced rebalancing. Internally, allocator is an implementation of the `org.elasticsearch.cluster.routing.allocation.allocator.ShardsAllocator` interface. Elasticsearch provides two types of allocators, which are as follows:

- `even_shard`
- `balanced` (the default one)

We can specify which allocator implementation we want to use by setting the `cluster.routing.allocation.type` property in the `elasticsearch.yml` file or by using the settings API. Let's look closer at the afore mentioned allocator types.

The `even_shard` ShardAllocator

The allocator that was also available in Elasticsearch prior to version 0.90.0. It focuses on ensuring that every node has the same number of shards (of course, this is not always possible). It also doesn't allow storing the primary shard and its replicas on the same node. When reallocation is needed and the `even_shard` `ShardAllocator` is used, Elasticsearch moves shards from the most occupied nodes to ones that are less occupied as long as the cluster is not fully balanced or no more moves can be done. The important information is that this allocator doesn't work on the indices level, which means that it doesn't take into consideration where different shards from the same index are placed as long as a shard and its replica are on different nodes.

The balanced ShardAllocator

This is the newest allocator that was introduced in Elasticsearch 0.90.0. It does allocation based on several weights, which we can control. Compared to the `even_shard` allocator we discussed earlier, it introduces additional possibilities of tuning allocation process by exposing some parameters, which can be changed dynamically using the cluster update API. The parameters that we can control are as follows:

- `cluster.routing.allocation.balance.shard`: Its default value is 0.45
- `cluster.routing.allocation.balance.index`: Its default value is 0.5
- `cluster.routing.allocation.balance.primary`: Its default value is 0.05
- `cluster.routing.allocation.balance.threshold`: Its default value is 1.0

The preceding parameters define the behavior of the `balanced` allocator. Starting from the first, we have the weight of the factor based on the total number of shards, next we have the weight of the factor based on the shards of the given index, and finally the weight of the factor based on primary shards. We will leave the threshold explanation for now. The higher the weight of a particular factor, the more important it will be, and the more influence it will have when Elasticsearch will take a decision about shard relocation.

The first factor tells Elasticsearch how important for us is the number of shards allocated to each node are similar. The second factor does the same, but not for all shards, but for shards of the same index. The third factor tells Elasticsearch how important is to have the same number of primary shards allocated to the nodes. So, if having the same amount of primary shards distributed among nodes in your cluster is very important, you should increase the value of the `cluster.routing.allocation.balance.primary` weight and probably decrease the value of all the other weights excluding threshold.

Finally, if the sum of all the calculated factors multiplied by the weights we've given is higher than the defined threshold then shards of such indexes needs to be reallocated. If for some reason you want to avoid considering one or more factors, just set their weight to 0.

The custom ShardAllocator

The built-in allocators may not fit in your deployment scenario. For example, you could need something that during allocation takes into consideration differences in sizes of the indices. Another example is a big cluster with various hardware components, different CPUs, RAM amount, or disk space. All of these factors can lead to inefficiently distributed data on nodes.

Happily, there is a possibility to implement your own solution. In this case, the `cluster.routing.allocation.type` setting should be set to a fully qualified class name, which implements the `org.elasticsearch.cluster.routing.allocation.allocator.ShardsAllocator` interface.

Deciders

To understand how shard allocators decide when shard should be moved and to which node it should be moved, we should discuss the internals of Elasticsearch, which are called deciders. These are the brains where the allocation decisions are made. Elasticsearch allows us to use many deciders simultaneously and all of them will vote on the decision. There is a rule consensus, for example, if one decider votes against reallocations of a shard, that shard cannot be moved. The list of deciders is constant and cannot be changed without affecting the Elasticsearch code. Let's see which deciders are used by default.

SameShardAllocationDecider

As its name suggests, this decider disallows situations where copies of the same data (the shard and its replica) are placed on the same node. The reason for this is obvious: we don't want to keep the backup of our data at the same place as the master data. However while talking about this decider we should mention the `cluster.routing.allocation.same_shard.host` property. It controls whether Elasticsearch should care about the physical machine on which the shard is placed. By default, it is set to `false`, because many nodes can run on the same server, which is running multiple virtual machines. When set to `true` this decider will disallow allocating a shard and its replicas on the same physical machine. It may seem strange, but think about virtualization and the modern world, where the operating system cannot determine on which physical machine it works. Because of this, it is better to rely more on settings such as `index.routing.allocation` properties family described in the *Adjusting shard allocation* section in this chapter.

ShardsLimitAllocationDecider

`ShardsLimitAllocationDecider` makes sure that for a given index there is no more than a given number of shards on a single node. The number is defined by the `index.routing.allocation.total_shards_per_node` property setting which can be set in the `elasticsearch.yml` file or updated on live indices using the index update API. The default value is `-1`, which means that no restrictions are applied. Note that lowering this value forces reallocation and causes additional load on the cluster for the duration of rebalancing.

FilterAllocationDecider

`FilterAllocationDecider` is used when we add the properties that are controlling shard allocation, which means the ones that match the `*.routing.allocation.*` name pattern. You can find more information about how this decider works in the *Adjusting shard allocation* section of this chapter.

ReplicaAfterPrimaryActiveAllocationDecider

The decider that causes that ElasticSearch will start allocating replicas only when the primary shards are allocated.

ClusterRebalanceAllocationDecider

`ClusterRebalanceAllocationDecider` allows changing conditions when rebalancing will be done according to the current cluster state. The decider can be controlled by the `cluster.routing.allocation.allow_rebalance` property, which can take the following values:

- `indices_all_active`: Its default value indicates that rebalancing can be done only when all the existing shards in the cluster are allocated
- `indices primaries_active`: This setting specifies that rebalancing can be done when primary shards are allocated
- `always`: This setting specifies that rebalancing is always allowed even when primaries and replicas are not allocated

Note that these settings cannot be changed at runtime.

ConcurrentRebalanceAllocationDecider

`ConcurrentRebalanceAllocationDecider` allows throttling relocation operations based on the `cluster.routing.allocation.cluster_concurrent_rebalance` property. With the help of the mentioned property, we can set up the number of concurrent relocations that should happen at once on the given cluster. The default value is 2, which means that no more than two shards can be moved at the same time in our cluster. Setting the value to -1 turns off throttling, which means that the number of concurrent rebalance is not limited.

DisableAllocationDecider

`DisableAllocationDecider` is another decider that exposes possibility of tuning its behavior and adjust it to our needs. In order to do that, we can change the following settings (statically in both `elasticsearch.yml` and using the cluster settings API):

- `cluster.routing.allocation.disable_allocation`: This setting allows us to disable all allocation
- `cluster.routing.allocation.disable_new_allocation`: This setting allows us to disable new primary shard allocation
- `cluster.routing.allocation.disable_replica_allocation`: This setting allows us to disable replicas allocation

All these settings are by default set to `false`. They are quite handy when you want to have full control when allocations happen. For example, you can disable reallocations when you want to quickly reconfigure and restart several nodes. In addition, remember that even though you can set the preceding properties in `elasticsearch.yml`, it usually makes sense to use the update API here.

AwarenessAllocationDecider

`AwarenessAllocationDecider` is responsible for handling the awareness allocation functionality. Whenever you use the `cluster.routing.allocation.awareness.attributes` settings, this decider will jump in. More information about how it works can be found in the *Adjusting shard allocation* section of this chapter.

ThrottlingAllocationDecider

`ThrottlingAllocationDecider` is similar to `ConcurrentRebalanceAllocationDecider` discussed earlier. This decider allows us to limit the load generated by the allocation process. In this case, we are allowed to control the recovery process by using the following properties:

- `cluster.routing.allocation.node_initial primaries_recoveries`: This property defaults to 4. It describes the number of initial primary shard recovery operations allowed on a single node.
- `cluster.routing.allocation.node_concurrent_recoveries`: This property defaults to 2. It defines the number of concurrent recovery operations on a single node.

RebalanceOnlyWhenActiveAllocationDecider

`RebalanceOnlyWhenActiveAllocationDecider` allows rebalancing process to happen only when all the shards are active in a single shard replication group (which means the primary shard and its replicas).

DiskThresholdDecider

`DiskThresholdDecider` introduced in ElasticSearch 0.90.4 allows us to allocate shards on the basis of free disk space available on the server. By default, it is disabled and we must set the `cluster.routing.allocation.disk.threshold_enabled` property to `true` in order to enable it. This decider allows us to configure thresholds when a shard can be allocated to a node and when ElasticSearch should try to relocate shard to another node.

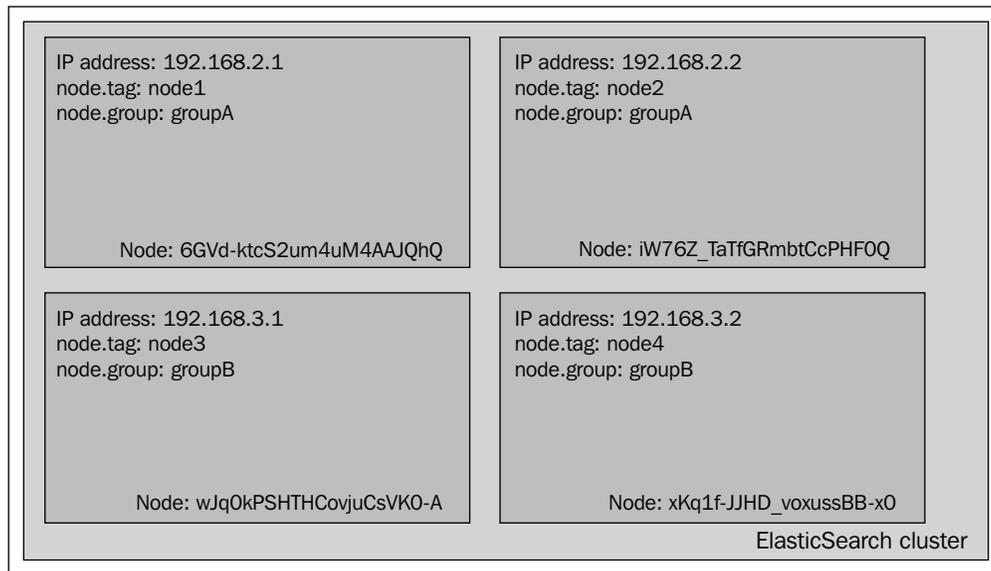
The `cluster.routing.allocation.disk.watermark.low` property allows us to specify the threshold or an absolute value when shard allocation is possible. For example, the default `0.7` value, informs ElasticSearch that new shards can be allocated to a node when the disk space is less than 70 percent.

The `cluster.routing.allocation.disk.watermark.high` property allows us to specify the threshold or an absolute value when a shard allocator will try to relocate the shard to another node. By default, it is set to `0.85`, which means that ElasticSearch will try to reallocate the shard when disk usage rises above 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 (for example, `0.7` or `0.85`) or to an absolute value (for example, `1000mb`). In addition, all the properties mentioned in this section can be set both statically in `elasticsearch.yml` and updated dynamically using the ElasticSearch API.

Adjusting shard allocation

In the *ElasticSearch Server* book we talked about how to manually force shard allocation, how to cancel it, and how to move shards around the cluster with a single API command. However, that's not the only thing ElasticSearch allows us to use when it comes to shard allocation, we are also allowed to define a set of rules on which shard allocation will work. For example, let's say that we have a four-node cluster, which looks as follows:



As you can see, our cluster is built of four nodes. Each node was bound to a specific IP address and each node was given the `tag` property and a `group` property (added to `elasticsearch.yml` as the `node.tag` and `node.group` properties). This cluster will serve the purpose of showing how shard allocation filtering works. The `group` and `tag` properties can be given names whatever you want, you just need to prefix your desired property name with the `node` name, for example if you like to use a property name, `party`, you need to just add `node.party: party1` to your `elasticsearch.yml` file.

Allocation awareness

Allocation awareness allows us to configure shards and their replicas allocation with the use of generic parameters. In order to illustrate how allocation awareness works we will use our example cluster. For the example to work, we should add the following property to the `elasticsearch.yml` file:

```
cluster.routing.allocation.awareness.attributes: group
```

This will inform Elasticsearch to use the `node.group` property as the awareness parameter.

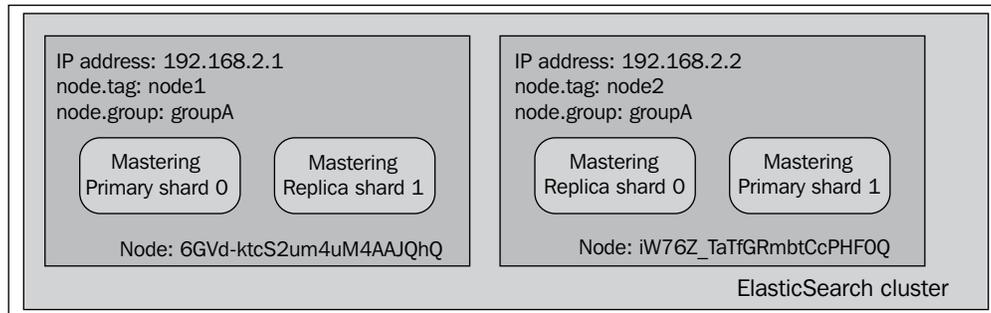
 One can specify multiple attributes while setting the `cluster.routing.allocation.awareness.attributes` property. For example:

```
cluster.routing.allocation.awareness.attributes: group, node
```

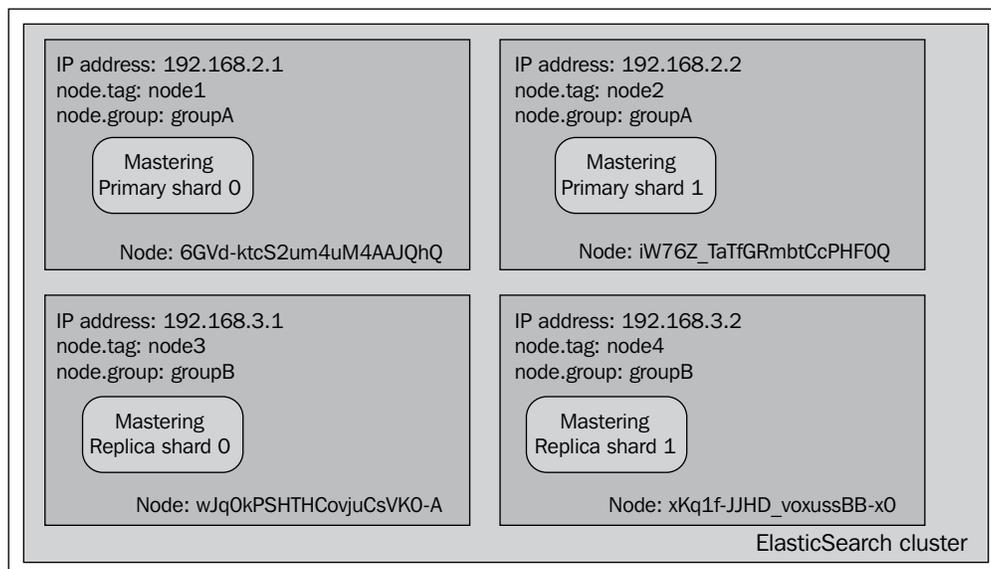
After that, let's start the first two nodes, the ones with the `node.group` parameter equal to `groupA` and let's create an index by running the following command:

```
curl -XPOST 'localhost:9200/mastering' -d '{
  "settings" : {
    "index" : {
      "number_of_shards" : 2,
      "number_of_replicas" : 1
    }
  }
}'
```

After executing that command our two-node cluster will look more or less similar to the following diagram:



As you can see, the index was divided between two nodes evenly. Now, let's see what happens when we launch the rest of the nodes (the ones with `node.group` set to `groupB`):



Notice the difference: the primary shards were not moved from their original allocation nodes, but the replica shards were moved to the nodes with a different `node.group` value. That's exactly right. While using shard allocation awareness, Elasticsearch won't allocate shards and replicas to the nodes with the same value of the property used to determine the allocation awareness (which in our case is the `node.group` value). One of the example usages of this functionality is dividing the cluster topology between virtual machines or physical locations, to be sure that you don't have a single point of failure.

 Please remember that while using allocation awareness, shards will not be allocated to the node that doesn't have the expected attributes set. So in our example, a node without setting the `node.group` property will not be taken into consideration by the allocation mechanism.

Forcing allocation awareness

Forcing allocation awareness can come in handy when we know in advance how many values our awareness attributes can take and we don't want more replicas than needed to be allocated in our cluster, for example not to overload our cluster with too many replicas. For that, we can force allocation awareness to be active only for certain attributes. We can specify those values by using the `cluster.routing.allocation.awareness.force.zone.values` property and providing a list of comma-separated values to it. For example, if we like for allocation awareness to only use the `groupA` and `groupB` values of the `node.group` property, we should add the following code to the `elasticsearch.yml` file:

```
cluster.routing.allocation.awareness.attributes: group
cluster.routing.allocation.awareness.force.zone.values: groupA, groupB
```

Filtering

ElasticSearch allows us configure allocation for the whole cluster or for the index level. In case of cluster allocation we can use the following properties prefixes:

- `cluster.routing.allocation.include`
- `cluster.routing.allocation.require`
- `cluster.routing.allocation.exclude`

When it comes to index-specific allocation, we can use the following properties prefixes:

- `index.routing.allocation.include`
- `index.routing.allocation.require`
- `index.routing.allocation.exclude`

The previously mentioned prefixes can be used with the properties that we've defined in the `elasticsearch.yml` file (our `tag` and `group` properties) and with a special property called `_ip` that allows us to match or exclude using node's IP address, for example:

```
cluster.routing.allocation.include._ip: 192.168.2.1
```

If we like to include nodes with a `group` property matching the value `groupA`, we should set the following property:

```
cluster.routing.allocation.include.group: groupA
```

Notice that we've used the `cluster.routing.allocation.include` prefix and we've concatenated it with the name of the property, which is `group` in our case.

But what those properties mean?

If you look closely at the parameters mentioned previously, you will notice that there are three kinds of them.

- `include`: This type will result in including all the nodes with this parameter defined. If multiple `include` conditions are visible then all the nodes that match at least a single of those conditions will be taken into consideration while allocating shards. For example, if we add two `cluster.routing.allocation.include.tag` parameters to our configuration, one with the value of `node1` and second with the value of `node2`, we would end up with indices (actually their shards) being allocated to the first and the second node (counting from left to right). To sum up, the nodes that have the `include` allocation parameter type, we will take into consideration ElasticSearch while choosing the nodes to place shards on, but that doesn't mean that ElasticSearch will put shards on them.
- `require`: This property was introduced in the ElasticSearch 0.90 type of the allocation filter. It requires all the nodes to have the value that matches the value of this property. For example, if we add one `cluster.routing.allocation.require.tag` parameter to our configuration with the value of `node1` and a `cluster.routing.allocation.require.group` parameter with the value of `groupA` we would end up with shards allocated only to the first node (the one with IP address of `192.168.2.1`).

- `exclude`: This property allows us to exclude nodes with given properties from the allocation process. For example, if we set `cluster.routing.allocation.include.tag` to `groupA`, we would end up with indices being allocated only to nodes with IP addresses, `192.168.3.1` and `192.168.3.2` (the third and the fourth node in our example).



Property values can use simple wildcard characters. For example, if we like to include all the nodes that have the `group` parameter value beginning with `group`, we should set the `cluster.routing.allocation.include.group` property to `group*`. In the example cluster case, it will result in matching nodes with the `groupA` and `groupB` group parameter values.

Runtime allocation updating

In addition to setting all the discussed properties in the `elasticsearch.yml` file we can also use the update API to update those settings in real time, when the cluster is already running.

Index-level updates

In order to update settings for a given index (for example our `mastering` index) we should run the following command:

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.require.group": "groupA"
}'
```

As you can see, the command was sent to the `_settings` endpoint for a given index. You can include multiple properties in a single call.

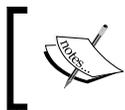
Cluster-level updates

In order to update settings for the whole cluster we should run the following command:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "cluster.routing.allocation.require.group": "groupA"
  }
}'
```

As you can see, the command was sent to the `_cluster/settings` endpoint. You can include multiple properties in a single call. Please remember that the `transient` name in the preceding command means that the property will be forgotten after restarting the cluster. If you want to avoid that and set this property as a permanent one, use the `persistent` property instead of the `transient` one. An example command, which will persist the settings between restarts would look as follows:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent" : {
    "cluster.routing.allocation.require.group": "groupA"
  }
}'
```



Please note that running the preceding commands, depending on the command and where your indices are located, can result in shards being moved between the nodes.

Defining total shards allowed per node

In addition to the previously mentioned properties, we are also allowed to define how many shards (primaries and replicas) for an index can be allocated per node. In order to do that, one should set the `index.routing.allocation.total_shards_per_node` property to a desired value. For example in the `elasticsearch.yml` file we should set:

```
index.routing.allocation.total_shards_per_node: 4
```

This would result in maximum of four shards per index being allocated to a single node.

This property can also be updated on a live cluster using the update API as follows:

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.total_shards_per_node": "4"
}'
```

And now, let's see a few examples of how the cluster would look, while creating a single index and having the allocation properties used in the `elasticsearch.yml` file.

Inclusion

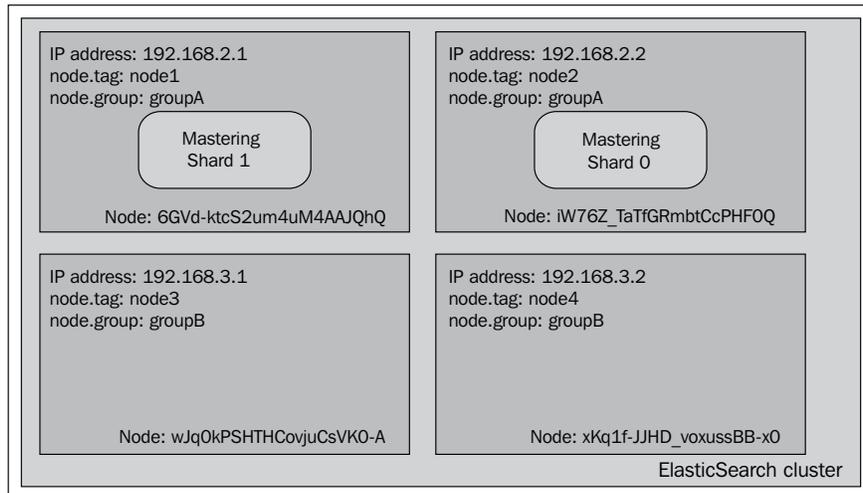
Now let's use our example cluster to see how the allocation inclusion works. Let's start by creating the `mastering` index by using the following command:

```
curl -XPOST 'localhost:9200/mastering' -d '{
  "settings" : {
    "index" : {
      "number_of_shards" : 2,
      "number_of_replicas" : 0
    }
  }
}'
```

After that let's try running the following command:

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.include.tag": "node1",
  "index.routing.allocation.include.group": "groupA",
  "index.routing.allocation.total_shards_per_node": 1
}'
```

If we visualize the response of the index status we will notice that the cluster looks similar to the the following diagram:



As you can see, the **Mastering** index shards are allocated to the nodes with the **tag** property set to **node1** or the **group** property set to **groupA**.

Requirements

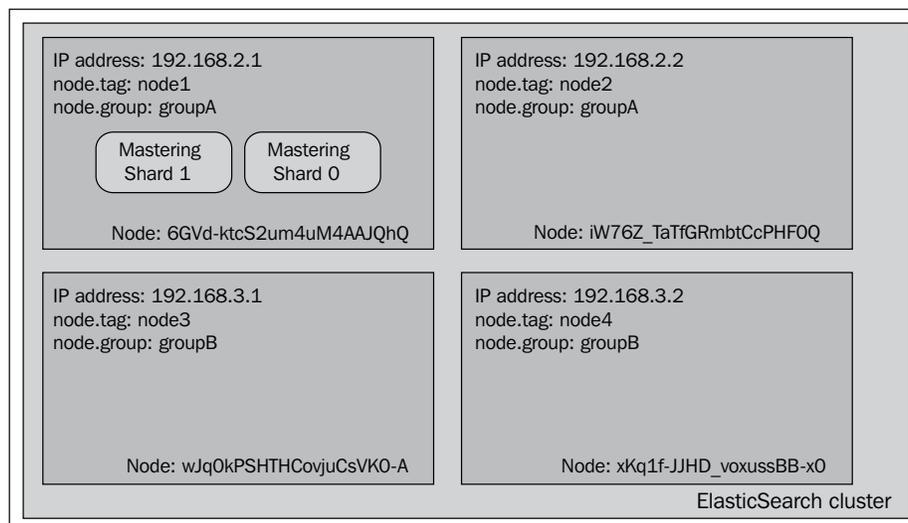
Now let's re-use our example cluster (let's assume we don't have any index there again) to see how the allocation requirement works. Let's again start by creating the **mastering** index using the following command:

```
curl -XPOST 'localhost:9200/mastering' -d '{
  "settings" : {
    "index" : {
      "number_of_shards" : 2,
      "number_of_replicas" : 0
    }
  }
}'
```

After that let's try running the following command:

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.require.tag": "node1",
  "index.routing.allocation.require.group": "groupA"
}'
```

If we visualize the response of the index status command we will see that the cluster looks as follows:



As you can see the view is different than the one when using `include`. This is because we tell Elasticsearch to allocate shards of the **Mastering** index only to the nodes that match both `require` parameters and in our case the only node that matches both is the first node.

Exclusion

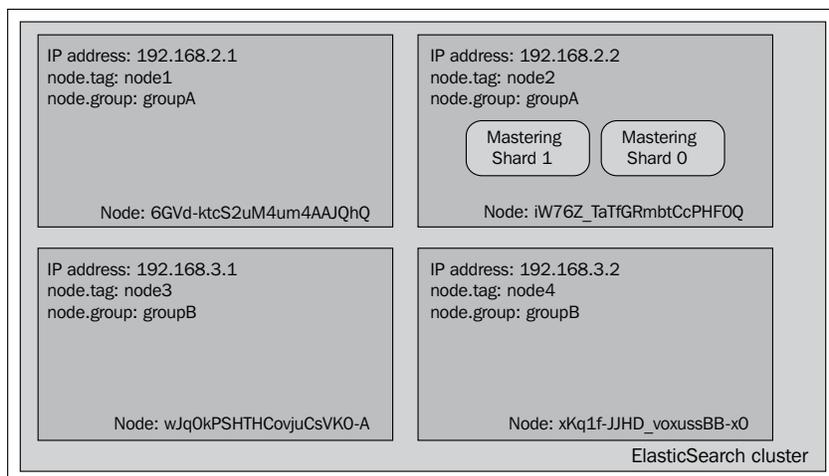
Again let's start with a clear example cluster and after that we will create the `mastering` index by using the following command:

```
curl -XPOST 'localhost:9200/mastering' -d '{
  "settings" : {
    "index" : {
      "number_of_shards" : 2,
      "number_of_replicas" : 0
    }
  }
}'
```

After that let's try running the following command to test allocation exclusion:

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.exclude.tag": "node1",
  "index.routing.allocation.require.group": "groupA"
}'
```

And again let's look at our cluster now:



As you can see, we said that we require the **group** property to be equal to **groupA** and we want to exclude the node with **tag** equal to **node1**. This resulted in a shard of the **Mastering** index being allocated to the node with IP address **192.168.2.2**, which is what we wanted.

Additional shard allocation properties

In addition to what we've already discussed, Elasticsearch allows us to use a few properties when it comes to shard allocation. Let's discuss them and see what else we can control.

- `cluster.routing.allocation.allow_rebalance`: This property allows us to control when rebalancing will take place based on the status of all the shards in the cluster. The possible values are: `always`, using which rebalancing will happen when its needed without looking at the state of the shards of the indices (be careful with this value as it can cause high load), `indices primaries active`, using which rebalancing will happen as soon as all the primary shards are active, and `indices all active`, using which rebalancing will happen after all the shards (primaries and replicas) are allocated. The default value is `indices all active`.
- `cluster.routing.allocation.cluster_concurrent_rebalance`: This property defaults to 2 and specifies how many concurrent shards may be rebalanced at the same time in our cluster. Setting this property to higher values may result in high I/O, increased network activity, and increased nodes load.
- `cluster.routing.allocation.node_initial_primaries_recoveries`: This property allows us to specify how many primary shards can be concurrently recovered per node. Because of the fact that the primaries recovery is usually fast, we can set this property to higher values without putting too much pressure on the node itself. This property defaults to 4.
- `cluster.routing.allocation.node_concurrent_recoveries`: This property defaults to 2 and specifies how many concurrent recoveries are allowed per node. Please remember that specifying too many concurrent recoveries per node may result in very high I/O activity.
- `cluster.routing.allocation.disable_new_allocation`: This property, by default is set to `false`. It allows us to disable allocation of new shards (both the primary and the replicas) for newly created indices. This property can be useful when you want to delay allocation of newly created indices or shards for some reason. We can also disable allocation of only the new shards for a given index by setting the `index.routing.allocation.disable_new_allocation` property to `true` in that index settings.

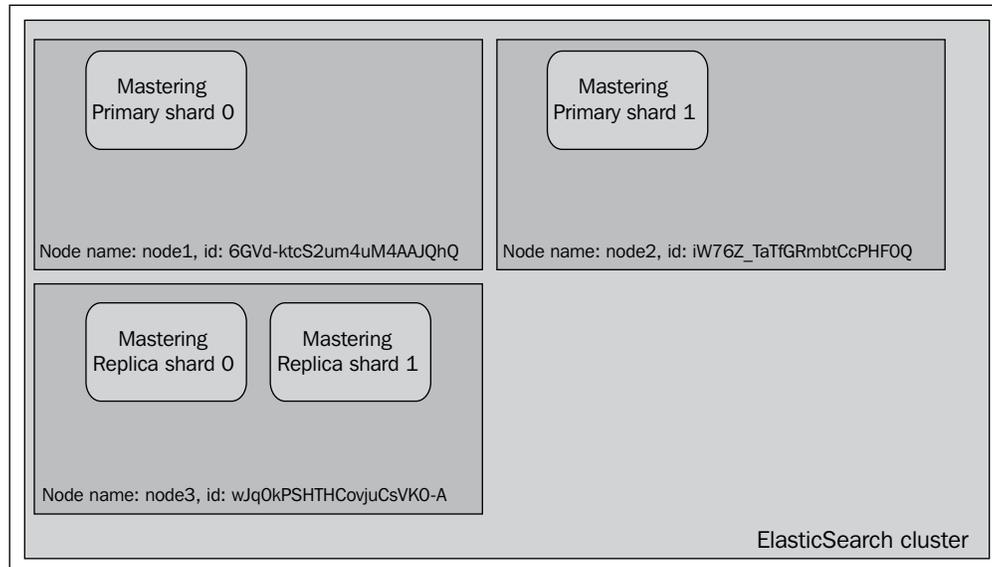
- `cluster.routing.allocation.disable_allocation`: This property, by default is set to `false` and allows us to disable allocation for already created primary and replica shards. Please note that promoting a replica shard to a primary one (if the primary one is not existent) isn't an allocation and thus such an operation will be permitted even if this property is set to `true`. This property can be useful when you want to disable allocation for newly created indices for some time. We can also completely disable allocation of shards for a given index by setting the `index.routing.allocation.disable_allocation` property to `true` in that index settings.
- `cluster.routing.allocation.disable_replica_allocation`: This property, by default, is set to `false`. When set to `true` it will disallow the process of replica shards allocation to nodes. This can be useful when one wants to stop allocating replicas for some time. We can also disable replica allocation for a given index by setting the `index.routing.allocation.disable_replica_allocation` property to `true` in that index settings.

All the preceding properties can be set in both the `elasticsearch.yml` file and by using the update settings API. However in practice, you use only the update settings to work with properties such as `cluster.routing.allocation.disable_new_allocation`, `cluster.routing.allocation.disable_allocation`, or `cluster.routing.allocation.disable_replica_allocation`.

Query execution preference

Let's forget about the shard placement and how to configure it, at least for a moment. In addition to all that fancy stuff that ElasticSearch allows us to set for shards and replicas we also have the possibility to specify where our queries (and other operations, for example, the real time get) should be executed.

Before we get into details, let's look at our example cluster:



As you can see, we have three nodes and a single index called **Mastering**. Our index is divided into two primary shards and there is one replica for each primary shard.

Introducing the preference parameter

In order to control where the query (and other operations) we are sending will be executed, we can use the `preference` parameter, which can be set to one of the following values:

- `_primary`: Using this property, the operation we are sending will only be executed on primary shards. So if we would send a query against the `mastering` index with the `preference` parameter set to the `_primary` value we would have it executed on the nodes with names `node1` and `node2`. For example, if you know that your primary shards are in one rack and the replicas are in other racks, you may want to execute the operation on primary shards to avoid network traffic.

- `_primary_first`: This option is similar to the `_primary` value's behavior, but with a failover mechanism. If we ran a query against the master index with the `preference` parameter set to the `_primary_first` value we would have it executed on the nodes with names `node1` and `node2`, however if one (or more) of the primary shard fails, the query will be executed against the other shard, which in our case is allocated to a node named `node3`. As we said, this is very similar to the `_primary` value, but with additional fall back to replicas if the primary shard is not available for some reason.
- `_local`: Elasticsearch will prefer to execute the operation on a local node if possible. For example, if we would send a query to `node3` with the `preference` parameter set to `_local`, we would end up having that query executed on that node. However, if we would send the same query to `node2`, we would end up with one query executed against the primary shard numbered 1 (which is located on that node) and the second part of the query will be executed against `node1` or `node3` where the shard numbered 0 resides. This is especially useful while trying to minimize network latency, while using the `_local` preference we ensure that our queries be executed locally whenever possible (for example when running a client connection from a local node or sending a query to a node).
- `_only_node:wJq0kPSHTHCovjuCsVK0-A`: This operation will be only executed against a node with the provided identifier (which is `wJq0kPSHTHCovjuCsVK0-A` in this case). So in our case, the query would be executed against two replicas located on `node3`. Please remember that if there aren't enough shards to cover all the index data, the query will be executed against only the shard available in the specified node. For example, if we would set the `preference` parameter to `_only_node:6GVd-ktcS2um4uM4AAJQhQ` we would end up having our query executed against a single shard. This can be useful for examples where we know that one of our nodes is more powerful than the other ones and we want some of the queries to be executed only on that node.
- `_prefer_node:wJq0kPSHTHCovjuCsVK0-A`: This option sets the `preference` parameter to the `_prefer_node:` value followed by a node identifier (which is `wJq0kPSHTHCovjuCsVK0-A` in our case) will result in Elasticsearch preferring the mentioned node while executing the query, but if some shards are not available on the preferred node Elasticsearch will send the appropriate query parts to nodes where the shards are available. Similar to the `_only_node` option, `_prefer_node` can be used while choosing a particular node, however with a fall back to other nodes.

- `_shards:0,1`: This is the preference value that allows us to identify which shards should the operation be executed against (in our case, it will be all the shards, because we only have shards 0 and 1 in the `mastering` index). This is the only preference parameter value that can be combined with the other mentioned values. For example, in order to locally execute our query against the 0 and 1 shard, we should concatenate the `0,1` value with `_local` using the `;` character, so the final value of the preference parameter should look like this: `0,1;_local`. Allowing us to execute the operation against a single shard can be useful for diagnosis purposes.
- `custom`, string value: This is a custom value that will guarantee that the query with the same value will be executed against the same shards. For example, if we send a query with the preference parameter set to the `mastering_elasticsearch` value we would end up having the query executed against primary shards located on nodes named `node1` and `node2`. If we send another query with the same preference parameter value, then the second query will be again executed against the shards located on nodes named `node1` and `node2`. This functionality can help us in cases where we have different refresh rates and we don't want our users to see different results while repeating requests.

There is one more thing missing, which is the default behavior. What ElasticSearch will do by default is randomize the operation between shards and replicas. If we sent many queries we would end up having the same (or almost the same) number of queries run against each of the shard and replicas.

Using our knowledge

As we are slowly approaching the end of the fourth chapter we need to get something that is closer to what you can encounter during your everyday work. Because of that we have decided to divide the real-life example into two sections. In this section, you'll see how to combine the knowledge we've got so far to build a fault-tolerant and scalable cluster based on some assumptions. Because this chapter is mostly about configuration, we will concentrate on that. The mappings and your data may be different, but with similar amount data and queries hitting your cluster the following sections may be useful for you.

Assumptions

Before we go into the juicy configuration details let's make some basic assumptions with which using which we will configure our ElasticSearch cluster.

Data volume and queries specification

Let's assume that we have an online library that currently sells about 100,000 books in different languages.

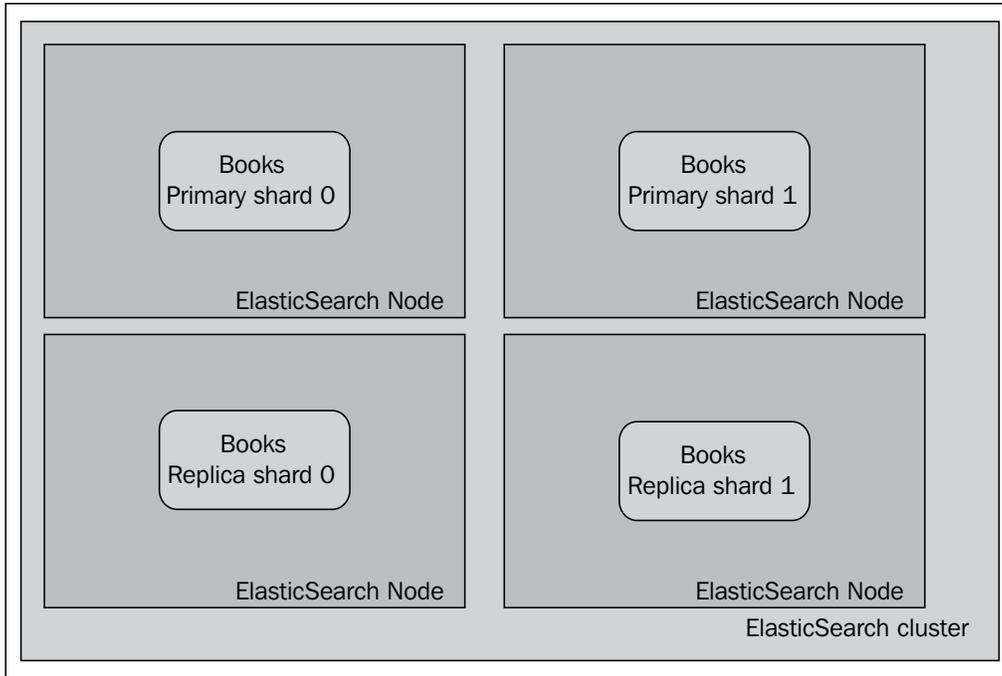
We also expect for the average query response time to be less or equal to 200 milliseconds in order not to force our users to wait too long for the search or browse for the results to be rendered.

So now, let's get back to the expected load. We did some performance tests (which are beyond the scope of this book) and we've managed to learn that our four-node cluster is behaving the best when we have our data divided into two shards and we have one replica for each of the created shards.



You'll probably want to do performance tests on your own. To do this, you can use one of the open source tools to run queries against your cluster, for example, Apache JMeter (<http://jmeter.apache.org/>) or ActionGenerator (<https://github.com/sematext/ActionGenerator>). In addition to that you can either use the ElasticSearch API to look at the statistics recorded by using a plugin similar to ElasticSearch paramedic (<https://github.com/karmi/elasticsearch-paramedic>), or BigDesk (<https://github.com/lukas-vlcek/bigdesk>), or use a complete monitoring and alerting solution similar to SPM for ElasticSearch from Sematext (<http://sematext.com/spm/elasticsearch-performance-monitoring/index.html>). All of these will provide the view on how your cluster behaves during performance tests and where are the bottlenecks. In addition to those mentioned, you'll probably want to monitor the JVM garbage collector's work and how the operating system behaves (some of the mentioned tools do that for you).

So we want our cluster to look similar to the following diagram:



Of course, the exact placement of shards and their replicas may be different, but the logic behind it stays the same, that is, we want to have a single shard per node.

Configuration

And now we'll create the configuration for our cluster and discuss in detail why we used each of the specified properties. Let's start:

```
cluster.name: books
# node configuration
node.master: true
node.data: true
node.max_local_storage_nodes: 1
# indices configuration
index.number_of_shards: 2
index.number_of_replicas: 1
index.routing.allocation.total_shards_per_node: 1
# instance paths
path.conf: /usr/share/elasticsearch/conf
path.plugins: /usr/share/elasticsearch/plugins
path.data: /mnt/data/elasticsearch
path.work: /usr/share/elasticsearch/work
path.logs: /var/log/elasticsearch
# swapping
bootstrap.mlockall: true
#gateway
gateway.type: local
gateway.recover_after_nodes: 3
gateway.recover_after_time: 30s
gateway.expected_nodes: 4
# recovery
cluster.routing.allocation.node_initial primaries recoveries: 1
cluster.routing.allocation.node_concurrent recoveries: 1
indices.recovery.concurrent_streams: 8
# discovery
discovery.zen.minimum_master_nodes: 3
# search and fetch logging
index.search.slowlog.threshold.query.info: 500ms
index.search.slowlog.threshold.query.debug: 100ms
index.search.slowlog.threshold.fetch.info: 1s
index.search.slowlog.threshold.fetch.debug: 200ms
# JVM garbage collection work logging
monitor.jvm.gc.ParNew.info: 700ms
monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

Now let's see what these values mean.

Node-level configuration

With the node-level configuration we've specified a cluster name (using the `cluster.name` property) that will identify our cluster. If we have multiple clusters on the same network, the nodes with the same cluster name will try to connect to each other and form a cluster. Next, we say that this particular node can be elected as a master node (the `node.master: true` property) and that it can hold the indices data (`node.data: true`). In addition to that, by setting the `node.max_local_storage_nodes` property to 1 we say that we don't want more than a single instance of Elasticsearch running on a single node.

Indices configuration

Because we will only have a single index and we are not planning more indices for now, we decided to set the default number of shards to 2 (the `index.number_of_shards` property) and the default number of replicas to 1 (the `index.number_of_replicas` property). In addition to that we've set the `index.routing.allocation.total_shards_per_node` property to 1, which means that for each index Elasticsearch will place a single shard on a node, which in case of our four-node cluster will result in an even placement of shards per node.

The directories layout

We've installed Elasticsearch in `/usr/share/elasticsearch` and because of that the `conf`, `plugins`, and the `work` directories are configured to be present in that directory. The data will be placed on a hard drive specially designated for that cause and which is available under the `/mnt/data/elasticsearch` mount point. Finally, the logfiles will be stored in the `/var/log/elasticsearch` directory. By having the directory layout that way while we do updates, we will only need to think about the `/usr/share/elasticsearch` directory and not touch the rest.

Gateway configuration

As you know, the gateway is the module responsible for the storage of our indices and their metadata. In our case we've chosen the suggested and the only non-deprecated type of gateway, which is the `local` (`gateway.type` property). We said that we want the recovery process to start as soon as there are three nodes (the `gateway.recover_after_nodes` property) and start 30 seconds after at least 3 nodes are connected to each other (the `gateway.recover_after_time` property). In addition to that, we've informed Elasticsearch that our cluster will consist of four nodes by setting the `gateway.expected_nodes` property to 4.

Recovery

One of the crucial configuration options when it comes to Elasticsearch is the recovery configuration. Although it is not needed every day, because you don't tend to restart Elasticsearch day-to-day and of course you don't want your cluster to fail often. However such situations happen and it's better to be prepared. So let's discuss what we've used. We've set the `cluster.routing.allocation.node_initial_primaries_recoveries` property to 1, which means that we only allow for a single primary shard to be recovered per node at once. This is all right, because we will have only a single node per Elasticsearch server. However, please remember that this operation is fast using the `local` gateway type and thus can be set to larger values if we have more than a single primary shard per node. We've also set the `cluster.routing.allocation.node_concurrent_recoveries` property to 1 to again limit the number of recoveries happening at the same time per node (we have a single shard per node so we are not hit by this at all, but again if you have more shards per node and your I/O allows that you can set it to a larger value). In addition to that we've set the `indices.recovery.concurrent_streams` property to 8, because during our initial tests of the recovery process we've seen that our network and servers can easily use eight concurrent streams while recovering a shard from the peer shard, which basically means that eight index files will be read concurrently.

Discovery

When it comes to the discovery module configuration we've only set a single property, the `discovery.zen.minimum_master_nodes` to 3. It specifies the minimum master-eligible nodes that are needed to form a cluster. It should be set to at least 50 percent of our nodes + 1, which in our case results to the value of 3. It will prevent us from facing a situation where the node from our cluster, because of some failure, will be disconnected and will form a new cluster with the same name (so called a split-brain situation). Such situations are dangerous because they can result in a data corruption, because two newly formed clusters will be indexing and changing the data at the same time.

Logging slow queries

One of the things that may be very useful while working with Elasticsearch is having your queries logged only when they are executed for a certain period of time, or longer. Keep in mind that this log is not telling about the total execution time of the query, but per shard execution, which means that only the partial execution time. In our case, we want the `INFO` level logging to log queries and real-time get requests to be logged when they are executed for longer than 500 milliseconds and one second for the real-time get requests. For debugging purposes, we have set those values to 100 milliseconds and 200 milliseconds. The following is the configuration section responsible for that part:

```
index.search.slowlog.threshold.query.info: 500ms
index.search.slowlog.threshold.query.debug: 100ms
index.search.slowlog.threshold.fetch.info: 1s
index.search.slowlog.threshold.fetch.debug: 200ms
```

Logging garbage collector work

Finally, because we start with no monitoring solutions, (at least for the start) we want to see how garbage collection is behaving. To be perfectly clear, we want to see if and when the garbage collection takes too much of time. In order to do that we've included the following lines to the `elasticsearch.yml` file:

```
monitor.jvm.gc.ParNew.info: 700ms
monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

While using the `INFO` level logging, Elasticsearch will write the information about garbage collection working for too long when the concurrent mark sweep works for five seconds or more, and for the younger generation collection working for more than 700 milliseconds. We've also added the `DEBUG` level logging for cases we want to debug and fix problems.



If you don't know what young generation garbage collection is or what concurrent mark sweep is, please refer to the Oracle Java documentation available at <http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>.

Memory setup

Till now we didn't talk about the RAM memory setup, so we will do that now. Let's assume that our nodes have 16 GB RAM each. The general recommendation is that we shouldn't set the JVM heap size to more than 50 percent of the total memory available, which is true in our case. We would set the `xms` and `xmx` Java properties to `8g` and for our case it should be enough, because our index size is not very high, and we don't have the parent-child relationships in our data because we don't facet on high cardinality fields. In the configuration shown earlier, we've set up Elasticsearch to include the garbage collector's information, however for long-term monitoring you may want to use monitoring tools such as SPM (<http://sematext.com/spm/index.html>) or Munin (<http://munin-monitoring.org/>).



We've mentioned the general rule, that is, 50 percent of the available physical memory may go to JVM and the rest should be left for the operating system. As with most of the rules, this one too is right for most of the cases. But let's imagine that our index would take about 30 GB of disk space and we would have 128 GB of RAM memory, but because of the amount of parent-child relationships and high cardinality fields facing, we end up having out-of-memory exceptions even with 64 GB of RAM dedicated to the JVM heap. Should we avoid adding more than 50 percent of the total available memory for JVM in such case? In our opinion the answer is no, but it all depends on the particular case, in the previously mentioned one the index size is far less than the free memory available after giving JVM the 64 GB out of 128 GB, so we can extend that, but we should remember to leave enough memory so that swapping is not happening in our system.

One more thing

There is one more thing we didn't talk about: the `bootstrap.mlockall` property. Setting this property to `true` allows Elasticsearch to lock the heap memory and ensure that the memory will never be swapped. While setting the `bootstrap.mlockall` property to `true` it is recommended to set the `ES_MIN_MEM` and `ES_MAX_MEM` variables to the same value and ensure that the server has enough free physical memory to start Elasticsearch and leave enough memory for the operating system to be able to work flawlessly. We will discuss more about it in the *Avoid swapping on Unix-like systems* section in *Chapter 6, Fighting with Fire*.

Changes are coming

Let's imagine now that our service is a great success. The traffic increases. And not only that, a big company wants to cooperate with us. This large supplier is not selling their own books, but only provides them to the retailers. The data we can expect from him is about 2 million books, so we will have 20 times of the data we have right now (when it comes to number of documents). We have to prepare our business for these changes, which also means altering our Elasticsearch cluster, so that our users have the same, or better, search experience. What can we do? The easier things will go first. Without additional work we can change (increase or decrease) the number of replicas. This will prepare us for more number of queries, which will of course put more pressure on Elasticsearch. The drawback is that additional replicas need more disk space. We also should make sure that the additional replicas can be allocated on the cluster nodes (refer to the formula in the *Choosing the right amount of shards and replicas* section). And please remember about performance testing: the resulting throughput always depends on many factors that cannot be described using the mathematical formulas.

What about sharding? As we said before, we cannot change the number of shards on a living index. If we over allocate shards, we have the room for expected growth. But in our case we have 2 shards, which were fine for 100,000. But it is too small to handle 2,100,000 (the ones which we have already handled and the additional ones) in a timely manner. Who could have imagined such a success? So, let's think about the solution that will allow us to handle data growth, but will also limit the time when our service is unavailable, because unavailability means loss of money.

Reindexing

The first option is to remove the old index and create a new one with a greater number of shards. This is the simplest solution but our service will be unavailable during re-indexing. Let's say that in our case, preparing documents for indexing is quite costly and the time of indexing of the whole database is long. The businessmen in our company say that stopping the whole service for the time needed for re-indexing is unacceptable. The second thought is that we can create the second index, feed it with data, and then switch the application to a new one. As an option, we can use aliasing to provide the new index without affecting the application's configuration. But there is a small issue, creating a new index requires additional disk space. Of course, we will have new machines with bigger hard drives (we have to index new "big data") but before they arrive, we should finish all the time-consuming tasks. We decide to search for another simpler solution.

Routing

Maybe routing will be handy in our case? The obvious gain from using routing is the possibility to create effective queries that return only books from our base dataset or data that belongs to our business collaborate (because routing allows us to hit only a part of our index). However, what we have to remember is to apply an appropriate filter, routing doesn't ensure that data from both sources are not in the same shard. Unfortunately, in our case this is another dead end, introducing routing again needs re-indexing. So again, we throw that solution to a trashcan next to our desk.

Multiple Indices

Let's start with the basic question, why we need only a single index and why we need to change the current part of the system. The answer is because we want to search in all the documents, to find out whether they come from our initial data source or the partner data source. Please note that Elasticsearch gives us the option of searching using multiple indices without additional penalty. We can use multiple indices in the API endpoint, for example, `/books,partner1/`. There is also a more elastic way that will allow us to add another partner fast and easily without the need of changing anything at the application side and without any downtime. We can use aliases for defining virtual indices what won't require changes in the application code.

After brainstorming, we've decided to choose the last solution with some additional improvements to put less pressure on the Elasticsearch cluster during indexing. What we did is disabling the refresh rate and removing replicas:

```
curl -XPUT localhost:9200/books/_settings -d '{
  "index" : {
    "refresh_interval" : -1,
    "number_of_replicas" : 0
  }
}'
```

Of course, after indexing we change it to its previous value (for refresh: `1s`). The one issue was that Elasticsearch disallows changing the name of the index, which causes a small downtime to service for changing the index name in configuration at the application side.

Summary

In this chapter we've learned how to choose the correct shards of replicas for our Elasticsearch deployment and we've seen how routing works when it comes to querying and indexing. We've seen how the new shard allocator works and how we can configure it to match our needs. We configured the allocation mechanism to our needs and we've learned how to choose query execution preference to specify which nodes our operations should be executed. Finally, we used the knowledge we got to configure a real-life example cluster, and we extended it when there needed.

In the next chapter we'll focus on more Elasticsearch configuration options: we will see how to configure memory usage and choose the right directory implementation. We will look at the Gateway and Discovery modules configuration and why they are very important. In addition to that we'll learn how to configure indices recovery and what information we can get about Lucene segments. Finally we will look at Elasticsearch caches.

5

ElasticSearch Administration

In the previous chapter we looked at how to choose the right amount of shards and replicas for our deployment, what is over sharding, and when we can go for it. We discussed routing in greater detail and we now know how the newly introduced shard allocator works and how we can alter its work. In addition to that, we learned how to choose which shards our queries should be executed at. Finally we've used all the knowledge to configure the fault tolerant and scalable clusters and how to extend our cluster for growing the application. By the end of this chapter you will have learned:

- How to choose the right directory implementation to allow ElasticSearch access the underlying I/O system in the most effective way
- How to configure the Discovery module to avoid potential problems
- How to configure the Gateway module to match our needs
- What the Recovery module gives us and how to alter its configuration
- How to look at the segments information
- What ElasticSearch caching looks like, what it is responsible for, how to use it, and alter its configuration

Choosing the right directory implementation – the store module

The store module is one of the modules that we usually don't pay much attention to when configuring our cluster, however it is very important. It allows us to control how data in the index is stored, by using persistent (on disk) storage or by using the transient one (in memory). Most of the store types in ElasticSearch are mapped to an appropriate Apache Lucene Directory class (http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/Directory.html). The directory is used to access all the files the index is built of, so it is crucial to properly configure it.

Store type

ElasticSearch exposes four store types that we can use. Let's see what they provide and how we can leverage their features.

The simple file system store

The simplest implementation of the directory class that is available is implemented using a random access file (Java `RandomAccessFile`: <http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html>) and maps to the `SimpleFSDirectory` (http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/SimpleFSDirectory.html) in Apache Lucene. It is sufficient for very simple applications. However, the main bottleneck will be multithreaded access, which has poor performance. In case of ElasticSearch it is usually better to use the new IO-based system store instead of the simple filesystem store. However, if you like to use this system store you should set the `index.store.type` property to `simplefs`.

The new IO filesystem store

This store type uses the directory class implementation based on the `FileChannel` (<http://docs.oracle.com/javase/7/docs/api/java/nio/channels/FileChannel.html>) from the `java.nio` package and maps to `NIOFSDirectory` in Apache Lucene (http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/NIOFSDirectory.html). The discussed implementation allows multiple threads to access the same files concurrently without performance degradation. In order to use this store one should set the `index.store.type` property to `niofs`.



Please remember that because of some bugs that exist in the JVM machine for Microsoft Windows it is very probable that the new IO filesystem store will suffer from performance problems while running on Microsoft Windows. More information about that bug can be found at: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6265734.

The MMap filesystem store

This store type uses the Apache Lucene `MMapDirectory` (http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/MMapDirectory.html) implementation. It uses the `mmap` system call (<http://en.wikipedia.org/wiki/Mmap>) for reading and randomly accessing a file for writing. It uses a portion of the available virtual memory address space in the process equal to the size of the file being mapped. It doesn't have any locking, so it is scalable when it comes to multithread access. When using `mmap` to read index files for the operating system it looks as if it is already cached (it was mapped to the virtual space). Because of that when reading a file from the Apache Lucene index, that file doesn't need to be loaded into the operating system's cache and thus the access is faster. This basically allows Lucene and thus Elasticsearch to directly access the I/O cache, which should result in faster access to the index files.

It is worth noting that the MMap filesystem store works best on 64-bit environments and should only be used on 32-bit machines when you are sure that the index is small enough and the virtual address space is sufficient. In order to use this store one should set the `index.store.type` property to `mmapfs`.

The memory store

This is the only store type that is not based on the Apache Lucene directory. The memory store allows us to store all the index files in the memory, so the files are not stored on the disk. This is crucial, because it means that the index data is not persistent: it will be removed whenever a full cluster restart happens. However if you need a small, very fast index that can have multiple shards and replicas and can be rebuilt very fast, the memory store type may be the option you are looking for. In order to use this store one should set the `index.store.type` property to `memory`.



The data stored in the memory store, similar to all the other stores is replicated among all the nodes that can hold data.

Additional properties

When using the memory store type we also have some degree of control over the caches, which are very important while using the memory store. Please remember that all the following settings are set per node:

- `cache.memory.direct`: This property defaults to `true` and specifies if the memory store should be allocated outside of the JVM's heap memory. It is usually a good idea to leave it to the default value, so that the heap is not overloaded with data.
- `cache.memory.small_buffer_size`: This property defaults to 1 KB and defines small buffer size: the internal memory structure used for holding segments information and deleted documents information.
- `cache.memory.large_buffer_size`: This property defaults to 1 MB and defines large buffer size: the internal memory structure used for holding index files other than segments information and deleted documents.
- `cache.memory.small_cache_size`: This property defines smaller cache size: the internal memory structure used for caching of index segments information and deleted documents information. It defaults to 10 MB.
- `cache.memory.large_cache_size`: This property defines a large cache size: the internal memory structure used for caching information about index other than index segments information and deleted documents information. It defaults to 500 MB.

The default store type

By default, ElasticSearch uses filesystem-based storage. However, different store types are chosen for different operating systems: however the one chosen by ElasticSearch will still be based on filesystem. For example, for 32-bit Microsoft Windows the `simplefs` type will be used, `mmapfs` will be used when ElasticSearch is running on Solaris, and Microsoft Windows 64-bit and the `niofs` will be used for the rest of the world.

 If you are looking for some information that comes from experts on how they see which directory implementation to use, please look at the <http://blog.thetaphi.de/2012/07/use-lucenes-mmapdirectory-on-64bit.html> post written by *Uwe Schindler* and <http://jprante.github.io/applications/2012/07/26/Mmap-with-Lucene.html> by *Jörg Prante*.

Usually the default store type will be the one that you want to use. However, sometimes it is worth considering using the MMap filesystem store type, especially when you have plenty of memory and your indices are big. This is because when using `mmap` to access the index file, will cause the index files to be cached only once and be reused both by Apache Lucene and the operating system.

Discovery configuration

As we already mentioned multiple times, Elasticsearch is built to work in the cluster. This is the main difference when comparing to other solutions available as open source and which have the same role as Elasticsearch. Other solutions can be used (easier or harder) in a multinode, distributed architecture: for Elasticsearch it is an everyday life. It also minimizes work needed to set up the cluster, thanks to discovery mechanisms.

The main assumption is that the cluster is automatically formed by the nodes which declare the same `cluster.name` setting. This allows us to have several independent clusters in the same network. The drawback of this automatic discovery is that sometimes someone forgets to change this setting and by accident join someone else's cluster. In such a situation, Elasticsearch may rebalance the cluster and move some data to the newly joined node. When that node is shut down, magically some data in the cluster may disappear.

Zen discovery

Zen discovery is the default mechanism responsible for discovery of Elasticsearch and available by default. The default Zen discovery configuration uses multicast to find the other nodes. This is a very convenient solution: just start new Elasticsearch node and if everything works, that node will be joined to the cluster if it has the same cluster name and is visible by the other nodes in that cluster. If not, you should check your `publish_host` or `host` settings to make sure that Elasticsearch listens to the proper network interface.

Sometimes multicast is not available for various reasons or you don't want to use it for the previously mentioned reason. In bigger clusters the multicast discovery may generate too much unnecessary traffic and this is sometimes a valid reason for not using it. For these cases Zen discovery introduces the second discovery method the unicast mode. Let's stop here for a moment and describe configuration for these modes.



If you want to know more about the differences between multicast and unicast ping methods please refer: <http://en.wikipedia.org/wiki/Multicast> and <http://en.wikipedia.org/wiki/Unicast>.

Multicast

As mentioned, this is the default mode. When the node is not a part of the cluster (for example it was just started or restarted) it will send a multicast ping request which will basically inform all the visible nodes and clusters that the node is available and is ready to be a part of the cluster.

The multicast part of the Zen discovery module exposes the following settings:

- `discovery.zen.ping.multicast.address` (default: all available interfaces): This interface is used for communication given as address or the interface name.
- `discovery.zen.ping.multicast.port` (default: 54328): This port is used for communication.
- `discovery.zen.ping.multicast.group` (default: 224.2.2.4): This represents a multicast address to send messages to.
- `discovery.zen.ping.multicast.buffer_size` (default: 2048)
- `discovery.zen.ping.multicast.ttl` (default: 3): It defines the time to live for a multicast message. Every time when packet crosses route, the TTL is decreased. This allows limiting an area where transmission can be received. Note that the routers can have assigned the threshold values as compared with TTL which ensures that the TTL value is not exactly the number of routers which packet can jump over.
- `discovery.zen.ping.multicast.enabled` (default: true): Setting this property to `false` turns off multicast. You should disable multicast if you are planning to use the unicast discovery method.

Unicast

When you switch off multicast as described previously, you can safely use the unicast part. When the node is not a part of the cluster (was just restarted, started or left the cluster because of some error) it will send a ping request to all the addresses specified in the configuration and will inform all those nodes that it is available for being a part of the cluster.

The configuration is very simple and is as follows:

- `discovery.zen.ping.unicats.hosts`: This configuration represents the initial list of nodes in the cluster. The list can be defined as a list or as an array of hosts. Every host can be given a name (or IP address), have port, or port range added. For example the value of this property can look similar to this: `["master1", "master2:8181", "master3 [80000-81000]"]`. So basically the hosts' list for the unicast discovery doesn't need to be a complete list of Elasticsearch nodes in your cluster, because once the node is connected to one of the mentioned nodes, it will be informed about all the others that are forming the cluster.
- `discovery.zen.ping.unicats.concurrent_connects` (default: 10): This configuration specifies the maximum number of concurrent connection unicast discovery will use.

Minimum master nodes

One of the very important properties when it comes to the discovery module is the `discovery.zen.minimum_master_nodes` property, which allows us to set the number of master eligible nodes that should be present in order to form the cluster. It allows us to prevent a so called split-brain situation, where because of some error (for example network issues), instead of a single cluster we have more of them with the same name. You can imagine two clusters (that should be a single one) indexing different data and problems it can cause. Because of that is it suggested to use the `discovery.zen.minimum_master_nodes` property and set it to at least 50 percent + 1 number of your nodes in the cluster. For example if you have 9 nodes in your cluster, that all are master-eligible, we should set the `discovery.zen.minimum_master_nodes` property to 5. So the minimal cluster that will be allowed to elect a master needs to have five master-eligible nodes present.

Zen discovery fault detection

ElasticSearch runs two detection processes while it is working. The first process is sending the ping requests from the master node to all the other nodes in the cluster to check if they are operational. The second process is the reverse of that: each of the nodes sends ping requests to the master in order to verify that it is still up and running and performing its duties. However, if we have a slow network or our nodes are in different hosting locations the default configuration may not be sufficient. Because of that ElasticSearch discovery module exposes the following properties, which we can change:

- `discovery.zen.fd.ping_interval`: This property defaults to 1s and specifies the interval of how often the node will send ping requests to the target node.
- `discovery.zen.fd.ping_timeout`: This property defaults to 30s and specifies how long the node will wait for the sent ping request to be responded to. If your nodes are 100 percent utilized or your network is slow you may consider increasing that property's value.
- `discovery.zen.fd.ping_retries`: This property defaults to 3 and specifies the number of ping request retries before the target node will be considered not operational. You can increase that value if your network has a high number of packets lost (or you can fix your network).

Amazon EC2 discovery

The Amazon store, in addition to selling goods has a few popular services as selling storage or computing power in the pay-as-you-go model. In this second case, called EC2, Amazon provides server instances and of course they can be used for installing and running the ElasticSearch cluster (among many other things as those are normal Linux machines). This is convenient: you pay for instances that are needed in order to handle the current traffic or speed up calculations and you shut down unnecessary instances when the traffic is lower. ElasticSearch works on EC2, but due to the nature of the environment some features may work slightly differently. One of those features that work differently is discovery, because Amazon EC2 doesn't support multicast discovery. Of course we can switch to unicast discovery: it will work but we will lose automatic node detection and in most cases we don't want to lose that functionality. However, there is an alternative: we can use the Amazon EC2 plugin, a plugin that combines the multicast and unicast discovery methods using the Amazon EC2 API.



Make sure that during set up of EC2 instances you set up communication between them (by default on port 9200 and 9300). This is crucial to have Elasticsearch communication and thus cluster functioning. Of course, this communication depends also on the `network.bind_host` and `network.publish_host` (or `network.host`) settings.

EC2 plugin's installation

The installation of a plugin is as simple as most of the plugins. In order to install it we should run the following command:

```
bin/plugin install cloud-aws
```

EC2 plugin's configuration

This plugin provides the following configuration settings that we need to provide in order for the EC2 discover to work:

- `cluster.aws.access_key`: It is the Amazon access key, one of the credential values you can find in the Amazon configuration panel
- `cluster.aws.secret_key`: It is the Amazon secret key: similar to the previously mentioned `access_key` it can be found in the EC2 configuration panel

The last thing is to inform Elasticsearch that we want to use new discovery type by setting the `discovery.type` property to the `ec2` value and turn off multicast.

Optional EC2 discovery configuration options

The previously mentioned settings are sufficient to run the EC2 discovery, but in order to control the EC2 discovery plugin's behavior Elasticsearch exposes the following additional settings:

- `cloud.aws.region`: This specifies the region for connecting with Amazon web services. You can choose a region adequate to the region where your instance resides. For example `eu-west-1` for Ireland. The possible values are: `eu-west-1`, `us-east-1`, `us-west-1`, and `ap-southeast-1`.
- `cloud.aws.ec2.endpoint`: Instead of defining a region, you can enter an address of the AWS endpoint, for example: `ec2.eu-west-1.amazonaws.com`.

- `discovery.ec2.ping_timeout` (default: 3s): This specifies the time to wait for the response for the ping message sent to the other node. After that time, the non-responsive node will be considered dead and will be removed from the cluster. Increasing this value make sense while dealing with network issues or having many EC2 nodes.

EC2 nodes scanning configuration

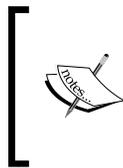
The last group of settings we want to mention allows us to configure a very important thing while building a cluster with EC2: the ability to filter the available ElasticSearch nodes in our Amazon network. The ElasticSearch EC2 plugin exposes the following properties that can help us to configure its behavior:

- `discovery.ec2.host_type` (defaults to `private_ip`): This property allows us to choose the host type which will be used to communicate with the other nodes in the cluster. The values we can use are `private_ip` (the default one, the private IP address will be used for communication), `public_ip` (the public IP address will be used for communication), `private_dns` (the private host name will be used for communication), and `public_dns` (the public host name will be used for communication).
- `discovery.ec2.tag`: This property defines a group of settings. When you launch your Amazon EC2 instances, you can define tags, which can describe the purpose of the instance, for example: customer name or environment type. Then you use these defined settings to limit the discovery nodes. Let's say you define a tag named `environment` with a `qa` value. While configuring you can now specify: `discovery.ec2.tag.environment: qa` and only the nodes running on instances with this tag will be considered for discovery.
- `discovery.ec2.groups`: This property specifies a list of security groups. Only the nodes that fall within those groups can be discovered and included in the cluster.
- `discovery.ec2.availability_zones`: This property specifies a list of available zones. Only the nodes with the specified available zones will be discovered and included in the cluster.
- `discovery.ec2.any_group` (defaults to `true`): Setting this property to `false` will force the EC2 discovery plugin to discover only those nodes that reside in an Amazon instance that fall into all of the defined security groups. The default value requires only a single group to be matched.

- There is one more property we would like to mention, the `cloud.node.auto_attributes` property. When `cloud.node.auto_attributes` is set to `true`, all the above information can be used as attributes while configuring shard placement. You can find more about shard placement in *Chapter 4, Index Distribution Architecture*, in the *Adjusting shard allocation* section.

Gateway and recovery configuration

The gateway module allows us to store all the data that is needed for Elasticsearch to work properly. That means that not only the data in the Apache Lucene indices is stored, but also all the metadata (for example index allocation settings) along with the mappings configuration for each index. Whenever the cluster's state is changed, for example when the allocation properties are changed, the cluster's state will be persisted by using the gateway module. When the cluster is started up, its state will be loaded and applied using the gateway module.



One should remember that while configuring different nodes and different gateway types, indices will use the gateway type configuration present on the given node. If an index state should not be stored using the gateway module one should explicitly set the index gateway type to `none`.

Gateway recovery process

Let's say that explicitly, the recovery process is used by Elasticsearch to load the data stored with the use of gateway module, in order for Elasticsearch to work. Whenever a full cluster's restart occurs the gateway process kicks in, in order to load all the relevant information we've mentioned: the metadata, the mappings, and of course all the indices. During shard recovery Elasticsearch copies the data between nodes, this data is of course the Lucene indices, metadata, and the transaction log that is used to recover not yet indexed documents.

Elasticsearch allows us to configure when the actual cluster data, metadata, and mappings should be recovered using the gateway module. For example, we need to wait for a certain amount of master eligible or data nodes to be present in the cluster before starting the recovery process. However, one should remember that when the cluster is not recovered all the operations performed on it will not be allowed. This is done in order to avoid modification conflicts.

Configuration properties

Before we continue with the configuration, we would like to say one more thing about ElasticSearch nodes. ElasticSearch nodes can play different roles: they can have a role of data nodes: the one that holds data, they can have a master role, and in addition to handling queries such nodes (one in a given cluster) will be responsible for managing the cluster. Of course, a node can be configured to neither be a master or a data node and in such case, the node will be only used as aggregator node that will have user queries. By default, each ElasticSearch node is data and master-eligible, but we can change that behavior. In order for the node to not be master-eligible, one should set the `node.master` property to `false` in the `elasticsearch.yml` file. In order for the node to not be data-eligible, one should set the `node.data` property to `false` in the `elasticsearch.yml` file.

In addition to that ElasticSearch allows us to use the following properties in order to control how the gateway module behaves:

- `gateway.recover_after_nodes`: This property represents an integer number that specifies how many nodes should be present in the cluster in order for the recovery to happen. For example, when set to 5 at least 5 nodes (no matter if they are data or master eligible nodes) must be present in order for the recovery process to start.
- `gateway.recover_after_data_nodes`: This property specifies an integer number that allows us to set how many data nodes should be present in the cluster in order for the recovery process to start.
- `gateway.recover_after_master_nodes`: This property specifies another gateway configuration option allowing us to set, how many master nodes should be present in the cluster in order for the recovery to start.
- `gateway.recover_after_time`: This property allows us to set the waiting time before the recovery process starts after the conditions are met.

Let's imagine that we have six nodes in our cluster from which four are data-eligible. We also have an index that is built of three shards, which are spread across the cluster. The last two nodes are master-eligible, but they are only used for querying and can't hold data. What we would like to configure is the recovery process to be delayed until all the data nodes are present in the cluster for at least 3 minutes. So our gateway configuration would look as follows:

```
gateway.recover_after_data_nodes: 4
gateway.recover_after_time: 3m
```

Expectations on nodes

In addition to already mentioned properties we can also specify the properties, which will force the recovery to happen. These are as follows:

- `gateway.expected_nodes`: This property specifies the number of expected nodes to be present in the cluster for the recovery to start immediately. If you don't need the recovery to be delayed it is advised to set this property to the number of nodes (or at least most of them) that the cluster will be formed from, because that will guarantee that the latest cluster's state is recovered.
- `gateway.expected_data_nodes`: This property specifies the number of expected data-eligible nodes to be present in the cluster for the recovery process to start immediately.
- `gateway.expected_master_nodes`: This property specifies the number of expected master-eligible nodes to be present in the cluster for the recovery process to start immediately.

Now let's get back to our previous example. We know that when all the six nodes are connected and in the cluster we want the recovery to start. So in addition to the previous configuration we would add the following property:

```
gateway.expected_nodes: 6
```

So the whole configuration would look as follows:

```
gateway.recover_after_data_nodes: 4
gateway.recover_after_time: 3m
gateway.expected_nodes: 6
```

Local gateway

With the release of Elasticsearch 0.20 (and some of the releases from 0.19 versions) all the gateway types, apart from the default, `local`, were deprecated and it is not advised to use those, because they will be removed in the future version of Elasticsearch. If you want to avoid full data reindexation you should only use the `local` gateway type and this is why we won't discuss all the other types.

The `local` gateway type uses a local storage available on a node to store the metadata, mappings, and indices. In order to use this gateway type, the local storage available on the node, there needs to be enough disk space to hold the data with no memory caching.

The persistence to the local gateway is different from the other gateways that are currently present (but deprecated). The writes to that gateway is done in a synchronous manner to ensure that no data will be lost during the write process.



In order to set the type of gateway that should be used, one should use the `gateway.type` property, which by default is set to `local`.

Backing up the local gateway

ElasticSearch up to the 0.90.5 (and including it) version doesn't support an automatic backup of the data stored with the `local` gateway type. However, sometimes it is crucial to do backups, for example when you want to upgrade your cluster to a newer version and you would like to be able to roll back if something bad happens. In order to do that, you should perform the following actions:

- Stop the indexing that is happening on your ElasticSearch cluster (this may mean stopping the rivers or any external application that is sending data to ElasticSearch).
- Flush all the not yet indexed data using the Flush API.
- Create at least one copy of each shard that is allocated in the cluster, which is the minimum to be able to get the data back in case something bad happens. However, if you like to have it as simple as it can be you can copy the whole data directory from every node in your cluster that is eligible to hold data.

Recovery configuration

We told that we can use the gateway to configure the behavior of the ElasticSearch recovery process, but in addition to that, ElasticSearch allows us to configure the recovery process itself. We did mention some of the recovery configuration options already while talking about the shard allocation in the *Altering default shard allocation behavior* section in *Chapter 4, Index Distribution Architecture*. However, we decided that it would be good to mention the properties we can use in the section dedicated to gateway and recovery.

Cluster-level recovery configuration

The recovery configuration is specified mostly at the cluster level and allows us to set general rules for the recovery module to work with. These settings are as follows:

- `indices.recovery.concurrent_streams`: This property defaults to 3 and specifies the number of concurrent streams that are allowed to be opened in order to recover a shard from its source. The higher the value of this property the more pressure will be put on the networking layer, however the recovery will be faster, depending on your network usage and throughput.
- `indices.recovery.max_bytes_per_sec`: This property, by default is set to 20 MB and specifies the maximum number of data that can be transferred during shard recovery per second. In order to disable data transfer limiting, one should set this property to 0. Similar to the number of concurrent streams this property allows us to control network usage of the recovery process. Setting this property to higher values may result in higher network utilization, but also faster recovery process.
- `indices.recovery.compress`: This property is set to `true` by default and allows us to define if Elasticsearch should compress the data that is transferred during recovery process. Setting this to `false` may lower the pressure on the CPU, but will also result in more data being transferred over the network.
- `indices.recovery.file_chunk_size`: This property specifies the chunk size used to copy the shard data from the source shard. By default, it is set to 512 KB and is compressed if the `indices.recovery.compress` property is set to `true`.
- `indices.recovery.translog_ops`: This property defaults to 1000 and specifies how many transaction log lines should be transferred between shards in a single request during the recovery process.
- `indices.recovery.translog_size`: This property specifies the chunk size used to copy shard transaction log data from the source shard. By default, it is set to 512 KB and is compressed if the `indices.recovery.compress` property is set to `true`.



In the versions prior to Elasticsearch 0.90.0, there was the `indices.recovery.max_size_per_sec` property that could be used, but it was deprecated and it is suggested to use the `indices.recovery.max_bytes_per_sec` property instead. However, if you are using Elasticsearch older than 0.90.0 it may be worth remembering.

All the mentioned settings can be updated using the cluster update API or set in the `elasticsearch.yml` file.

Index-level recovery settings

In addition to the values mentioned previously there is a single property that can be set on per index basis. The property can be set both in the `elasticsearch.yml` file and using the indices update settings API and it is called `index.recovery.initial_shards`. In general, ElasticSearch will only recover a particular shard when there is a quorum of shards present and if that quorum can be allocated. A quorum is 50 percent of the shards for the given index plus one. By using the `index.recovery.initial_shards` property we can change what ElasticSearch will take as a quorum. This property can be set to one of the following values:

- `quorum`: This value implies that 50 percent plus one shards needs to be present and be allocable.
- `quorum-1`: This value implies that 50 percent of the shards for a given index needs to be present and be allocable.
- `full`: This value implies that all of the shards for the given index needs to be present and be allocable.
- `full-1`: This value implies that 100 percent minus one shards for the given index needs to be present and be allocable.
- `integer value`: This value represents any integer, for example 1, 2 or 5 which will specify the number of shards that are needed to be present and that can be allocated. For example, setting this value to 2 will mean that at least two shards need to be present and ElasticSearch needs at least 2 shards to be allocable.

It is good to know about this property, but in most cases the default value will be sufficient for most deployments.

Segments statistics

In the *Segment merging under control* section of *Chapter 3, Low-level Index Control*, we've discussed the possibilities of adjusting Apache Lucene segments merge processes to match our needs. In addition to that, in *Chapter 6, Fighting with Fire*, in the *When it is too much for the I/O - throttling explained* section, we will further discuss configuration possibilities. However, in order to be aware what needs to be adjusted we should at least see how the segments of our index or indices look.

Introducing the segments API

In order to look deeper into Lucene segments Elasticsearch provides the segments API, which we can access by running an HTTP GET request to the `_segments` REST endpoint. For example, if we like to get the view of all the segments of the indices present in our cluster we should run the following command:

```
curl -XGET 'localhost:9200/_segments'
```

If we like to only view the segments for a mastering index, we should run the following command:

```
curl -XGET 'localhost:9200/mastering/_segments'
```

We can also view segments of multiple indices at once by running the following command:

```
curl -XGET 'localhost:9200/mastering,books/_segments'
```

The response

The response of the segments API call is always shard-oriented. That's because our index is built of one or more shards (and their replicas) and as you already know each shard is a physical Apache Lucene index. Let's assume that we already have an index called `mastering` and that it has some documents indexed in it. During that index creation, we've specified that we want it to be built of a single shard and it should not have any replicas: it's only a test index, so it should be enough.

Let's check how the index segments look by running the following command:

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

In response we would get the following JSON (where we truncated a bit):

```
{
  "ok" : true,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "indices" : {
    "mastring" : {
      "shards" : {
        "0" : [ {
          "routing" : {
            "state" : "STARTED",
            "primary" : true,
```

```
    "node" : "Cz4RFYP5RnudkXzSwe-WGw"
  },
  "num_committed_segments" : 1,
  "num_search_segments" : 8,
  "segments" : {
    "_0" : {
      "generation" : 0,
      "num_docs" : 62,
      "deleted_docs" : 0,
      "size" : "5.7kb",
      "size_in_bytes" : 5842,
      "committed" : true,
      "search" : true,
      "version" : "4.3",
      "compound" : false
    },
    ...
    "_7" : {
      "generation" : 7,
      "num_docs" : 1,
      "deleted_docs" : 0,
      "size" : "1.4kb",
      "size_in_bytes" : 1482,
      "committed" : false,
      "search" : true,
      "version" : "4.3",
      "compound" : false
    }
  }
}
}
```

As we can see, ElasticSearch returned plenty of useful information that we can analyze. The top-level information is the index name and the shard that we are looking at. In our case, we see that we have a single shard, a 0 one, that is started and working ("state" : "STARTED") which is a primary shard ("primary" : true) and which is placed on a node with the Cz4RFYP5RnudkXzSwe-WGw identifier.

The next information we can see is the number of committed segments (the `num_committed_segments` property) and the number of search segments (the `num_search_segments` property). The committed segments are the ones that have a commit command run on them, which means that they are persistent to a disk and read-only. The search segments are the ones that are searchable.

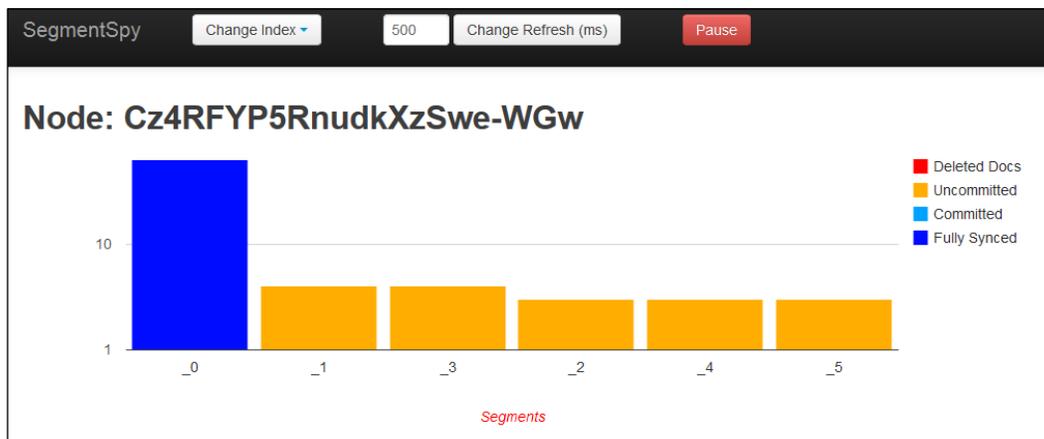
After that we will have a list of segments. Each segment is characterized by the following information:

- `number`: This property specifies the number of the segment which is the name of the JSON object grouping other segment characteristics (for example, `_0`, `_1`, and so on).
- `generation`: This property specifies index generation, a number showing us how "old" the segment is. For example, the segment with generation of 0 was created as the first one, then the one with the generation of 1 was created, and so on.
- `num_docs`: This property specifies the number of documents indexed in the segment.
- `deleted_docs`: This property specifies the number of documents that are marked as deleted and those will be removed while merging the segments.
- `size`: This property specifies the size of the segment on the disk.
- `size_in_bytes`: This property specifies the segment size in bytes.
- `committed`: This property is set to `true` if the segment is committed and set to `false`, when the segment is not yet committed.
- `search`: This property specifies the segment searchable by ElasticSearch.
- `version`: This property specifies the Lucene version used to create this index. Just a side note: although a given ElasticSearch version will always use a single version of Lucene it may happen that different segments are created by different Lucene versions. This can happen when you've upgraded ElasticSearch and it is built on a different version of Lucene. In such cases, the older segments will be rewritten while merging the segments to the newer version.
- `compound`: This property specifies the segment in the compound format (is there a single file used to hold all the segment information).

Visualizing segments information

The first thing that comes to mind when we look at all that information that is returned by the segments API is: hey let's visualize it. If you'd like to do that yourself you can always go and do it, but there is a nice plugin called SegmentSpy that is dedicated to visualize your segments (<https://github.com/polyfractal/elasticsearch-segmentspy>) and utilizes the API we just discussed.

After installing the plugin, by pointing our web browser to `http://localhost:9200/_plugin/segmentspy/` and choosing the index we are interested in we would see a screen similar to the following screenshot:



As you can see, the plugin visualizes the information provided by the segments API and can come in handy when we want to look at the segments, but we don't want to analyze the whole JSON response provided by ElasticSearch.

Understanding ElasticSearch caching

Caching is one of those things that we usually don't pay attention to while using an already configured and working ElasticSearch cluster (actually not only ElasticSearch). The caches play an important role in ElasticSearch. They allow us to effectively store filters and reuse them, use parent child functionality, use faceting, and of course sort on indexed fields effectively. In this section, we will look at the filter cache and the field data cache that are one of the most important caches and as we would think knowing how they work is very useful while tuning our ElasticSearch cluster.

The filter cache

The filter cache is the one responsible for caching the results of filters used in queries. For example let's look at the following query:

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "term" : {
          "category" : "romance"
        }
      }
    }
  }
}
```

It will return all the documents that have the `romance` term in the `category` field. As you can see, we've used the `match_all` query combined with a filter. Now, after the initial query, every query with the same filter present in it will reuse the results of our filter and save the precious I/O and CPU resources.

Filter cache types

There are two types of filter caches in Elasticsearch: index level and node-level filter cache. So, we can basically choose to configure the filter cache to be dependent on the index and on a node (which is the default behavior). Since we can't always predict where the given index will be allocated (actually its shards and replicas), it is not recommended to use the index-based filter cache because we can't predict the memory usage then.

Index-level filter cache configuration

ElasticSearch allows us to use the following properties to configure the index-level filter cache behavior:

- `index.cache.filter.type`: This property sets the type of the cache, which can take the values of `resident`, `soft` and `weak` and `node` (the default one). The entries in the `resident` cache can't be removed by JVM unless we want them to be removed (either by using API, setting the maximum size or expiration time) and is basically recommended because of this (filling up the filter cache can be expensive). The `soft` and `weak` filter cache types can be cleared by JVM when it lacks memory, with the difference that when clearing up memory, JVM will choose the weaker reference objects first and then the one using the `soft` reference. The `node` property says that the cache will be controlled on a node level (refer the *Node-level filter cache configuration* section in this chapter).
- `index.cache.filter.max_size`: This property specifies the maximum number of cache entries that can be stored in a cache (default is `-1`, which means unbounded). You need to remember that this setting is not applicable for the whole index, but for a single segment of a shard for such index, so the memory usage will differ depending on how many shards (and replicas) there are (for the given index) and how many segments the index contains. Generally the default, unbounded filter cache is fine with the `soft` type and proper queries that are paying attention to make the caches reusable.
- `index.cache.filter.expire`: This property specifies the expiration time of an entry in the filter cache, which is by default unbounded (set to `-1`). If we want our filter cache to expire if not accessed, we can set the maximum time of inactivity. For example, if we would like our cache to expire after 60 minutes we should set this property to `60m`.



If you want to read more about the `soft` and `weak` references in Java, please refer to the Java documentation, especially the Javadocs for those two types: <http://docs.oracle.com/javase/7/docs/api/java/lang/ref/SoftReference.html> and <http://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html>.

Node-level filter cache configuration

The default and recommended filter cache type, which is configured for all shards allocated to a given node (set using the `index.cache.filter.type` property to the `node` value or not setting that property at all). Elasticsearch allows us to use the `indices.cache.filter.size` property to configure the size of this cache. We can either use a percentage value as `20%` (which is the default value) or a static memory value as `1024mb`. If we use the percentage value, Elasticsearch will calculate it as a percentage of the maximum heap memory given to a node.

The node-level filter cache is an **LRU** cache type (**Least Recently Used**), which means that while removing cache entries, the ones that were used the least number of times will be thrown away in order to make place for the newer entries.

The field data cache

Field data cache is used when we want to send queries that involve faceting or sorting on the field value. What Elasticsearch does is, it loads all the values for the field we are using in memory, and by doing this Elasticsearch is able to provide fast document-based access to these values. There are two things one should remember: the field data cache is usually expensive to build from the hardware resource's point of view, because the data for the whole field needs to be loaded into memory and that requires both I/O operations and CPU resources.



One should remember that for every field that we sort on or use faceting on, the data needs to be loaded into the memory: each and every term. This can be expensive, especially for the fields that are high cardinality ones, the ones with numerous different terms in them.

Index-level field data cache configuration

Similar to index-level filter cache, we can also use the index-level field data cache, but again it is not recommended to do so because of the same reasons: it is hard to predict which shards or which indices will be allocated to which nodes. Because of that we can't predict the amount of memory that will be used for caching each index and we can run into memory issues.

However, if you know what you are doing and what you want to use, resident or soft field data cache, you can use the `index.fielddata.cache.type` property and set it to `resident` or `soft`. As we already discussed during filter cache's description, the entries in the resident cache can't be removed by JVM unless we want them to be and it is basically recommended to use this cache type when we want to use index-level field data cache. Rebuilding the field data cache is expensive and will affect the ElasticSearch query's performance. The `soft` field data cache types can be cleared by JVM when it lacks memory.

Node-level field data cache configuration

ElasticSearch Version 0.90.0 allows us to use the following properties to configure the node-level field data cache, which is the default field data cache if we don't alter the configuration:

- `indices.fielddata.cache.size`: This property specifies the maximum size of the field data cache either as a percentage value as `20%` or an absolute memory size as `10gb`. If we use the percentage value, ElasticSearch will calculate it as a percentage of the maximum heap memory given to a node. By default, the field data cache size is unbounded.
- `indices.fielddata.cache.expire`: This property specifies the expiration time of an entry in the field data cache, which is by default set to `-1`, which means that the entries in the cache won't be expired. If we want our field data cache to expire if not accessed, we can set the maximum time of inactivity. For example, if we like our cache to expire after 60 minutes we should set this property to `60m`.



If we want to be sure that ElasticSearch will use the node-level field data cache we should set the `index.fielddata.cache.type` property to the `node` value or not set that property at all.

Filtering

In addition to the previously mentioned configuration options, Elasticsearch allows us to choose which field values are loaded into the field data cache. This can be useful in some cases, especially if you remember that sorting and faceting use the field data cache to calculate the results. Elasticsearch allows us to use two types of field data loading filtering: by term frequency, by using `regex`, or by combining both of them.

One of the examples when field data filtering can be useful and when you may want to exclude the terms with the lowest frequency from the results is faceting. For example, we may need to do this, because we know that we have some terms in the index having spelling mistakes and those are lower cardinality terms for sure. We don't want to bother calculating faceting for them, so we can remove them from the data, correct them in our data source, or remove them from the field data cache by filtering. It will not only exclude them from the results returned by Elasticsearch but also make the field data memory footprint lower, because less data will be stored in the memory. Now let's look at the filtering possibilities.

Adding field data filtering information

In order to introduce the field data cache filtering information, we need to add an additional object to our mappings field definition: the `fielddata` object with its child object, `filter`. So our extended field definition, for some abstract `tag` field, would look as follows:

```
"tag" : {
  "type" : "string",
  "index" : "not_analyzed",
  "fielddata" : {
    "filter" : {
      ...
    }
  }
}
```

We will see what to put in the `filter` object in the following sections:

Filtering by term frequency

Filtering by term frequency allows us to only load the terms that have frequency higher than the specified minimum (the `min` parameter) and lower than the specified maximum (the `max` parameter). The term frequency bounded by the `min` and `max` parameters is not specified for the whole index, but per segment, which is very important, because those frequencies will differ. The `min` and `max` parameters can be specified either as a percentage (for example 1 percent is `0.01` and 50 percent is `0.5`) or as an absolute number.

In addition to that we can include the `min_segment_size` property that specifies the minimum number of documents a segment should contain in order to be taken into consideration while building the field data cache.

For example if we would like to store in the field data cache only the terms that come from segments with at least 100 documents and the terms that have a segment term frequency between 1 percent to 20 percent we should have mappings similar to the following ones:

```
{
  "book" : {
    "properties" : {
      "tag" : {
        "type" : "string",
        "index" : "not_analyzed",
        "fielddata" : {
          "filter" : {
            "frequency" : {
              "min" : 0.01,
              "max" : 0.2,
              "min_segment_size" : 100
            }
          }
        }
      }
    }
  }
}
```

Filtering by regex

In addition to filtering by the term frequency, we can also filter by the `regex` expression. In such a case, only the terms that match the specified `regex` will be loaded into the field data cache. For example, if we only like to load the data from the `tag` field that are probably Twitter tags (starting with the `#` character), we should have the following mappings:

```
{
  "book" : {
    "properties" : {
      "tag" : {
        "type" : "string",
        "index" : "not_analyzed",
        "fielddata" : {
          "filter" : {
            "regex" : "^#.*"
          }
        }
      }
    }
  }
}
```

Filtering by regex and term frequency

Of course we can combine the previously discussed filtering methods. So, if we like to have the field data cache responsible for holding the `tag` field data, only those terms that start with the `#` character, that comes from a segment with at least 100 documents and that have segment term frequency between 1 percent to 20 percent, we should have the following mappings:

```
{
  "book" : {
    "properties" : {
      "tag" : {
        "type" : "string",
        "index" : "not_analyzed",
        "fielddata" : {
          "filter" : {
            "frequency" : {
              "min" : 0.1,
              "max" : 0.2,
              "min_segment_size" : 100
            }
          },
        }
      }
    }
  }
}
```

```
    "regex" : "^#.*"  
  }  
}  
}  
}  
}
```



Please remember that the field data cache is not built during indexing, but can be rebuilt while querying and because of that we can change filtering during runtime by updating the `fielddata` section using the mappings API. However, one has to remember that after changing the field data loading filtering settings the cache should be cleared using the clear cache API described in the *Clearing the caches* section in this chapter.

The filtering example

So now let's get back to the example from the beginning of the filtering section. What we want to do is exclude the terms with the lowest frequency from faceting results. In our case the lowest ones are the ones that have the frequency lower than 50 percent. Of course, this frequency is very high, but in our example we only use four documents, in production you'll want to have different values: lower ones. In order to do that, we will create a `books` index with the following commands:

```
curl -XPOST 'localhost:9200/books' -d '{  
  "settings" : {  
    "number_of_shards" : 1,  
    "number_of_replicas" : 0  
  },  
  "mappings" : {  
    "book" : {  
      "properties" : {  
        "tag" : {  
          "type" : "string",  
          "index" : "not_analyzed",  
          "fielddata" : {  
            "filter" : {  
              "frequency" : {  
                "min" : 0.5,  

```

```

        "max" : 0.99
      }
    }
  }
}
}
}'

```

Now let's index some sample documents using the bulk API:

```

curl -s -XPOST 'localhost:9200/_bulk' --data-binary '
{ "index": { "_index": "books", "_type": "book", "_id": "1" } }
{"tag":["one"]}
{ "index": { "_index": "books", "_type": "book", "_id": "2" } }
{"tag":["one"]}
{ "index": { "_index": "books", "_type": "book", "_id": "3" } }
{"tag":["one"]}
{ "index": { "_index": "books", "_type": "book", "_id": "4" } }
{"tag":["four"]}
'

```

Now let's check a simple term's faceting by running the following query (because as we already discussed faceting uses the field data cache to operate):

```

curl -XGET 'localhost:9200/books/_search?pretty' -d ' {
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "tag" : {
      "terms" : {
        "field" : "tag"
      }
    }
  }
}'

```

The response for the preceding query would be as follows:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  .
  .
  .
  "facets" : {
    "tag" : {
      "_type" : "terms",
      "missing" : 1,
      "total" : 3,
      "other" : 0,
      "terms" : [ {
        "term" : "one",
        "count" : 3
      } ]
    }
  }
}
```

As you can see terms faceting was only calculated for the `one` term and the `four` term was omitted. If we assume that the `four` term was misspelled, then we achieved what we wanted.

Clearing the caches

As we've mentioned earlier, that while changing the field data filtering it is crucial to clear the caches after the changes are done. Also this can be useful, when you want to change some of the queries where you explicitly set the cache key. ElasticSearch allows us to clear the caches using the `_cache` rest: the endpoint, whose usage will be discussed now.

Index, indices, and all caches clearing

The simplest thing we can do is just clear all the caches by running the following command:

```
curl -XPOST 'localhost:9200/_cache/clear'
```

Of course, as we are used to, we can choose a single index or multiple indices to clear the caches for them. For example, if we like to clear the cache for the `mastering` index we should run the following command:

```
curl -XPOST 'localhost:9200/mastering/_cache/clear'
```

And if we like to clear caches for the `mastering` and `books` indices we should run the following command:

```
curl -XPOST 'localhost:9200/mastering,books/_cache/clear'
```

Clearing specific caches

In addition to the clearing caches methods mentioned previously, we can also clear only a cache of a given type. The following caches can be cleared:

- `filter`: This cache can be cleared by setting the `filter` parameter to `true`. In order to exclude this cache type from clearing one we should set this parameter to `false`.
- `field_data`: This cache can be cleared by setting the `field_data` parameter to `true`. In order to exclude this cache type from clearing one we should set this parameter to `false`.
- `bloom`: In order to clear the bloom cache (used in posting formats that use the bloom filter, which was discussed in the *Using Codecs* section in *Chapter 3, Low-level Index Control*), the `bloom` parameter should be set to `true`. In order to exclude this cache type from clearing one we should set this parameter to `false`.

For example, if we like to clear the field data cache for the `mastering` index, but leave the `filter` cache and the `bloom` cache untouched, we can run the following command:

```
curl -XPOST 'localhost:9200/mastering/_cache/clear?field_data=true&filter=false&bloom=false'
```

Clearing fields-related caches

In addition to clearing all the caches or specific ones, we are allowed to clear the cache that is only specific to given fields. In order to do that we need to add the `fields` parameter to our request, with the value of all the fields we want to clear the caches for separated by the comma character. For example, if we would like to clear the caches for the `title` and `price` fields of the `mastering` index we should run the following request:

```
curl -XPOST 'localhost:9200/mastering/_cache/clear?fields=title,price'
```

Summary

In this chapter we've learned how to choose the right directory implementation to allow ElasticSearch to access the underlying I/O system in the most effective way. We've seen how to configure the nodes discovery module both using the multicast and unicast methods. We discussed the gateway module which allows us to control when the cluster recovery kicks in and of course we've looked at the recovery module and its configuration. In addition to that, we've learned how to analyze segment-level information returned by ElasticSearch. Finally we've seen how ElasticSearch caching works, how to alter it, and control the way the field data cache is built.

In the next chapter we'll fight with fire: we'll learn how to deal with troublesome situations. We will start with discussing how garbage collection works in Java, how to monitor garbage collector's work, and how to look at the information provided by JVM. In addition to that we'll discuss throttling, which allows us to control how much ElasticSearch and the underlying Apache Lucene library will be allowed to stress out the I/O subsystem. We'll see what warmers can give us in terms of query's performance and we'll learn how to use them. Finally we'll learn how to use the hot threads API provided by ElasticSearch and how the ElasticSearch API can be used to provide useful information and statistics while diagnosing and fighting problems.

6

Fighting with Fire

In the previous chapter we looked at ElasticSearch administration properties a bit deeper. We learned how to choose the right Lucene Directory implementation and which of the available implementations is the right choice in our environment. We also learned how ElasticSearch Discovery module works, what multicast and unicast discovery is all about, and how to use Amazon EC2 discovery. In addition to that we now know what the Gateway module is and how to configure its recovery behavior. We've used some additional ElasticSearch API that we think may be of use and we've seen how to check what segments our indices are built of. We've also seen how to visualize that information. Finally, we've learned what the cache types in ElasticSearch are, what they are used for, how to configure them, and how to clear them with the use of ElasticSearch API. By the end of this chapter, you will have learned:

- What is garbage collector, how it works, and how to diagnose problems that it can cause
- How we can control the amount of I/O operations used by ElasticSearch
- How can warmers speed up our queries and an example of such case
- What are hot threads and how to get the list of them
- Which ElasticSearch API can be useful when diagnosing problems with nodes and cluster

Knowing the garbage collector

You know that Elasticsearch is a Java application and because of that it runs in the Java Virtual Machine. Each Java application is compiled into a so called byte code, which can be executed by the JVM. In the most general way of thinking, you can imagine that the JVM is just executing other programs and controls their behavior. But this is not what you will care about unless you are developing plugins for Elasticsearch, which we will discuss in *Chapter 9, Developing Elasticsearch Plugins*. What you will care about is the garbage collector – the piece of JVM that is responsible for memory management. When objects are dereferenced, they can be removed from memory by the garbage collector, when the memory is running low, the garbage collector starts working, and so on. In this section, we will see how to configure garbage collector, how to avoid memory swapping, how to log garbage collector behavior, diagnose problems, and use some Java tools that will show us how it all works.



You can learn more about the architecture of Java Virtual Machine at many places on the World Wide Web, for example, on Wikipedia: http://en.wikipedia.org/wiki/Java_virtual_machine

Java memory

When we specify the amount of memory using the `Xms` and `Xmx` parameters (or the `ES_MIN_MEM` and `ES_MAX_MEM` properties), we specify the minimum and maximum size of Java Virtual Machine heap space. It is basically a reserved space of physical memory that can be used by the Java program, which in our case is Elasticsearch. A Java process will never use more heap memory than we've specified with the `Xmx` parameter (or the `ES_MAX_MEM` property). When a new object is created in Java application, it is placed in the heap memory. After it is no longer used, garbage collector will try to remove that object from heap to free the memory space and for JVM to be able to reuse it in the future. You can imagine, that if you don't have enough heap memory for your application to create new objects on heap, then bad things will happen, such as JVM will throw the `OutOfMemory` exception, which is a sign that something is wrong with the memory, that is, either we don't have enough of it or we have some memory leak and we don't release the object that we don't use.

The JVM memory is divided into the following regions:

- **Eden space:** It is the part of the heap memory where the JVM initially allocates most of the object types.
- **Survivor space:** It is the part of the heap memory that stores objects which survived the garbage collection of the eden space heap part. The survivor space is divided into survivor space 0 and survivor space 1.

- **Tenured generation:** It is the part of the heap memory that holds objects that were living for some time in the survivor space heap part.
- **Permanent generation:** It is the non-heap memory that stores all the data for the virtual machine itself, such as the classes and methods for objects.
- **Code cache:** It is the non-heap memory that is present in the HotSpot JVM that is used for compilation and storage of native code.

The previous classification can be simplified. The eden space and the survivor space are called the young generation heap space, and the tenured generation is often called old generation.

The life cycle of Java object and garbage collections

In order to see how garbage collector works, let's get through the lifecycle of a sample Java object.

When a new object is created in a Java application, it is placed in the young generation heap space, inside the eden space part. Then when the next young generation garbage collection is run and the object survives that collection (so basically if it was not one time use object and the application still needs it), it will be moved to the survivor part of the young generation heap space (first to survivor 0 and then, after another young generation garbage collection, to survivor 1).

After living sometime in the survivor 1 space, the object is moved to tenured generation heap space, so it will now be a part of the old generation. From now on the young generation garbage collector won't be able to move that object in heap space. Now that object will be living in the old generation until our application decides that it is not needed anymore. In such case, when the next full garbage collection comes in, it will be removed from the heap space and will make place for new objects.

Based on the previous paragraph, we can say (and it is actually true) that, at least till now, Java uses generational garbage collection; the more garbage collections our object will survive the further it gets promoted. Because of that we can say that there are two types of garbage collectors working side-by-side: the young generation garbage collector (also called minor) and the old generation garbage collector (also called major).

Dealing with garbage collection problems

When dealing with garbage collection problems, the first thing you need to identify is the source of the problem. It is not straightforward work and usually requires some effort from the system administrator or the people responsible for handling the cluster. In this section, we will show two methods of observing and identifying problems with garbage collector: first is turning on logging for garbage collector in Elasticsearch and the second is using the `jstat` command, which is present in most Java distributions.

Turning on logging of garbage collection work

ElasticSearch allows us to observe periods when garbage collector is working too long. In the default `elasticsearch.yml` configuration file, you can see the following entries, which are commented out by default:

```
monitor.jvm.gc.ParNew.warn: 1000ms
monitor.jvm.gc.ParNew.info: 700ms
monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.warn: 10s
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

As you can see the configuration specifies three log levels and the thresholds for each of them. For example, for the `info` logging level, if the young generation collection will take 700 milliseconds or more, Elasticsearch will write information to logs. In case of old generation, it will be written to logs if it will take more than 5 seconds.

What you'll see in the logs is something like this:

```
[EsTestNode] [gc] [ConcurrentMarkSweep] [964] [1] duration [14.8s],
collections [1]/[15.8s], total [14.8s]/[14.8s], memory [8.6gb] -
>[3.4gb]/[11.9gb], all_pools {[Code Cache] [8.3mb] -
>[8.3mb]/[48mb]} {[Par Eden Space] [13.3mb]
>[3.2mb]/[266.2mb]} {[Par Survivor Space] [29.5mb]
>[0b]/[33.2mb]} {[CMS Old Gen] [8.5gb]->[3.4gb]/[11.6gb]} {[CMS
Perm Gen] [44.3mb] ->[44.2mb]/[82mb]}
```

As you can see, the previous line from the logfile says that it is about the `ConcurrentMarkSweep` garbage collector, so it's about old generation collection. We can see that the total collection time took 14.8 seconds. Before the garbage collection operation, there was 8.6gb of heap memory used (out of 11.9gb). After garbage collection work, the amount of heap memory used was reduced to 3.4gb. After that you can see the information in more detailed statistics about which parts of the heap were taken into consideration by the garbage collector: code cache, eden space, survivor space, old generation heap space, and the perm generation space.

When turning on logging of garbage collector work at a certain threshold, we can see when things are not running the way we would like by just looking at the logs. However, if you would like to see more, Java comes with a tool for that, the `jstat` command.

Using JStat

Running the `jstat` command to look at how our garbage collector works is as simple as running the following command:

```
jstat -gcutil 123456 2000 1000
```

The `-gcutil` switch tells the command to monitor garbage collector work, the `123456` is the virtual machine identifier on which ElasticSearch is running, `2000` is the interval in milliseconds between samples, and `1000` is the number of samples to be taken. So in our case, the previous command will be running for a little more than 33 minutes ($2000 * 1000 / 1000 / 60$).

In most cases, the virtual machine identifier will be similar to your process ID or even the same, but not always. In order to check what Java processes are running and what their virtual machines identifiers are, one can just run a `jps` command which is provided with most JDK distributions. A sample command would be like this:

```
jps
```

And the result would be as follows:

```
16232 Jps  
11684 ElasticSearch
```

In the result of the `jps` command, we see that each line contains of JVM identifier followed by the process name. If you want to learn more about the `jps` command, please refer to Java documentation: <http://docs.oracle.com/javase/7/docs/technotes/tools/share/jps.html>



Please remember to run the `jstat` command from the same account ElasticSearch is running or if that is not possible, run the `jstat` command with administrator privileges (for example, using `sudo` command on Linux systems). It is crucial to have access rights to the process running ElasticSearch, or the `jstat` command won't be able to connect to that process.

Now, let's look at a sample output of the `jstat` command:

| S0 | S1 | E | O | P | YGC | YGCT | FGC | FGCT | GCT |
|-------|------|-------|------|-------|-----|-------|-----|-------|-------|
| 12.44 | 0.00 | 27.20 | 9.49 | 96.70 | 78 | 0.176 | 5 | 0.495 | 0.672 |
| 12.44 | 0.00 | 62.16 | 9.49 | 96.70 | 78 | 0.176 | 5 | 0.495 | 0.672 |
| 12.44 | 0.00 | 83.97 | 9.49 | 96.70 | 78 | 0.176 | 5 | 0.495 | 0.672 |
| 0.00 | 7.74 | 0.00 | 9.51 | 96.70 | 79 | 0.177 | 5 | 0.495 | 0.673 |
| 0.00 | 7.74 | 23.37 | 9.51 | 96.70 | 79 | 0.177 | 5 | 0.495 | 0.673 |
| 0.00 | 7.74 | 43.82 | 9.51 | 96.70 | 79 | 0.177 | 5 | 0.495 | 0.673 |
| 0.00 | 7.74 | 58.11 | 9.51 | 96.71 | 79 | 0.177 | 5 | 0.495 | 0.673 |

The previous example comes from the Java documentation and we decided to take it because it nicely shows what `jstat` is all about. Let's start by describing what each of the columns means:

- S0: It indicates survivor space 0 utilization as a percentage of space capacity
- S1: It indicates survivor space 1 utilization as a percentage of space capacity
- E: It indicates eden space utilization as a percentage of space capacity
- O : It indicates old space utilization as a percentage of space capacity
- YGC: It indicates the number of young garbage collection events
- YGCT: It indicates time of young garbage collections in seconds
- FGC: It indicates the number of full garbage collections
- FGCT: It indicates time of full garbage collections in seconds
- GCT: It indicates total garbage collection time in seconds

And now, let's get back to our example. As you can see, there was a young garbage collection event after sample three and before sample four. We can see that the collection took 0.001 of a second (0.177 YGCT in fourth sample minus 0.176 YGCT in third sample). We also know that the collection promoted objects from eden space (which is 0% in fourth sample and was 83.97% in third sample) to old generation heap space (which was increased from 9.49% in third sample to 9.51% in fourth sample). And this example shows you how you can analyze the output of `jstat`. Of course, it can be time consuming and requires some knowledge about how garbage collector works and what is stored in the heap. However, sometimes it is the only way to see the reason ElasticSearch can get stuck at certain moments.

Please remember that if you ever see that Elasticsearch is not working correctly, the `S0`, `S1`, or `E` columns are at 100 percent, and the garbage collector working is not able to handle those heap spaces, than either your young is too small and you should increase it (of course if you have sufficient physical memory available) or you run into some memory problems, such as memory leaks, when some resources are not releasing unused memory. On the other hand, when your old generation space is at 100 percent and garbage collector is struggling with releasing it (frequent garbage collections), but it can't, then it probably means that you just don't have enough heap space for your Elasticsearch node to operate properly. In such case, what you can do without changing your index architecture is to increase the heap space that is available for JVM that is running Elasticsearch (for more information about JVM parameters please refer to <http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html>).

Creating memory dumps

One additional thing that we didn't mention till now is the ability to dump the heap memory to a file. Java allows us to get a snapshot of memory for the given point in time and we can use that snapshot to analyze what is stored in memory and find problems. In order to dump Java process memory, one can use the `jmap` (<http://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html>) command, for example, like this:

```
jmap -dump:file=heap.dump 123456
```

The `123456`, in our case, is the identifier of the Java process we want to get the memory dump for and `-dump:file=heap.dump` specifies that we want the dump to be stored in the file named `heap.dump`. Such dump can be further analyzed in specialized software, such as `jhat` (<http://docs.oracle.com/javase/7/docs/technotes/tools/share/jhat.html>) but usage of such programs is beyond the scope of this book.

More information on garbage collector work

Tuning garbage collection is not a simple process. The default options set for us in Elasticsearch deployment are usually sufficient for most cases and the only thing you'll need to do is adjust the amount of memory for your nodes. The topic of tuning the garbage collector work is beyond the scope of this book, it is very broad and called black magic by some developers. However, if you would like to read more about garbage collector, what the options are, and how they affect your application, I can suggest a great article that can be found at <http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html>. Although the article at the link is concentrated on Java 6, most of the options, if not all, can be successfully used with deployments running on Java 7.



There is one thing to remember, what you usually try to aim for is more and smaller garbage collections rather than one, but longer. This is because you want your application to be running at the same constant performance and the garbage collector work to be transparent for Elasticsearch. When a big garbage collection happens, it can stop the world garbage collection event, where Elasticsearch will be frozen for a short period of time, which will make your queries very slow and will stop your indexing process for some time.

Adjusting garbage collector work in Elasticsearch

We now know how garbage collector works and how to diagnose problems with it, so it would be also nice to know how we can change Elasticsearch startup parameters to change how garbage collector works. It depends on how you run Elasticsearch. We will look at the two most common ones: the standard startup script provided with Elasticsearch distribution package and when using service wrapper.

Using standard startup script

When using a standard startup script, in order to add additional JVM parameters, we should include them in the `JAVA_OPTS` environment property. For example, if we wanted to include `-XX:+UseParNewGC -XX:+UseConcMarkSweepGC` to our Elasticsearch startup parameters in Linux like systems, we would do the following:

```
export JAVA_OPTS="-XX:+UseParNewGC -XX:+UseConcMarkSweepGC"
```

In order to check if the property was properly considered, we can just run another command:

```
echo $JAVA_OPTS
```

And the previous command should result in the following output in our case:

```
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC
```

Service wrapper

ElasticSearch allows user to install service wrapper as a service using Java service wrapper (<https://github.com/elasticsearch/elasticsearch-servicewrapper>). If you are using the service wrapper, setting up the JVM parameters is different when compared to the method shown previously.

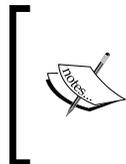
What we need to do is modify the `elasticsearch.conf` file which will probably be located in `/opt/elasticsearch/bin/service/` (if your ElasticSearch was installed in `/opt/elasticsearch`). In the mentioned file, you will see the following properties:

```
set.default.ES_HEAP_SIZE=1024

wrapper.java.additional.1=-Delasticsearch-service
wrapper.java.additional.2=-Des.path.home=%ES_HOME%
wrapper.java.additional.3=-Xss256k
wrapper.java.additional.4=-XX:+UseParNewGC
wrapper.java.additional.5=-XX:+UseConcMarkSweepGC
wrapper.java.additional.6=-XX:CMSInitiatingOccupancyFraction=75
wrapper.java.additional.7=-XX:+UseCMSInitiatingOccupancyOnly
wrapper.java.additional.8=-XX:+HeapDumpOnOutOfMemoryError
wrapper.java.additional.9=-Djava.awt.headless=true
```

The first property is responsible for setting the heap memory size for ElasticSearch, while the rest are additional JVM parameters. If you would like to add another parameter, you can just add another `wrapper.java.additional`, followed by dot and the next available number, for example:

```
wrapper.java.additional.10=-server
```



One thing to remember is that tuning garbage collector work is not something that you do once and forget. It requires experimenting as its work is very dependent on your data, queries, and all that combined. Don't be afraid to make changes when something is wrong, but also observe them and look how ElasticSearch works after making changes.

Avoiding swapping on Unix-like systems

Although this is not strictly about garbage collection and heap memory usage, we think that it is crucial to see how to disable swap. Swapping is a process of writing memory pages to disk (swap partition in Unix based systems), when the amount of physical memory is not sufficient or the operating system decides that for some reason it is better to have some part of RAM memory written into disk. If the swapped memory pages will be again needed, operating system will load them from swap partition and allow processes to use them. As you can imagine, such a process takes time and resources.

When using Elasticsearch, we have to avoid its process memory from being swapped. You can imagine that having the parts of memory used by Elasticsearch written to disk and then again reading from it can hurt the performance of both searching and indexing. Because of that Elasticsearch allows us to turn off swapping for it. In order to do that one should set `bootstrap.mlockall` to `true` in the `elasticsearch.yml` file.

But the previous setting is only a beginning. You also need to ensure that the JVM resizes the heap by setting the `Xmx` and `Xms` parameters to the same values (you can do that by specifying the same values for the `ES_MIN_MEM` and `ES_MAX_MEM` environment variables for Elasticsearch). Please also remember that you need to have enough physical memory to handle the settings you've set.

Now, if we run Elasticsearch, we can run into the following message in the logs:

```
[2013-06-11 19:19:00,858] [WARN ] [common.jna           ]
Unknown mlockall error 0
```

This means that our memory locking is not working. So now, let's modify two files on our Linux operating system (this will require administration rights). We assume that the user that will run Elasticsearch is `elasticsearch`.

First, we modify `/etc/security/limits.conf` and we add the following entries:

```
elasticsearch - nofile 64000
elasticsearch - memlock unlimited
```

The second thing is modifying the `/etc/pam.d/common-session` file and adding the following:

```
session required pam_limits.so
```

After re-logging to the `es` user account, you should be able to start Elasticsearch and not see the `mlockall error` message.

When it is too much for I/O – throttling explained

In the *Choosing the right directory implementation - the store module* section in *Chapter 5, Elasticsearch Administration*, we've talked about the store type, which means we are now able to configure the store module to match our needs. However, we didn't write everything about the store module – we didn't write about throttling.

Controlling I/O throttling

As you remember from the *Segment merging under control* section in *Chapter 3, Low-level Index Control*, Apache Lucene stores the data in the immutable segments files that can be read many times, but written only once. The merge process is asynchronous and in general, should not interfere with indexing and searching, from a Lucene point of view. However, problems may occur because merging is expensive when it comes to I/O; it requires reading the segments that are going to be merged and writing new ones. If searching and indexing happen concurrently, it can be too much for the I/O subsystem, especially on systems with low I/O. And this is where the throttling kicks in; we can control how much I/O Elasticsearch will use.

Configuration

Throttling can be configured both on a node level and on the index level, so you can either configure how many resources a node will use or how many resources will be used for the index.

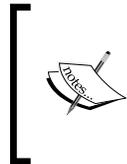
Throttling type

In order to configure throttling type on node level, one should use the `indices.store.throttle.type` property, which can take the value of `none`, `merge`, and `all`. The `none` value will tell Elasticsearch that no limiting should take place and is the default value. The `merge` value tells Elasticsearch that we want to limit I/O usage for merging the nodes, and the `all` value specifies that we want to limit all the store module based operations.

In order to configure throttling type on the index level, one should use the `index.store.throttle.type` property which can take the same values as the `indices.store.throttle.type` property with an additional one, the `node`. The `node` value will tell Elasticsearch that instead of using per index throttling limiting, we will use the node level configuration. This is the default value.

Maximum throughput per second

In both cases, when using index or node level throttling, we are able to set the maximum bytes per second that I/O can use. For the value of this property, we can use 10mb, 500mb, or anything that we need. For the index level configuration, we should use the `index.store.throttle.max_bytes_per_sec` property and for the node level configuration, we should use the `indices.store.throttle.max_bytes_per_sec` property.



The previous mentioned properties can be set both in `elasticsearch.yml` file and also can be updated dynamically using the cluster update settings for the node level configuration and using the index update settings for index level configuration.

Node throttling defaults

On the node level, since Elasticsearch 0.90.1, throttling is enabled by default. The `indices.store.throttle.type` property is set to `merge` and the `indices.store.throttle.max_bytes_per_sec` property is set to 20mb. Elasticsearch versions before 0.90.1 don't have throttling enabled by default.

Configuration example

Now, let's imagine that we have a cluster that consists of four Elasticsearch nodes and we want to configure throttling for the whole cluster. By default, we want the merge operation not to process more than 50 megabytes per second for a node. We know that we can handle such operations without affecting search performance and this is what we are aiming at. In order to achieve this, we would run the following request:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent" : {
    "indices.store.throttle.type" : "merge",
    "indices.store.throttle.max_bytes_per_sec" : "50mb"
  }
}'
```

In addition to that we have a single index called `payments`, that is very rarely used and we've placed it on the smallest machine in the cluster. This index doesn't have replicas and is built of a single shard. What we would like to do for that index is limit the merges to process a maximum of 10 megabytes per second. So, in addition to the previous command, we would run one like this:

```
curl -XPUT 'localhost:9200/payments/_settings' -d '{
  "index.store.throttle.type" : "merge",
  "index.store.throttle.max_bytes_per_sec" : "10mb"
}'
```

However, this command alone won't work, because currently Elasticsearch doesn't refresh the throttle settings for the index. So, we will close and open our `payments` index to force refresh by running the following commands:

```
curl -XPOST 'localhost:9200/payments/_close'
curl -XPOST 'localhost:9200/payments/_open'
```

After running the previous commands, we can check our index settings by running the following command:

```
curl -XGET 'localhost:9200/payments/_settings?pretty'
```

In response, we should get the following JSON code:

```
{
  "payments" : {
    "settings" : {
      "index.number_of_shards" : "5",
      "index.number_of_replicas" : "1",
      "index.version.created" : "900099",
      "index.store.throttle.type" : "merge",
      "index.store.throttle.max_bytes_per_sec" : "10mb"
    }
  }
}
```

As you can see, after updating the index setting, closing the index, and opening it again, we've finally got our settings working.

Speeding up queries using warmers

If you worked with Elasticsearch, and we assume you did, it is very probable that you've heard of or used warmers API. This is a functionality that allows us to add queries that will be used to warm up index segments. And this is exactly what warmer is (a query or queries that are registered in Elasticsearch and which are used to prepare the index for searching). In this section, we will recall how we can add warmers, how we can manage them, and what they can be used for.

Reason for using warmers

One of the questions you may be asking yourself is if the warmers are really that useful. The answer to this question actually depends on your data and your queries, but in general, they are useful. As we've mentioned earlier (for example, during cache discussion in the *Understanding Elasticsearch caching* section in *Chapter 5, Elasticsearch Administration*), Elasticsearch needs to preload some data into the caches in order to be able to use certain features, such as parent-child relationships, faceting, or field-based sorting. The preload can take time and resources, which will make your queries slower for some time. What's more is that if your index is changing rapidly, the caches will need to be refreshed often and the query performance will suffer even more.

That's why, Elasticsearch 0.20 introduced the warmers API. Warmers are standard queries that are run against cold segments (not yet used) before Elasticsearch will allow searching on them. This is also done not only during start up, but also whenever a new segment is committed. Because of that, with proper warming queries, we can preload all the needed data into caches and also warm up operating system I/O cache (by reading the cold segment). By doing this, when the segment is finally exposed to queries, we can be sure that we will get the optimal search performance and all the needed data will be ready.

At the end of the warmers section, we will show you a simple example of how warmers can improve initial query performance and notice the difference yourself.

Manipulating warmers

ElasticSearch allows us to create warmers, retrieve them, and of course, delete them. Each warmer is associated with a given index or index and type. We can include warmers with the index creation request, include them in our templates, or use the PUT Warmers API to create them. Finally, we can completely disable warmers without the need of deleting them. So if we don't want them to be running only for a certain amount of time, we can do that easily.

Using the PUT Warmer API

The simplest way to add warmers to your index, or index and type is to use the PUT Warmer API. In order to do that we need to send a PUT HTTP request to the `_warmer` REST end point with the query in the request body. For example, if we would like to add a simple `match_all` query with some terms faceting as a warmer for an index, named `mastering`, and type, named `doc`, we could use the following command:

```
curl -XPUT 'localhost:9200/mastering/doc/_warmer/testWarmer' -d '{
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "nameFacet" : {
      "terms" : {
        "field" : "name"
      }
    }
  }
}'
```

As you can see, each warmer has its own name which should be unique (in the previous case, it is `testWarmer`) and which we can use to retrieve or delete it.

If we want to add the same warmer, but for the whole `mastering` index, we would have to omit the type name and the command would look like this:

```
curl -XPUT 'localhost:9200/mastering/_warmer/testWarmer' -d '{
  ...
}'
```

Adding warmers during index creation

In addition to using Put Warmer API, we can define warmers during index creation. In order to do that we need to add additional warmers section on the same level as mappings. For example, if we would like to create the `mastering` index with the `doc` document type and add the same warmer as we used in the Put Warmer API, we would send the following request:

```
curl -XPUT 'localhost:9200/mastering' -d '{
  "warmers" : {
    "testWarmer" : {
      "types" : ["doc"],
      "source" : {
        "query" : {
          "match_all" : {}
        },
        "facets" : {
          "nameFacet" : {
            "terms" : {
              "field" : "name"
            }
          }
        }
      }
    }
  },
  "mappings" : {
    "doc" : {
      "properties" : {
        "name" : { "type": "string", "store": "yes", "index":
          "analyzed" }
      }
    }
  }
}'
```

As you can see, in addition to the `mappings` section, we've included a `warmers` one, which is used to provide warmers for the index we are creating. Each warmer is identified by its name (`testWarmer` in this case) and has two properties: `types` and `source`. The `types` property is an array of document types in the index, which the warmer should be used to. If we want the warmer to be used for all document types, we should leave that array empty. The `source` property should contain our query source. We can have multiple warmers included in a single index create request.

Adding warmers to templates

ElasticSearch also allows us to include warmers for templates in the very same manner as we would add those when creating an index. For example, if we would like to include a warmer for a sample template, we would run the following command:

```
curl -XPUT 'localhost:9200/_template/templateone' -d '{
  "warmers" : {
    "testWarmer" : {
      "types" : ["doc"],
      "source" : {
        "query" : {
          "match_all" : {}
        },
        "facets" : {
          "nameFacet" : {
            "terms" : {
              "field" : "name"
            }
          }
        }
      }
    }
  },
  "template" : "test*"
}'
```

Retrieving warmers

All the defined warmers can be retrieved using the GET HTTP method and sending a request to the `_warmer` REST endpoint, which should be followed by a name. We can retrieve a single warmer by using its name, for example, like this:

```
curl -XGET 'localhost:9200/mastering/_warmer/warmerOne'
```

Or we can use a wildcard character to retrieve all warmers, whose names start with the given phrase. For example, if we would like to get all warmers that start with the `w` character, we could use the following command:

```
curl -XGET 'localhost:9200/mastering/_warmer/w*'
```

Finally, we can get all warmers for the given index with the following command:

```
curl -XGET 'localhost:9200/mastering/_warmer/'
```

Of course, we can also include the document type in all the previous commands to operate not on the warmers for the whole index, but only for the ones for the desired type.

Deleting warmers

Similar to the ways Elasticsearch allows us to retrieve the warmers, we can delete them by using the DELETE HTTP method and the `_warmer` REST endpoint. For example, if we want to remove a warmer named `warmerOne` from the `mastering` index, we would run the following command:

```
curl -XDELETE 'localhost:9200/mastering/_warmer/warmerOne'
```

We can also delete all the warmers with the name starting with the given phrase. For example, if we want to delete all the warmers that start with the `w` letter and belong to the `mastering` index, we would run the following command:

```
curl -XDELETE 'localhost:9200/mastering/_warmer/w*'
```

And we can also delete all warmers for the given index. We do that by sending the DELETE HTTP method to the `_warmer` REST endpoint without providing the warmer name. For example, deleting all the warmers for the `mastering` index could be done with the following command:

```
curl -XDELETE 'localhost:9200/mastering/_warmer/'
```

And of course, we can also include the type in all the previous commands to operate not on the warmers for the whole index, but only for the ones for the desired type just like we did while retrieving warmers.

Disabling warmers

If you don't want to use your warmers, but you don't want to delete them, you can use the `index.warmer.enabled` property and set it to `false`. You can set it in the `elasticsearch.yml` file or by using the update setting API, for example, like this:

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.warmer.enabled": false
}'
```

If you would like to use the warmers again, the only thing you need to do is change the `index.warmer.enabled` property and set it to `true`.

Testing the warmers

In order to test the warmers, let's run a simple test. I've created a simple index with the following command:

```
curl -XPUT localhost:9200/docs -d '{
  "mappings" : {
    "doc" : {
      "properties" : {
        "name": { "type": "string", "store": "yes", "index":
          "analyzed" }
      }
    }
  }
}'
```

In addition to that I've created a second type, called `child`, which were acting as child documents of the previously created `doc` type documents. To do that I've used the following command:

```
curl -XPUT 'localhost:9200/docs/child/_mapping' -d '{
  "child" : {
    "_parent":{
      "type" : "doc"
    },
    "properties" : {
      "name": { "type": "string", "store": "yes", "index":
        "analyzed" }
    }
  }
}'
```

After that I've indexed a single document with the `doc` type and about 80,000 documents of type `child`, which were pointing to that document using the `parent` request parameter.

Querying without warmers present

After the indexation process ended restart ElasticSearch and run the following query:

```
{
  "query" : {
    "has_child" : {
      "type" : "child",
      "query" : {
        "term" : {
          "name" : "document"
        }
      }
    }
  }
}
```

As you can see, it is a simple query that returns the parent document with a given term in at least one of the child's documents. The response returned by ElasticSearch was as follows:

```
{
  "took" : 479,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "docs",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"name":"Test 1234"}
    } ]
  }
}
```

The execution time was 479 milliseconds. Sounds pretty high, right? If we would run the same query once again, the execution time would drop.

Querying with warmer present

In order to improve the initial query performance, we need to introduce a simple warmer that will not only warm the I/O cache, but also force Elasticsearch to load the parent documents identifiers into memory to allow faster parent child queries. As we know, Elasticsearch does that during the first query with the given relationship. So with that information, we can use the following command to add our warmer:

```
curl -XPUT 'localhost:9200/docs/_warmer/sampleWarmer' -d '{
  "query" : {
    "has_child" : {
      "type" : "child",
      "query" : {
        "match_all" : {}
      }
    }
  }
}'
```

Now, if we restart Elasticsearch and run the same query as we did in the test without the warmer, we will get more or less the following result:

```
{
  "took" : 38,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "docs",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"name":"Test 1234"}
    } ]
  }
}
```

Now we can see how warmer improved the execution time of the query right after ElasticSearch was restarted. The query without the warmers took almost half of a second, while the query executed with the warmers present was executed faster than 40 milliseconds.

Of course, the performance gain is not only from the fact that the ElasticSearch was able to load the document identifiers to memory, but also because operating system was able to cache index segments. Nevertheless, the performance gain is significant and if you use queries that can be warmed up (for example, use heavy filtering, parent-child relationships, faceting, and so on), it's a good way to go.

Very hot threads

When you are in trouble and your cluster works slower than usual and uses large amounts of CPU power, you know you need to do something to make it work again. This is the case when the Hot Threads API can give you necessary information to find the root of problems. A hot thread is a Java thread that uses high CPU volume and executes for longer period of time. Hot Threads API returns information about which part of ElasticSearch code are hot spots from the CPU side or where ElasticSearch is stuck for some reason.

When using Hot Threads API, you can examine all nodes, a selected few of them, or particular node using the `/_nodes/hot_threads` or `/_nodes/{node or nodes}/hot_threads` endpoints. For example, to look at hot threads on all the nodes, we would run the following command:

```
curl localhost:9200/_nodes/hot_threads
```

The API supports the following parameters:

- `threads` (default: 3): It is the number of threads that should be analyzed. The ElasticSearch will take the specified number of the most "hot" threads by looking at the information determined by the `type` parameter.
- `interval` (default: 500ms): ElasticSearch checks threads two times to calculate the percentage of time spent in particular thread on operation defined by the `type` parameter. The time between these checks is defined by the `interval` parameter.

- `type` (default: `cpu`): It is the type of thread state to examine. The API can check CPU time taken by given thread (`cpu`), time in `BLOCKED` state (`block`), or at `WAITING` (`wait`) state. If you would like to know more about the thread states, see: <http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.State.html>
- `snapshots` (default: 10): It is the number of stack trace (nested sequence of method calls at a certain point of time) snapshots to take.

It is time to move on to the example of how to use the Threads API with the earlier mentioned parameters. For example, the following command will tell ElasticSearch to examine threads in the `WAITING` state with an interval of one second:

```
curl 'localhost:9200/_nodes/hot_threads?type=wait&interval=1s'
```

Hot Threads API usage clarification

Unlike other API responses where you can expect JSON to be returned, the Hot Threads API returns formatted text, where you can distinguish several sections. Before we see this, we would like to tell you something about the logic behind Hot Threads API. ElasticSearch takes all the running threads and collects various information about CPU time spent in each thread, number of times particular thread was blocked or in waiting state, how long it was blocked or in waiting state, and so on. The next thing is waiting for a particular amount of time (specified by the `interval` parameter). After that time passes, ElasticSearch collects the same information once again and sorts it on the basis of the time threads that were running (in descending order). Of course, the mentioned time is measured for a given operation type specified by the `type` parameter. After that the first N threads (where the N is the number of threads specified by the `threads` parameter) are analyzed by ElasticSearch. What ElasticSearch does is that at every few milliseconds, it takes a few snapshots (the number of snapshots is specified by the `snapshot` parameter) of stack traces of the threads. The last thing that needs to be done is grouping of stack traces to visualize changes in the thread state.

Hot Threads API response

Now, let's go through the sections returned by the Hot Threads API. For example, the following screenshot is a fragment of a just started Elasticsearch Hot Threads API response:

```
> curl -XGET 'localhost:9200/_nodes/hot_threads'
::: [Ozone][_EDivmEeQZG1hbCrKB9ljQ][inet[/192.168.0.100:9300]]

  0,1% (300micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][transport_client_timer][T#1]{Hashed wheel timer #1}'
  10/10 snapshots sharing following 5 elements
    java.lang.Thread.sleep(Native Method)
    org.elasticsearch.common.netty.util.HashedWheelTimersWorker.waitForNextTick(HashedWheelTimer.java:467)
    org.elasticsearch.common.netty.util.HashedWheelTimersWorker.run(HashedWheelTimer.java:376)
    org.elasticsearch.common.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
    java.lang.Thread.run(Thread.java:722)

  0,0% (228micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][timer]'
  10/10 snapshots sharing following 2 elements
    java.lang.Thread.sleep(Native Method)
    org.elasticsearch.threadpool.ThreadPool$EstimatedTimeThread.run(ThreadPool.java:607)

  0,0% (103micros out of 500ms) cpu usage by thread 'elasticsearch[Ozone][http_server_worker][T#4]{New I/O worker #31}'
  10/10 snapshots sharing following 15 elements
    sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
    sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:159)
    sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
    sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
    sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
    org.elasticsearch.common.netty.channel.socket.nio.SelectorUtil.select(SelectorUtil.java:64)
    org.elasticsearch.common.netty.channel.socket.nio.AbstractNioSelector.select(AbstractNioSelector.java:409)
    org.elasticsearch.common.netty.channel.socket.nio.AbstractNioSelector.run(AbstractNioSelector.java:206)
    org.elasticsearch.common.netty.channel.socket.nio.AbstractNioWorker.run(AbstractNioWorker.java:88)
    org.elasticsearch.common.netty.channel.socket.nio.NioWorker.run(NioWorker.java:178)
    org.elasticsearch.common.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
    org.elasticsearch.common.netty.util.internal.DeadLockProofWorker$1.run(DeadLockProofWorker.java:42)
    java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1110)
    java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:603)
    java.lang.Thread.run(Thread.java:722)

> |
```

Now let's discuss the Hot Threads API response by looking at a more useful example of response than the one shown in the previous screenshot:

The first section of the response can be a thread as a general header and looks like this:

```
::: [Cecilia][0d7etUE3TQuVi9kGRsOpw][inet[/192.168.0.100:9300]]
```

Thanks to it, we can see which node it talks about, which is handy when Hot Threads API call goes to many nodes.

The next lines can be divided into sections which start with something like this:

```
22.5% (112.3ms out of 500ms) cpu usage by thread
  'elasticsearch[Cecilia][search][T#993]'
```

In our case, we see that the thread, named `search`, takes 22.5% of all CPU time at the time when the measurement was done. The `cpu usage` command indicates that we are using `type` equal to `cpu` (other values you can expect here are `block usage` for threads in blocked state and `wait usage` for threads in waiting states). The thread name is very important here (thanks to it we can guess what functionality of ElasticSearch causes problems). In our example, we see that this thread is about searching (the `search` value). Other example values that you can expect to see are: `recovery_stream` (for recovery module events), `cache` (for caching events), `merge` (for segments merging threads), `index` (for data indexing threads), and so on.

The next part of the Hot Threads API response is a section starting with the following information:

```
10/10 snapshots sharing following 5 elements
```

The previous information will be followed by the stack trace. In our case, `10/10` means that 10 snapshots taken has the same stack trace, which in most cases mean that for the current thread, all examination time was spent in the same part of ElasticSearch code.

Real-life scenarios

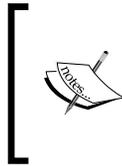
The section you are about to read is designed in a different way than the previous ones. What we wanted is to get you off describing ElasticSearch features and instead, take you on a short trip where you can see how to use different API to see what is happening to your cluster. Please note that the examples given in the following sections are real life examples we encountered.

Slower and slower performance

So, we deployed our application to production and it is a great success. After some time, our system administrator comes in and says that our monitoring system reports some performance degradation, by some he means at least 30 percent higher query latency. Maybe it is nothing that we should worry about (actually we should!), but in the broader perspective, we have to do something with it.

Let's connect to our cluster and gather some statistics. We can do that by running the following command:

```
curl 'localhost:9200/_stats?pretty'
```



The detailed description of this command is beyond the scope of this book. If you want more information, please see our previous book: *ElasticSearch Server* on Packt's website or look at <http://www.elasticsearch.org/guide/reference/api/admin-indices-stats/>.

The previous command returns much information regarding all indices and also broken down by each of the indices present in the cluster. Our system administrator has the habit of running this command once every hour and stores the result. This allows us to examine what has changed from previously run inspections (it is worth having historical statistical information, isn't it?) and actually see what is happening. In our case, the indices have approximately the same number of documents and they are almost the same when it comes to disk space as in historical data; so we can assume that it's nothing regarding indexing. Indexation statistics also doesn't show anything interesting. But statistics of get and search operations are quite enigmatic:

```
"get" : {
  "total" : 408709,
  "time" : "3.6h",
  "time_in_millis" : 13048877,
  "exists_total" : 359320,
  "exists_time" : "2.9h",
  "exists_time_in_millis" : 10504331,
  "missing_total" : 49389,
  "missing_time" : "42.4m",
  "missing_time_in_millis" : 2544546,
  "current" : 0
},
"search" : {
  "query_total" : 136704,
  "query_time" : "15.1h",
  "query_time_in_millis" : 54427259,
  "query_current" : 0,
  "fetch_total" : 84127,
  "fetch_time" : "10.8m",
  "fetch_time_in_millis" : 648088,
  "fetch_current" : 0
}
```

Believe it or not, but comparing with the older information, we see that volume of information read from Elasticsearch increased. Let's do some math: during 3.6 hours the server handles 408709 get requests and the request average takes about 32 ms. For searching, we see that the total amount of time Elasticsearch spent on querying was 15.1 hours during which 136704 searches were handled, which gives us an average search request time of 400ms. Those numbers can't be used to say anything about how our traffic is distributed; so the average response times can be used for comparison, but not for drawing conclusions about performance. For us, it is important that the numbers are increasing (in the same time frame we have more searches and gets served comparing to historical data). What we would like to achieve is to still have the same average response time that was before the increase in traffic. Probably the easiest way for preparing for more traffic is by dividing that traffic to more nodes than we have now. This means that we need to set up new nodes and place our indices shards there; so what we need to do is raise the number of replicas. Of course, we can do that easily with Elasticsearch. For example, to do that for our `messages` index, we would run the following command (assuming we did not have more than two replicas before the command was run):

```
curl -XPUT localhost:9200/messages/_settings -d '{
  "index.number_of_replicas" : 3
}'
```

After the previous command, we can check our cluster state by running the following command:

```
curl localhost:9200/_cluster/state?pretty
```

Our cluster was already properly balanced and the previous command reported that we now have unassigned shards:

```
{
  "state" : "UNASSIGNED",
  "primary" : false,
  "node" : null,
  "relocating_node" : null,
  "shard" : 1,
  "index" : "messages"
}
```

If you use additional reporting plugins, such as paramedic or head, you will see this more clearly way: our index contains unallocated replicas, which will be placed on a new node (or nodes) if they will join the cluster. Now if we would run a new node, after a while, Elasticsearch will start and assign those unassigned shards (or some of them, depending on the settings) to this node and that node will automatically handle the incoming requests. The only thing to do is ask our system administrator about new statistics after a while and check if the load of machine decreases (or use some automated monitoring system and do it yourself).

Heterogeneous environment and load imbalance

Another case we would like to share is when we saw reports indicating overall performance degradation during indexing. This was a problem because there was a strong business requirement that put emphasis on fast publication of new messages in the application. We had to improve indexing speed and reduce or eliminate its impact on performance. The first idea we had was to move indexing to the stronger server.

As in the previous example, we can use admin indices stats API to gather some information about traffic. Again, we used the following command:

```
curl 'localhost:9200/_stats?pretty'
```

In the reply, the most interesting fragment (or the one we were focused on) was the following one:

```
"indexing" : {
  "index_total" : 1475514,
  "index_time" : "1.2h",
  "index_time_in_millis" : 4400308,
  "index_current" : 167,
  "delete_total" : 2608506,
  "delete_time" : "1.3h",
  "delete_time_in_millis" : 4997302,
  "delete_current" : 0
}
```

This is a quite good result but we want more. As we said before, we move the indexing to a more powerful server. It is not a problem; we have several servers with various hardware and we can move data between them (in order to learn more about the process of shard allocation, please refer to the entire *Chapter 4, Index Distribution Architecture*). In the beginning, our Elasticsearch cluster was built from equal nodes and each of them had the same responsibilities. Of course, there is a master node, but every node can be elected for this role. The question is: what can we do if we want to have specialized nodes? In the *Gateway and recovery configuration* section of *Chapter 5, Elasticsearch Administration*, we described the `node.data` and `node.master` settings. Let's recall this and look at which roles can be given for Elasticsearch node by using the following mentioned settings:

- `node.data = true` and `node.master = true`: These are the standard situation. The node can hold data, process queries, and can be elected as a master node.
- `node.data = true` and `node.master = false`: In this case, the node will never be elected as a master but can hold data and process queries.
- `node.data = false` and `node.master = true`: In this case, node can be a master and can process queries but data will not be put on such nodes. However, such nodes can process queries.
- `node.data = false` and `node.master = false`: Such nodes will never be elected as a master and will contain no data, but can be used as process queries.

We would like to stop here and give some small clarification. As you see, the node does not have to hold data to process queries due to distributed nature of Elasticsearch. When you send a query to the node, it is forwarded to nodes that hold shards necessary to execute the query and get the data (you can read about which shard are chosen by Elasticsearch and how to control it in the *Query execution preference* section of *Chapter 4, Index Distribution Architecture*). The replies from the nodes holding the data are merged and processed by one of the Elasticsearch nodes and sent back to the client. In the more complicated query (for example, faceting), this merging and processing is a resource consuming task. We use faceting a lot in our application, so we decided to separate part of the nodes as so called aggregator nodes: without data and master responsibility, so they only process queries. Thanks to it, we can send queries only to these nodes and not stress the data nodes with query aggregation and thus, give data nodes additional processing power to handle greater load from indexing.

My server is under fire

The last example we would like to discuss is again about an issue. Under certain conditions, a random Elasticsearch node from the cluster is heavily loaded for no visible reason. When everything failed, you have been asked for help (OK, we were, but let's assume that you are a consultant for now). This time, finding problems took some time. You waited for the problem to occur again (because there were no monitoring software present and the nodes were restarted) and just after that you fired the following command:

```
curl localhost:9200/_nodes/hot_threads
```

You saw that there are a very large amount of searches, few indexing threads and even less threads regarding caches. In addition to that we've noticed a thread responsible for segments merge operation. That can be a good catch and we should check it. After a while, by using the `iostat` operating system command (we were working on Linux based nodes), we confirmed that something strongly uses I/O operations: both reads and writes. This leads to accumulation of incoming search requests, increase response time, and finally, the node becomes unresponsive because of that. "Let's buy SSD drives"; this is one of the solutions we could propose, which is a very good idea, but for now, you decide to limit the I/O operations used by the segments merge process. You can find more information about this in the *When it is too much for the I/O - throttling explained* section of this chapter. In order to adjust throttling, we used the following command:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent" : {
    "indices.store.throttle.type" : "merge",
    "indices.store.throttle.max_bytes_per_sec" : "20mb"
  }
}'
```

This, in fact, fixed the problem. The one thing that we experimented with was the appropriate value of throttling, but finally, we decided to go with 20mb, which was enough in our case, both when it comes to merging the speed and merging the process, without interfering with the overall node work.

Summary

In this chapter, we've learned what garbage collector is, how it works, and how to diagnose problems with it when they happen. We've also seen how to limit and control the amount of I/O operations used by the store module and what the query speed improvements warmers can bring us. We've learned what hot threads are, how to get them using Elasticsearch API, and how to interpret the response Elasticsearch gives us. Finally, we've used Elasticsearch API to get the statistics that helped us diagnose problems with Elasticsearch.

In the next chapter, we'll concentrate on improving user search experience in general. We will use the newly introduced suggest API to correct the user spelling mistakes and we will use faceting to allow users to quickly find what they are looking for. We will also see how to improve the query relevance with different types of queries that are available in Elasticsearch.

7

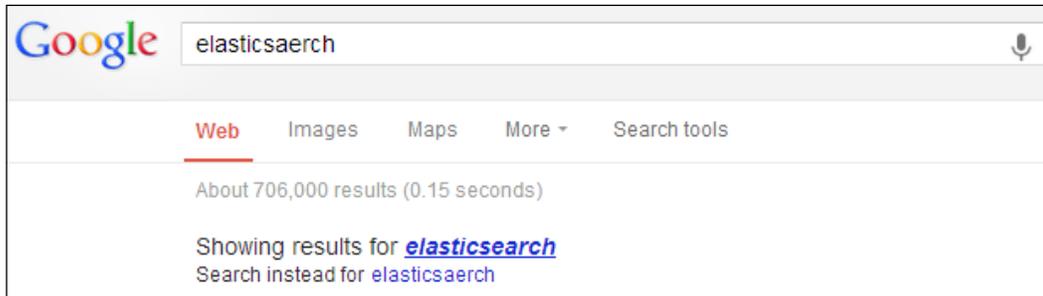
Improving the User Search Experience

In the previous chapter we learned what is a garbage collector, how it works, and how to diagnose problems with it when they happen. We've discussed how to limit and control the amount of I/O operations used by the store module and what query speed improvement warmers can bring us. We've learned what are hot threads, how to get them using the Elasticsearch API, and how to interpret the response Elasticsearch gives us. We've used the Elasticsearch API to get the statistics that helped us to diagnose problems associated with Elasticsearch. In this chapter we'll focus on the user search experience. By the end of this chapter, you will have learned:

- How to use the Elasticsearch Suggest API to correct user spelling mistakes
- How to use `term suggester` to suggest single words
- How to use the `phrase suggester` to suggest whole phrases
- How to configure suggest capabilities to match your needs
- How to use `completion suggester` for the autocomplete functionality
- How to improve query relevance by using different Elasticsearch functionalities

Correcting user spelling mistakes

One of the simplest ways to improve user search experience is to correct their spelling mistakes, either automatically or by just showing the correct query phrase and allowing the user to use it. For example, this is what Google shows us when we enter `elasticsaerch` instead of `elasticsearch`:



Since 0.90.0.Beta1, Elasticsearch allows us to use the Suggest API to correct user spelling mistakes. However, the documentation states that this functionality is still under development and it may change drastically in the upcoming Elasticsearch versions.

Test data

For the purpose of this section, we decided that we need a bit more data than 10 documents. In order to get the data we needed, we decided to use the Twitter river plugin to index some public tweets from Twitter. First, we need to install the plugin by running the following command:

```
bin/plugin -install elasticsearch/elasticsearch-river-twitter/1.4.0
```

And then we run the following command:

```
curl -XPUT 'localhost:9200/_river/my_twitter_river/_meta' -d '{
  "type" : "twitter",
  "twitter" : {
    "oauth" : {
      "consumer_key" : "****",
      "consumer_secret" : "****",
      "access_token" : "****",
      "access_token_secret" : "****"
    }
  }
}
```

```
},  
"index" : {  
  "index" : "twitter",  
  "type" : "status",  
  "bulk_size" : 100  
}  
'}
```

In order to get your authentication details you need to log in to <https://dev.twitter.com/apps/>, create a new Twitter application, and create an access token after that. I've indexed about 100,000 documents using the river to the `twitter` index.

Getting into technical details

The Suggest API is not the simplest one available in ElasticSearch. In order to get the desired suggestion, we can either add a new `suggest` section to the query or we can use a specialized REST endpoint that ElasticSearch exposes. In addition to that we have two suggest implementations that allow us to correct user spelling mistakes with various options that we can tune, depending on the use case (and one more implementation that was introduced in ElasticSearch 0.90.3, but we will get to that later). All this gives us a powerful and a flexible mechanism that we can use in order to make our search better.

Of course, the suggest functionality works on our data, so if we have small set of documents in the index the appropriate suggestion may not be found. While dealing with less data, ElasticSearch has fewer words in the index and because of that, it has fewer candidates for suggestions. On the other hand, the more the data, the bigger the possibility that we will have data that has some mistakes, however as we will see, ElasticSearch handles that pretty well.



Please note that the layout of this chapter is a bit different. We start by showing you a simple example on how to query for suggestions and how to interpret Suggest API response without getting much into all the configuration options. This is because we don't want to overwhelm you with technical details, but we want to show you what you can achieve. The nifty configuration parameters come later.

Suggesters

Before we continue with querying and analyzing the responses, we will write a few words about the available suggester type: the functionality responsible for finding a suggestion while using the Elasticsearch Suggest API. Elasticsearch allows us to use three suggesters currently: the `term` one, the `phrase` one, and the `completion` one. The first two allow us to correct spelling mistakes, while the third one allows us to develop very fast and autocomplete functionality. With Elasticsearch 0.90.3 we have the possibility of using the prefix-based suggester which is very handy for implementing the autocomplete functionality and which we will discuss in the *Completion suggester* section. However, for now, let's not focus on any particular suggester type, but let's look at the query possibilities and the responses returned by Elasticsearch. We will try to show the general principles and then we will get into more details about each of the available suggesters.

Using the `_suggest` REST endpoint

The first possibility where we can get suggestions for a given text is using a dedicated `_suggest` REST endpoint. What we need to provide is the text to analyze and the type of used suggester (`term` or `phrase`). So, if we like to get suggestions for the words `graphics designner` (we've made the spelling mistake purposefully), we will run the following query:

```
curl -XPOST 'localhost:9200/twitter/_suggest?pretty' -d '{
  "first_suggestion" : {
    "text" : "graphics designner",
    "term" : {
      "field" : "_all"
    }
  }
}'
```

As you can see, each suggestion request is sent to Elasticsearch in its own object, with the name we chose (in the preceding case it is `first_suggestion`). Next we specify the text we want, the suggestion to be returned by using the `text` parameter. Finally we add the `suggester` object, which is either `term` or `phrase` currently. The `suggester` object contains its configuration, which for the `term` suggester used in the preceding example, is the field we want to use for suggestions (the `field` property).

We can also send more than one suggestion at a time, by adding multiple suggestion names. For example, if in addition to the preceding suggestion, we also like to include one for the word, `worcing` we should use the following command:

```
curl -XPOST 'localhost:9200/twitter/_suggest?pretty' -d '{
  "first_suggestion" : {
    "text" : "graphics designer",
    "term" : {
      "field" : "_all"
    }
  },
  "second_suggestion" : {
    "text" : "worcing",
    "term" : {
      "field" : "description"
    }
  }
}'
```

Understanding the REST endpoint suggester response

Let's now look at the example response we can expect from the `_suggest` REST endpoint call. Although the response will differ for each suggester type, let's look at the response returned by ElasticSearch for the first command we've sent previously that used the `terms` suggester:

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "first_suggestion" : [ {
    "text" : "graphics",
    "offset" : 0,
    "length" : 8,
    "options" : [ {
      "text" : "graphic",
      "score" : 0.85714287,
      "freq" : 9
    } ]
  } ]
}
```

```
    }, {
      "text" : "desiganer",
      "offset" : 9,
      "length" : 9,
      "options" : [ {
        "text" : "designer",
        "score" : 0.875,
        "freq" : 60
      } ], {
        "text" : "designers",
        "score" : 0.7777778,
        "freq" : 7
      }, {
        "text" : "desaigner",
        "score" : 0.7777778,
        "freq" : 1
      }, {
        "text" : "designed",
        "score" : 0.75,
        "freq" : 3
      } ]
    } ]
  } ]
}
```

As you can see in the preceding response, the `term_suggester` returns a list of possible suggestions for each term that was present in the `text` parameter of our `first_suggestion` section. For each term, the `term_suggester` ElasticSearch will return an array of possible suggestions with additional information. Looking at the data returned for the `desiganer` term we can see the original word (the `text` parameter), its offset in the original `text` parameter (`offset` parameter), and its length (`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 characterized 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 may be.

- `freq`: This property defines the frequency of the suggestion. The frequency represents how many times the word appears in the documents, in the index where we are running the suggestion query against. The higher the frequency, the more documents have the suggested word in its fields and the higher the chance that the suggestion is the one we are looking for.

 Please remember that the phrase suggester response will differ from the one returned by the `terms` suggester and we will discuss response of the phrase suggester later in this section.

Including suggestions requests in a query

In addition to using the `_suggest` REST endpoint, we can include the `suggest` section in addition to the `query` section in a normal query sent to Elasticsearch. For example, if we want to get the same suggestion we've got in the first example, but during query execution we should send the following query:

```
curl -XGET 'localhost:9200/twitter/_search?pretty' -d '{
  "query" : {
    "match_all" : {}
  },
  "suggest" : {
    "first_suggestion" : {
      "text" : "graphics designaner",
      "term" : {
        "field" : "_all"
      }
    }
  }
}'
```

There is also one more possibility: if we have the same suggestion text, but we want multiple suggestion types, we can embed our suggestions in the `suggest` object and place the `text` property as the `suggest` object option. For example, if we like to get suggestions for the `graphics designaner` text for the `description` field and for `_all` field we should run the following command:

```
curl -XGET 'localhost:9200/twitter/_search?pretty' -d '{
  "query" : {
    "match_all" : {}
  },
  "suggest" : {
    "text" : "graphics designaner",
    "first_suggestion" : {
      "term" : {
        "field" : "_all"
      }
    },
    "second_suggestion" : {
      "term" : {
        "field" : "description"
      }
    }
  }
}'
```

Suggester response

As you can guess, the response will include both query results and the suggestions, and you are right. The response for the preceding command would be as follows:

```
{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 50175,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "twitter",
      "_type" : "status",
```

```

    "_id" : "346196614853046274",
    "_score" : 1.0
  },
  ...
} ]
},
"suggest" : {
  "first_suggestion" : [ {
    "text" : "graphics",
    "offset" : 0,
    "length" : 8,
    "options" : [ {
      "text" : "graphic",
      "score" : 0.85714287,
      "freq" : 9
    } ]
  } ], {
    "text" : "desiganer",
    "offset" : 9,
    "length" : 9,
    "options" : [ {
      "text" : "designer",
      "score" : 0.875,
      "freq" : 60
    } ], {
      "text" : "designers",
      "score" : 0.7777778,
      "freq" : 7
    } ], {
      "text" : "desaigner",
      "score" : 0.7777778,
      "freq" : 1
    } ], {
      "text" : "designed",
      "score" : 0.75,
      "freq" : 3
    } ]
  } ]
}
}
}

```

As we can see we've got both the search results and the suggestions whose structure we've already discussed earlier in this section.

We now know how to make a query with suggestions returned or how to use the `_suggest` REST endpoint. Let's now get into more details of each of the available suggesters type.

The term suggester

The `term` suggester works on the basis of edit distance, which means that the suggestion with fewer characters that need to be changed or removed to make the suggestion look as the original word is the best one. For example, let's take the words `worl` and `work`. In order to change the `worl` term to `work` we need to change the `l` letter to `k`, so it means a distance of 1. The text provided to the suggester is of course analyzed and then terms are chosen to be suggested.

Configuration

We have already seen how the term suggester works and what it can give us. Now let's discuss its configuration options.

Common term suggester options

The common term suggester options can be used for all the suggester implementations that are based on the `term` suggester. Currently those are the phrase suggesters and of course the base term suggester. The available options are as follows:

- `text`: This option represents the text for which we want to get the suggestions. This parameter is required in order 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. For example, if we only want to consider the `title` field terms in suggestions we should set this parameter's value to `title`.
- `analyzer`: This option represents the name of the analyzer which should be used to analyze the text provided in the `text` parameter. If not set, Elasticsearch will use the analyzer used for the field provided by the `field` parameter.
- `size`: This option represents the maximum number of suggestions that is allowed to be returned by each term provided in the `text` parameter. Default is 5.
- `sort`: This option allows us to specify how suggestions should be sorted in the result returned by Elasticsearch. By default, it is set to `score`, which tells Elasticsearch that the suggestions should be sorted by the suggestion score first, next by the suggestion document frequency, 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.

- `suggest_mode`: This is another suggestion parameter that allows us to control which suggestions should be included in the ElasticSearch's response. Currently there are three values that can be passed to this parameter: `missing`, `popular`, and `always`. The default `missing` value will inform ElasticSearch to generate suggestions to only those words provided in the `text` parameter that doesn't exist in the index. If this property will be set to `popular`, then the `term` suggester will only suggest terms that are more popular (existing in more number of documents) than the original term for which the suggestion was generated. The last value, `always` will result in the suggestion generated for each of the words in the `text` parameter.

Additional term suggester options

In addition to the common term suggest options ElasticSearch allows us to use additional ones that will only make sense for the `term` suggester. Those options are as follows:

- `lowercase_terms`: This option, when set to `true` will inform ElasticSearch to lowercase all the terms that are produced from the `text` field after analysis.
- `max_edits`: The default value of this option is 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. Setting this value to 1 can result in fewer suggestions or no suggestions at all for words with many spelling mistakes, `prefix_len`: because usually spelling mistakes do not appear at the beginning of the word. ElasticSearch allows us to set how many initial characters of the suggestion must match with the initial characters of the original term. By default, this property is set to 1. If we are struggling with suggester's performance, increasing this value will improve the overall performance, because fewer suggestions will only be processed.
- `min_word_len`: The default of this option is 4 and specifies the minimum number of characters a suggestion must have in order to be returned on 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 (this is because of the fact that terms are held in different shards for our indices unless we have a single-shard index created) being calculated but will also result in the spellchecker's performance degradation.

- `max_inspections`: This option defaults to 5 and specifies how many candidates will ElasticSearch inspect, in order to find the words that can be used as suggestions. ElasticSearch will inspect a maximum of `shard_size` values multiplied by the `max_inspections` candidates for suggestions. Setting this property to values higher than the default 5 may improve the suggester's accuracy, but can also degrade the performance.
- `min_doc_freq`: This option defaults to 0, which means not enabled. It allows us to limit the returned suggestions to only those that appear in the number of documents higher than the value of this parameter (this is per shard value, and not a globally counted one). For example, setting this parameter to 2, will result in a suggestion that appears in at least two documents in a given shard. Setting this property to values higher than 0 can improve returned suggestions quality, however it can also result in some suggestion not being returned because it has low shard document frequency. This property can help us in removing suggestions that come from a few documents and may be erroneous. This parameter can be specified as a percentage: if we want to do so its value must be less than 1. For example 0.01 means 1%, which again means that the minimum frequency of the given suggestion needs to be higher than 1% of the total term frequency (of course per shard).
- `max_term_freq`: This option defaults to 0.01 and specifies the maximum number of documents the term from the `text` field should exist, for it to be considered as a candidate for spellchecking. Similar to the `min_doc_freq` parameter, it can be either provided as an absolute number (for example, 4 or 100) or it can be a percentage value if it is beyond 1 (for example 0.01 means 1%). Please remember that this is also a per-shard frequency. The higher the value of this property the better the overall performance of the spellchecker. In general, this property is very useful when we want to exclude terms that appear in many documents from spellchecking, because those are usually correct terms.
- `accuracy`: This option defaults to 0.5 and can be a number between 0 to 1. This option specifies how similar the term should be when compared with the original one. The higher the value, the more similar the terms need to be. This value is used in comparison during string distance calculation for each of the terms from the original input.

- `string_distance`: This option specifies which algorithm should be used to compare how similar terms are to each other. This is an expert setting. The following options are available: the `internaldefault` comparison algorithm based on optimized implementation of the *Damerau Levenshtein* similarity algorithm, `damerau_levenshtein` is the implementation of the Damerau Levenshtein string distance algorithm (http://en.wikipedia.org/wiki/Damerau-Levenshtein_distance), `levenshtein` which is an implementation of Levenshtein distance (http://en.wikipedia.org/wiki/Levenshtein_distance), `jarowinkler` which is an implementation of the Jaro-Winkler distance algorithm (http://en.wikipedia.org/wiki/Jaro-Winkler_distance) and finally the `ngram`, which is an n-gram based distance algorithm.



Because we've used the `terms suggester` during the initial examples, we decided to skip showing how to query the `terms suggester` and how the response would look like in this place. If you want to see how to query this suggester and what the response looks like, please refer to the beginning of the `suggesters` section.

The phrase suggester

The `term suggester` provides a great way to correct user spelling mistakes on a per term basis. However, if we want to get back the phrases, it is not possible using that suggester. That's why the `phrase suggester` was introduced. It is built on top of the `term suggester` and adds additional phrase calculation logic to it, so that whole phrases can be returned instead of individual terms. It uses the n-gram based language models to calculate how good the suggestion is and will probably be a better choice for suggesting whole phrases instead of the `term suggester`. The n-gram approach divides terms in the index into grams: word fragments are built of one or more letters. For example, if we want to divide the word `mastering` into bi-grams (two letter n-gram) it would look similar to this: `ma as st te er ri in ng`.



If you want to read more about the n-gram language models please refer to the Wikipedia's article available at http://en.wikipedia.org/wiki/Language_model#N-gram_models and continue from there.

The usage example

Before we continue with all the possibilities we have to configure the `phrase` suggester, let's start by showing the example of how to use it. This time we will run a simple query to the `_search` endpoint with only `suggest`s section in it. We do that by running the following command:

```
curl -XGET 'localhost:9200/twitter/_search?pretty' -d '{
  "suggest" : {
    "text" : "graphics designer",
    "our_suggestion" : {
      "phrase" : {
        "field" : "_all"
      }
    }
  }
}'
```

As you can see in the preceding command it is almost the same as we did while using the `term` suggester, but instead of specifying the `term` suggester's type, we've specified the `phrase` type. The response of the preceding command would be as follows:

```
{
  "took" : 58,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 50175,
    "max_score" : 1.0,
    "hits" : [
      ...
    ]
  },
  "suggest" : {
```

```

"our_suggestion" : [ {
  "text" : "graphics designer",
  "offset" : 0,
  "length" : 18,
  "options" : [ {
    "text" : "graphics designer",
    "score" : 4.665424E-5
  }, {
    "text" : "graphics designers",
    "score" : 2.3448094E-5
  }, {
    "text" : "graphics designed",
    "score" : 1.9354089E-5
  }, {
    "text" : "graphic designer",
    "score" : 1.7957824E-5
  }, {
    "text" : "graphics designer",
    "score" : 1.3458714E-5
  } ]
} ]
}
}

```

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 as the suggestion for each term from the `text` field it is already combined and ElasticSearch returns whole phrases. Of course, we can configure additional parameters in the `phrase` section and now we will look at what parameters are available for usage. Of course, by default the returned suggestions are sorted by their score.

Configuration

The `phrase_suggester` configuration parameter can be divided into three groups: basic parameter which defines the general behavior, the smoothing models configuration for balancing the n-gram's weights, and the candidate generators configuration which are responsible for producing the list of terms suggestions that will be used to return final suggestions.

Basic configuration

Because the `phrase` suggester is based on the term suggester it can also use some of the configuration options provided by it. Those options are: `text`, `size`, `analyzer`, and `shard_size`. Please refer to the term suggester's description earlier in this chapter to find out what they mean.

In addition to the properties mentioned previously, the `phrase` suggester exposes the following basic options:

- `gram_size`: This option specifies the maximum size of the n-gram that is stored in the field specified by the `field` property. If the given field doesn't contain n-grams, this property should be set to `1` or not passed with the suggestion request. If not set, Elasticsearch will try to detect the correct value of this parameter by itself. For example, for fields using a `shingle` filter (<http://www.elasticsearch.org/guide/reference/index-modules/analysis/shingle-tokenfilter/>) the value of this parameter will be set to the `max_shingle_size` property (of course, if not set explicitly).
- `confidence`: This option specifies the parameter which allows us to limit the suggestion based on their score. The value of this parameter is applied to the score of the input phrase (the score is multiplied by the value of this parameter) and this score is used as a threshold for generated suggestions. If a suggestion score is higher than the calculated threshold it will be included in the returned results, if not then it will be dropped. For example, setting this parameter to `1.0` (which is its default value) will result in suggestions that are scored higher than the original phrase. On the other hand, setting it to `0.0` will result in the suggester returning all the suggestions (limited by the `size` parameter) no matter what their score is.
- `max_errors`: This property allows us to specify the maximum number (or percentage) of terms which can be erroneous (not correctly spelled) in order to create a correction using it. The value of this property can be either an integer number such as `1`, `5` or a float between `0` and `1` which will be treated as a percentage value. If we set a float, it will specify the percentage of terms that can be erroneous, for example a value of `0.5` will mean 50%. If we specify an integer number, for example `1`, `5` Elasticsearch will treat it as the maximum number of erroneous terms. 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 the terms in the resulting bigram field.
- `force_unigrams`: This option defaults to `true` and specifies if the spellchecker should be forced to use a gram size of `1` (unigram).

- `token_limit`: This option defaults to 10 and specifies the maximum number of tokens the corrections list can have in order to it to be returned. Setting this property to a value higher than the default one may improve the suggester's accuracy at the cost of its performance.
- `real_word_error_likelihood`: This option specifies a percentage value, which defaults to 0.95 and specifies how likely it is that a term is misspelled even though it exists in the dictionary (built of the index). The default value of 0.95 informs Elasticsearch that 5% of all terms that exist in its dictionary are misspelled. Lowering the value of this parameter will result in more terms being taken as misspelled ones even though they may be correct.

Configuring smoothing models

Smoothing model is a functionality of the `phrase` suggester whose responsibility is to measure the balance between the weights of infrequent n-grams which don't exist in the index and the frequent ones that do exist in the index. It is rather an expert option and if you want to modify those, you should check suggester responses for your queries in order to see if your suggestions are proper for your case. Smoothing is used in language models to avoid situations where probability of a given term is equal to zero. Elasticsearch `phrase` suggester supports multiple smoothing models.



You can find out more about language models at http://en.wikipedia.org/wiki/Language_model.

In order to set which smoothing model we want to use we need to add an object called `smoothing` and include a smoothing model name we want to use inside it. Of course, we can include the properties we need or want to set for the given smoothing model. There are three smoothing models available in Elasticsearch. For example we can run the following command:

```
curl -XGET 'localhost:9200/_search?pretty&size=0' -d '{
  "suggest" : {
    "text" : "graphics designer",
    "generators_example_suggestion" : {
      "phrase" : {
        "analyzer" : "standard",
        "field" : "_all",
        "smoothing" : {
          "linear" : {
            "trigram_lambda" : 0.1,
```

```
    "bigram_lambda" : 0.6,  
    "unigram_lambda" : 0.3  
  }  
}  
}  
}  
}  
}'
```

Let's look at which smoothing models can be used with the `phrase_suggester` in ElasticSearch.

Stupid backoff

Stupid backoff is the default smoothing model used by the ElasticSearch phrase suggester. In order to alter it or force its usage we need to use the `stupid_backoff` name. The stupid backoff smoothing model is an implementation that will use a lower ordered n-gram (and will give it a discount equal to the value of the `discount` property) if higher order n-gram count is equal to 0. To illustrate the example, let's assume that we have a bigram `ab` and a unigram `c` which are common and they exist in our index used by suggester. However, we don't have the trigram `abc` present. What the stupid backoff model will do is use the `ab` bigram model, because the `abc` doesn't exist and of course the `ab` bigram model will be given a discount equal to the value of the `discount` property.

The stupid backoff model provides a single property that we can alter: the `discount`. By default it is set to `0.4` and it is used as a discount factor for the lower-ordered, n-gram model.

You can read more about the n-gram smoothing models by visiting at http://en.wikipedia.org/wiki/N-gram#Smoothing_techniques and http://en.wikipedia.org/wiki/Katz's_back-off_model (which is similar to the stupid backoff model described).

Laplace

The Laplace smoothing model is also called additive smoothing. When used (in order to use it, we need to use the `laplace` value as its name), a constant value equal to the value of the `alpha` parameter (which is by default `0.5`) will be added to counts to balance the weights of frequent and infrequent n-grams.

As mentioned, the Laplace smoothing model can be configured using the `alpha` property, which is by default set to `0.5`. The usual values for this parameter are typically equal or below `1.0`.

You can read more about additive smoothing at http://en.wikipedia.org/wiki/Additive_smoothing.

Linear interpolation

Linear interpolation is the last smoothing model that takes the values of the lambdas provided in the configuration and uses them to calculate weights of trigrams, bigrams and unigrams. In order to use the linear interpolation smoothing model, we need to provide the name, `linear` in the `smoothing` in the suggester query object and provide three parameters: `trigram_lambda`, `bigram_lambda`, and `unigram_lambda`. The sum of the values of the three mentioned parameters must be equal to `1`. Each of these parameters is weights for a given type of n-grams, for example the `bigram_lambda` parameter value will be used as a weight for bigrams.

Configuring candidate generators

In order to return possible suggestions for a term from a text provided in the `text` parameter Elasticsearch uses the so called candidate generators. You can think of candidate generators as term suggesters although they are not exactly the same: they are similar, because they are used for every single term in the provided query given to a suggester. After the candidate terms are returned they are scored in combination with suggestions for other terms from the text and this way the phrase suggestions are built.

Direct generators

Currently direct generator is the only candidate generator available in Elasticsearch although we can expect more of them to be present in the future. Elasticsearch allows us to provide multiple direct generators in a single phrase suggester request. We can do that by providing the list named `direct_generators`. For example we can run the following command:

```
curl -XGET 'localhost:9200/_search?pretty&size=0' -d '{
  "suggest" : {
    "text" : "graphics designer",
    "generators_example_suggestion" : {
      "phrase" : {
        "analyzer" : "standard",
        "field" : "_all",
```

```
"direct_generator" : [
  {
    "field" : "_all",
    "suggest_mode" : "always",
    "min_word_len" : 2
  },
  {
    "field" : "_all",
    "suggest_mode" : "always",
    "min_word_len" : 3
  }
]
}
```

The response should be very similar to the one previously shown so we decided to omit it.

Configuring direct generators

Direct generators allow us to configure their behavior by using similar parameters to those that are exposed by terms suggester. Those common configuration parameters are: the `field` (which is required), `size`, `suggest_mode`, `max_edits`, `prefix_length` (which is the same as the `prefix_len` parameter), `min_word_len` (in this case defaults to 4), `min_word_len`, `max_inspections`, `min_doc_freq`, `max_term_freq`. Please refer to the term suggester description to see what those parameters mean.

In addition to the mentioned properties, direct generators allow us to use the `pre_filter` and `post_filter` properties. These two properties allow us to provide an analyzer name that Elasticsearch will use. The analyzer specified by the `pre_filter` property will be used for each term passed to the direct generator and the filter specified by the `post_filter` property will be used after it is returned by the direct generator, just before these terms are passed to the phrase scorer for scoring.

For example, we can use the filtering functionality of the direct generators to include synonyms just before the suggestions are passed to the scorer, by using the `post_filter` property. For example, let's update our Twitter index setting to include simple synonyms and let's use them in filtering. To do that we start with updating the settings with the following commands:

```
curl -XPOST 'localhost:9200/twitter/_close'
curl -XPUT 'localhost:9200/twitter/_settings' -d '{
  "settings" : {
    "index" : {
      "analysis": {
        "analyzer" : {
          "sample_synonyms_analyzer": {
            "tokenizer": "standard",
            "filter": [
              "sample_synonyms"
            ]
          }
        },
        "filter": {
          "sample_synonyms": {
            "type" : "synonym",
            "synonyms" : [
              "graphics => made"
            ]
          }
        }
      }
    }
  }
}'

curl -XPOST 'localhost:9200/twitter/_open'
```

We need to first close the index, update the setting, and then open it again because Elasticsearch won't allow us to change settings on the opened indices.

Now we can test our direct generator using synonyms with the following command:

```
curl -XGET 'localhost:9200/_search?pretty&size=0' -d '{
  "suggest" : {
    "text" : "made desiganer",
    "generators_with_synonyms" : {
      "phrase" : {
        "analyzer" : "standard",
        "field" : "_all",
        "direct_generator" : [
          {
            "field" : "_all",
            "suggest_mode" : "always",
            "post_filter" : "sample_synonyms_analyzer"
          }
        ]
      }
    }
  }
}'
```

The response of the preceding command should be as follows:

```
{
  "took" : 27,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 50175,
    "max_score" : 1.0,
    "hits" : [ ]
  },
  "suggest" : {
    "generators_with_synonyms" : [ {
```

```

    "text" : "made desiganer",
    "offset" : 0,
    "length" : 14,
    "options" : [ {
      "text" : "made designer",
      "score" : 1.3923876E-4
    }, {
      "text" : "made designers",
      "score" : 6.893026E-5
    }, {
      "text" : "make desiganer",
      "score" : 6.1075225E-5
    }, {
      "text" : "made designed",
      "score" : 5.1168725E-5
    }, {
      "text" : "made desaigner",
      "score" : 5.1012223E-5
    } ]
  } ]
}

```

As you can see, instead of the `graphics` term, the `made` term was returned in the result of the `phrase` suggester and this is what we were looking for in this example. So, our synonyms configuration was taken into consideration. However, please remember that the synonyms was taken before the scoring of the fragments, so it can happen that the suggestions with the synonyms are not the ones that are scored the most and you will not be able to see them in the suggester's results.

Completion suggester

With the release of Elasticsearch 0.90.3 we were given the opportunity to use a prefix-based suggester. This allows us to create the autocomplete functionality in a very effective way, because complicated structures are stored in the index instead of being calculated during query time. Although this suggester is not about correcting user spelling mistakes we thought that it will be good to show at least a simple example of this highly efficient suggester.

The logic behind completion suggester

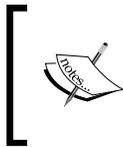
The prefix suggester is based on the data structure called **FST (Finite State Transducer)** (http://en.wikipedia.org/wiki/Finite_state_transducer). Although it is highly efficient, it may require significant resources to build on systems with large amount of data in them, systems that ElasticSearch is perfectly suitable for. If we like to build these structures on the nodes after each restart or cluster state change we may lose performance. Because of that ElasticSearch creators decided to create FST-like structures during index time and store it in the index so that it can be loaded into memory when needed.

Using completion suggester

In order to use a prefix-based suggester we need to properly index our data with a dedicated field type called `completion`, which stores the FST-like structures in the index. In order to illustrate how to use this suggester, let's assume that we want to create an autocomplete feature to allow us to show book authors, that we store in an additional index. In addition to the author's name we want to return the identifiers of the books she/he wrote, by searching them with an additional query. 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`, which is the name of the author and the `ac` field, which is the field we will use for autocompletion. The `ac` field is the one we are interested in. We've defined it using the `completion` type, which will result in storing the FST-like structures in the index. In addition to that we've used the `simple` analyzer for both index and query time. The last thing is the payload, the additional information we will return along with the suggestion, in our case it will be an array of book identifiers.



The type property for the field we will use for autocomplete is mandatory and should be set to `completion`. By default, the `search_analyzer` and `index_analyzer` properties will be set to `simple` and the `payloads` property will be set to `false`.

Indexing data

In order to index 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 their 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" : [ "121211" ] }
  }
}'
```

Notice the structure of the data for the `ac` field. We provide the `input`, `output` and `payload` properties. The `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 for building the FST alike structures and will be used for matching the user input to decide if the document should be returned by suggester. The `output` property is used to inform the suggester for which data should be there in the document.



Please remember that the `payload` property must be a JSON object that starts with the `{` character and ends with `}` character.

If the `input` and `output` property is the same in your case and you don't want to store payloads, you may index the documents just like you usually index your data. For example, the command to index our first document would look as follows:

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
  "name" : "Fyodor Dostoevsky",
  "ac" : [ "Fyodor Dostoevsky" ]
}'
```

Querying data

Finally let's look how to query our indexed data. If we like to find documents that have the author's name 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 in the `_suggest` endpoint, because we don't want to run a standard query, we are just interested in autocomplete results. The rest of the query is exactly the same as the standard suggester query run against the `_suggest` endpoint, with the query type set to `completion`.

The results of the preceding command looks like this:

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 10,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 1.0, "payload" : {"books":["123456","123457"]}
    } ]
  } ]
}
```

As you can see, in response, we've got the document we were looking for along with the payload information.

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 when you have multiple shards for your index or your index is composed of multiple segments. 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, not a float, the one similar to the boost for queries and documents. The greater the `weight` property's value, the more important the suggestion is. This gives us plenty of opportunities to control how the returned suggestions will be sorted.

For example, if we like 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" : 80
  }
}'
```

Now if we run our example query the results would be as follows:

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 10,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 80.0, "payload" : {"books":["123456","123457"]}
    } ]
  } ]
}
```

Look how the score of the result changed. In our initial example it was 1.0 and now it is 80.0: because we've set the `weight` parameter to 80 during indexing.

Additional parameters

There are two additional parameters supported by the prefix suggester that we didn't talk till now: `preserve_separators` and `preserve_position_increments`, both can be set to `true` or `false`. Setting the `preserve_separators` to `false` suggester will omit separators, for example, whitespace (of course proper analysis is required). Setting the `preserve_position_increments` property to `false` is needed if the first word in the suggestion is a stop word and we are using an analyzer that throws these stop words away. For example, if we have `The Clue` as our document, then the `The` word will be discarded by the analyzer, by setting the `preserve_position_increments` property to `false`, the suggester will be able to return our document by specifying `c` as the `text`.

Improving query relevance

ElasticSearch, in fact search engines in general, are used for searching. Of course some use cases may require to browse some portion of the data indexed, but in order to use most of them scoring should come to play. As we said in the *Default Apache Lucene scoring explained* section of *Chapter 2, Power User Query DSL*. ElasticSearch leverages the Apache Lucene library document scoring capabilities and allows us to use different query types to manipulate score of results returned by our queries. What's more, we can change the low level algorithm used to calculate score that was described in the *Altering Apache Lucene scoring* section of *Chapter 3, Low-level Index Control*.

Given all this, when we start designing our queries, we usually go for the simplest query that return the documents we want. However, given all the things we can do in ElasticSearch when it comes to scoring control, such queries return results that are not the best when it comes to user search experience. That's because ElasticSearch can't guess what our business logic is and what documents are the ones that are the best from our point of view when running a query. In this section we will try to follow a real-life example of query relevance tuning. We want to make this chapter a bit different, instead of only giving you insight, we have decided to give you a full example of how query tuning process may look like. Although some of the examples you find in this section may be general purpose ones, when using them in your own application, make sure that they make sense for you.

Just to give you a little insight on what is coming, we will start with a simple query that returns the results we want, we will alter the query introducing different ElasticSearch queries to make the results better, we will use filters, we will lower the score of the documents we think of as garbage and finally we will introduce faceting to render drill-down menus for users to allow results narrowing. Finally, as a bonus, we will see how we can look at changes we make to our queries and measure the change of users search experience.

The data

Of course in order to show you the results of the query modifications we do, we need data. We would love to show you the real-life data we were working with, but we can't, that's understandable. However there is a solution to that: for the purpose of this book we decided to index Wikipedia data. In order to do that, we've installed the Wikipedia river, by running the following command:

```
bin/plugin -install elasticsearch/elasticsearch-river-wikipedia/1.1.0
```

The Wikipedia river will create the `wikipedia` index for us if there is non-existing. However, we know that we need to adjust the index fields and in order not to reindex the data we create the index upfront. In order to do that we use the following command:

```
curl -XPOST 'localhost:9200/wikipedia/' -d '{
  "settings": {
    "index": {
      "analysis": {
        "analyzer": {
          "keyword_ngram": {
            "tokenizer": "ngram",
            "filter": ["lowercase"]
          }
        }
      }
    }
  },
  "mappings": {
    "page" : {
      "properties" : {
        "category" : {
          "type" : "multi_field",
          "fields" : {
            "category" : {"type" : "string", "index" : "analyzed"},
            "untouched" : {"type" : "string", "index" : "not_analyzed" }
          }
        }
      }
    },
    "disambiguation" : { "type" : "boolean" },
  }
}
```

```

    "link" : { "type" : "string", "store" : "no", "index" : "not_
analyzed" },
    "redirect" : { "type" : "boolean" },
    "special" : { "type" : "boolean" },
    "stub" : { "type" : "boolean" },
    "text" : {"type" : "string", "index" : "analyzed"},
    "title" : {
      "type" : "multi_field",
      "fields" : {
        "title" : {"type" : "string", "index" : "analyzed"},
        "simple" : {"type" : "string", "index" : "analyzed", "analyzer" :
"simple" },
        "ngram" : {"type" : "string", "index" : "analyzed", "analyzer" :
"keyword_ngram" }
      }
    }
  }
}
}'

```

For now, what we have to know is that we have a `page` type that we are interested in and whether that represents a Wikipedia page. We will use two fields for searching: the `text` and `title` fields. The first one holds the content of the page and the second one is responsible for holding its title.

What we have to do next is start the Wikipedia river. Because we were interested in the latest data in order to instantiate the river and start indexing we've used the following command:

```

curl -XPUT 'localhost:9200/_river/wikipedia/_meta' -d '{
  "type" : "wikipedia"
}'

```

And that's all; Elasticsearch will index the newest Wikipedia dump available to the index called `wikipedia`. All we have to do is wait. We were not patient, and we decided that we'll only index the first 10 million documents and after the Wikipedia river hits that number of documents we deleted it. We've checked the final number of documents by running the following command:

```

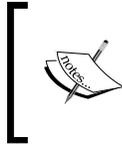
curl -XGET 'localhost:9200/wikipedia/_search?q=*&size=0&pretty'

```

The response was as follows:

```
{
  "took" : 24,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 10425183,
    "max_score" : 1.0,
    "hits" : [ ]
  }
}
```

We can see that we have 10,425,183 documents in the index.



When running examples from this chapter, please consider the fact that the data we've indexed changes over time, so the examples shown in this chapter may result in a different document if we run it after some time.

The quest for improving relevance

After we have our indexed data we are ready to begin the process of searching. We will start from the beginning, using a simple query that will return the results we are interested in. After having that, we will try to improve query relevance. We will also try to pay attention to performance and notice about the performance changes when they will probably happen.

The standard query

As you know, by default, Elasticsearch includes the content of the documents in the `_all` field. So, why we need to bother with specifying multiple fields in a query, when we can use a single one, right? Going in that direction, let's assume that we've constructed the following query and now we send it to Elasticsearch to retrieve our precious documents by using the following command:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query" : {
    "match" : {
      "_all" : {
        "query" : "australian system",
        "operator" : "OR"
      }
    }
  }
}'
```

Because we are only interested in getting the `title` field (ElasticSearch will use the `_source` field to return the `title` field, because the `title` field is not stored) we've added the `fields=title` request parameter and of course we want it to be in human friendly formatting, so we add the `pretty` parameter.

However, the results were not as perfect as we would like them to be. The top documents were as follows (the whole response can be found in the `response_query_standard.json` file provided with the book):

```
{
  ...
  "hits" : {
    "total" : 562264,
    "max_score" : 3.3271418,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "3706849",
      "_score" : 3.3271418,
```

```
    "fields" : {
      "title" : "List of Australian Awards"
    }
  }, {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "26663528",
    "_score" : 2.9786692,
    "fields" : {
      "title" : "Australian rating system"
    }
  }, {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "7239275",
    "_score" : 2.9361649,
    "fields" : {
      "title" : "AANBUS"
    }
  },
  ...
]
}
```

While looking at the title of the second document it seems that it is more relevant than the first one, isn't it? Let's try to improve things.

The Multi match query

What we can do first is not use the `_all` field at all. The reason for this is because we need to tell Elasticsearch what importance each of the fields have. For example, in our case, the `title` field is more important than the content of the field, which is stored in the `text` field. In order to inform that to Elasticsearch we will use the `multi_match` query. In order to send such a query to Elasticsearch we will use the following command:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query" : {
    "multi_match" : {
      "query" : "australian system",
      "fields" : [ "title^100", "text^10", "_all" ]
    }
  }
}'
```

The top results of the preceding query were as follows (the whole response can be found in `response_query_multi_match.json` file provided with the book):

```
{
  ...
},
"hits" : {
  "total" : 562264,
  "max_score" : 5.3996744,
  "hits" : [ {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "7239222",
    "_score" : 5.3996744,
    "fields" : {
      "title" : "Australian Antarctic Building System"
    }
  }, {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "26663528",
    "_score" : 5.3996744,
    "fields" : {
      "title" : "Australian rating system"
    }
  }, {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "21697612",
    "_score" : 5.3968987,
    "fields" : {
      "title" : "Australian Integrated Forecast System"
    }
  },
  ...
]
}
```

What we did is instead of running the query against a single `_all` field, we chose to run it against the `title`, `text`, and the `_all` fields. In addition to that we've introduced boosting: the higher the boost value, the more important the field will be (default boost value for a field is 1.0). So we said that the `title` field is more important than the `text` field and the `text` field is more important than `_all`.

If you look at the results now, they seem to be a bit more relevant, but still not as good as we would like them to be. For example, look at the first and second documents. The first document's title is Australian Antarctic Building System and the second document's title is Australian rating system, and so on. I would like the second document to be higher than the first one.

Phrases comes into play

The next idea that should come into our minds is introduction of phrase queries so that we can overcome the problem that was described previously. However, we still need the documents that don't have phrases included in the results, just below the ones with the phrases present. So, we need to modify our query by adding the `bool` query on top. Our current query will come into the `must` section and the phrase query will go into the `should` section. An example command that sends the modified query would look as follows:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query" : {
    "bool" : {
      "must" : [
        {
          "multi_match" : {
            "query" : "australian system",
            "fields" : [ "title^100", "text^10", "_all" ]
          }
        }
      ],
      "should" : [
        {
          "match_phrase" : {
            "title" : "australian system"
          }
        },
        {
          "match_phrase" : {
            "text" : "australian system"
          }
        }
      ]
    }
  }
}'
```

Now if we would look at the top results, they are as follows (the whole response can be found in the `response_query_phrase.json` file provided with the book):

```
{
  ...
},
"hits" : {
  "total" : 562264,
  "max_score" : 3.5905828,
  "hits" : [ {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "363039",
    "_score" : 3.5905828,
    "fields" : {
      "title" : "Australian honours system"
    }
  }, {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "7239222",
    "_score" : 1.7996382,
    "fields" : {
      "title" : "Australian Antarctic Building System"
    }
  }, {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "26663528",
    "_score" : 1.7996382,
    "fields" : {
      "title" : "Australian rating system"
    }
  }, {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "21697612",
    "_score" : 1.7987131,
    "fields" : {
      "title" : "Australian Integrated Forecast System"
    }
  },
  ...
]
}
```

However, our results are still not as good as we would like to have them, although it is a bit better. That's because we don't have all the phrases matched. What we can do is introduce the `slop` parameter, which will allow us to define how many words in between can be present for a match to be considered as a phrase match. For example, our `australian system` query will be considered a phrase match for a document with title `australian education system` and with a `slop` of 1 or more. So let's send our query with the `slop` parameter present by using the following command:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query" : {
    "bool" : {
      "must" : [
        {
          "multi_match" : {
            "query" : "australian system",
            "fields" : [ "title^100", "text^10", "_all" ]
          }
        }
      ],
      "should" : [
        {
          "match_phrase" : {
            "title" : {
              "query" : "australian system",
              "slop" : 1
            }
          }
        },
        {
          "match_phrase" : {
            "text" : {
              "query" : "australian system",
              "slop" : 1
            }
          }
        }
      ]
    }
  }
}
```

And now let's look at the results (the whole response can be found in `response_query_phrase_slop.json` file provided with the book):

```
{
  ...
  "hits" : {
    "total" : 562264,
    "max_score" : 5.4896,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "7853879",
      "_score" : 5.4896,
      "fields" : {
        "title" : "Australian Honours System"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "363039",
      "_score" : 5.4625454,
      "fields" : {
        "title" : "Australian honours system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "11268858",
      "_score" : 4.7900333,
      "fields" : {
        "title" : "Wikipedia:Articles for deletion/Australian
university system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "26663528",
      "_score" : 3.6501765,
      "fields" : {
        "title" : "Australian rating system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "12324081",
      "_score" : 3.6483011,
```

```
    "fields" : {
      "title" : "Australian Series System"
    }
  },
]
}
}
```

It seems that the results are now better. However, we can always do some more tweaking and see if we can get some more improvements.

Let's throw the garbage away

What we can do now is we can remove the garbage from our results. We can do that by removing redirect documents and special documents (for example, the ones that are marked for deletion). In order to do that we will introduce a filter, not to mess with the scoring of other results, (because filters are not scored), but to be able to cache filter results (automatically by ElasticSearch) and reuse them in our queries. The command that sends our query with filters will look as follows:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d
'{
  "query" : {
    "bool" : {
      "must" : [
        {
          "multi_match" : {
            "query" : "australian system",
            "fields" : [ "title^100", "text^10", "_all" ]
          }
        }
      ],
      "should" : [
        {
          "match_phrase" : {
            "title" : {
              "query" : "australian system",
              "slop" : 1
            }
          }
        },
        {
          "match_phrase" : {
            "text" : {
              "query" : "australian system",
              "slop" : 1
            }
          }
        }
      ]
    }
  }
}
```

```

    }
  ]
}
},
"filter" : {
  "bool" : {
    "must_not" : [
      {
        "term" : {
          "redirect" : "true"
        }
      },
      {
        "term" : {
          "special" : "true"
        }
      }
    ]
  }
}
}'

```

And the results returned by it will look as follows:

```

{
  ...
  "hits" : {
    "total" : 474076,
    "max_score" : 5.4625454,
    "hits" : [ {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "363039",
      "_score" : 5.4625454,
      "fields" : {
        "title" : "Australian honours system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "12324081",
      "_score" : 3.6483011,
      "fields" : {
        "title" : "Australian Series System"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "13543384",
      "_score" : 3.6483011,

```

```
      "fields" : {
        "title" : "Australian Arbitration system"
      }
    }, {
      "_index" : "wikipedia",
      "_type" : "page",
      "_id" : "24431876",
      "_score" : 3.5823703,
      "fields" : {
        "title" : "Australian soccer league system"
      }
    },
    ...
  ]
}
```

Isn't it better?

And now we boost

If you ever need to boost the importance of the phrase queries that we've introduced we can do that by wrapping a phrase query with the `custom_boost_factor` query. For example, if we like to have a phrase for the `title` field to have a boost of 1000, we need to change the following part of the preceding query:

```
...
{
  "match_phrase" : {
    "title" : {
      "query" : "australian system",
      "slop" : 1
    }
  }
}
...
```

We will change it to the following one:

```
...
{
  "custom_boost_factor" : {
    "query" : {
      "match_phrase" : {
        "title" : {
          "query" : "australian system",
          "slop" : 1
        }
      }
    }
  }
}
```

```

    },
    "boost_factor" : 1000
  }
}
...

```

Making a misspelling-proof search

If you look back at the mappings, you will see that we have the `title` field defined as `multi_field` and one of the fields is analyzed with a defined `ngram` analyzer. By default, it will create bigrams, so from the word `system` it will create the `sy ys st` `te em` bigrams. Imagine that we could drop some of them during searching to make our search misspelling-proof. For the purpose of showing how we can do this, let's take a simple misspelled query sent with the following command:

```

curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query" : {
    "query_string" : {
      "query" : "austrelia",
      "default_field" : "title",
      "minimum_should_match" : "100%"
    }
  }
}'

```

The results returned by Elasticsearch would be as follows:

```

{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : null,
    "hits" : [ ]
  }
}

```

We've sent a query that is misspelled against the `title` field and because there is no document with the misspelled term we didn't get any results. So now let's leverage the `title.ngram` field capabilities and let's omit some of the bigrams, so that ElasticSearch can find some documents. Our command with a modified query looks as follows:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query" : {
    "query_string" : {
      "query" : "austrelia",
      "default_field" : "title.ngram",
      "minimum_should_match" : "85%"
    }
  }
}'
```

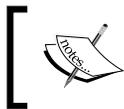
What we did is we changed the `default_field` property from `title` to `title.ngram` in order to inform ElasticSearch, the one with bigrams indexed. In addition to that, we've introduced the `minimum_should_match` property and we've set it to 85%. This allows us to inform ElasticSearch that we don't want all the terms produced by the analysis process to match, but only percent of them, we don't care which.



Lowering the value of the `minimum_should_match` property will give us more documents, but a less accurate search. Setting the value of the `minimum_should_match` property to a higher one will result in the decrease of the documents returned, but they will have more bigrams similar to the query ones and thus will be more relevant.

The top results returned by the preceding query are as follows (the whole result's response can be found in a file called `response_ngram.json` provided with the book):

```
{
  ...
},
"hits" : {
  "total" : 67633,
  "max_score" : 1.9720218,
  "hits" : [ {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "11831449",
    "_score" : 1.9720218,
    "fields" : {
      "title" : "Aurelia (Australia)"
    }
  }, {
    "_index" : "wikipedia",
    "_type" : "page",
    "_id" : "2568010",
    "_score" : 1.8479118,
    "fields" : {
      "title" : "Australian Kestrel"
    }
  },
  ...
]
}
```



If you would like to see how to use the ElasticSearch suggester to handle spellchecking please refer to the *Correcting user spelling mistakes* section in this chapter.

Drill downs with faceting

The last thing we want to mention about is faceting. You can do multiple things with it, for example; calculating histograms, statistics for fields, geo distance ranges, and so on. However, one thing that can help your users to get the data they are interested in is terms faceting. For example, if you go to `amazon.com` and enter the `kids shoes` query you would see the following screenshot:

The screenshot shows the Amazon search interface for the query "kids shoes". The search bar at the top contains "kids shoes" and a "Go" button. Below the search bar, there are several faceted navigation sections on the left side:

- Departments**: A dropdown menu set to "Department".
- Shoes**: A list of sub-categories including "Boys' Sneakers", "Girls' Sneakers", "Boys' Running Shoes", "Girls' Running Shoes", and "Girls' Sport Sandals".
- Shipping Option**: A link to "Free Super Saver Shipping".
- Brand**: A list of brand checkboxes including New Balance, Puma, DC, Keen, crocs, Timberland, Reebok, Saucony, Skechers, Tsukihoshi, TOMS, Stride Rite, and Adidas.

The main content area displays the search results for "kids shoes". It includes a "Related Searches" section with links to "boys shoes", "girls shoes", and "toddler shoes". Below this, it shows "Showing 1 - 16 of 70,717 Results" and "Top Results for 'kids shoes'". The top results are displayed as a grid of four shoe images with their respective category labels: "Boys' Sneakers", "Boys' Running Shoes", "Girls' Sneakers", and "Girls' Running Shoes".

The first product listing is for a "Puma Voltaic 3 V Kids Running Shoe (Toddler/Littl)". It features a large image of the shoe, a price range of "\$25.00 - \$69.91" with a Prime logo, and a star rating of 4.5 stars (53 reviews). Additional details include "Eligible for FREE S" and "Some sizes/colors".

You can narrow the results by brand (left side of the page). The list of brands is not static, it is generated on the basis of the results returned. We can achieve the same with terms faceting in ElasticSearch.

So now let's get back to our Wikipedia data. Let's assume that we like to allow our users to choose the category of documents they want to see after the initial search. In order to do that, we add the `facets` section to our query (however in order to simplify the example let's use the `match_all` query instead of our complicated one) and send the new query with the following command:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query" : {
    "match_all" : {}
  },
  "facets" : {
    "category_facet" : {
      "terms" : {
        "field" : "category.untouched",
        "size" : 10
      }
    }
  }
}'
```

As you can see, we've run the facet calculation on the `category.untouched` field, because terms faceting is calculated on the indexed data. If we run it on the `category` field we will get a single term in the faceting result, and we want the whole category to be present. The faceting section of the results returned by the preceding query looks as follows (the entire result's response can be found in a file called `response_query_facets.json` provided with the book):

```
"facets" : {
  "category_facet" : {
    "_type" : "terms",
    "missing" : 84741,
    "total" : 2054646,
    "other" : 1990915,
    "terms" : [ {
      "term" : "Living people",
      "count" : 46796
    }, {
      "term" : "Year of birth missing (living people)",
```

```
    "count" : 3450
  }, {
    "term" : "Australian rules footballers from Victoria
(Australia)",
    "count" : 2837
  }, {
    "term" : "Windows games",
    "count" : 2051
  }, {
    "term" : "English-language films",
    "count" : 1907
  }, {
    "term" : "Australian rugby league players",
    "count" : 1754
  }, {
    "term" : "Australian Labor Party politicians",
    "count" : 1630
  }, {
    "term" : "Unincorporated communities in West Virginia",
    "count" : 1369
  }, {
    "term" : "Unincorporated communities in Indiana",
    "count" : 1228
  }, {
    "term" : "Australian television actors",
    "count" : 709
  } ]
}
}
```

By default, we've got the faceting results sorted on the basis of the `count` property, which tells us how many documents belong to that particular category. Now if our user wants to narrow down its results to the `English-language films` category we need to send the following query:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d
'{
  "query" : {
    "match_all" : {}
  },
  "filter" : {
    "term" : {
      "category.untouched" : "English-language films"
    }
  },
  "facets" : {
    "category_facet" : {
      "terms" : {
        "field" : "category.untouched",
        "size" : 10
      },
      "facet_filter" : {
        "term" : {
          "category.untouched" : "English-language films"
        }
      }
    }
  }
}'
```

Notice one thing: in addition to the standard filter, we've also included a `facet_filter` section for our `category_facet` class. We need to do that in order to have faceting narrowed down: that's right, by default, standard filters that narrow down the query results don't narrow down faceting and we want to show our users the next level of nesting.

Summary

In this chapter we've learned how to correct user spelling mistakes both by using the `term suggester` and the `phrase suggester`, so now we know what to do in order to avoid empty pages that are results of misspelling. In addition to that, we've improved our users' query experience by improving the query relevance. We started with a simple query, and then we added multi match queries, phrase queries, boosts, and used query slop. We've seen how to filter our garbage results and how to improve phrase match importance. We've used n-grams to avoid misspellings as an alternate method to using Elasticsearch suggesters. We've also discussed how to use faceting to allow our users to narrow down the search results and thus simplify the ways in which they can find the desired documents or products.

In the next chapter we will finally concentrate on the Elasticsearch Java API. We will learn how to connect to local and external Elasticsearch clusters, we will learn how to index data by sending the documents in batches and one by one. We will also discuss update API that allows us to update the already indexed documents, and of course we won't forget about running queries which we will describe as well as we can. Finally we will talk about handling errors returned by the Elasticsearch API calls and how to run the administrative commands we already talked about.

8

ElasticSearch Java APIs

In the previous chapter, we learned how to improve a user's search experience in general. We saw how to use term suggester to suggest single terms and how to use the phrase suggester which allows us to suggest whole phrases. We also changed their configuration to match our needs. In addition to that, we've used a Wikipedia example and we've improved simple query relevance, by going step-by-step and introducing more and more complicated elements to it, such as phrases, filters, and facets. Finally we've touched the surface of search analytics and we've seen what software we can use to measure and observe. In this chapter we will focus on the ElasticSearch Java API. By the end of this chapter you will have learned how to use the ElasticSearch Java API to:

- Connect to the local or remote ElasticSearch cluster by using the client object
- Index your data, one document at a time and in batches
- Update your document contents
- Run different queries that are available in ElasticSearch
- Handle errors returned by ElasticSearch
- Run administrative commands to return the cluster's status and perform administrative tasks

Introducing the ElasticSearch Java API

Till now in our examples we have used the RESTful API, however it is not always the most convenient method of connecting to ElasticSearch and using this search server. In this chapter we would like to introduce you to the world of ElasticSearch Java API and show you what you can do with it.

Comparing to the ElasticSearch REST API, the Java API is less portable when it comes to integration and limits you to JVM-based programming languages. However, we are almost certain, that it is not a problem for you, because you can skip that chapter. In addition to less portability and JVM commitment, using the REST API you can connect to different major ElasticSearch versions if you are aware of the differences in REST endpoints and response returned by ElasticSearch. In case of the Java API you would have to include two different versions of ElasticSearch library in a single application, which is hard (it would require your own class loader implementation). Luckily there are not too many use cases, when you need to use multiple clusters that are not compatible.

Besides the preceding points discussed, for Java world, the API library provided by ElasticSearch is a ready-to-use solution. We can plug additional JAR archive to our project and start communicating with ElasticSearch without the need of a boilerplate code preparation for connecting the HTTP layer using JSON. The library provides two ways of connecting to ElasticSearch: by creating a local node that will join the cluster and by using transport. We will describe each of these methods later in this chapter.

In terms of possibilities, with the Java API, in addition to what we can do with the REST API there are some more of them: from inserting, getting and deleting documents, through search, bulk imports, obtaining information about statistics and the state of the cluster, and using administrating operation. There is also some operation not available in the traditional API similar to combine operations in bulk requests. In fact, the Java API is the same code that is internally used by ElasticSearch to perform its work.

Before we go on with the details about the Elasticsearch API, we would like to inform you about one important thing. The API we are going to discuss is very broad. You can expect that, since we have already written, Elasticsearch uses that API internally. Basically all the information and features we've described in the previous chapters and those described in the previously published *ElasticSearch Server* book are handled by using this API (and of course not only it). It means that showing all methods and all calls available by the Java API would result in duplication of the information we already discussed and this approach wouldn't be a good idea. To be honest, we tried. And we failed to do that, because it was massive and we were at the beginning. This chapter was bigger and bigger and we saw that we cannot describe everything. So we chose another way: instead of describing every method we will try to show how to utilize your existing knowledge and migrate it to Java word. We hope that by doing so, you will be able to see how to use the Java API provided by Elasticsearch and you will know where to look if the information provided by this book is insufficient.

The code

For this chapter, we created the Maven project (<http://maven.apache.org>) and every aspect we described is connected with a particular JUnit (<http://junit.org/>) test in that project. Of course, you can use your favorite IDE to look at and test the code: we have used Eclipse (<http://eclipse.org/>). From the command line you can run every single test using the following command (of course if you have Maven installed):

```
mvn -Dtest=com.elasticsearchserverbook.api.connect.ClientTest test
```

In the preceding example we executed tests defined in the `com.elasticsearchserverbook.api.connect.ClientTest` class. When you look at the code, note that it doesn't contain any error handling pieces. This is intended, because we wanted to increase readability. However, please remember to add error handling before you use that code in a real application.



There is one important thing you should remember while using the Java API. Because the API we are about to discuss uses a low-level way of communication with Elasticsearch nodes you have to be sure that the version of the API at the client side matches the version of the nodes. If the versions are different you may run into issues, for example, not being able to connect or problems with response deserialization.

Connecting to your cluster

As we already said there are two ways of connecting to the ElasticSearch cluster while using the Java API. Both of these methods utilize an appropriate instance of the `Client` (from `org.elasticsearch.client.Client`) interface: the main entry point for all functionalities exposed by the ElasticSearch API. We will discuss the `Client` interface in depth later in this chapter, now it's time to describe how to connect ElasticSearch in a more detailed way.

Becoming the ElasticSearch node

The first and the basic method of connecting to the ElasticSearch node can be surprising for someone who didn't have an experience with the ElasticSearch Java API. The idea is that our application can be a part of ElasticSearch cluster as any other node. Of course, (or at least in most cases) we don't want our application to hold any data or be a master node. However, all the other consequences of being a node apply. For example, the node we've created knows where the relevant shard is placed and how to route queries in the best way. This limits the number of round trips between client and the cluster to only the ones that are needed. Let's see an example of the Java code. First, let's look at the following imports statements:

```
import static org.elasticsearch.node.NodeBuilder.nodeBuilder;
import org.elasticsearch.client.Client;
import org.elasticsearch.node.Node;
```

We have the `Client` interface, the `Node` interface, and the `NodeBuilder` class to set up our connection. The following snippet creates a client instance:

```
Node node = nodeBuilder().clusterName("escluster2").client(true).
node();
Client client = node.client();
```

We have used the `NodeBuilder` class for creating a node. It allows us to prepare a desired configuration of the node and then set it up. The important information is the name of the cluster to which we want to connect to. We use the `clusterName()` method to provide that information. If we omit this, the default name will be used (which is `elasticsearch`) and you may accidentally connect to a different cluster than the one you actually want (if such cluster existing on the network is visible in the created `Client`). The second thing is the `client()` method call. Using it, we will be able to create a node which will act as a client node, so it will be not allowed to hold. If you omit this, strange things can happen unless you really know what you are doing. Your cluster can lose its data or some portion of it! In fact the cluster can decide to migrate some shards to your computer and when you turn off your `Node` instance, the cluster will see that one node is down. If you have several clients holding data, the parts of indices may be easily lost. For example if you don't have replicas for your index and the shard is migrated to your client, after shutdown you will lose your data for sure. So, please remember to always use the `client(true)` method unless you really know what you are doing.

It is also worth mentioning the possibility to call the `local(true)` method. Thanks to it, `ElasticSearch` will act as a standard node but only within the boundaries of the current Java virtual machine. It means that several nodes can form cluster but only if all of them are running in the same process. This is quite handy in the integration testing without need to set up external nodes for tests and need to ensure separation between the tests running in parallel.

 We said that the node created by the `NodeBuilder` class acts as every other node in the cluster. This also means that during bootstrap it reads the `elasticsearch.yml` configuration file from the classpath. Every setting (including the cluster's name) may be defined in this file, but these settings are overwritten by the values put explicitly in the code.

If you run the test that we've put it in the `com.elasticsearchserverbook.api.connect.ClientTest` class, you should see something as follows in the `ElasticSearch` server logs you are trying to connect to:

```
[2013-07-14 12:41:17,878] [INFO ] [cluster.service      ]
[Jon Spectre] added { [Hayden, Alex] [_bVTEUBVRDajOKmNp-Au0w]
[inet [/192.168.0.100:9301]] {client=true, data=false}, }, reason: zen-
disco-receive(join from node[ [Hayden, Alex] [_bVTEUBVRDajOKmNp-Au0w]
[inet [/192.168.0.100:9301]] {client=true, data=false}])
```

In our example, the `Jon Spectre` node detected the client connection from the node named, `Hayden, Alex`. After finishing its work, we should close the connection and leave the cluster. We can do that by running the following code:

```
node.close();
```

The preceding code shuts down the local node.

Using the transport connection method

This method involves connecting to the cluster without joining the cluster. You will see in this moment that the client can connect to several nodes and use them in the round-robin fashion. This means that every request will go to a different node and this node is required to route this request to a proper place (of course, this is in addition to any routing done by the ElasticSearch node itself). Let's look at the code now. As usual, we will start by showing which import statements are needed:

```
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.ImmutableSettings;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport.InetSocketTransportAddress;
```

And, now the main part of the code illustrating the transport method connection is as follows:

```
Settings settings = ImmutableSettings.settingsBuilder()
    .put("cluster.name", "escluster2").build();
TransportClient client = new TransportClient(settings);
client.addTransportAddress(new InetSocketTransportAddress("127.0.0.1",
    9300));
```

This is an easy way. First we created the needed settings by using the `settingsBuilder()` static method of the `ImmutableSettings` class. In this case, the only setting we need is the cluster's name (we will describe more transport client capabilities after this example). After that, we will create a `TransportClient` object using the settings we've created earlier. After creating the client object we should add the addresses of the nodes that we want our `TransportClient` class to connect to. We do that by calling the `addTransportAddress` method of the `TransportClient` class and passing the `InetSocketTransportAddress` object. In order to create the `InetSocketTransportAddress` object, we've provided the IP address of the server on which the ElasticSearch node is running and the port on which the transport layer of ElasticSearch is listening. Please note that the port is not `9200`, which we've used to communicate with ElasticSearch using the HTTP REST API, but it is `9300`, which is the default port on which the transport layer listens.

And now let's get back to the settings we need to set on the `TransportClient` class:

- `client.transport.sniff` (default: `false`): If set to `true`, `ElasticSearch` will read the information about the nodes that build the cluster, so you don't have to provide the addresses of all the nodes during transport client creation. `ElasticSearch` will be smart enough to detect active nodes and dynamically add them to the list.
- `client.transport.ignore_cluster_name` (default: `false`): If set to `true`, `ElasticSearch` will ignore the cluster name and connect to the cluster even if its name doesn't match. This can be dangerous, because you can end up using a cluster that you didn't intend to use in the first place.
- `client.transport.ping_timeout` (default: 5s): This is the time to wait for a ping's reply from the node. If the network latency between the client and the cluster is high and you have connectivity problems, you may need to increase the default value.
- `client.transport.nodes_sampler_interval` (default: 5s): This is the time interval for checking nodes availability. Similar to the previous property, if the network latency between the client and the cluster is high and you have connectivity problems, you may need to increase the default value.

 Please remember that similar to the previously described connection method, when we create the `TransportClient` instance, `ElasticSearch` will automatically try to read the `elasticsearch.yml` file from the classpath.

Choosing the right connection method

So now you already know how to connect to the `ElasticSearch` cluster using the Java API provided by `ElasticSearch`: both by creating a client node as well as using the transport client. Which of them is better? From one side, the first method complicates the startup sequence: the client has to join the cluster and establish connections with the other nodes. It takes time and resources. However, setup operations can be executed faster, because all the information about the cluster, its indices, and shards are available to such client node. On the other side, we have the `TransportClient` object. It starts faster and requires less number of resources, for example, fewer socket connections. However sending queries and data is more resource demanding: the `TransportClient` object is not aware of all the information regarding cluster and index topology, so it won't be able to send the data to the correct node right away: it will send the data to one of the initial transport nodes and the `ElasticSearch` will have to do the rest. In addition, the `TransportClient` object requires you to specify a list of initial addresses to connect to.

While choosing the right connection method one should remember one thing - it is not always possible to connect to your cluster using the first of the described methods. For example if the ElasticSearch cluster you want to connect is in a different network, the only way to connect to it using the Java API will be using the `TransportClient` object.

Anatomy of the API

As we said before, the client interface is the key for communication with the cluster. In practice, this means that with most of the operations you want to do, you will start with calling a method, for example, `prepareX()`. Let's start from a simple operation which will fetch a particular record. We start with the needed imports:

```
import org.elasticsearch.action.get.GetResponse;
import org.elasticsearch.client.Client;
```

And now, the following is the code snippet that will fetch a document from the library index, type `book`, and identifier `1`:

```
GetResponse response = client
    .prepareGet("library", "book", "1")
    .setFields("title", "_source")
    .execute().actionGet();
```

The preceding code did some interesting things. We called the `prepare` method, as warned before. In this case the `prepareGet()` method prepares the fetching request of a particular document with the particular type from the given index. In fact this method returns the builder object which allows us to set additional parameters and settings, in our case we set fields that should be returned for the document using the `setFields()` method. After preparation, we can create the request from the builder object to use it in future (using the `request()` method) or just fire our query using the `execute()` call, we do the second. And this is the most interesting part: the ElasticSearch API is asynchronous by nature. This means that the `execute()` call doesn't wait for the results returned by ElasticSearch and immediately returns control to the caller block and query is run in the background. In this example, we simply use the `actionGet()` method, which waits for query execution to end and return the fetched data. This was simple, but in more complicated systems this is not enough. Now let's look at the example of how we can use the asynchronous API. We start with the relevant import directive:

```
import org.elasticsearch.action.ListenableActionFuture;
```

Now let's modify our previous example to something as follows:

```

ListenableActionFuture<GetResponse> future = client
    .prepareGet("library", "book", "1")
    .setFields("title", "_source")
    .execute();
future.addListener(new ActionListener<GetResponse>() {
    @Override
    public void onResponse(GetResponse response) {
        System.out.println("Document: " + response.getIndex()
            + "/"
            + response.getType()
            + "/"
            + response.getId());
    }
    @Override
    public void onFailure(Throwable e) {
        throw new RuntimeException(e);
    }
});

```

The first line is similar to the previous example. We prepared the Get request and we've executed it, but we didn't want to wait for execution, so we did not call the `actionGet()` method. The `execute()` method returned an object of the `ListenableActionFuture` type, which we stored in the `future` variable. The `ListenableActionFuture` type defines the `addListener()` method allowing us to inform API, which code should be run when the response of the Get request is returned. In the example, we used an anonymous object inherited from the `ActionListener<GetResponse>` interface. This interface forces us to define two methods: `onResponse()` called by Elasticsearch when a request is executed successfully and the response is returned, and the `onFailure()` method is executed if an error occurs. Any code defined in these methods will be run in future, independently from the code defined after the `addListener()` call. The asynchronous task can be tested using the `isDone()` method to check if the request is already done or interrupted using `cancel()`.

Now let's look closer at the response object. Every API class has its own dedicated object holding appropriate information returned by Elasticsearch. In case of a GET request, the API methods include checking if the desired document was found, get information about the document, and its fields or document source. We will see some more examples in the rest of the chapter.

We now have the basic knowledge about conventions used by the ElasticSearch API to communicate with the cluster. In the next subchapters we will review all the available operations provided by the API.

 Please note that you are not forced to run the `actionGet()` method after the `execute()` method, just like we discussed in the *Anatomy of the API* section of this chapter. If you are interested in asynchronous behavior, you can still use the futures.

CRUD operations

We will have a closer look into the API operations, the **CRUD (create, retrieve, update, and delete document)** commands. Let's begin from the retrieve document call.

Fetching documents

Let's begin from the retrieve document call. We've already seen the following example when we discussed API anatomy:

```
GetResponse response = client
    .prepareGet("library", "book", "1")
    .setFields("title", "_source")
    .execute().actionGet();
```

While in preparation, after setting the index name, type name (may be null if we don't care what the type is), and identifier we will have the tool object. This builder object is an instance of `org.elasticsearch.action.get.GetRequestBuilder` and allows us to set the following additional information:

- `setFields(String)`: This method specifies which fields should be returned in the document. By default, this API method will return only document source. Note that lack of `_source` on the list of returned fields causes the `sourceXXX()` methods not to be working (as discussed in the next point).
- `setIndex(String)`, `setType(String)`, `setId(String)`: These methods set the index name, type of the document, and its identifier respectively. These methods are handy when you want to reuse the builder object or use the `prepareGet()` method version with no arguments.

- `setRouting(String)`: This method sets the routing value which is used to determine which shard should be used to execute the request. For more information about routing, please refer to the *Routing explained* section in *Chapter 4, Index Distribution Architecture*.
- `setParent(String)`: This method sets the parent document identifier in a parent-child relationship. This is equivalent to setting routing to the parent identifier.
- `setPreference(String)`: This method sets the query preference. For example, if the possible values are: `_local`, `_primary`, or the custom value. For more information about query preference, please refer to the *Query execution preference* section in *Chapter 4, Index Distribution Architecture*.
- `setRefresh(Boolean)`: This method controls if refreshing should be done before the operation. By default it is set to `false`, which means Elasticsearch will not perform the refresh operation before executing the GET request.
- `setRealtime(Boolean)`: This method is a request considered as a real-time get operation. By default, it is set to `true`, which informs Elasticsearch that the request is a real-time GET one.

When it comes to the response of the GET request, the most interesting methods for the `GetResponse` object are as follows:

- `exists()`: This method returns the information if document is found.
- `getIndex()`: This method returns the requested index name.
- `getType()`: This method returns the document type name.
- `getId()`: This method returns the requested document identifier.
- `getVersion()`: This method returns the document's version number.
- `isSourceEmpty()`: This method returns the information if the source of the document is available or not in the current and returned document. The possible returned values are `true` (when the source exists) and `false` (when the source is not existent in the returned document).
- `getSourceXXX()`: This family of methods allows to obtain the source document in various forms, for example as a text (`getSourceAsString()`), map (`getSourceAsMap()`), or bytes array (`getSourceAsBytes()`).
- `getField(String)`: This method returns the object describing field, which allows us to get field name, its value, or multiple values for multivalued fields.

Handling errors

Of course, we should always be prepared that something will go wrong, at least when we create software. Bad things can happen and in such cases. The important part of all the applications is the code responsible for handling errors. ElasticSearch handling of errors is slightly inconsistent (at least in 0.90.3 and previous). The API sometimes throws a checked exception in several calls' response containing a dedicated method to check the status of the operation, but in others these methods doesn't exist. So be prepared for some surprises. In case of all CRUD operations, ElasticSearch throws unchecked exceptions which inherit from `org.elasticsearch.ElasticSearchException`.

Indexing documents

We know how to fetch documents by using the GET requests, but it would be nice to actually have something to fetch from the index. So now we will look at how we can prepare indexing operations using the ElasticSearch Java API. As usual we will start with the needed imports:

```
import org.elasticsearch.action.index.IndexResponse;
import org.elasticsearch.client.Client;
```

And now we will create a simple code snippet that will index a document to the library index, under the book type and will give it an identifier of 2. The code looks as follows:

```
IndexResponse response = client.prepareIndex("library", "book", "2")
    .setSource("{ \"title\": \"Mastering ElasticSearch\"}")
    .execute().actionGet();
```

The general principle is the same as we discussed during the GET request: we will prepare the indexing request using a dedicated builder object (the object returned by the `prepareIndex()` call is of the `org.elasticsearch.action.index.IndexRequestBuilder` type), executing the request (`execute()` method) and waiting for response by calling the `actionGet()` method. The only ugly thing is the document source we've used to pass the whole document structure, but don't worry. The ElasticSearch API has some methods to programmatically create JSON documents which we will show. Note that the client object also has the `prepareIndex()` method version that doesn't require you to pass the document identifier. In such case, ElasticSearch will generate the identifier automatically.

The discussed builder object allows us to set the following information:

- `setSource()`: This method allows us to set the source of the document. The Elasticsearch API defines a few methods with the same name, but with a different argument type. In our example we've used the version where the input is a text: a `String` object. The other takes an array of bytes, key-value pairs, a map with fields and its values, or the `XContentBuilder` class allowing us to construct any JSON document.
- `setIndex(String)`, `setType(String)`, `setId(String)`: These methods are the same as in the GET request building, they allow us to set the index name, its type, and document identifier.
- `setRouting(String)`, `setParent(String)`: These methods are the same as the ones with the same names in the Get request API, please refer to the *Fetching documents* section in this chapter for further information about them.
- `setOpType()`: This method exists in two variants. The first takes the string argument: `index` or `create`. The second uses enumeration for this (`org.elasticsearch.action.index.IndexRequest.OpType.INDEX` or `org.elasticsearch.action.index.IndexRequest.OpType.CREATE`). Both variants of this method allow us to choose Elasticsearch's behavior when a document with a given identifier already exists in the index. The default behavior is to overwrite the previous document with the contents of the new one. When the argument is `create` (or an equivalent enumeration value), then the index operation will fail if a document with a given identifier already exists.
- `setCreate(Boolean)`: This method is a shortcut of `setOpType()`. When the argument is `true`, the indexing will fail if a document with the given identifier already exists. The default value is `false`: the new document will overwrite the previous or an already existing one.
- `setRefresh(Boolean)`: This method controls if refresh should be executed after the operation. By default it is set to `false`, which informs Elasticsearch not to perform the refresh operation before executing the GET request.

- `setReplicationType()`: This method exists in two variants. The first takes the `String` argument which can be one of the following values: `sync`, `async`, or `default`. The second variant of this method takes one of the enumeration values: `org.elasticsearch.action.support.replication.ReplicationType.SYNC`, `org.elasticsearch.action.support.replication.ReplicationType.ASYNC` or `org.elasticsearch.action.support.replication.ReplicationType.DEFAULT`. It allows us to control the replication type during indexation. By default the index operation is considered done when all the replicas have performed the operation (the `sync` value or type). The second option is returning from the operation without waiting for replicas (the `async` value or type). The third option is informing ElasticSearch to behave in a way according to the node's configuration (the `default` value or type).
- `setConsistencyLevel()`: This method defines how many replicas should be active to allow the index operation to be performed. The possible values are: `org.elasticsearch.action.WriteConsistencyLevel.DEFAULT` (use node setting), `org.elasticsearch.action.WriteConsistencyLevel.ONE` (only one replica is sufficient), `org.elasticsearch.action.WriteConsistencyLevel.QUORUM` (at least 50 percent + 1 defined replicas needs to be present), `org.elasticsearch.action.WriteConsistencyLevel.ALL` (all replicas must be available).
- `setVersion(long)`: This method allows us to define the version of the document which will be updated during indexing. If the document with a given identifier doesn't exist or the version is different, the update operation will fail. Thanks to this the application that makes sure that nobody changed the given document while reading the document and updating it.
- `setVersionType(VersionType)`: This method informs ElasticSearch which versioning type will be used: `org.elasticsearch.index.VersionType.INTERNAL` (the default) or `org.elasticsearch.index.VersionType.EXTERNAL`. This affects the way of comparing the version number by the server.
- `setPercolate(String)`: This method causes the percolator to be checked on the indexed document. The argument of this parameter is the query to limit percolated queries. The `*` value means that all queries should be checked.
- `setTimestamp(String)`: If the `_timestamp` field is enabled while mapping, ElasticSearch will automatically generate information about the last modification time. This method allows you to set your own value for this timestamp.

- `setTTL(long)`: If the `_ttl` field is enabled while mapping, Elasticsearch allows us to set the time in milliseconds, after which the document will be removed from the index.

The `IndexResponse` class returned by the indexing provides the following useful methods:

- `getIndex()`: This method returns the requested index name
- `getType()`: This method returns the document type name
- `getId()`: This method returns the indexed document's identifier
- `getVersion()`: This method returns the indexed document's version number
- `getMatches()`: This method returns the list of percolate queries that matched the document or `null` if no percolation was requested

Updating documents

The third operation we wanted to discuss from the CRUD operations is an update of the document. Again we are starting with necessary imports:

```
import org.elasticsearch.action.update.UpdateResponse;
import org.elasticsearch.client.Client;
```

In our example we will use the `Map` class and `Maps` helper provided by Elasticsearch and because of that we need the following additional imports:

```
import java.util.Map;
import org.elasticsearch.common.collect.Maps;
```

And now let's look at the code that will change the title field of our previously indexed document (the one that was indexed in the previous *Indexing documents* section):

```
Map<String, Object> params = Maps.newHashMap();
params.put("ntitle", "ElasticSearch Server Book");

UpdateResponse response = client.prepareUpdate("library", "book", "2")
    .setScript("ctx._source.title = ntitle")
    .setScriptParams(params)
    .execute().actionGet();
```

Again, as in the GET and create operations, the general rule is the same. We run the `prepareUpdate` method and we pass the index name (`library` in our case), document type (`book` in our case), and the identifier of the document we want to update (`2` in our case). After that we set the script that will be used by ElasticSearch by using the `setScript` method, we set the script parameters by using the `setScriptParams` method and we run the update request waiting for it to finish and return the `UpdateResponse` object.

The update request builder (which is an object of the `org.elasticsearch.action.update.UpdateRequestBuilder` type) contains the following useful methods we can use:

- `setIndex(String)`, `setType(String)`, `setId(String)`: These methods are same as in the GET request building, allows us to set the index name, its type, and document identifier.
- `setRouting(String)`, `setParent(String)`: These methods are the same as discussed in the Get request API, please refer to the *Fetching documents* section in this chapter for further information about these methods.
- `setScript(String)`: This method sets the script used for changing the document we want to alter.
- `setScriptLang(String)`: This method sets the information about the type of language used in the provided script.
- `setScriptParams(Map<String, Object>)`: As you are aware, the script we pass to ElasticSearch can use variables, whose values can be defined using this method. The keys of the map, provided as the method's arguments are the parameter names and the values are those parameter values.
- `addScriptParam(String, Object)`: Instead of using `setScriptParams`, you can use a series of the `addScriptParam` method calls which in every occurrence defines one variable. Please note that repetition of the parameter with the same name will overwrite the previous definition.
- `setFields(String...)`: This method sets which fields should be returned in the document as a GET response. Note that if you set the fields by yourself you should also define `_source` on the list to make the `sourceXXX()` methods working.

- `setRetryOnConflict(int)`: The default is 0. In Elasticsearch updating a document means retrieving the previous value, modifying its structure, removing the previous document and indexing the new, updated one. It means that during this process of fetching and writing the new value, the target document may change, for example by the other application. Elasticsearch detects this by comparing the document's version number and returns error. As an alternative, it may retry the operation. The number of this retries can be defined by this method.
- `setRefresh(Boolean)`: This method controls if refresh should be executed after the operation. By default it is set to `false`, which means that the refresh operation will not be triggered.
- `setReplicationType()`: This method exists in two variants. The first takes the `String` argument which can be one of the following values: `sync`, `async`, or `default`. The second variant of this method takes one of the enumeration values: `org.elasticsearch.action.support.replication.ReplicationType.SYNC`, `org.elasticsearch.action.support.replication.ReplicationType.ASYNC`, or `org.elasticsearch.action.support.replication.ReplicationType.DEFAULT`. It allows us to control the replication type during update. By default the update operation is considered done when all the replicas have performed the operation (the `sync` value or type). The second option is returning from the operation without waiting for the replicas (the `async` value or type). The third option is informing Elasticsearch to behave in a way according to node's configuration (the `default` value or type).
- `setConsistencyLevel()`: This method defines how many replicas should be active to allow the update operation to be performed. The possible values are: `org.elasticsearch.action.WriteConsistencyLevel.DEFAULT` (use node setting), `org.elasticsearch.action.WriteConsistencyLevel.ONE` (only one replica is sufficient), `org.elasticsearch.action.WriteConsistencyLevel.QUORUM` (at least 50 percent + 1 defined replicas needs to be present), `org.elasticsearch.action.WriteConsistencyLevel.ALL` (all replicas must be available).
- `setPercolate(String)`: This method causes the percolator to be run on the indexed document. The parameter is the query to limit percolated queries. The `*` value means that all the queries are to be checked.

- `setDoc()`: This family of methods allows to set the partial document, which should be merged with a document from the index. This document will be ignored if the script is defined. ElasticSearch provides versions of this method which requires document in `String`, an array of bytes, `XContentBuilder`, or map of fields. In addition to that the document may be given as the `IndexRequest` object.
- `setUpsertRequest()`: This family of methods defines the document that should be indexed when a requested document doesn't exist in the index. The possible argument types are the same as in the `setDoc()` methods.
- `setSource()`: Instead of using multiple methods such as `setScript()` and `setScriptParams()`, you can prepare the whole request body and it will be parsed to set values for the script, script params, doc, upsert, lang, and information about the document as upsert (as given in the next point).
- `setDocAsUpsert(Boolean)`: The default: `false`. If set to `true`, if the document doesn't exist, the value used in `setDoc()` methods will be used as the new document.

The `UpdateResponse` class object returned by the update request provides the following information:

- `getIndex()`: This method returns the requested index name.
- `getType()`: This method returns the document type name.
- `getId()`: This method returns an updated document identifier.
- `getVersion()`: This method returns an updated document version.
- `getMatches()`: This method returns the list of percolate queries that matched the document or `null` if no percolation was requested.
- `getResult()`: This method returns the GET result, which holds information about the updated document. Note that this request will be available if you have used the `setFields()` method while creating the request.

Deleting documents

The last basic operation in CRUD is the deletion of a document. You probably can easily guess how the code should look like. In imports there is a class responsible for the reply:

```
import org.elasticsearch.action.delete.DeleteResponse;
import org.elasticsearch.client.Client;
```

The main code is also obvious, which is as follows:

```
DeleteResponse response = client.prepareDelete("library", "book", "2")
    .execute().actionGet();
```

Just as we did before, we will prepare the request: the delete request now and we will give it the information about the index, the type, and the document identifier. The request builder (instance of `org.elasticsearch.action.delete.DeleteRequestBuilder`) has no new methods apart from the ones we discussed before, but let's recall them once again:

- `setIndex(String), setType(String), setId(String)`: As in all the other builders we discussed, the mentioned methods allow us to set the index name, its type, and the document identifier.
- `setRouting(String), setParent(String)`: These methods are the same as in the GET request, please refer to the *Fetching documents* section in this chapter for further information about these methods.
- `setRefresh(Boolean)`: This method controls if refresh should be executed after the operation. By default it is set to `false`, which means that the refresh operation will not be executed.
- `setVersion(long)`: This method allows us to define the version of the document, which will be deleted during document deletion. If the document with the given identifier doesn't exist or the version is different, the delete operation fails. Thanks to this, the application makes sure that nobody changes the given document.
- `setVersionType(VersionType)`: This method informs ElasticSearch which versioning type will be used: `org.elasticsearch.index.VersionType.INTERNAL` (the default value) or `org.elasticsearch.index.VersionType.EXTERNAL`. This affects the way of comparing the version number by the server.
- `setReplicationType()`: This method exists in two variants. The first takes the `String` argument which can be one of the following values: `sync`, `async`, or `default`. The second variant of this method takes one of the enumeration values: `org.elasticsearch.action.support.replication.ReplicationType.SYNC`, `org.elasticsearch.action.support.replication.ReplicationType.ASYNC`, or `org.elasticsearch.action.support.replication.ReplicationType.DEFAULT`. It allows us to control the replication type during deletion. By default the delete operation is considered done when all replicas have performed the operation (the `sync` value or type). The second option is returning from the operation without waiting for replicas (the `async` value or type). The third option is informing ElasticSearch to behave in a way according to the node's configuration (the default value or type).

- `setConsistencyLevel()`: This method defines how many replicas should be active to allow the delete operation to be performed. The possible values are: `org.elasticsearch.action.WriteConsistencyLevel.DEFAULT` (use node setting), `org.elasticsearch.action.WriteConsistencyLevel.ONE` (only one replica is sufficient), `org.elasticsearch.action.WriteConsistencyLevel.QUORUM` (at least 50% + 1 defined replicas needs to be present), `org.elasticsearch.action.WriteConsistencyLevel.ALL` (all replicas must be available).

The `DeleteResponse` response, returned after running the delete operation contains the following methods:

- `getIndex()`: This method returns the requested index name
- `getType()`: This method returns the document type name
- `getId()`: This method returns the deleted document's identifier
- `getVersion()`: This method returns the deleted document's version number
- `isNotFound()`: This method returns `true` if the request did not find the document for deletion

Querying ElasticSearch

Right now we should be able to use the ElasticSearch Java API to perform basic CRUD operations, such as creating and deleting documents. But wouldn't it be nice to be able to search the data we have in our indices? Let's look at the search possibilities exposed by the ElasticSearch Java API.

Preparing a query

You may be already expecting this, but again the structure code that allows us to make a query using the Java API is very similar, almost identical when compared it with CRUD operations. In the base query the following imports will be handy:

```
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.search.SearchHit;
```

Besides the `client` class and response, you will see the `SearchHit` class responsible for holding a single document that matches the criteria and additional metadata as a score value. The following code shows a simple query returning all the documents from the `library` index. For each document, the `title` and `_source` fields are fetched. After getting the response back, we iterate over the returned page of results showing identifiers of the matched documents. Let's look at the code that does that:

```
SearchResponse response = client.prepareSearch("library")
    .addFields("title", "_source")
    .execute().actionGet();

for(SearchHit hit: response.getHits().getHits()) {
    System.out.println(hit.getId());
    if (hit.getFields().containsKey("title")) {
        System.out.println("field.title: "
            + hit.getFields().get("title").getValue());
    }
    System.out.println("source.title: "
        + hit.getSource().get("title"));
}
```

When we iterate over the returned documents, we read the `title` field's value two times. The first value of the `title` field is read from the `title` field of the document, in this case the query must have the definition of the returned fields, as you can see we can use the `addFields()` method to define which fields will be returned. The second time we read the `title` field contents, we use the source of the document. Please remember that by default, if no fields are defined, `_source` will be returned. In our case we need to add this in addition to the `title` field. Of course, `_source` should be enabled in mapping which is the default behavior.

Building queries

The empty query which matches all the documents is boring, you have to admit it. You know Elasticsearch provides many ways for querying the index (or multiple indices) to let us get the desired results. The API takes complete advantage of these possibilities and exposes a set of `setQuery()` methods. Similar to the already described `setSource()` method, they allow you to use various forms of setting the query: as a `String`, as an array of bytes, `Map`, or `XContentBuilder`. But we will focus on using the `QueryBuilder` interface, precisely speaking on the `QueryBuilders` class which helps us to create the `QueryBuilder` implementations. Let's look at a simple example. As usual we will start with the needed code imports:

```
import org.elasticsearch.index.query.QueryBuilder;
import org.elasticsearch.index.query.QueryBuilders;
```

And now we are ready to build queries. Let's create a dismax query that will combine two queries: a simple term query and a prefix query. The following example does that by using the `QueryBuilders` class:

```
QueryBuilder queryBuilder = QueryBuilders
    .dismaxQuery()
    .add(QueryBuilders.termQuery("title", "Elastic"))
    .add(QueryBuilders.prefixQuery("title", "el"));

System.out.println(queryBuilder.toString());
SearchResponse response = client.prepareSearch("library")
    .setQuery(queryBuilder)
    .execute().actionGet();
```

Thanks to the `QueryBuilders` helper class. We've prepared a dismax query which contains two queries: a term query and a prefix query. In our example we have used the `toString()` method which allows us to show the JSON representation of the generated query which could be used with the REST API:

```
{
  "dis_max" : {
    "queries" : [ {
      "term" : {
        "title" : "Elastic"
      }
    }, {
      "prefix" : {
        "title" : "el"
      }
    } ]
  }
}
```

After creating the query, we've created the search request, applied `QueryBuilder` containing our query to it using the `setQuery()` method and we've executed that request. And that's it. The `QueryBuilder` object has many methods for building many kinds of queries including those which combine multiple queries together, as the previously mentioned dismax query builder where we passed another builder to the `add()` method. Let's briefly go through the `QueryBuilder` class to see what it offers. All of these methods return their own builders with dedicated methods.



Please note that due to a large number of functions, queries, parameters and others, we described only a few queries which can be built using the API. Remember that the principle for building all the queries is the same. We also assume that you already know and understand the RESTful version of the API. If not, take a look at the documentation available at the ElasticSearch website or read our previous *ElasticSearch Server* book.

Using the match all documents query

The simplest query available in ElasticSearch and thus in its Java API is the `matchAllQuery()` query. As the name suggests, this query matches all the documents from the index (or indices) and allows you to set the boost value and norms for the given field. As you can guess these parameters are directly available in the builder. For example:

```
queryBuilder = QueryBuilders.matchAllQuery()
    .boost(11f).normsField("title");
```

The preceding code snippet generates the following query in the JSON format:

```
{
  "match_all" : {
    "boost" : 11.0,
    "norms_field" : "title"
  }
}
```

The match query

Let's do something opposite of what we just did and instead of making a query with the Java API and showing the JSON we will try to build the Java API query that matches the following match query in the JSON format:

```
{
  "match" : {
    "message" : {
      "query" : "a quick brown fox",
      "operator" : "and",
      "zero_terms_query": "all"
    }
  }
}
```

When looking at the `QueryBuilders` class you will easily find the `matchQuery()` method which takes two arguments: the field name and its value. The methods for setting the operator and `zero_terms_query` are also available. The Java version of the preceding JSON query would look as follows:

```
queryBuilder = QueryBuilders
    .matchQuery("message", "a quick brown fox")
    .operator(Operator.AND)
    .zeroTermsQuery(ZeroTermsQuery.ALL);
```

It is quite simple, isn't it?

Using the geo shape query

Let's look at another example and again we will start with the query in the JSON format. Let's assume that we would like to build the following query using the Java builders available in ElasticSearch:

```
{
  "query": {
    "geo_shape": {
      "location": {
        "shape": {
          "type": "envelope",
          "coordinates": [[13, 53],[14, 52]]
        }
      }
    }
  }
}
```

As you probably expect right now, for this kind of query the `QueryBuilders` class also has an appropriate method. This time it is the `geoShapeQuery()` method. So, in order to build the preceding JSON query using the `QueryBuilders` class, we can write a code similar to the following one:

```
queryBuilder = QueryBuilders.geoShapeQuery("location",
    ShapeBuilder.newRectangle()
        .topLeft(13, 53)
        .bottomRight(14, 52)
        .build());
```

The `geoShapeQuery()` method takes two arguments: the name of the field and the `Shape` object, which can be created using the `ShapeBuilder` helper. Please remember, if some method takes an argument which is a dedicated type, try to find the builder for this object. It really helps in coding. Note that the presented example is true for Version 0.90.3 of `ElasticSearch`. For the 1.0 version the syntax is a little bit different and looks as follows:

```
QueryBuilders.geoShapeQuery("location",
    ShapeBuilder.newEnvelope()
        .topLeft(13, 53)
        .bottomRight(14, 52)
    );
```

Note that the `ShapeBuilder` class now in the 1.0 version of `ElasticSearch` is in a different package as compared to the previous versions.

Paging

When the result list is large, it is necessary to use paging to limit the number of documents fetched in a single request and go through the subsequent ones. In the API, it is simple because the `SearchRequestBuilder` class provides two dedicated methods for this: the `setFrom(int)` method for declaring offset and the `setSize(int)` method for defining the page size. Let's take a look at the following example:

```
SearchResponse response = client.prepareSearch("library")
    .setQuery(QueryBuilders.matchAllQuery())
    .setFrom(10)
    .setSize(20)
    .execute().actionGet();
```

The preceding request will return 20 documents (of course if they exist) skipping the first 10 documents. Note that, by default `ElasticSearch` returns the first 10 documents. The `SearchHits` class contains the `totalHits()` method which allows us to obtain information about the total number of documents fulfilling the criteria of our query.

Sorting

In order to define sorting, the ElasticSearch Java API provides two variants of the `addSort()` method. Let's look at the following example which illustrates both the versions of the method:

```
SearchResponse response = client.prepareSearch("library")
    .setQuery(QueryBuilders.matchAllQuery())
    .addSort(SortBuilders.fieldSort("title"))
    .addSort("_score", SortOrder.DESC)
    .execute().actionGet();
```

The first version used accepts the `SortBuilder` abstract class. Looking at the other places in the API, you can guess that there is a dedicated class for creating a particular instance of `SortBuilder`, the `SortBuilders` class. In the previous example, we saw the definition of sorting by the `title` field using the `fieldSort(String)` method. The second example of the sort method takes the name of the field and the `SortOrder` enumeration value which defines the sorting order.

In addition to the described sort methods, ElasticSearch provides a method to allow script-based sorting: the `scriptSort(String, String)` method, and spatial-based sorting: the `geoDistanceSort(String)` method.

Filtering

From the API's perspective, creating filters is very similar to creating queries and this is true for both the Java API as well as the REST API. Let's add a few filters by using the Java API:

```
FilterBuilder filterBuilder = FilterBuilders
    .andFilter(
        FilterBuilders.existsFilter("title").filterName("exist"),
        FilterBuilders.termFilter("title", "elastic")
    );
SearchResponse response = client.prepareSearch("library")
    .setFilter(filterBuilder)
    .execute().actionGet();
```

We have created the `FilterBuilder` class using the `FiltersBuilders` utility. You will again spot the same pattern as in the other part of the `ElasticSearch` API. In this example, we've created two filters (exists filter and term filter) and wrapped them together using the `andFilter()` method. If we print the contents of the `FilterBuilder` class in order to see how the JSON form of our query will look, we would see something as follows:

```
{
  "and" : {
    "filters" : [ {
      "exists" : {
        "field" : "title",
        "_name" : "exist"
      }
    }, {
      "term" : {
        "title" : "elastic"
      }
    } ]
  }
}
```

You can check this by calling the `toString()` method on our builder.

The next lines after creating the filter are responsible for query execution. And again this is almost the same as with the query execution handling with one change: we've set the filter using the `setFilter()` method.

It is worth mentioning that the API provides several methods named `setFilter()`. The remaining six allows us to define the filters using JSON in various forms (byte array, `String`, `XContentBuilder`) or generate it from `Map`. In our example we've used the `filterName()` methods that allows us to set the name of a filter. Thanks to this, you can check which filters matched a particular document from the result set returned by `ElasticSearch`. For this, look at the `matchedFilters()` method in the `SearchHit` interface. As for searching, the list of possible filter types is very extensive and all the ones that are available in `ElasticSearch` can be built using the Java API.

Faceting

Defining faceting is in general the same as filtering, so it is very similar to what we've already seen. In detail, we have some differences in the naming convention, but once you are used to the Java API of ElasticSearch you'll be able to easily make requests with faceting. Of course, we will use `FacetBuilders` in order to get the `FacetBuilder` object and we will use the `setFacets()` method to add facets to our request. The mentioned method gets JSON or Map objects and constructs facets using that information. Let's now look at an example code:

```
FacetBuilder facetBuilder = FacetBuilders
    .filterFacet("test")
    .filter(FilterBuilders.termFilter("title", "elastic"));

SearchResponse response = client.prepareSearch("library")
    .addFacet(facetBuilder)
    .execute().actionGet();
```

We've created the filter facet as an example, but as you may have guessed the API provides multiple possibilities of using faceting, just like the REST API does from the terms facet, through query, range, geo, statistical to histogram faceting. Note that it is different to the previous example, the `toString()` method in builder presents nothing interesting. The `SearchResult` object contains the `getFacets()` method which allows you to analyze the data returned by faceting.

Highlighting

In a modern, search-based application, searching in the fields containing a lot of text, showing the user which and where the phrase was found is a valuable feature. The Java API also exposes the highlighting functionality, which allows us to do that. Let's look at the following example:

```
SearchResponse response = client.prepareSearch("wikipedia")
    .addHighlightedField("title")
    .setQuery(QueryBuilders.termQuery("title", "actress"))
    .setHighlighterPreTags("<1>", "<2>")
    .setHighlighterPostTags("</1>", "</2>")
    .execute().actionGet();
```

In the example, we created a simple query searching for the `actress` word in the `title` field. The `title` field is also this field which we would like to see as highlighted. Because of that, we use the `addHighlightedField()` method call and we pass it the field name we are interested in. We also defined how the words that are found should be marked: in the example we chose simple HTML-like tagging with the numbers.

In the response, every search hit contains the information about highlighting connected with such hit, so in order to print out the highlighted fragments we can use a code similar to the following one:

```
for(SearchHit hit: response.getHits().getHits()) {
    HighlightField hField = hit.getHighlightFields().get("title");
    for (Text t : hField.fragments()) {
        System.out.println(t.string());
    }
}
```

The preceding code snippet would result in the following output:

```
Academy Award for Best Supporting <1>Actress</1>
```

Please note the highlighted term. It was surrounded with the tags `<1>` and `</1>`, which we've defined while creating our query by highlighting it.

Suggestions

In the *Correcting user spelling mistakes* section of *Chapter 7, Improving the User Search Experience* we've have used the following query to show how a suggester works:

```
{
  "query" : {
    "match_all" : {}
  },
  "suggest" : {
    "first_suggestion" : {
      "text" : "graphics designer",
      "term" : {
        "field" : "_all"
      }
    }
  }
}
```

Now we will try to rewrite this JSON query into the Java API. Take a look at the following example code:

```
SearchResponse response = client.prepareSearch("wikipedia")
    .setQuery(QueryBuilders.matchAllQuery())
    .addSuggestion(new TermSuggestionBuilder("first_suggestion")
        .text("graphics designer")
        .field("_all")
    )
    .execute().actionGet();
```

The first part with `setQuery()` is nothing interesting, because we are just using the standard `match_all` query. In the next line, we construct `TermSuggestionBuilder` by calling its constructor and passing in the suggester name: we will need it in order to see which suggester returned what results. The last thing is setting the field used to generate suggestion (the `_all` field). Now let's check the response our suggester generated:

```
for( Entry<? extends Option> entry : response.getSuggest()
    .getSuggestion("first_suggestion").getEntries()) {
    System.out.println("Check for: "
        + entry.getText()
        + ". Options:");
    for( Option option : entry.getOptions()) {
        System.out.println("\t" + option.getText());
    }
}
```

As you can see for every named suggestion you can get a list of available options and exactly the same information as described in the *Correcting user spelling mistakes* section of *Chapter 7, Improving the User Search Experience*.

Counting

In the examples we've shown previously we were interested in the documents that match a given criteria. However, sometimes the list of documents doesn't matter: we only want to know the number of them that match our criteria. In such cases, we should use the count request because it is wise in terms of performance: we skip sorting and retrieving the documents from the index. The example code for getting the document count looks as follows:

```
CountResponse response = client.prepareCount("library")
    .setQuery(QueryBuilders.termQuery("title", "elastic"))
    .execute().actionGet();
```

The difference of counting from searching is the `prepareCount()` method call instead of `prepareSearch()`. That's it!

Scrolling

If you want to use the scrolling feature in order to fetch large data sets using the Elasticsearch Java API, please look at the following code example:

```
SearchResponse responseSearch = client.prepareSearch("library")
    .setScroll("1m")
    .setSearchType(SearchType.SCAN)
    .execute().actionGet();

String scrollId = responseSearch.getScrollId();
SearchResponse response = client.prepareSearchScroll(scrollId)
    .execute().actionGet();
```

In the example you will see two requests: the first one defines the query and the scroll attributes such as the time when our scroll should be valid (using the `setScroll()` method). In addition to that, we need to specify the search type: we switch to the scan mode that omits sorting and fetching documents, so the first query returns only the number of documents found and the scroll identifier which can be used in the second query.

Then we retrieve the identifier of the scroll by using the `getScrollId()` method of the `SearchResponse` object. Finally, we can run the second query using the `prepareSearchScroll()` method. We pass it with the already retrieved scroll identifier and in result we get the `SearchResponse` object that will contain our documents set.

Performing multiple actions

Till now we've looked at how to perform single action requests, such as the search operations or indexing documents one by one. However as you know, Elasticsearch provides bulk functionalities, which enables us to index multiple documents with a single request, run multiple queries, or delete documents using a query. Let's look at how to use these functionalities by using the Elasticsearch Java API.

Bulk

ElasticSearch bulk API allows you to pack together multiple indexes, delete and update requests in one request, and analyze response of these requests separately. Let's look at the following example:

```
BulkResponse response = client.prepareBulk()
    .add(client.prepareIndex("library", "book", "5")
        .setSource("{ \"title\" : \"Solr Cookbook\"}")
        .request())
    .add(client.prepareDelete("library", "book", "2").request())
    .execute().actionGet();
```

The preceding request will add one document to the library book (the one with the identifier of 5) and will delete one document (the one with the identifier of 2). In response, we get an array of the `org.elasticsearch.action.bulk.BulkItemResponse` objects available by calling the `getItems()` method. You should loop through this array and check the status of particular operation using the `isFailed()` method (which returns `true` if errors happened during a particular operation) and `getFailure()` for additional information about the failure. Each of this responses also contains the `getResponse()` method which returns the `IndexResponse` or `DeleteResponse` objects, which you already know from the index and delete operations.

The delete by query

The API gives us the possibility of deleting multiple documents: the ones that match a given query. Let's look at the following example:

```
DeleteByQueryResponse response = client.
prepareDeleteByQuery("library")
    .setQuery(QueryBuilders.termQuery("title", "ElasticSearch"))
    .execute().actionGet();
```

The preceding code will result in deleting the documents from the `library` index: the ones that match the given term query.

Multi GET

While the bulk index allows us to index multiple documents, the multi get request allows us to group several GET requests in a single call and thus return multiple documents in response. Let's look at the following example:

```
MultiGetResponse response = client.prepareMultiGet()
    .add("library", "book", "1", "2")
    .execute().actionGet();
```

It will result in returning the documents from the `library` index, with the book type that has identifiers 1 or 2.

The response object for the preceding query allows us to use the `getResponses()` method which returns an array of the `org.elasticsearch.action.get.MultiGetItemResponse` object. The mentioned object is similar to the one already described in the bulk API, but `getResponse()` returns the `GetResponse` object describing a particular GET operation.

Multi Search

The last API we want to mention is the one that allows us to group multiple query requests together. As you can easily guess it groups multiple searches in one requests, for example let's look at the following code:

```
MultiSearchResponse response = client.prepareMultiSearch()
    .add(client.prepareSearch("library", "book").request())
    .add(client.prepareSearch("news").
        .setFilter(FilterBuilders.termFilter("tags", "important")))
    .execute().actionGet();
```

As in the multi get operation, the response contains the `getResponses()` method which returns an array of the `org.elasticsearch.action.search.MultiSearchResponse.Item` objects. As you can guess, this object has the `getResponse()` method returning a particular search response and the `isFailure()` method informing us about the status of the operation.

Percolator

The percolator is a search turned upside down. We can use the document to find queries that match that document. Let's say that we have the index called `prcltr`. Then we can index the query to the `_percolator` index with the `prcltr` type by using the following code example:

```
client.prepareIndex("_percolator", "prcltr", "query:1")
    .setSource(XContentFactory.jsonBuilder()
        .startObject()
        .field("query",
            QueryBuilders.termQuery("test", "abc"))
        .endObject())
    .execute().actionGet();
```

In the preceding example our query has an identifier, `query:1` and it is a simple query checking the `test` field for the `abc` value. Now that we have our percolator prepared, we can send a document to it by using the following code fragment:

```
PercolateResponse response = client.preparePercolate("prcltr", "type")
    .setSource(XContentFactory.jsonBuilder()
        .startObject()
            .startObject("doc")
                .field("test").value("abc")
            .endObject()
        .endObject())
    .execute().actionGet();
```

As you see the document that we've sent should match the query stored in the percolator. We can check it using the `getMatches()` method. The following simple code fragment illustrates how that can be achieved:

```
for (String match : response.getMatches()) {
    System.out.println("Match: " + match);
}
```

The preceding snippet should return the `query:1` value, an identifier of our query. This is the only one example which illustrates how the percolator works. There are more features and all of them are available in the Java API.

ElasticSearch 1.0 and higher

There is one thing you should know. While writing this book, the version of ElasticSearch was 0.90.3. Because of the changes that will be done in ElasticSearch 1.0 the preceding examples will not even be compiled. The `_percolator` index is not an index anymore but type. So when using ElasticSearch 1.0 the preceding index operation should look as follows:

```
client.prepareIndex("prcltr", "_percolator", "query:1")
```

The percolate interface will also be changed. In ElasticSearch 1.0 the query to the percolator should look as the one given in the following example:

```
PercolateResponse response = client.preparePercolate()
    .setIndices("prcltr")
    .setDocumentType("type")
    .setPercolateDoc(PercolateSourceBuilder
        .docBuilder()
            .setDoc(XContentFactory.jsonBuilder())
```

```
.startObject()
  .field("test").value("abc")
  .endObject())
.execute().actionGet();
```

There are also some changes done in the way the results are retrieved. In ElasticSearch 1.0, the percolator returns the `Match` type object we can use for getting the matched queries. So our code that fetches the matched queries will look as follows:

```
for (Match queryName : response.getMatches()) {
    System.out.println("Match: " + queryName.getId());
}
```

The explain API

The last thing about querying ElasticSearch is the explain API. The explain API helps us in finding relevant issues and looking why a given document was or wasn't found. Let's look at the following example:

```
ExplainResponse response = client
    .prepareExplain("library", "book", "1")
    .setQuery(QueryBuilders.termQuery("title", "elastic"))
    .execute().actionGet();
```

What we did should be obvious by know, but let's discuss it. We prepared the explain request by running the `prepareExplain` method. In our example we are checking if the document with identifier equals to 1 will be found in reply for a simple term query.

The response consists of the `getExplanation()` method which returns an `org.apache.lucene.search.Explanation` object. This Lucene object describes the way the score was calculated. The structure of the information this object contains is dependent on how the query is constructed and which components query contains. Unfortunately information is sometimes almost cryptic, but very useful to determine why a given document was found for this particular query.

Building JSON queries and documents

Let's now take a step back from running requests themselves and look on how to construct JSON queries and documents. We will start recalling the example by creating a new document:

```
IndexResponse response = client
    .prepareIndex("library", "book", "2")
    .setSource("{\"title\": \"Mastering ElasticSearch\"}")
    .execute().actionGet();
```

The `setSource()` argument doesn't look too clear, does it? Fortunately, ElasticSearch also has the API for creating JSON documents. It is handy not only for creating input data, but also for querying ElasticSearch as practically all calls to the server allows to set the request payload directly. Let's look at the example of this API usage. We will start with the required imports:

```
import org.elasticsearch.common.xcontent.XContentBuilder;
import org.elasticsearch.common.xcontent.XContentFactory;
```

The following is the code:

```
Map<String, Object> m = Maps.newHashMap();
m.put("1", "Introduction");
m.put("2", "Basics");
m.put("3", "And the rest");
XContentBuilder json = XContentFactory.jsonBuilder().prettyPrint()
    .startObject()
    .field("id").value("2123")
    .field("lastCommentTime", new Date())
    .nullField("published")
    .field("chapters").map(m)
    .field("title", "Mastering ElasticSearch")
    .array("tags", "search", "ElasticSearch", "nosql")
    .field("values")
    .startArray()
    .value(1)
    .value(10)
    .endArray()
    .endObject();
```

This preceding built document describes the information about this book (more or less). First, we created the map with chapter descriptions, which we will use. Then we start the JSON definition by using the `XContentFactory.jsonBuilder()` call which will result in creating a proper factory. As you can see, the called methods imitate the desired JSON document structure (we add additional indentation to show it better). The API provides methods such as `startObject()` or `startArray()` in order for us to indicate that the next calls are related to the content of an object or an array. This allows us to create nested structures by nesting calls to the mentioned methods. End of definition of the given structure is marked by an appropriate call to `endObject()` or `endArray()`. In fact, the API automatically closes the open tags at the end of the definition, so in our case we can open it (using it influences the clarity of the code so it's better not to rely on this feature).

The next important thing is the definition of fields. The ElasticSearch API provides the `field()` method that allows us to define the name of the attribute (as we did with the `id` field, where the field value was defined using the `value()` method call) or define the field name and value at once (as we did for `title` field). In this second case, we can define the nested structure by using the `map()` method. There is also a dedicated `nullField()` method which allows us to define fields with null value. The last thing worth mentioning is the `array()` method which defines the array type field with the given name and with values defined by the second and subsequent arguments.

As you can see, the usage of the JSON builder is simple and convenient (also because all `field()` methods takes various types of parameters from `Boolean` to `Date`, which is suitable for various data types). What you should pay attention to is the correct order of function called, so the resulting JSON object is the one you are looking for.

When you have the builder defined, you can easily generate the string representation of the defined JSON by using the `string()` method. It generates a one-line JSON code, but if you want something more readable you can add a call to the `prettyPrint()` method (it has to be called just after creating the). We did this and `string()` method called for our code example generates the following JSON:

```
{
  "id" : "2123",
  "lastCommentTime" : "2013-08-11T09:27:43.446Z",
  "published" : null,
  "chapters" : {
    "3" : "And the rest",
    "2" : "Basics",
    "1" : "Introduction"
  },
  "title" : "Mastering ElasticSearch",
  "tags" : [ "search", "ElasticSearch", "nosql" ],
  "values" : [ 1, 10 ]
}
```

The administration API

On previous pages you saw how to search your data and how to manipulate documents using the Java API. However this is not the only functionality that ElasticSearch has to offer. In the rest of the following chapter we will see how to use the ElasticSearch Java API in order to perform administration tasks and monitor ElasticSearch state. ElasticSearch divides administration operation into two groups - cluster administration and indices administration. We will start with the first one.



Please note that the administration API of ElasticSearch is very broad and we just can't describe all the methods and responses, because we would have to write another book, just in order to describe the API. However we will try showing you the point where you can start.

The cluster administration API

The cluster administration API is exposed by the `org.elasticsearch.client.ClusterAdminClient` interface. You can get its implementation by running the `admin().cluster()` method on the `Client` interface, just as we did in the following example:

```
ClusterAdminClient cluster = client.admin().cluster();
```

The work with this interface is similar to what we did when we used the `Client` interface. The `ClusterAdminClient` interface defines several `prepareXXX()` methods returning builders, which after configuration return appropriate request objects. Similar to searching, you can use this object to send request to ElasticSearch in a synchronous or asynchronous way and after that use a dedicated response object. Let's review the available possibilities.

The cluster and indices health API

The cluster and indices health API allows us to fetch the basic health information about the cluster or its indices:

```
ClusterHealthResponse response = client.admin().cluster()  
    .prepareHealth("library")  
    .execute().actionGet();
```

The `prepareHealth()` method takes the names of indices to check (or no names at all, in such case the health for the whole cluster will be returned). In response, you can read information about the cluster status, number of assigned shards, number of shards, and replicas for a particular index.

The cluster state API

The cluster state API allows us to fetch information about the cluster: routing, shard allocation, and mappings. For example, the following code will send a request that will fetch the complete state information of the cluster:

```
ClusterStateResponse response = client.admin().cluster()
    .prepareState()
    .execute().actionGet();
```

The update settings API

The update settings API allows to set cluster-wide configuration parameters. The configuration is divided into transient settings (which are lost after the restart of the entire cluster) and permanent settings, where changes will be preserved even if the cluster is fully restarted. Remember that the list of parameters that can be updated dynamically is limited. The following example illustrates how we can change the `indices.ttl.interval` property using the discussed API:

```
Map<String, Object> map = Maps.newHashMap();
map.put("indices.ttl.interval", "10m");

ClusterUpdateSettingsResponse response = client.admin().cluster()
    .prepareUpdateSettings()
    .setTransientSettings(map)
    .execute().actionGet();
```

The reroute API

The reroute API allows you to move shards between nodes, cancel the shard allocation, or force shard allocation. The following example shows two allocation commands: the `move` command and the `cancel` command:

```
ClusterRerouteResponse response = client.admin().cluster()
    .prepareReroute()
    .setDryRun(true)
    .add(
        new MoveAllocationCommand(new ShardId("library", 3),
            "G3czOt4HQbKZT1RhpPCULw",
            "PvHtEMuRSJ6rLJ27AW3U6w"),
        new CancelAllocationCommand(new ShardId("library", 2),
            "G3czOt4HQbKZT1RhpPCULw",
            true))
    .execute().actionGet();
```

As you can see, each allocation command has its own class that we pass to the `add` method of the builder returned by `prepareReroute()` method call. Please also note that in order to specify the shard which we are interested in, we use the `ShardId` class which we will construct by invoking the constructor and passing the name of the index and the shard number.

One additional thing to notice is the `setDryRun(true)` method call in the preceding example. This method prevents the execution of allocation commands. Here, ElasticSearch only checks if the given allocation is possible. This is handy to test if we can run the given command in our cluster (you can find more about shard allocation in *Chapter 4, Index Distribution Architecture*, in the *Adjusting shard allocation* section).

The nodes information API

The nodes info API provides information about a particular node or nodes given by a node identifier or for all the nodes in the cluster. This information can contain details about the Java virtual machine, the operating system including the network information (such as IP address or Ethernet address or information about plugins). For example, the following code will fetch the node information including the plugin and network information:

```
NodesInfoResponse response = client.admin().cluster()
    .prepareNodesInfo()
    .setNetwork(true)
    .setPlugin(true)
    .execute().actionGet();
```

Note that this particular information is available only when directly requested. In the example, the response contained information about the network and plugins because of the `setNetwork()` and `setPlugin()` methods. There is also an `all()` method that turns on fetching all the information.

The node statistics API

The node statistics call is very similar to nodes info, but returns statistics about ElasticSearch utilization, such as indices statistics, filesystem, HTTP module, and Java virtual machine. The following example will fetch all the statistics regarding nodes:

```
NodesStatsResponse response = client.admin().cluster()
    .prepareNodesStats()
    .all()
    .execute().actionGet();
```

Similar to nodes info API, if we want a particular information to be returned, we should inform ElasticSearch about it by using the appropriate method or use the `all()` method to include all the information.

The nodes hot threads API

The nodes hot threads API allows you to examine the nodes when something goes wrong and the CPU utilization exceeds the norm. More information about this API can be found in *Chapter 6, Fighting with Fire* in the *Very hot threads* section. An example usage of this API in Java looks as follows:

```
NodesHotThreadsResponse response = client.admin().cluster()
    .prepareNodesHotThreads()
    .execute().actionGet();
```

In addition to the no-argument version of the `prepareNodesHotThreads()` method, it allows us to define one or more node identifiers as arguments to limit the information to the given nodes.

The nodes shutdown API

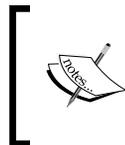
The nodes shutdown API is a simple API that allows us to shut down the selected nodes (or all of them). If we want, we can add the delay for the shutdown operation. For example the, following code will shut down the whole cluster immediately:

```
NodesShutdownResponse response = client.admin().cluster()
    .prepareNodesShutdown()
    .execute().actionGet();
```

The search shards API

The search shards API is another simple API which allows us to check which shards will be used for handling the request. It allows us to set the routing parameter value, so it is very handy when using custom routing. For example, the following code will check which shard (or shards) of the `library` index will be used to handle the request with the routing value of `12`:

```
ClusterSearchShardsResponse response = client.admin().cluster()
    .prepareSearchShards()
    .setIndices("library")
    .setRouting("12")
    .execute().actionGet();
```



If you want to learn more about this API please refer to <http://elasticsearchserverbook.com/elasticsearch-0-90-search-shards-api/> where we have written a simple blog post about this API.

The Indices administration API

In order to handle the administration requests of the indices, the ElasticSearch API uses the `org.elasticsearch.client.IndicesAdminClient` interface. You can get the implementation of this interface from the `Client` object by using the following code:

```
IndicesAdminClient cluster = client.admin().indices();
```

The work with this interface is again similar to what we've seen in the *The cluster administration API* section. This interface defines several `prepareXXX()` methods and these are the entry points to create requests. Let's review the available API requests.

The index existence API

The index existence API allows us to check if the indices given in the `prepareExists` call exists in the cluster. An example usage of this API is illustrated in the following code:

```
IndicesExistsResponse response = client.admin().indices()
    .prepareExists("books", "library")
    .execute().actionGet();
```

The Type existence API

Type existence API is very similar to the index existence API, but instead of checking the existence of index or indices, we can check the existence of types in a given index or indices. Note that indices should exist in order for the API to successfully return the information. The following is an example call using this API which checks for the existence of the `book` type in the `library` index:

```
TypesExistsResponse response = client.admin().indices()
    .prepareTypesExists("library")
    .setTypes("book")
    .execute().actionGet();
```

The indices stats API

The indices stats API provides information about indices, documents, storage, and operations such as `get`, `search` and `indexing`, `warmers` and `merge process`, `flushing`, and `refreshing`. This information is divided on the basis of type, and a particular information should be requested while creating the request. For example, the following code illustrates how to get all the information about the `library` index:

```
IndicesStatsResponse response = client.admin().indices()
    .prepareStats("library")
    .all()
    .execute().actionGet();
```

Index status

The indices stats API gives various information about the requested indices. In addition to that, when requested the result is enriched by information about the recovery status of shards and snapshotting. For example, to get the status information for the `library` index we should run the following code:

```
IndicesStatusResponse response = client.admin().indices()
    .prepareStatus("library")
    .setRecovery(true)
    .setSnapshot(true)
    .execute().actionGet();
```

Segments information API

The segments API returns low-level information about Lucene segments for the given index or indices, for example:

```
IndicesSegmentResponse response = client.admin().indices()
    .prepareSegments("library")
    .execute().actionGet();
```

Creating an index API

The create index API can be used to create a new index and set the mappings or settings for it. For example, the following code illustrates how to create the `news` index, with the number of shards equal to 1 along with the provided mappings:

```
CreateIndexResponse response = client.admin().indices()
    .prepareCreate("news")
    .setSettings(ImmutableSettings.settingsBuilder()
        .put("number_of_shards", 1))
    .addMapping("news", XContentFactory.jsonBuilder()
        .startObject()
            .startObject("news")
                .startObject("properties")
                    .startObject("title")
                        .field("analyzer", "whitespace")
                        .field("type", "string")
                    .endObject()
                .endObject()
            .endObject()
        .endObject()
    .execute().actionGet();
```

Deleting an index

The delete API allows us to irreversibly delete one or more indices. The following code illustrates how to delete the index called `news`:

```
DeleteIndexResponse response = client.admin().indices()
    .prepareDelete("news")
    .execute().actionGet();
```

Closing an index

The close API allows us to close the unused indices and thus free our node and cluster resources such as CPU cycles and memory. For example, the following code illustrates how to close the `library` index:

```
CloseIndexResponse response = client.admin().indices()
    .prepareClose("library")
    .execute().actionGet();
```

Opening an index

The open API allows us to open the previously closed index. For example, the following code will result in the closing of the `library` index:

```
OpenIndexResponse response = client.admin().indices()
    .prepareOpen("library")
    .execute().actionGet();
```

The Refresh API

The Refresh API allows us to perform a refresh on a given index or indices. For more information about what refresh is, please refer to the *NRT, flush, refresh and transaction log* section in *Chapter 3, Low-level Index Control*. The following example illustrates how to perform the refresh operation on the index called `library`:

```
RefreshResponse response = client.admin().indices()
    .prepareRefresh("library")
    .execute().actionGet();
```

The Flush API

The flush API performs flush on given indices. For more information refer to the *NRT, flush, refresh and transaction log* section in *Chapter 3, Low-level Index Control*. The example is as follows:

```
FlushResponse response = client.admin().indices()
    .prepareFlush("library")
    .setFull(false)
    .execute().actionGet();
```

The Optimize API

The Optimize API invokes the segments merge process on the given indices. Just like the RESTful version of this API, we are allowed to set the target number of segments or only remove the deleted documents from the index. For example, the following code snippet executes the optimize operation on the `library` index, specifies that the maximum number of segments is 2. The flush should be performed after the operation and we are not only interested in removing the deleted documents:

```
OptimizeResponse response = client.admin().indices()
    .prepareOptimize("library")
    .setMaxNumSegments(2)
    .setFlush(true)
    .setOnlyExpungeDeletes(false)
    .execute().actionGet();
```

The put mapping API

The put mapping API allows us to create or change mapping for a particular index or for multiple indices at once. However, please remember that the index we are specifying must exist. The following example illustrates creating the mapping for the news type in the news index:

```
PutMappingResponse response = client.admin().indices()
    .preparePutMapping("news")
    .setType("news")
    .setSource(XContentFactory.jsonBuilder()
        .startObject()
            .startObject("news")
                .startObject("properties")
                    .startObject("title")
                        .field("analyzer", "whitespace")
```

```
        .field("type", "string")
        .endObject()
    .endObject()
    .endObject()
    .endObject()
    .execute().actionGet();
```

The delete mapping API

It is the reverse method to put mapping. By using the delete mapping API we are allowed to delete mappings from one or more indices. For example, the following code illustrates how to delete the mappings for the `news` type in the `news` index:

```
DeleteMappingResponse response = client.admin().indices()
    .prepareDeleteMapping("news")
    .setType("news")
    .execute().actionGet();
```

The gateway snapshot API

The gateway snapshot API allows us to force ElasticSearch to perform snapshot for given index or indices. Note that this API works only for shared gateway types, which are in fact deprecated now:

```
GatewaySnapshotResponse response = client.admin().indices()
    .prepareGatewaySnapshot("news")
    .execute().actionGet();
```

The aliases API

The aliases API provides methods for managing the ElasticSearch aliases. Let's take a look at the following example:

```
IndicesAliasesResponse response = client.admin().indices()
    .prepareAliases()
    .addAlias("news", "n")
    .addAlias("library", "elastic_books",
        FilterBuilders.termFilter("title", "elasticsearch"))
    .removeAlias("news", "current_news")
    .execute().actionGet();
```

As you see, in our example we created multiple aliases with the same request. We've added an `n` alias for the `news` index and the `elastic_books` alias for the `library` index. In addition to that, while creating the `elastic_books` alias we also specified additional `termfilter` which will be used along with the alias. We've also removed the `current_news` alias for the `news` index.

The get aliases API

The get aliases API allows us to list the currently defined aliases. Let's look at the following example:

```
IndicesGetAliasesResponse response = client.admin().indices()
    .prepareGetAliases("elastic_books", "n")
    .execute().actionGet();
```

The preceding code will result in showing information about two aliases, the `n` and the `elastic_book` aliases. The nice thing about this is that we are allowed to use asterisks. Thanks to it, we can create request, for example, the `prepareGetAliases("*")` request which will list all the available aliases. The described API works in Version 0.90.3 of Elasticsearch. In the 1.0 version this is a little different: the object that holds the response for this request has a `GetAliasesResponse` type instead of `IndicesGetAliasesResponse`.

The aliases exists API

The aliases exists API allows us to check whether at least one of the listed aliases exists. Just as we did with the get aliases we can use asterisks in alias names. For example, the following code snippet results in a request that checks if the aliases starting with the names, `elastic` or named `unknown` exist:

```
AliasesExistResponse response = client.admin().indices()
    .prepareAliasesExist("elastic*", "unknown")
    .execute().actionGet();
```

The clear cache API

The clear cache API allows us to clear certain types of caches for a given index or indices. For example, the following code clears the filter cache, the identifier cache, and the field data cache for the `library` index and the `title` field:

```
ClearIndicesCacheResponse response = client.admin().indices()
    .prepareClearCache("library")
    .setFieldDataCache(true)
    .setFields("title")
    .setFilterCache(true)
    .setIdCache(true)
    .execute().actionGet();
```

The update settings API

The update settings API allow us to update the settings for particular indices or all of them. For example, the following code will set the number of replicas (the `index.number_of_replicas` property) to 2 for the `library` index:

```
UpdateSettingsResponse response = client.admin().indices()
    .prepareUpdateSettings("library")
    .setSettings(ImmutableSettings.builder()
        .put("index.number_of_replicas", 2))
    .execute().actionGet();
```

The analyze API

This API is very handy when we want to see how the analysis process is done for the given analyzer, tokenizer, and filters. For example, the following code will result in a request that checks how the analysis will be performed for the `ElasticSearch Servers` phrase in the `library` index using the `whitespace` tokenizer and the `nGram` filter:

```
AnalyzeResponse response = client.admin().indices()
    .prepareAnalyze("library", "ElasticSearch Servers")
    .setTokenizer("whitespace")
    .setTokenFilters("nGram")
    .execute().actionGet();
```

The put template API

The put template API allows you to configure the new templates for use by ElasticSearch while creating the new indices. Of course, the templates can hold mappings and settings. For example, the following code will result in creating a new template called `my_template` that will be used for any index starting with `product`. Each index created using this template will have two replicas, one shard and a type called `item` with a single `title` field:

```
PutIndexTemplateResponse response = client.admin().indices()
    .preparePutTemplate("my_template")
    .setTemplate("product*")
    .setSettings(ImmutableSettings.builder()
        .put("index.number_of_replicas", 2)
        .put("index.number_of_shards", 1))
    .addMapping("item", XContentFactory.jsonBuilder()
        .startObject()
            .startObject("item")
```

```
        .startObject("properties")
        .startObject("title")
        .field("type", "string")
        .endObject()
        .endObject()
        .endObject()
        .endObject()
        .execute().actionGet();
```

The delete template API

With the delete template API you can delete a single template or several templates given by a pattern. For example, the following code deletes all the templates whose names starts with `my_`:

```
DeleteIndexTemplateResponse response = client.admin().indices()
    .prepareDeleteTemplate("my_*")
    .execute().actionGet();
```

The validate query API

The validate query API can be used to check if the query you want to send to ElasticSearch is valid. The response of the API provides the information about whether the query is correct or not. By calling the `setExplain` method and passing a `true` value to it, the API will also provide information about what is wrong with the query. The following code illustrates the usage of this API:

```
ValidateQueryResponse response = client.admin().indices()
    .prepareValidateQuery("library")
    .setExplain(true)
    .setQuery(XContentFactory.jsonBuilder()
        .startObject()
        .field("name").value("elastic search")
        .endObject().bytes())
    .execute().actionGet();
```

The put warmer API

The put warmer API allows us to create a warmer for a given index or several indices. For more information about the warmers, please refer to the *Speeding up queries using warmers* section in *Chapter 6, Fighting with Fire*. The following example shows how to create a warmer called `library_warmer` that will execute a match for all the queries with `termsfacet` on the `tags` parameter filed in the `library` index:

```
PutWarmerResponse response = client.admin().indices()
    .preparePutWarmer("library_warmer")
    .setSearchRequest(client.prepareSearch("library")
        .addFacet(FacetBuilders
            .termsFacet("tags").field("tags")))
    .execute().actionGet();
```

The delete warmer API

If you can put a warmer, you can also delete it and the delete warmer API provides such functionality. For example, if we like to delete all the warmers that start with the `library_` prefix we should use the following code:

```
DeleteWarmerResponse response = client.admin().indices()
    .prepareDeleteWarmer()
    .setName("library_*")
    .execute().actionGet();
```

Summary

In this chapter we've learned how to use the Java API that is provided with ElasticSearch. We started with discussing how to connect to local and remote ElasticSearch cluster, how to index data in two ways - both one document after another and by using batches. We've updated API to alter already indexed documents and we've seen how to make queries by using Java API. We've learned how to handle errors returned by ElasticSearch Java API. In addition to that we've run administrative commands from our Java code and we've learned how to perform administrative tasks exposed by the API.

In the next chapter we will get more into the internals of ElasticSearch and we will learn how to extend its functionality. We will learn how to develop a river plugin that will be managed by ElasticSearch. In addition to that we will extend the analysis capabilities of ElasticSearch by developing a custom tokenizer and filter. Finally, we will see how to develop a custom search plugin that will extend the ElasticSearch functionality.

9

Developing Elasticsearch Plugins

In the previous chapter we learned how to use the Java API that is provided with Elasticsearch. We started with discussing how to connect to local and remote Elasticsearch cluster and how to index data in two ways (both one document after another and by using batches). We updated the API to alter already indexed documents and we've seen how to make queries by using Java API. We learned how to handle errors returned by Elasticsearch Java API. In addition to that we've run administrative commands from our Java code and we learned how to perform administrative tasks exposed by the API. In this chapter, we will focus on extending Elasticsearch capabilities by writing custom plugins. By the end of this chapter, you will be able to:

- Set up a general Apache Maven project that will automatically package the code you've created, so it can be installed by Elasticsearch
- Develop a custom River plugin to index your data
- Create custom tokenizer that can be used during querying and indexing
- Develop a custom filter that will allow custom filtering

Creating the Apache Maven project structure

Before we start with showing how to develop a custom Elasticsearch plugin, we would like to discuss a way to package it, so it can be installed by Elasticsearch by using the `plugin` command. In order to do that, we will use Apache Maven (<http://maven.apache.org/>) project management software. It aims to make your build process easier, provide unifying build system, manage dependencies, and so on.



Please note that even though the chapter you are currently reading was written and tested using Elasticsearch 0.90.3, it was also tested with 1.0.0Beta release. If you would like the code examples to work with Elasticsearch 1.0.0Beta version, you should modify Elasticsearch version in the `pom.xml` file.

Please also remember that the book you are holding in your hands is not about Maven, but Elasticsearch, and we will keep Maven related information to the required minimum.



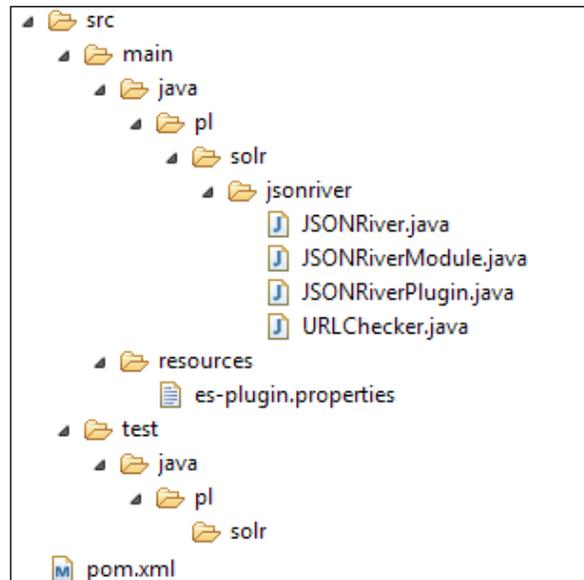
Installing Apache Maven is a straightforward task. We assume that you already have it installed. However, if you have problems with it, please consult <http://maven.apache.org/> for more information.

Understanding the basics

The result of a Maven build process is an artifact. Each artifact is defined by its identifier, its group, and version. This is crucial when working with Maven, because each dependency you'll use will be identified by those three mentioned properties.

Structure of the Maven Java project

The idea behind Maven is quite simple; you create a project structure that looks something like the following screenshot:



You can see that the code is placed in the `src` folder (the code is in the `main` folder and the unit tests are located in the `test` folder). Although you can change the default layout, Maven tends to work best with the default layout.

The idea of POM

In addition to the code, you can see a file named `pom.xml` that is located in the root directory in the previous image. This is a project object model file that describes the project, its properties, and dependencies. That's right; you don't need to manually download dependencies if they are available in one of the available Maven repositories. During its work, Maven will download them and use it. All you need to care about is writing an appropriate `pom.xml` section that will inform Maven which dependencies should be used.

For example, see the following Maven pom.xml file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>pl.solr</groupId>
  <artifactId>jsonriver</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>jsonriver</name>
  <url>http://solr.pl</url>

  <properties>
    <elasticsearch.version>0.90.3</elasticsearch.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.elasticsearch</groupId>
      <artifactId>elasticsearch</artifactId>
      <version>${elasticsearch.version}</version>
    </dependency>
  </dependencies>
</project>
```

This is a simplified version of a pom.xml file that we will extend in the rest of the chapter. You can see that it starts with the root `project` tag and then defines the group identifier, the artefact identifier, version, and packaging method (in our case, the standard build command will create a jar file). In addition to that, we've specified a single dependency: the Elasticsearch library version 0.90.3.

Running the build process

In order to run the build process, what we need to do is simply run the following command in the directory where the `pom.xml` file is present:

```
mvn clean package
```

It will result in running Maven. It will clean all the generated content in the working directory, compile, and package our code. Of course, if we have unit tests, they will have to pass in order for the package to be built. The built package will be written into the `target` directory created by Maven.



If you want to learn more about Maven lifecycle, please refer to <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>.

Introducing the assembly Maven plugin

In order to build the zip file that will contain our plugin code, we need to package it. By default, Maven doesn't support pure zip files packaging, so in order to make it all work, we will use the Maven Assembly plugin (you can find more about the plugin at <http://maven.apache.org/plugins/maven-assembly-plugin/>). In general, the described plugin allows us to aggregate the project output along with its dependencies, documentations, and configuration files into a single archive.

In order for the plugin to work, we need to add the `build` section to our `pom.xml` file that will contain information about the assembly plugin, the jar plugin (which is responsible for creating proper jar), and the compiler plugin, because we want to be sure that the code will be readable by Java 6. In addition to this, let's assume that we want our archive to be put into the `target/release` directory of our project. The relevant section of the `pom.xml` file should look like this:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.3</version>
      <configuration>
        <finalName>elasticsearch-${project.name}-
          ${elasticsearch.version}</finalName>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```
</plugin>

<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-assembly-plugin</artifactId>
<version>2.2.1</version>
<configuration>
  <finalName>elasticsearch-${project.name}-
    ${elasticsearch.version}</finalName>
  <appendAssemblyId>>false</appendAssemblyId>
  <outputDirectory>${project.build.directory}
    /release/</outputDirectory>
  <descriptors>
  <descriptor>assembly/release.xml</descriptor>
  </descriptors>
</configuration>
<executions>
<execution>
  <id>generate-release-plugin</id>
  <phase>package</phase>
  <goals>
    <goal>single</goal>
  </goals>
</execution>
</executions>
</plugin>

<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
  </configuration>
</plugin>
</plugins>
</build>
```

If you look closely at the assembly plugin configuration, you'll notice that we specify the assembly descriptor, called `release.xml`, in the `assembly` directory. This file is responsible for specifying what kind of archive we want to have as the output. Let's put the following `release.xml` file in the `assembly` directory of our project:

```
<?xml version="1.0"?>
<assembly>
  <id>bin</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <dependencySets>
    <dependencySet>
      <unpack>false</unpack>
      <outputDirectory>./</outputDirectory>
      <useProjectArtifact>false</useProjectArtifact>
      <useTransitiveFiltering>true</useTransitiveFiltering>
      <excludes>
        <exclude>org.elasticsearch:elasticsearch</exclude>
        <exclude>junit:junit</exclude>
      </excludes>
    </dependencySet>
  </dependencySets>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>elasticsearch-${project.name}-
          ${elasticsearch.version}.jar</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

Again, we don't need to know all the details, however it is nice to understand what is going on, even on the general level only. So, the previous file tells Maven Assembly plugin that we want our archive to be packed with zip (`<format>zip</format>`) and we want the JUnit and Elasticsearch libraries to be excluded (the `exclude` section), because they will already be present in Elasticsearch on which we will install the plugin. In addition to that we've specified that we want our project jar to be included (the `includes` section).



If you want to see the full project structure, with the full `pom.xml` file and all the needed files, please look at the code of this chapter provided with this book at Packt's website.

Creating a custom river plugin

The next extension we will develop to Elasticsearch will be a custom river plugin. As you know, the rivers are Elasticsearch functionality (usually plugins) that allow to index data from different sources, such as Wikipedia, Twitter, and database. In this simple example, we will develop a river that will be able to write last modified date of a given website and will check and update that data periodically and the period will be configurable. The data that will be fetched by the river will be stored in the index. So let's see what we need to do and how to do it.

Implementation details

To achieve the requirements we've described previously, we will need to develop the following parts:

- Module definition class, which extends the `AbstractModule` class from the `org.elasticsearch.common.inject` package; we will call it `JSONRiverModule`.
- Plugin definition class, which extends the `AbstractPlugin` class from the `org.elasticsearch.plugins` package; we will call it `JSONRiverModule`.

- The river itself, which extends the `AbstractRiverComponent` class from the `org.elasticsearch.river` package and implements `River` interface from the `org.elasticsearch.river` package; we will call it `JSONRiver`.
- A class that we will use to periodically check a given URL address; we will develop it as a `Runnable` interface implementation, because we will use thread executors to run it. We will call that class `URLChecker`.



We assume that you already have a Java project created and that you are using Maven, just like we did in the *Creating Apache Maven project structure* section in the beginning of this chapter. If you would like to use an already created and working example and start from there, please look at the code of this book that is available with the book on Packt's website.

In addition to that, we will need a library that will allow a quick and simple way to connect to the given URL and read the returned headers, for example, `HttpComponents` from Apache (<http://hc.apache.org/>). To use it, we just need to add another dependency to our Maven `pom.xml` file, so its dependencies section looks like this:

```
<dependencies>
  <dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>${elasticsearch.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.2.5</version>
  </dependency>
</dependencies>
```

Implementing the URLChecker class

Let's start by implementing the logic behind URL checking, because it is more or less detached from the actual river logic. The whole code for the URLChecker class looks as follows:

```
public class URLChecker implements Runnable {
    private Client client;
    private ESLogger logger;
    private String url;
    private int time;
    private HttpClient httpClient;

    public URLChecker(Client client, String url, String time,
        ESLogger logger) {
        this.client = client;
        this.url = url;
        this.time = Integer.parseInt(time);
        this.logger = logger;
        this.httpClient = new DefaultHttpClient();
    }

    @Override
    public void run() {
        while (true) {
            logger.info("Checking {}", url);
            HttpHeaders headMethod = new HttpHeaders(url);
            try {
                HttpResponse httpResp = httpClient.execute(headMethod);
                int respCode = httpResp.getStatusLine().getStatusCode();
                if (respCode < HttpStatus.SC_OK || respCode >
                    HttpStatus.SC_MULTI_STATUS) {
                    logger.error("Error, got {} code", respCode);
                } else {
                    Header header = httpResp.getFirstHeader("Last-Modified");
                    if (header != null) {
                        indexLastModified(header.getValue());
                    } else {
                        logger.warn("{} didn't return last modified", url);
                    }
                }
            } catch (Exception ex) {
                logger.error("Error during URLChecker execution", ex);
            }
        }
    }
}
```

```

    } finally {
        headMethod.releaseConnection();
    }
    logger.info("Sleeping for {} minutes", time);
    try {
        Thread.sleep(1000 * 60 * time);
    } catch (InterruptedException ie) {
        logger.error("Thread interrupted", ie);
    }
}
}

protected void indexLastModified(String modified) throws
    IOException {
    client
        .prepareIndex("urls", "url", url)
        .setSource(XContentFactory.jsonBuilder().startObject()
            .field("url", url)
            .field("modified", modified).endObject())
        .execute().actionGet();
}
}

```

We start with the constructor, which is very simple. We just store references to the objects we will use and we instantiate `DefaultHttpClient`, which will be used to make requests which will read the URL headers.

The next thing is the `run` method. If you look at the `Runnable` interface of Javadocs (<http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html>), you'll notice that this method will be executed when the thread execution will be to is started and this is exactly what we want. We want the logic in that method to be running until the river is loaded and because of that we are using the following loop:

```
while (true) {
```

This loop will run the code inside itself until interrupted or unchecked exception is thrown.

In the loop itself we create a new `HTTP HEAD` method, which will be used to fetch headers from the given URL and we execute that method using the `execute()` method of the earlier initialized `DefaultHttpClient`. After that we check if the request was properly executed and processed by checking the response code and we read the value of the `Last-Modified` header, which is what we wanted to do. After that, if the read header value is not null, we call the `indexLastModified()` method, which will index that data.

 If you want to read in detail how Apache HttpComponents library works at the Java code level, please refer to its tutorial available at the following URL: <http://hc.apache.org/httpclient-legacy/tutorial.html>

There is one more thing that we wanted to explicitly discuss. At the end of the discussed loop, we can see the following code fragment:

```
try {
    Thread.sleep(1000 * 60 * time);
} catch (InterruptedException ie) {
    logger.error("Thread interrupted", ie);
}
```

Because our code is running in a virtually endless loop, we need some kind of sleep mechanism that will prevent our header retrieving code to be executed endlessly as soon as Elasticsearch can execute it. If we would run it without sleep, we would just hit the URL constantly and waste CPU resources, but indexing the data again and again. This is why we execute the `sleep()` method of the `Thread` class, which allows us to suspend thread execution for a given amount of milliseconds. After the given time has passed, the execution will be resumed.

The last thing regarding the `URLChecker` class is the data indexing method, which looks as follows:

```
protected void indexLastModified(String modified) throws
    IOException {
    client
        .prepareIndex("urls", "url", url)
        .setSource(XContentFactory.jsonBuilder().startObject()
            .field("url", url)
            .field("modified", modified).endObject())
        .execute().actionGet();
}
```

We index the URL's last modified date to the `urls` index, under the `url` type. If you've read *Chapter 8, Elasticsearch Java APIs*, you should be familiar with the previous code; if you didn't, we strongly recommend to go through that chapter.

Implementing the JSONRiver class

The `JSONRiver` class, the one which is responsible for the main river logic, extends the `AbstractRiverComponent` class from the `org.elasticsearch.river` package and implements the `River` interface from the `org.elasticsearch.river` package.

The `AbstractRiverComponent` class allows us to use `ElasticSearch` logging capabilities and settings without the need of having them initialized and the `River` interface forces us to implement the `start()` and `stop()` methods which are called when the river is starting (the `start()` method) and when it is being stopped (the `stop()` method). The whole code of the class looks as follows:

```
public class JSONRiver extends AbstractRiverComponent implements
    River {
    private URLChecker urlChecker;
    private Thread urlCheckerThread;

    @Inject
    protected JSONRiver(RiverName riverName, RiverSettings settings,
        Client client) {
        super(riverName, settings);
        String urlToCheck = (String) settings.settings().get("url");
        String timeBetweenChecks = (String)
            settings.settings().get("time");
        this.urlChecker = new URLChecker(client, urlToCheck,
            timeBetweenChecks, logger);
    }

    @Override
    public void start() {
        logger.info("Starting JSONRiver river");
        urlCheckerThread =
            Executors.daemonThreadFactory(settings.globalSettings(),
                "jsonriver_thread").newThread(
                urlChecker);
        urlCheckerThread.start();
    }

    @Override
    public void close() {
        urlCheckerThread.interrupt();
    }
}
```

Again we start with the constructor. In it, we pass the `settings()` method (which contains the river settings) and the name of the river to the super class constructor. We read the properties that will be passed when the river is registered and we instantiate the `URLChecker` object instance. We will omit discussing how to read settings, because it is a call to the `settings` method of the `Settings` type object.

The `start()` method is the method `ElasticSearch` will call when starting the river. We start by logging a simple information and then we use executors to create a new thread from our `URLChecker` class which implements the `Runnable` interface:

```
urlCheckerThread =  
    EsExecutors.daemonThreadFactory(settings.globalSettings(),  
    "jsonriver_thread").newThread(urlChecker);
```

The previous code uses the `EsExecutors` class, which is an `ElasticSearch` way of running daemon threads inside the `ElasticSearch` nodes. We call the static `daemonThreadFactory()` method and we pass the global settings and the name of the thread to it (choose the one that fits you and will be unique). Such a call returns a `ThreadFactory` type object (from the `java.util.concurrent` JDK package, <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadFactory.html>), which allows us to create a new thread from an object implementing the `Runnable` interface. The `newThread()` method return `Thread` that we can start by simply running its `start()` method, which we do as follows:

```
urlCheckerThread.start();
```

This will result in starting the actual thread and finally executes its `run()` method. In our case, it will be the `run()` method of our `URLChecker` class.

Finally, the `stop` method will be called by `ElasticSearch` when the river should be stopped, for example, during `ElasticSearch` node shutdown or when the river is deleted. What we do is we use the `interrupt()` method of the `Thread` class, which interrupts thread execution (more about it can be found at <http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>).

Implementing the JSONRiverModule class

The `JSONRiverModule` class is responsible for binding the river class and telling ElasticSearch that it should be a singleton. We want only a single river instance we create to be running inside the cluster. The whole class is very simple and its code looks as follows:

```
public class JSONRiverModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(River.class).to(JSONRiver.class).asEagerSingleton();
    }
}
```

As you can see, the `JSONRiverModule` class extends the `AbstractModule` class from the `org.elasticsearch.common.inject` package and only overwrites a single method: `configure()`. In its body, we bind the general `River` class to our `JSONRiver` and we say that we want that to be bound as a singleton.



If you want to learn more about ElasticSearch modules binding, please refer to the Javadoc of the `Binder` class located in the `org.elasticsearch.common.inject` package.

Implementing the JSONRiverPlugin class

The `JSONRiverPlugin` is a class in the code that is used by ElasticSearch to initialize the plugin itself. It extends the `AbstractPlugin` class from the `org.elasticsearch.plugin` package. Because we are making an extension, we are obliged to implement the following code parts:

- A standard constructor that will take a single argument; in our case, it will be empty
- The `onModule()` method that includes the code that will add our custom river plugin, so that ElasticSearch will know about it
- The `name()` method that names of our plugin
- The `description()` method which gives a short description of our plugin

What we need to do when creating the `AbstractPlugin` class implementation is create three methods, `onModule()`, `name()`, and `description()`, just like described previously. The `name()` and `description()` methods are very simple and they are only responsible for returning the plugin's name and description. The whole class code looks like this:

```
public class JSONRiverPlugin extends AbstractPlugin {
    @Inject
    public JSONRiverPlugin(Settings settings) {
        super();
    }

    public void onModule(RiversModule module) {
        module.registerRiver("jsonriver", JSONRiverModule.class);
    }

    @Override
    public String name() {
        return "JSONRiver";
    }

    @Override
    public String description() {
        return "JSON river plugin";
    }
}
```

This time we've used the `onModule()` method to register our river plugin. As you can see, the `onModule` method takes a single argument, which is a `RiversModule` class object and it exposes a `registerRiver()` method. We call this method, giving it the type of the river and the module class responsible for binding the plugin.

Informing Elasticsearch about the JSONRiver plugin class

Once we have our code ready, we can add the property file that will inform Elasticsearch which is the class registering our plugin: the one we've called `JSONRiverPlugin`. In order to do that, we create an `es-plugin.properties` file in the `src/main/resources` directory with the following content:

```
plugin=pl.solr.jsonriver.JSONRiverPlugin
```

We just specify the `plugin` property there, which should have a value of the class we use for registering our plugins (the one that extends the `ElasticSearch AbstractPlugin` class). This file will be included in the jar file that will be created during the build process and will be used by `ElasticSearch` during plugin load process.

Testing our river

We could stop here and assume that our river works, but we won't. We will build our plugin, install it, and finally test it to see what we can expect.

Building our river

We will start with building our plugin. In order to do that, we run a simple command:

```
mvn compile package
```

After it finishes, we can find the archive with the plugin at the `target/release` directory (assuming you are using a similar project setup to the one we've described in the beginning of the chapter).

Installing our river

In order to install the plugin, again we will use the `plugin` command that is located in the `bin` directory of `ElasticSearch` distributable package. Assuming that we have our plugin archive stored at the `/home/install/es/plugins` directory, we would run the following command (we run it from the `ElasticSearch` home directory):

```
bin/plugin --install jsonriver --url file:/home/gro/es/elasticsearch-  
jsonriver-0.90.3.zip
```

We need to install the plugin on all the nodes that are eligible for running rivers, so by default, all the `ElasticSearch` nodes in our cluster.

After we've the plugin installed, we need to restart our `ElasticSearch` instance we were making the installation on. After restart, we should see something like the following code in the logs:

```
[2013-08-21 21:01:17,576] [INFO ] [plugins  
[Antimatter] loaded [JSONRiver], sites []
```

As you can see, `ElasticSearch` informed us that the plugin, named `JSONRiver`, was loaded.

Initializing our river

After we are done with installation, we can initialize our river. In order to do that we need to run the command to create it, which looks as follows:

```
curl -XPUT 'localhost:9200/_river/jsonriver/_meta' -d '{
  "type" : "jsonriver",
  "url" : "http://lucene.apache.org",
  "time" : "1"
}'
```

As you can see, we create a `jsonriver` type river, that will check the `http://solr.pl` URL once every minute (the `time` property value set to 1). If all worked well, you should be able to see something like the following code in the logs on the master node:

```
[2013-08-21 21:01:37,623] [INFO ] [cluster.metadata      ]
  [Antimatter] [_river] creating index, cause [auto(index api)],
  shards [1]/[1], mappings []
[2013-08-21 21:01:37,926] [INFO ] [cluster.metadata      ]
  [Antimatter] [_river] update_mapping [jsonriver] (dynamic)
[2013-08-21 21:01:37,995] [INFO ] [cluster.metadata      ]
  [Antimatter] [_river] update_mapping [jsonriver] (dynamic)
```

Those messages say that the `_river` index was created. The river logs can be found on the node that the river is running at and should look similar to the following ones:

```
[2013-08-21 21:01:37,988] [INFO ] [pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver] [jsonriver] Strting JSONRiver river
[2013-08-21 21:01:37,994] [INFO ] [pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver] [jsonriver] Checking
  http://lucene.apache.org
[2013-08-21 21:01:38,677] [INFO ] [pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver] [jsonriver] Sleeping for 1 minutes
[2013-08-21 21:02:38,677] [INFO ] [pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver] [jsonriver] Checking
  http://lucene.apache.org
[2013-08-21 21:02:38,929] [INFO ] [pl.solr.jsonriver.JSONRiver]
  [Antimatter] [jsonriver] [jsonriver] Sleeping for 1 minutes
```



If you are not familiar with how to run rivers in Elasticsearch, please refer to the Elasticsearch Server book or to the Elasticsearch reference guide on rivers, which is available at the following URL: <http://www.elasticsearch.org/guide/reference/river/>

Checking if our JSON river works

We can finally check if our river indexed some data. Remember the code? We've said that we want the data to be written into the `urls` index; so let's check it. In order to do that we would run the following simple search that would match all docs from the index:

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

The following response is returned by ElasticSearch:

```
{
  "took" : 29,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "urls",
      "_type" : "url",
      "_id" : "http://lucene.apache.org",
      "_score" : 1.0, "_source" :
      { "url": "http://lucene.apache.org", "modified": "Tue, 03 Sep
        2013 21:21:03 GMT" }
    } ]
  }
}
```

As you can see, the data is present, so our river is working as it should be.

Creating custom analysis plugin

The last thing we want to discuss when it comes to custom ElasticSearch plugins is analysis process extension. We've chosen to show how to develop a custom analysis plugin because this is sometimes very useful, for example, when you want to have custom analysis process introduced that you use in your company or when you want to use Lucene analyzer or filter that is not present in ElasticSearch itself, or as a plugin for it. Because creating an analysis extension is more complicated as compared what we've seen so far, we decided to leave it until the end of the chapter.

Implementation details

Because developing a custom analysis plugin is the most complicated, at least from the Elasticsearch point of view and the number of classes we need to develop, we will have more things to do comparing to previous examples. We will need to develop the following things:

- The `TokenFilter` class extension (from the `org.apache.lucene.analysis` package) implementation that will be responsible for handling token reversing; we will call it `CustomFilter`
- The `AbstractTokenFilterFactory` extension (from the `org.elasticsearch.index.analysis` package) that will be responsible for providing our `CustomFilter` instance to Elasticsearch; we will call it `CustomFilterFactory`
- Custom analyzer, which will extend the `org.apache.lucene.analysis.Analyzer` class and will provide Lucene analyzer functionality; we will call it `CustomAnalyzer`
- The Analyzer provider, which we will call `CustomAnalyzerProvider` and extends `AbstractIndexAnalyzerProvider` from the `org.elasticsearch.index.analysis` package and which will be responsible for providing analyzer instance to Elasticsearch
- Extension of `AnalysisModule.AnalysisBinderProcessor` from the `org.elasticsearch.index.analysis` package, which will contain information about under which names our analyzer and token filter will be available in Elasticsearch; we will call it `CustomAnalysisBinderProcessor`
- An extension to the `AbstractComponent` class from the `org.elasticsearch.common.component` package, which will inform Elasticsearch which factories should be used for our custom analyzer and token filter; we will call it `CustomAnalyzerIndicesComponent`
- The `AbstractModule` extension (from the `org.elasticsearch.common.inject` package) that will inform Elasticsearch that our `CustomAnalyzerIndicesComponent` should be a singleton; we will call it `CustomAnalyzerModule`
- Finally, the usual `AbstractPlugin` extension (from the `org.elasticsearch.plugins` package) that will register our plugin; we will call it `CustomAnalyzerPlugin`

So, let's start discussing the code.

Implementing TokenFilter

The funniest thing with the currently discussed plugin is that the whole analysis work is actually done on Lucene level and what we need to do is write the `org.apache.lucene.analysis.TokenFilter` extension, which we will call `CustomFilter`. In order to do that, we need to initialize the super class and override the `incrementToken()` method. Our class will be responsible for reversing the tokens, so the logic we want our analyzer and filter to have. The whole implementation of our `CustomFilter` looks as follows:

```
public class CustomFilter extends TokenFilter {
    private final CharTermAttribute termAttr =
        addAttribute(CharTermAttribute.class);

    protected CustomFilter(TokenStream input) {
        super(input);
    }

    @Override
    public boolean incrementToken() throws IOException {
        if (input.incrementToken()) {
            char[] originalTerm = termAttr.buffer();
            if (originalTerm.length > 0) {
                StringBuilder builder = new StringBuilder(new
                    String(originalTerm).trim()).reverse();
                termAttr.setEmpty();
                termAttr.append(builder.toString());
            }
            return true;
        } else {
            return false;
        }
    }
}
```

The first thing we see in the implementation is the following line:

```
private final CharTermAttribute termAttr =
    addAttribute(CharTermAttribute.class);
```

It allows us to retrieve the text of the token we are currently processing. In order to get access to other token information, we need to use other attributes. The list of attributes can be found by looking at the classes implementing the Lucene `org.apache.lucene.util.Attribute` interface (http://lucene.apache.org/core/4_4_0/core/org/apache/lucene/util/Attribute.html). What you need to know now is that by using the static `addAttribute` method, we can bind different attributes and use them during token processing.

Then we have the constructor, which is only used for super class initialization; so we can skip discussing it.

Finally, there is the `incrementToken()` method, which returns `true` when there is a token in the token stream left to be processed and `false` if there is no token left to be processed. So what we do first is we check if there is a token to be processed by calling the `incrementToken()` method of `input`, which is the `TokenStream` instance stored in the super class. Then we get the term text by calling the `buffer()` method of the attribute we've bound in the first line of our class. If there is a text in the term (its length is higher than zero), we use a `StringBuilder` object to reverse the text, we clear the term buffer (by calling `setEmpty()` on the attribute), and we append the reversed text to the already emptied term buffer (we do that by calling the `append()` method of the attribute). After that we return `true`, because our token is ready to be processed further (on token filter level, we don't know if the token will be processed further or not, so we need to be sure we return the correct information, just in case).

Implementing the TokenFilter factory

The factory for our token filter implementation is one of the simplest classes in case of the discussed plugins. What we need to do is create an `AbstractTokenFilterFactory` (from `org.elasticsearch.index.analysis` package) extension that overrides a single `create()` method, in which we create our token filter. The code of this class looks as follows:

```
public class CustomFilterFactory extends
    AbstractTokenFilterFactory {
    @Inject
    public CustomFilterFactory(Index index, @IndexSettings Settings
        indexSettings, @Assisted String name, @Assisted Settings
        settings) {
        super(index, indexSettings, name, settings);
    }

    @Override
    public TokenStream create(TokenStream tokenStream) {
        return new CustomFilter(tokenStream);
    }
}
```

As you can see, the class is very simple. We start with the constructor, which is needed because we need to initialize the parent class. In addition to that, we have the `create` method, in which we create our `CustomFilter` class with the provided `TokenStream` object.

Implementing custom analyzer

We wanted to keep the example of implementation as simple as possible and because of that, we've decided not to complicate the analyzer implementation. To implement analyzer, we need to extend the abstract `Analyzer` class from the Lucene `org.apache.lucene.analysis` package and so, we did. The whole code of our `CustomAnalyzer` class looks as follows:



If you want see more complicated analyzer implementations please look at the source code of Apache Lucene, Apache Solr, and Elasticsearch.

```
public class CustomAnalyzer extends Analyzer {
    private final Version version;

    public CustomAnalyzer(final Version version) {
        this.version = version;
    }

    @Override
    protected TokenStreamComponents createComponents(String field,
        Reader reader) {
        final Tokenizer src = new WhitespaceTokenizer(this.version,
            reader);
        return new TokenStreamComponents(src, new CustomFilter(src));
    }
}
```

Our `CustomAnalyzer` class needs to hold information about the `Version` class. The `Version` class (http://lucene.apache.org/core/4_4_0/core/org/apache/lucene/util/Version.html) is used by Lucene to maintain version compatibility across releases of Lucene. We need it to initialize `WhitespaceTokenizer` which we use for data tokenization. We pass the `Version` type object in constructor, because we will be able to access which `Version` Elasticsearch uses when developing analyzer provider only.

The `createComponent()` method is the one we need to implement and it should return a `TokenStreamComponents` object (from the `org.apache.lucene.analysis` package) for a given field name (the `String` type object – first argument of the method) and data (the `Reader` type object – second method argument). What we do is create `Tokenizer` using the `WhitespaceTokenizer` class available in Lucene. This will result in the input data to be tokenized on whitespace characters. And then, we create the Lucene `TokenStreamComponents` object, to which we give the source of tokens (our previously created `Tokenizer`) and our `CustomFilter`. This will result in our `CustomFilter` to be used by the `CustomAnalyzer`.

Implementing analyzer provider

Analyzer provider is another provider implementation, in addition to the token filter factory we've created earlier. This time, we need to extend `AbstractIndexAnalyzerProvider` from the `org.elasticsearch.index.analysis` package in order for Elasticsearch to be able to create our analyzer. The implementation is very simple as we only need to implement the `get` method in which we should return our analyzer. The `CustomAnalyzerProvider` class code is shown as follows:

```
public class CustomAnalyzerProvider extends
    AbstractIndexAnalyzerProvider<CustomAnalyzer> {
    private final CustomAnalyzer analyzer;

    @Inject
    public CustomAnalyzerProvider(Index index, @IndexSettings
        Settings indexSettings, Environment env, @Assisted String
        name, @Assisted Settings settings) {
        super(index, indexSettings, name, settings);
        analyzer = new CustomAnalyzer(version);
    }

    @Override
    public CustomAnalyzer get() {
        return this.analyzer;
    }
}
```

As you can see, we've implemented the constructor in order to be able to initialize the super class. In addition to that, we are creating a single instance of our analyzer which we will return when ElasticSearch requests it. Please note that this is the class that is aware of the `Lucene Version` class, which we need for our analyzer. We don't need to worry because our analyzer is thread safe and thus, a single instance can be reused. In the `get` method, we are just returning our analyzer.

Before we go on, we would like to mention two things: the `@IndexSettings` and `@Assisted` annotations. The first one will result in index setting being injected as the `Settings` class object to the constructor; of course, it is done automatically. The `@Assisted` annotation results in the annotated parameter value to be injected from the argument of the factory method.

Implementing analysis binder

The binder is a part of our custom code that informs ElasticSearch under which names our analyzer and token filter will be available. Our `CustomAnalysisBinderProcessor` extends the `AnalysisModule.AnalysisBinderProcessor` class from `org.elasticsearch.index.analysis` and we override two methods of that class: `processAnalyzers`, in which we will register our analyzer and `processTokenFilters`, in which we will register our token filter. If we would have only analyzer or only token filter, we would only override a single method. The code of the `CustomAnalysisBinderProcessor` method looks like the following code:

```
public class CustomAnalysisBinderProcessor extends
    AnalysisModule.AnalysisBinderProcessor {
    @Override
    public void processAnalyzers (AnalyzersBindings
        analyzersBindings) {
        analyzersBindings.processAnalyzer ("mastering_analyzer",
            CustomAnalyzerProvider.class);
    }

    @Override
    public void processTokenFilters (TokenFiltersBindings
        tokenFiltersBindings) {
        tokenFiltersBindings.processTokenFilter ("mastering_filter",
            CustomFilterFactory.class);
    }
}
```

The first method, `processAnalyzers()`, takes a single `AnalysisBinding` object type, which we can use to register our analyzer under a given name. We do that by calling the `processAnalyzer` method of the `AnalysisBinding` object and pass it with the name under which our analyzer will be available and the implementation of `AbstractIndexAnalyzerProvider` which is responsible for creating our analyzer, which in our case is the `CustomAnalyzerProvider` class.

The second method, `processTokenFilters`, again takes a single `TokenFiltersBindings` class, which enables us to register our token filter. We do that by calling the `processTokenFilter` method and passing the name under which our token filter will be available, and the token filter factory class, which in our case is the `CustomFilterFactory`.

Implementing analyzer indices component

Analyzer indices component is a node level component that will allow our analyzer and token filter to be reused. However, we will tell Elasticsearch that the analyzer should be usable only when on indices level, not global one (just to show you how to do it). What we need to do is extend the `AbstractComponent` class from the `org.elasticsearch.common.component` package. In fact, we only need to develop a constructor for the class we called `CustomAnalyzerIndicesComponent`. The whole code for the mentioned class looks as follows:

```
public class CustomAnalyzerIndicesComponent extends
    AbstractComponent {
    @Inject
    public CustomAnalyzerIndicesComponent(Settings settings,
        IndicesAnalysisService indicesAnalysisService) {
        super(settings);
        indicesAnalysisService.analyzerProviderFactories().put(
            "mastering_analyzer",
            new PreBUILTAnalyzerProviderFactory("mastering_analyzer",
                AnalyzerScope.INDICES, new CustomAnalyzer(
                    Lucene.ANALYZER_VERSION)));

        indicesAnalysisService.tokenFilterFactories().put("mastering_filt
r",
            new PreBUILTTokenFilterFactoryFactory(new
                TokenFilterFactory() {
                @Override
                public String name() {
                    return "mastering_filter";
                }
            }));
    }
}
```

```

    }

    @Override
    public TokenStream create(TokenStream tokenStream) {
        return new CustomFilter(tokenStream);
    }
    }));
}
}

```

First of all, we pass the constructor argument to the super class in order to initialize it. After that we create a new analyzer, which is our `CustomAnalyzer`, by using the following code snippet:

```

indicesAnalysisService.analyzerProviderFactories().put (
    "mastering_analyzer",
    new PreBuiltAnalyzerProviderFactory("mastering_analyzer",
        AnalyzerScope.INDICES, new CustomAnalyzer(
            Lucene.ANALYZER_VERSION)));

```

As you can see, we've used the `indicesAnalysisService` object and its `analyzerProviderFactories()` method to get Map of `PreBuiltAnalyzerProviderFactory` (as a value and the name as a key in the map) and we've put a newly created `PreBuiltAnalyzerProviderFactory` object with the name of `mastering_analyzer`. In order to create the `PreBuiltAnalyzerProviderFactory` object, we've used our `CustomAnalyzer` and `AnalyzerScope.INDICES` enum value (from the `org.elasticsearch.index.analysis` package). The other values of `AnalyzerScope` enum are `GLOBAL` and `INDEX`. If you would like the analyzer to be globally shared, you should use `AnalyzerScope.GLOBAL` and you should use `AnalyzerScope.INDEX` to be created for each index.

In a similar way, we add our token filter, but this time we use the `tokenFilterFactories()` method of the `IndicesAnalysisService` object, which returns a Map of `PreBuiltTokenFilterFactoryFactory` as a value and a name (a string object) as a key. We put a newly created `TokenFilterFactory` with the name of `mastering_filter`.

Implementing analyzer module

Analyzer module is a simple class, called `CustomAnalyzerModule`, extending `AbstractModule` from the `org.elasticsearch.common.inject` package. It is used to tell Elasticsearch that our `CustomAnalyzerIndicesComponent` class should be used as singleton (we do that because it's enough to have a single instance of that class). Its code looks as follows:

```
public class CustomAnalyzerModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(CustomAnalyzerIndicesComponent.class).asEagerSingleton();
    }
}
```

As you can see, we implement a single `configure` method, which tells us to bind the `CustomAnalyzerIndicesComponent` class as singleton.

Implementing analyzer plugin

Analyzer plugin is the Elasticsearch plugin implementation which will inform Elasticsearch about the plugin. It should extend the `AbstractPlugin` class from the `org.elasticsearch.plugins` package and thus implements at least the `name` and `descriptions` methods. However, we want our plugin to be registered and that's why we implement two additional methods, which we can see in the following code snippet:

```
public class CustomAnalyzerPlugin extends AbstractPlugin {
    @Override
    public Collection<Class<? extends Module>> modules() {
        return ImmutableList.<Class<? extends
            Module>>of(CustomAnalyzerModule.class);
    }

    public void onModule(AnalysisModule module) {
        module.addProcessor(new CustomAnalysisBinderProcessor());
    }

    @Override
    public String name() {
        return "AnalyzerPlugin";
    }

    @Override
    public String description() {
        return "Custom analyzer plugin";
    }
}
```

The `name` and `description` methods are quite obvious as they are returning the name of the plugin and its description. The `onModule()` method adds our `CustomAnalysisBinderProcessor` to the `AnalysisModule` object provided to it.

The last method is the one we are not yet familiar with, the `modules` method:

```
public Collection<Class<? extends Module>> modules() {  
    return ImmutableList.<Class<? extends  
        Module>>of (CustomAnalyzerModule.class);  
}
```

We override that method from the super class in order to return a collection of modules that our plugin is registering. In this case, we are registering a single module class, the `CustomAnalyzerModule` class and we are returning a list with a single entry.

Informing Elasticsearch about our custom analyzer

Once we have our code ready, we need to add one additional thing – we need to let Elasticsearch know what the class registering our plugin is – the one we've called `CustomAnalyzerPlugin`. In order to do that, we create an `es-plugin.properties` file in the `src/main/resources` directory with the following content:

```
plugin=pl.solr.analyzer.CustomAnalyzerPlugin
```

We only specified the `plugin` property there, which should have a value of the class we use for registering our plugins (the one that extends the Elasticsearch `AbstractPlugin` class). This file will be included in the jar file that will be created during the build process and will be used by Elasticsearch during plugin load process.

Testing our custom analysis plugin

Now we want to test our custom analysis plugin just to be sure that everything works. In order to do that, we need to build our plugin, install it on all nodes in our cluster, and finally, use the Admin Indices Analyze API to see how our analyzer works. Let's do that.

Building our custom analysis plugin

We start with the easiest part: building our plugin. In order to do that, we run a simple command:

```
mvn compile package
```

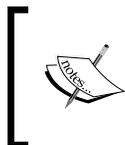
We tell Maven that we want the code to be compiled and packaged. After the command finishes, we can find the archive with the plugin in the `target/release` directory (assuming you are using similar project setup to the one we've describe in the beginning of the chapter).

Installing the custom analysis plugin

To install the plugin, we will use the `plugin` command, just like we did before. Assuming that we have our plugin archive stored in `/home/install/es/plugins` directory we would run the following command (we run it from the Elasticsearch home directory):

```
bin/plugin --install analyzer --url  
file:/home/install/es/plugins/elasticsearch-analyzer-0.90.3.zip
```

We need to install the plugin on all the nodes in our cluster, because we want Elasticsearch to be able to find our analyzer and filter no mater on which node the analysis process is done. If we don't install the plugin on all nodes, we can be certain that we will run into issues.



In order to learn more about installing Elasticsearch plugins, please refer to our previous book, *ElasticSearch Server* or to the official Elasticsearch documentation <http://www.elasticsearch.org/guide/reference/modules/plugins/>.

After we've the plugin installed, we need to restart our Elasticsearch instance we were making the installation on. After restart, we should see something like this in the logs:

```
[2013-08-24 21:45:49,344] [INFO ] [plugins  
[Tattletale] loaded [AnalyzerPlugin], sites []
```

With the previous log line, Elasticsearch informed us that the plugin, named `AnalyzerPlugin`, was successfully loaded.

Checking if our analysis plugin works

We can finally check if our custom analysis plugin works as it should. In order to do that we start with creating an empty index called test (index name doesn't matter). We do that by running the following command:

```
curl -XPOST 'localhost:9200/test/'
```

After that we use the Admin Indices Analyze API (<http://www.elasticsearch.org/guide/reference/api/admin-indices-analyze/>) to see how our analyzer works. We do that by running the following command:

```
curl -XGET  
'localhost:9200/test/_analyze?analyzer=mastering_analyzer&pretty' -d  
'mastering elasticsearch'
```

So, what we should see in response is two tokens, one which should be reversed `mastering`, `gniretsam` and the second one which should be reversed `elasticsearch`, `hcraftsalsale`. The response ElasticSearch returned looks like the following code:

```
{  
  "tokens" : [ {  
    "token" : "gniretsam",  
    "start_offset" : 0,  
    "end_offset" : 9,  
    "type" : "word",  
    "position" : 1  
  }, {  
    "token" : "hcraftsalsale",  
    "start_offset" : 10,  
    "end_offset" : 23,  
    "type" : "word",  
    "position" : 2  
  } ]  
}
```

As you can see, we've got exactly what we expected, so it seems that our custom analysis plugin works as intended.

Summary

In this chapter you've learned how to properly setup your Maven project to be able to automatically build your Elasticsearch plugins. You've seen how to develop a custom river plugin that can be used to index data while being run inside our Elasticsearch cluster. Finally, we've extended Elasticsearch analysis capabilities by creating a plugin that included a custom token filter and new analyzer.

We've reached the end of the book and because of that we wanted to write a small summary and say a few words to the brave reader who managed to get to the end. We wrote this book, because we felt that the resources available online are not enough. Of course, you could get into the source code, try on your own, and you would manage to get the knowledge; however, we wanted to make it easier. We went from introduction to Apache Lucene and Elasticsearch, through querying and data handling, both on the Lucene index and Elasticsearch level. We hope that by now, you know how Lucene works and how Elasticsearch uses it and you will find that knowledge worthy in your journey with this great search engine. We've discussed shard allocation process in greater details, so you know how it works, know how to control it, and alter its behavior when you need it. We've talked about some topics that can be useful when things are hot, such as I/O throttling, hot threads API, and how to speed up your queries.

We also decided to write the whole chapter, the longest one, on query relevance, user search experience, and a small introduction to search analytics. We hope that after reading this chapter, you'll be able to improve query relevance of your application and maybe start using search analytics to measure how your users behave and what are the possible points you can improve.

Finally, we dedicated two chapters to discuss Java development: how to use Elasticsearch Java API and how to extend Elasticsearch with your own plugins. Although we didn't describe the whole methods of the Java API that Elasticsearch provides (we would have to write another book just about it), we hope that you'll be able to use the API and you'll know where to look if you will need deeper knowledge. We also think that now, you'll be able to write your own plugins and even though we didn't write about all the possibilities, we hope that you'll be able to find the things we didn't write about.

Thank you for reading the book. We hope that you like it and that it brought you some knowledge that you were seeking and which you'll be able to use whether you use Elasticsearch professionally or just as a hobby.

Finally, please stop by from time-to-time to <http://elasticsearchserverbook.com/>. In addition to usual posts we make, we will publish the book fragments that didn't make it into the book or were cut down, because the book would be too broad.

Index

Symbols

- `_all` field 248
- `_cache_key` property 55
- `_local` property 138
- `_only_node` option 138
- operator 12
- + operator 12
- `_prefer_node` option 138
- `_primary_first` property 138
- `_primary` property 137
- `_routing` field 113
- `_source` field 48, 49
- `_suggest` REST endpoint
 - using 218

A

- AbstractComponent** class 334
- AbstractModule** class 322, 329
- AbstractPlugin** class 322, 330, 342
- AbstractRiverComponent** class 323, 327
- `acceptable_overhead_ratio` property 83
- accuracy option, term suggester 226
- ActionGenerator**
 - URL 140
- `actionGet()` method 272, 274, 276
- actions**
 - performing 295
- actions, performing**
 - bulk 296
 - delete by query 296
 - Multi GET 296
 - Multi Search 297
- `addFields()` method 285
- `addHighlightedField()` method 293

- additive smoothing**
 - about 232
 - URL 233
- `addListener()` method 273
- `addScriptParam(String, Object)` method 280
- `addSort()` method 290
- `addTransportAddress` method 270
- `admin().cluster()` method 302
- Admin Indices Analyze API**
 - URL 345
- administration API**
 - cluster administration API 302
 - Indices administration API 306
- `after_effect` parameter 77
- aliases** 117
- aliases API** 310
- aliases exists API** 311
- `all()` method 304
- allocation awareness**
 - about 126-128
 - forcing 128
- alpha parameter** 232
- always value** 122
- Amazon EC2 discovery**
 - about 158
 - configuration properties 162
 - gateway configuration 161
 - gateway recovery process 161
 - nodes expectations 163
 - nodes expectations 163
 - plugin configuration 159
 - plugin's installation 159
 - recovery configuration 161
- analyze API**
 - about 312
 - URL 91

- analyzer**
 - about 10
 - changing, during indexing 95, 96
 - changing, during searching 96, 97
- analyzer indices component**
 - implementing 340, 341
- analyzer module**
 - implementing 342
- analyzer option, term suggester 224**
- analyzer plugin**
 - implementing 342, 343
- analyzer provider**
 - implementing 338-340
- analyzerProviderFactories method 341**
- andFilter() method 291**
- AND operator 12**
- Apache**
 - URL 323
- Apache JMeter**
 - URL 140
- Apache Lucene**
 - about 8, 32, 33
 - architecture 8, 9
 - data, analyzing 10
 - Lucene query language 11
- Apache Lucene scoring**
 - altering 71
 - document, matching 26, 27
 - ElasticSearch viewpoint 29
- Apache Lucene TF/IDF scoring formula**
 - URL 28
- Apache Maven**
 - URL 316
- Apache Maven project**
 - about 316
 - build process, running 319
 - creating 316
 - Maven Assembly plugin 319-322
 - POM 317, 318
 - structure 317
- API**
 - anatomy 272, 273
- append method 336**
- array() method 301**
- AwarenessAllocationDecider 123**

B

- balanced ShardAllocator 120**
- basic_model parameter 77**
- BigDesk**
 - URL 140
- Binder class 329**
- bloom_default codec 81**
- bloom filter**
 - URL 81
- Bloom filter based codec properties**
 - delegate property 84
 - ffp property 84
- bloom_pulsing codec 81**
- books field 59**
- Boolean model**
 - URL 27
- Bootstrap process 18, 19**
- bootstrap.mlockall property 146**
- budget 99**
- buffer method 336**
- build process**
 - running 319
- bulk 296**

C

- calibrate_size_by_deletes property 102, 103**
- cancel command 303**
- candidate generators**
 - configuring 233
 - direct generator 233
- category_facet class 263**
- changes**
 - committing 86
- clear cache API 311**
- client class 285**
- Cluster 16**
- cluster administration API**
 - cluster and indices health API 302
 - cluster state API 303
 - nodes hot threads API 305
 - nodes information API 304
 - nodes shutdown API 305
 - node statistics API 304
 - reroute API 303
 - search shards API 305
 - update settings API 303

- cluster and indices health API** 302
- cluster-level recovery configuration**
 - indices.recovery.compress 165
 - indices.recovery.concurrent_streams 165
 - indices.recovery.file_chunk_size 165
 - indices.recovery.max_bytes_per_sec 165
 - indices.recovery.translog_ops 165
 - indices.recovery.translog_size 165
- cluster-level updates** 130, 131
- ClusterRebalanceAllocationDecider** 122
- cluster.routing.allocation.allow_rebalance** property 135
- cluster.routing.allocation.cluster_concurrent_rebalance** property 135
- cluster.routing.allocation.disable_allocation** property 136
- cluster.routing.allocation.disable_new_allocation** property 135
- cluster.routing.allocation.disable_replica_allocation** property 136
- cluster.routing.allocation.node_concurrent_recoveries** property 135
- cluster.routing.allocation.node_initial primaries_recoveries** property 135
- cluster state API** 303
- code** 267
- codec behavior**
 - bloom filter based codec properties 84
 - configuring 82
 - default codec properties 83
 - direct codec properties 83
 - memory codec properties 83
 - pulsing codec properties 83
- codecs**
 - posting formats 81
 - using 78
 - working 79, 80
- commit command** 86
- committed property** 169
- completion suggester**
 - about 238
 - additional parameters 243
 - custom weights 241, 242
 - data, indexing 239, 240
 - data, querying 240, 241
 - using 238, 239
- compound property** 169
- concurrent merge scheduler** 103
- ConcurrentRebalanceAllocationDecider** 123
- conditional modifications**
 - scripting, using 50
- Confidence option** 230
- configuration**
 - directories layout 143
 - discovery module configuration 144
 - garbage collector work, logging 145
 - gateway configuration 143
 - indices configuration 143
 - memory setup 146
 - node-level configuration 143
 - recovery 144
 - slow queries, logging 145
- constant_score_boolean rewrite method** 34
- constant_score_filter rewrite method** 34
- count property** 263
- cpu usage command** 207
- createComponent method** 338
- create method** 336
- CRUD operations**
 - documents, deleting 282, 283
 - documents, fetching 274, 275
 - documents, indexing 276, 278
 - documents, updating 279-282
- ctx variable** 50
- CustomAnalysisBinderProcessor**
 - method 339
- custom analysis plugin**
 - analysis binder, implementing 339, 340
 - analyzer indices component, implementing 340, 341
 - analyzer module, implementing 342
 - analyzer plugin, implementing 342, 343
 - analyzer provider, implementing 338, 339
 - building 344
 - creating 333
 - custom analyzer, implementing 337
 - installing 344
 - testing 343
 - TokenFilter factory, implementing 336
 - TokenFilter, implementing 335, 336
 - working 345
- custom analyzer**
 - implementing 337

CustomAnalyzer class 337
CustomAnalyzerModule class 343
CustomAnalyzerProvider class 338, 340
CustomFilter class 337
custom river plugin
 creating 322
 JSONRiver class, implementing 327, 328
 JSONRiverModule class, implementing 329
 JSONRiverPlugin class,
 implementing 329, 330
 URLChecker class, implementing 324-326
custom ShardAllocator 121
custom weights 241, 242

D

daemonThreadFactory method 328
Damerau Levenshtein string
 distance algorithm
 URL 227
data
 analyzing 10
 indexing 21, 239, 240
 querying 22, 240, 241
 sorting 43
data, analyzing
 indexing 11
 querying 11
data handling
 analyzer, changing during indexing 95, 96
 analyzer, changing during searching 96, 97
 default analysis 97
 example usage 94, 95
 input analysis 90-94
 pitfall 97
data, sorting
 with multivalued fields 44
 with multivalued geo fields 45, 46
 with nested objects 47, 48
deciders
 about 121
 AwarenessAllocationDecider 123
 ClusterRebalanceAllocationDecider 122
 ConcurrentRebalanceAllocationDecider 123
 DisableAllocationDecider 123
 DiskThresholdDecider 124
 FilterAllocationDecider 122

 RebalanceOnlyWhenActiveAllocation
 Decider 124
 ReplicaAfterPrimaryActiveAllocation
 Decider 122
 SameShardAllocationDecider 121
 ShardsLimitAllocationDecider 122
 ThrottlingAllocationDecider 124
default codec 81
default codec properties
 max_block_size property 83
 min_block_size property 83
default_field property 258
default refresh time
 changing 86
default similarity model
 choosing 75
delegate property 84
delete by query 296
deleted_docs property 169
delete mapping API 310
delete template API 313
delete warmer API 314
description field 11
description method 329, 343
DFR similarity
 configuring 77
direct codec 81
direct codec properties
 low_freq_cutoff property 83
 min_skip_count property 83
direct generators
 about 233
 configuring 234-237
directories layout 143
DisableAllocationDecider 123
discount_overlaps property 76
discovery configuration
 about 155
 Amazon EC2 discovery 158
 local gateway 163
 recovery configuration 164
 Zen discovery 155
discovery.ec2.any_group property 160
discovery.ec2.availability_zones
 property 160
discovery.ec2.groups property 160
discovery.ec2.host_type property 160

- discovery.ec2.tag property** 160
- discovery module configuration** 144
- discovery.zen.fd.ping_interval property** 158
- discovery.zen.fd.ping_retries property** 158
- discovery.zen.fd.ping_timeout property** 158
- DiskThresholdDecider** 124
- distribution property** 78
- Divergence from randomness similarity**
 - about 72
 - URL 72
- Document** 15
- document count** 294, 295
- documents**
 - building 300, 301
 - creating, Update API used 50, 51
 - deleting 283, 284
 - deleting, Update API used 50, 51
 - errors, handling 276
 - fetching 274, 275
 - indexing 276, 278
 - matching 26, 27
 - updating 279-282
- document type** 16
- Drill downs**
 - with faceting 260-263

E

- EC2 plugin's installation**
 - configuration 159
 - optional configuration options 159
 - scanning configuration 160
- Eclipse**
 - URL 267
- ElasticSearch**
 - about 15
 - administration 23
 - architecture 17, 18
 - caching behavior, changing 54
 - Cluster 16
 - communicating with 20
 - data, indexing 21
 - data, querying 22
 - Document 15
 - document count 294, 295
 - document type 16

- explain API 299
- faceting 292
- filtering 290, 291
- garbage collector work, adjusting 190
- gateway 17
- highlighting 292, 293
- Index 15
- index, configuring 23
- mapping 16
- monitoring 23
- node 16
- paging 289
- query, building 285, 286
- querying 284
- query, preparing 284, 285
- Replica 17
- scrolling 295
- shard 17
- sorting 290
- suggestion 293, 294
- working 18
- ElasticSearch 1.0** 298, 299
- ElasticSearch caching**
 - about 170
 - cached, clearing 180
 - clearing 180
 - field data cache 173
 - fields related caches, clearing 182
 - filter cache 171
 - index, clearing 181
 - indices, clearing 181
 - specific caches, clearing 181
- ElasticSearch cluster**
 - connecting to 268
 - ElasticSearch node 268-270
 - right connection method, choosing 271
 - transport connection method, using 270
- ElasticSearch documentation**
 - URL 344
- ElasticSearch Java API** 266
- ElasticSearch node**
 - connecting to 268-270
- ElasticSearch paramedic**
 - URL 140
- ElasticSearch Server**
 - URL 208

- ElasticSearch, working**
 - Bootstrap process 18, 19
 - failure detection 19
- even_shard ShardAllocator** 119
- example data** 35
- exclude property** 130
- exclusion** 134
- execute method** 326
- execute() method** 273, 274
- explain API** 299
- F**
- FacetBuilder object** 292
- facet filter** 65, 66
- faceting**
 - about 292
 - filters, using 61, 63
- failure detection** 19
- ffp property** 84
- field**
 - updating 49
- field data cache**
 - about 173
 - filtering 175
 - index-level field data cache
 - configuration 174
 - Node-level field data cache
 - configuration 174
- field() method** 301
- field option, term suggester** 224
- field parameter** 224
- fields**
 - querying 13
- FilterAllocationDecider** 122
- FilterBuilder class** 291
- filter cache**
 - about 171
 - index-level filter cache configuration 172
 - node-level filter cache configuration 173
 - types 171
 - used, for storing filter results 52, 53
- filtering**
 - about 128, 129, 175, 290, 291
 - by frequency term 176
 - by regex 177
 - by term frequency 177
- example 178-180
- field data filtering information, adding 175
- filters**
 - used, to optimize query 51
- Finite State Transducer.** *See* FST
- Flush API** 309
- flush command** 88
- flushing** 87
- force_unigrams option** 230
- freq_cut_off property** 83
- freq property** 221
- FST**
 - URL 238
- G**
- garbage**
 - removing 254-256
- garbage collector**
 - about 184
 - Java memory 184, 185
 - JStat, using 187-189
 - memory dumps, creating 189
 - problems, identifying with 186
 - swapping, avoiding on
 - Unix like systems 191, 192
- garbage collector work**
 - adjusting, in ElasticSearch 190
 - logging 145
 - logging, turning on 186
 - service wrapper 191
 - standard startup script, using 190
 - tuning 189, 190
- gateway** 17
- gateway configuration** 143
- gateway.expected_data_nodes property** 163
- gateway.expected_master_nodes property** 163
- gateway.expected_nodes property** 163
- gateway.recover_after_data_nodes property** 162
- gateway.recover_after_master_nodes property** 162
- gateway.recover_after_nodes property** 162
- gateway.recover_after_time property** 162
- gateway snapshot API** 310

- generation property 169
- geoDistanceSort(String) method 290
- geo shape query
 - using 288
- geoShapeQuery() method 288, 289
- get aliases API 311
- getExplanation() method 299
- getFacets() method 292
- getField(String) method 275
- getGetResult() method 282
- getId() method 275, 279, 282, 284
- getIndex() method 275, 279, 282, 284
- getItems() method 296
- getMatches() method 279, 282, 298
- getResponse() method 296
- GetResponse object 275, 297
- getResponses() method 297
- getScrollId() method 295
- getSourceXXX() method 275
- getType() method 275, 279, 282, 284
- getVersion() method 275, 279, 282, 284
- global property 67
- global scope 67, 68
- gram_size option 230
- group property 125

H

- heterogeneous environment 210, 211
- Hot Threads API
 - about 204
 - response 206, 207
 - using 205
- HTTP HEAD method 326

I

- IB similarity
 - configuring 78
- ImmutableSettings class 270
- include property 129
- inclusion 132
- incrementToken method 336
- index
 - about 15
 - closing 308

- configuring 23
 - deleting 308
 - opening 308
 - updating 86
- index API
 - creating 307
- index.compound_format property 101-103
- index creation
 - warmers, adding 198
- index existence API 306
- index.fielddata.cache.type property 174
- indexLastModified method 326
- Index-level filter cache configuration
 - index.cache.filter.expire property 172
 - index.cache.filter.max_size property 172
 - index.cache.filter.type property 172
- index-level updates 130
- index.merge.async_interval property 101
- index.merge.async property 101
- index.merge.policy.expunge_deletes_
 - allowed property 100
- index.merge.policy.floor_segment
 - property 100
- index.merge.policy.max_merge_at_once_
 - explicit property 100
- index.merge.policy.max_merged_segment
 - property 99, 100
- index.merge.policy.segments_per_tier
 - property 101
- index.query.bool.max_clause_count
 - property 34
- index.reclaim_deletes_weight property 101
- index.refresh_interval parameter 86
- IndexResponse class 279
- index status 307
- index.translog.disable_flush option 88
- index.translog.flush_threshold_ops
 - option 88
- index.translog.flush_threshold_period
 - option 88
- index.translog.flush_threshold_size
 - option 88
- index.warmer.enabled property 200
- Indices administration API
 - aliases API 310
 - aliases exists API 311

- analyze API 312
- clear cache API 311
- delete mapping API 310
- delete template API 313
- delete warmer API 314
- Flush API 309
- gateway snapshot API 310
- get aliases API 311
- index API, creating 307
- index, closing 308
- index, deleting 308
- index existence API 306
- index, opening 308
- index status 307
- indices stats API 306
- Optimize API 309
- put mapping API 309
- put template API 312
- put warmer API 314
- Refresh API 308
- segments information API 307
- Type existence API 306
- update settings API 312
- validate query API 313
- indices_all_active value 122**
- indicesAnalysisService object 341**
- indices.cache.filter.terms.expire_after_access property 60**
- indices.cache.filter.terms.expire_after_write property 60**
- indices.cache.filter.terms.size property 60**
- indices configuration 143**
- indices primaries_active value 122**
- indices stats API 306**
- indices.store.throttle.type property 193**
- InetSocketAddress object 270**
- Information based similarity 72**
- input analysis 90, 91**
- input property 240**
- interrupt method 328**
- interval parameter 204**
- inverted index 9**
- I/O throttling**
 - configuring 193
 - controlling 193

- I/O throttling configuration**
 - example 194, 195
 - maximum throughput per second 194
 - node throttling 194
 - throttling type 193
- isDone() method 273**
- isExists() method 275**
- isFailed() method 296**
- isFailure() method 297**
- isNotFound() method 284**
- isSourceEmpty() method 275**

J

- Javadocs**
 - URL 325
- Java memory 184, 185**
- Java object**
 - lifecycle 185
- Java Virtual Machine**
 - URL 184
- JDK package**
 - URL 328
- jhat**
 - URL 189
- jmap**
 - URL 189
- JSON object**
 - URL 15
- JSON queries**
 - building 300, 301
- JSONRiver class**
 - implementing 327, 328
- JSONRiverModule class**
 - implementing 329
- JSONRiverPlugin class**
 - implementing 329, 330
- JStat**
 - using 187-189
- jstat command 186, 187**
- JUnit**
 - URL 267
- JVM memory**
 - Code cache 185
 - Eden space 184
 - Permanent generation 185

- Survivor space 184
- Tenured generation 185

JVM parameters

- URL 189

K

knowledge

- data volume 140

L

lambda property 78

language models

- URL 231

Laplace smoothing model. *See* **addictive smoothing**

Least Recently Used. *See* **LRU**

Levenshtein distance

- URL 227

Linear interpolation smoothing model 233

load imbalance 210, 211

local gateway

- about 163
- backing up 164

local(true) method 269

log byte size merge policy 99

log byte size merge policy,

- configuration** 101

log doc merge policy 100

log doc merge policy, configuration 102

lowercase_terms option, term suggester 225

low_freq_cutoff property 83

LRU 173

LRU cache

- URL 60

Lucene 8

Lucene conceptual formula 27

Lucene practical formula 28

Lucene query language

- about 12
- fields, querying 13
- special characters, handling 14
- term modifiers 13

M

map() method 301

mapping 16

Mastering 137

match all documents query

- using 287

matchAllQuery() query 287

matchedFilters() method 291

match query 287

Maven Assembly plugin

- about 319-322
- URL 319

Maven lifecycle

- URL 319

Maven project

- URL 267

max_block_size property 83

max_edits option, term suggester 225

max_errors property 230

max_inspections option, term suggester 226

max_merge_docs property 102

maxMergeDocs property 102

max_merge_size property 102

max_term_freq option, term suggester 226

memory codec 81

memory codec properties

- acceptable_overhead_ratio property** 83
- pack_fst property** 83

memory setup 146

memory store

- cache.memory.direct property** 154
- cache.memory.large_buffer_size property** 154
- cache.memory.large_cache_size property** 154
- cache.memory.small_buffer_size property** 154
- cache.memory.small_cache_size property** 154

merge_factor property 101, 102

merge policy

- log byte size merge policy 99
- log doc merge policy 100
- tiered merge policy 99

- merge policy configuration**
 - log byte size merge policy 101
 - log doc merge policy 102
 - tiered merge policy 100, 101
- Mike McCandless**
 - URL 81
- Mike McCandless blog post**
 - URL 104
- min_block_size** property 83
- min_doc_freq** option, term suggester 226
- min_doc_freq** parameter 226
- minimum_should_match** property 258
- min_merge_docs** property 102
- min_merge_size** property 101
- min_segment_size** property 176
- min_skip_count** property 83
- min_word_len** option, term suggester 225
- misspelling proof search**
 - making 257-259
- MMap filesystem store** 153
- mode** parameter 44
- mode** property 46
- move command** 303
- Multi GET** 296
- MultiGet operation** 40, 41
- multi match query** 248, 249
- multiple indices**
 - about 148
 - versus multiple shards 108
- multiple routing values** 118
- multiple shards**
 - versus multiple indices 108
- Multi Search** 297
- MultiSearch operation** 41-43
- multivalued fields**
 - data, sorting with 44
- multivalued geo fields**
 - data, sorting with 45, 46
- Munin**
 - URL 146

N

- name** method 329, 343
- Near Real Time (NRT)** 18
- nested objects**
 - data, sorting with 47, 48

- newThread** method 328
- n-gram language model**
 - URL 227
- n-gram smoothing models**
 - URL 232
- node** 16
- NodeBuilder** class 269
- node.group** property 126, 128
- node-level configuration** 143
- Node-level field data cache configuration**
 - indices fielddata.cache.expire, property 174
 - indices fielddata.cache.size property 174
- nodes hot threads** API 305
- nodes information** API 304
- nodes shutdown** API 305
- node statistics** API 304
- node throttling** 194
- normalization** parameter 77
- NOT operator** 12
- nullField()** method 301
- number** property 169
- num_docs** property 169

O

- Okapi BM25 similarity**
 - about 72
 - configuring 77
 - URL 72
- onFailure()** method 273
- onModule** method 329, 330, 343
- operations**
 - MultiGet operation 39-41
 - MultiSearch operation 41-43
- Optimize** API 309
- org.apache.lucene.analysis.Analyzer**
 - class 334
- OR operator** 12
- over allocation**
 - about 106, 107
 - example 108

P

- pack_fst** property 83
- paging** 289
- path** parameter 96, 113

- path property** 58
- payload property** 240
- payments** 195
- percolator**
 - about 297, 298
 - ElasticSearch 1.0 298, 299
- per-field similarity**
 - setting 73, 74
- performance degradation** 207-210
- phrase query** 250-254
- phrase suggester**
 - about 227
 - configuring 229
 - example 228, 229
- phrase suggester configuration**
 - basic configuration 230, 231
 - candidate generators, configuring 233-237
 - smoothing models, configuring 231-233
- plugin command** 331, 344
- POM** 317, 318
- post_filter property** 235
- posting_format property** 80
- posting formats** 81
- prcltr** 297
- preference parameter** 137
- pre_filter property** 234
- prefix query** 29-31
- prepareCount() method** 295
- prepareExplain method** 299
- prepareGet() method** 272, 274
- prepareHealth() method** 302
- prepareNodesHotThreads() method** 305
- prepareSearchScroll method** 295
- preserve_position_increments**
 - parameter 243
- preserve_separators parameter** 243
- prettyPrint() method** 301
- processAnalyzer method** 340
- pulsing codec** 81
- Pulsing codec properties**
 - freq_cut_off property 83
- put mapping API** 309
- put template API** 312
- put warmer API** 314
- PUT Warmer API**
 - using 197

Q

- query**
 - about 36
 - building 115-117, 285, 286
 - geo shape query, using 288
 - logging 145
 - match all documents query, using 287
 - match query 287
 - optimizing, filters used 51
 - preparing 284, 285
 - rewriting 29
 - speeding up, warmers used 196
- QueryBuilder class** 285, 288
- query execution preference**
 - about 136, 137
 - preference parameter 137
- query object** 37, 60
- query relevance**
 - data 244, 245
 - improving 243
- query, rewriting**
 - prefix query 29-31
 - properties 33-35
- query_weight parameter** 38

R

- real life examples**
 - heterogeneous environment 210, 211
 - load imbalance 210, 211
 - performance degradation 207-210
- real-time GET operation** 89
- real_word_error_likelihood option** 231
- RebalanceOnlyWhenActiveAllocation**
 - Decider 124
- recovery configuration**
 - about 144, 164
 - cluster-level 165
 - Index-level 166
- Refresh API** 308
- re-indexing** 147
- release_dates field** 44
- remove() method** 50
- Replica** 17

**ReplicaAfterPrimaryActiveAllocation
Decider 122**

replicas 108

request() method 272

require property 129

reroute API 303

rescore

about 35

example data 35

parameters 38

query 36

query example 36, 37

summing up 39

rescore_mode parameter 38

rescore object 37

rescore_query_weight parameter 38

REST

URL 20

REST endpoint suggester response 219, 220

rewrite parameter 33

rewrite property 34

right connection method

using 271

river

building 331

initializing 332

installing 331

testing 331

working 333

RiversModule class 330

routing

about 148

advantages 113

aliases 117

indexing with 112-115

multiple routing values 118

queries, building 115-117

shards 109

testing 110-112

routing parameter 115

run method 325, 328

Runnable interface 323

run time allocation

cluster-level updates 130, 131

index-level updates 130

S

SameShardAllocationDecider 121

saturation 77

scheduling

about 103

concurrent merge scheduler 103

desired merge scheduler, setting 104

serial merge scheduler 104

score property 220

scoring_boolean rewrite method 34

script field 50

scripting

used, for conditional modifications 50

SearchHit class 285

SearchHits class 289

search property 169

SearchRequestBuilder class 289

SearchResponse object 295

SearchResult object 292

search shards API

about 305

URL 305

section field 43

segment merging

about 97, 98

segments information

visualizing 170

segments information API 307

segments merge 10

segments statistics

about 166

information, visualizing 170

segments API 167

segments API, response 167-169

Sematest

URL 140

Separator option 230

serial merge scheduler 104

service wrapper

about 191

URL 190

setConsistencyLevel() method 278, 281, 284

setCreate(Boolean) method 277

setDocAsUpsert(Boolean) method 282

setDoc() method 282

- setExplain method 313
- setFacets() method 292
- setFields() method 272
- setFields(String...) method 280
- setFields(String) method 274
- setFilter() method 291
- setIndex(String), setType(String),
 - setId(String) method 274, 277, 280, 283
- setNetwork() method 304
- setOpType() method 277
- setPercolate(String) method 278, 281
- setPlugin() method 304
- setPreference(String) method 275
- setQuery() method 285, 286
- setRealtime(Boolean) method 275
- setRefresh(Boolean)
 - method 275, 277, 281, 283
- setReplicationType() method 278, 281, 283
- setRetryOnConflict(int) method 281
- setRouting(String) method 275
- setRouting(String), setParent(String)
 - method 277, 280, 283
- setScriptLang(String) method 280
- setScriptParams(Map<String, Object>)
 - method 280
- setScript(String) method 280
- setSize(int) method 289
- setSource() method 277, 282
- setSource()method 285
- setTimestamp(String) method 278
- settings method 328
- setTTL(long) method 279
- setUpsertRequest() method 282
- setVersion(long) method 278, 283
- setVersionType(VersionType)
 - method 278, 283
- ShapeBuilder class 289
- shard 17
- shard allocation
 - adjusting 125
 - allocation awareness 126-128
 - altering 119
 - balanced ShardAllocator 120
 - custom ShardAllocator 121
 - even_shard ShardAllocator 119
 - exclusion 134
 - filtering 128, 129
 - inclusion 132
 - properties 135
 - requisites 133, 134
 - run time allocation, updating 130
 - ShardAllocator 119
 - total shards, defining 131
- ShardAllocator 119
- ShardId class 304
- Sharding 106, 107
- shards 109
- shard_size option, term suggester 225
- ShardsLimitAllocationDecider 122
- shingle filter
 - URL 230
- similarity model
 - chosen similarity model, configuring 76
 - configuring 74, 75
 - default similarity model, choosing 75, 76
 - Divergence from randomness 72
 - Information based 72
 - Okapi BM25 72
- single point of failure (SPOF) 18
- size_in_bytes property 169
- size option, term suggester 224
- size parameter 225
- size property 169
- sleep method 326
- slop parameter 252
- smoothing models
 - additive smoothing 232
 - configuring 231, 232
 - Linear interpolation smoothing model 233
 - Stupid backoff smoothing model 232
- snapshots parameter 205
- sorting 290
- sort option, term suggester 224
- special characters
 - handling 14
- specific caches, clearing
 - bloom 181
 - field_data 181
 - filter 181
- standard query 247, 248
- standard startup script
 - using 190

- start method** 327, 328
- stop method** 327
- store module**
 - about 151
 - store types 152
- store types**
 - memory store 153
 - MMap filesystem store 153
 - new IO filesystem store 152
 - simple file system store 152
- StringBuilder object** 336
- string_distance option, term suggester** 227
- string() method** 301
- Stupid backoff smoothing model** 232
- suggester** 218
- suggester object** 218
- suggester response** 222, 223
- suggestion**
 - working 293, 294
- suggestions requests**
 - including, in query 221, 222
- suggest_mode option, term suggester** 225
- swapping**
 - avoiding, on Unix like systems 191, 192

T

- tag property** 125
- templates**
 - warmers, adding to 199
- term frequency/inverse document frequency.** *See* TF/IDF
- term modifiers** 13
- terms lookup filter**
 - about 55-57
 - cache settings 60
 - performance 59
 - working 58, 59
- term suggester**
 - about 224
 - configuring 224
- term suggester configuration**
 - additional term suggester options 225-227
 - common term suggester options 224, 225
- term vectors** 9
- text option, term suggester** 224

- text parameter** 91, 218, 233
- text property** 220
- TF/IDF**
 - about 26
 - Lucene conceptual formula 27
 - Lucene practical formula 28
- TF/IDF similarity**
 - configuring 76
- Thread class** 326, 328
- threads parameter** 204
- ThrottlingAllocationDecider** 124
- throttling type** 193
- tiered merge policy** 99
- tiered merge policy, configuration** 100, 101
- time property** 332
- title field** 11, 43, 65, 293
- TokenFilter**
 - implementing 335, 336
- tokenFilterFactories method** 341
- TokenFilter factory**
 - implementing 336
- tokenizer** 10
- token_limit option** 231
- TokenStreamComponents object** 338
- TokenStream object** 337
- top_terms_boost_N rewrite method** 34
- top_terms_N rewrite method** 34
- toString() method** 286, 291, 292
- totalHits() method** 289
- transaction log**
 - configuring 87, 89
- TransportClient class** 270, 271
- TransportClient object** 270, 271
- transport connection method**
 - using 270
- Type existence API** 306
- type parameter** 204, 205
- type property** 239

U

- Unix like systems**
 - swapping, avoiding 191, 192
- Update API**
 - about 48
 - conditional modifications, scripting used 50

- field update 49
- used, for creating documents 50, 51
- used, for deleting documents 50, 51

update settings API 303, 312

URLChecker class

- implementing 324-326

use case 78

user spelling mistakes

- completion suggester 237
- correcting 216
- data, testing 216, 217
- technical details 217

V

validate query API 313

value() method 301

Vector Space Model

- URL 27

Version class

- URL 337

version property 169

version_type 41

W

warmers

- adding, during index creation 198
- adding, to templates 199
- deleting 200
- disabling 200
- manipulating 197
- need for 196
- PUT Warmer API, using 197
- querying with 203, 204
- querying without 202
- retrieving 199
- testing 201
- used, for speeding up queries 196

weight property 241

WhitespaceTokenizer class 338

window_size parameter 38

X

Xmx parameter 184

Y

year field 51

Z

Zen discovery

- about 155
- fault detection 158
- minimum master nodes 157
- multicast 156
- unicast 157



Thank you for buying Mastering Elasticsearch

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.

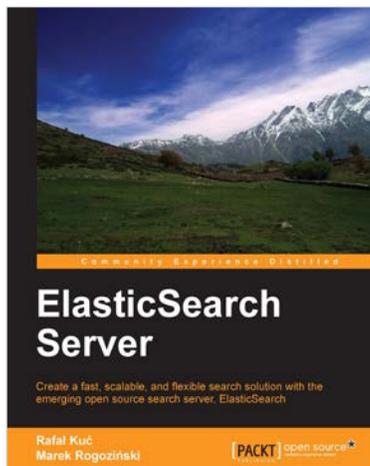
About Packt Open Source

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.

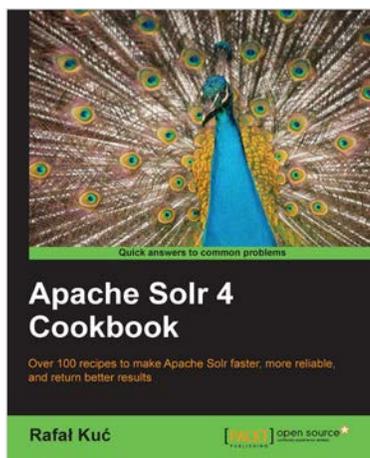


ElasticSearch Server

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

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

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



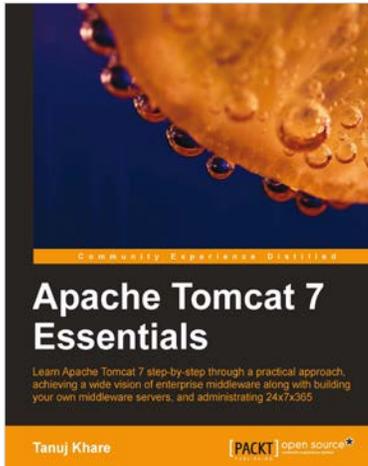
Apache Solr 4 Cookbook

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

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

1. Learn how to make Apache Solr search faster, more complete, and comprehensively scalable
2. Explore the versatility of Spring Python by integrating it with frameworks, libraries, and tools
3. Solve performance, setup, configuration, analysis, and query problems in no time
4. Get to grips with, and master, the new exciting features of Apache Solr 4

Please check www.PacktPub.com for information on our titles

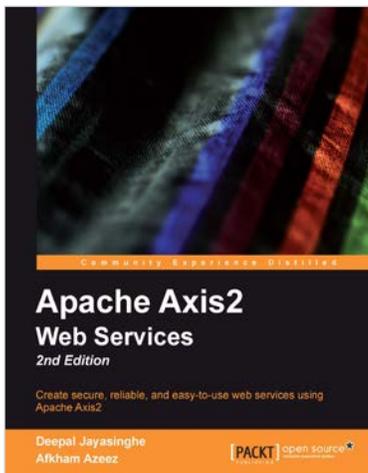


Apache Tomcat 7 Essentials

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

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

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



Apache Axis2 Web Services, 2nd Edition

ISBN: 978-1-84951-156-8 Paperback: 308 pages

Create secure, reliable and east-to-use web services using Apache Axis2

1. Extensive and detailed coverage of the enterprise ready Apache Axis2 Web Services / SOAP / WSDL engine
2. Attain a more flexible and extensible framework with the world class Axis2 architecture
3. Learn all about AXIOM - the complete XML processing framework, which you also can use outside Axis2

Please check www.PacktPub.com for information on our titles