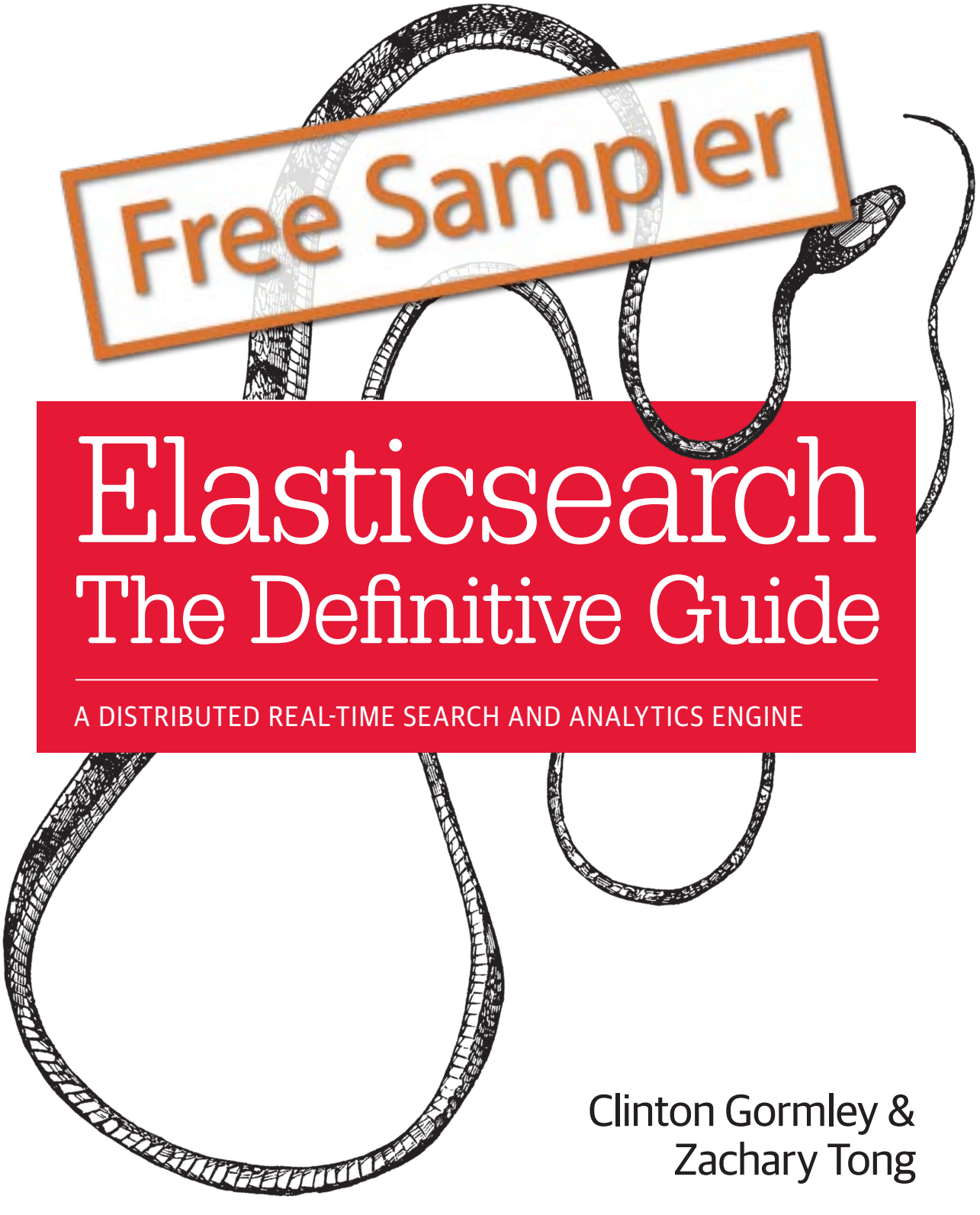
O'REILLY®




# Elasticsearch
## The Definitive Guide

A DISTRIBUTED REAL-TIME SEARCH AND ANALYTICS ENGINE

Clinton Gormley &
Zachary Tong

**Elasticsearch: The Definitive Guide**
by Clinton Gormley and Zachary Tong



Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

**Editors:** Mike Loukides and Brian Anderson
**Production Editor:** Shiny Kalapurakkel
**Proofreader:** Sharon Wilkey
**Indexer:** Ellen Troutman-Zaig

**Interior Designer:** David Futato
**Cover Designer:** Ellie Volkhausen
**Illustrator:** Rebecca Demarest

January 2015: First Edition

**Revision History for the First Edition**
2015-01-16: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449358549* for release details.



978-1-449-35854-9

[LSI]

# Table of Contents

**Foreword** ..... **xxi**

**Preface** ..... **xxiii**

## Part I. Getting Started

**1. You Know, for Search** ..... **3**

Installing Elasticsearch 4
Installing Marvel 5
Running Elasticsearch 5
Viewing Marvel and Sense 6
Talking to Elasticsearch 6
Java API 6
RESTful API with JSON over HTTP 7
Document Oriented 9
JSON 9
Finding Your Feet 10
Let's Build an Employee Directory 10
Indexing Employee Documents 10
Retrieving a Document 12
Search Lite 13
Search with Query DSL 15
More-Complicated Searches 16
Full-Text Search 17
Phrase Search 18
Highlighting Our Searches 19
Analytics 20
Tutorial Conclusion 23

Distributed Nature 23
Next Steps 24

**2. Life Inside a Cluster. . . . . 25**
An Empty Cluster 26
Cluster Health 26
Add an Index 27
Add Failover 29
Scale Horizontally 30
Then Scale Some More 31
Coping with Failure 32

**3. Data In, Data Out. . . . . 35**
What Is a Document? 36
Document Metadata 37
_index 37
_type 37
_id 38
Other Metadata 38
Indexing a Document 38
Using Our Own ID 38
Autogenerating IDs 39
Retrieving a Document 40
Retrieving Part of a Document 41
Checking Whether a Document Exists 42
Updating a Whole Document 42
Creating a New Document 43
Deleting a Document 44
Dealing with Conflicts 45
Optimistic Concurrency Control 47
Using Versions from an External System 49
Partial Updates to Documents 50
Using Scripts to Make Partial Updates 51
Updating a Document That May Not Yet Exist 52
Updates and Conflicts 53
Retrieving Multiple Documents 54
Cheaper in Bulk 56
Don't Repeat Yourself 60
How Big Is Too Big? 60

**4. Distributed Document Store. . . . . 61**
Routing a Document to a Shard 61

How Primary and Replica Shards Interact 62
Creating, Indexing, and Deleting a Document 63
Retrieving a Document 65
Partial Updates to a Document 66
Multidocument Patterns 67
Why the Funny Format? 69

**5. Searching—The Basic Tools. . . . . 71**
The Empty Search 72
hits 73
took 73
shards 73
timeout 74
Multi-index, Multitype 74
Pagination 75
Search *Lite* 76
The _all Field 77
More Complicated Queries 78

**6. Mapping and Analysis. . . . . 79**
Exact Values Versus Full Text 80
Inverted Index 81
Analysis and Analyzers 84
Built-in Analyzers 84
When Analyzers Are Used 85
Testing Analyzers 86
Specifying Analyzers 87
Mapping 87
Core Simple Field Types 88
Viewing the Mapping 89
Customizing Field Mappings 89
Updating a Mapping 91
Testing the Mapping 92
Complex Core Field Types 93
Multivalue Fields 93
Empty Fields 93
Multilevel Objects 94
Mapping for Inner Objects 94
How Inner Objects are Indexed 95
Arrays of Inner Objects 95

**7. Full-Body Search. . . . . 97**
Empty Search 97
Query DSL 98
Structure of a Query Clause 99
Combining Multiple Clauses 99
Queries and Filters 100
Performance Differences 101
When to Use Which 101
Most Important Queries and Filters 102
term Filter 102
terms Filter 102
range Filter 102
exists and missing Filters 103
bool Filter 103
match_all Query 103
match Query 104
multi_match Query 104
bool Query 105
Combining Queries with Filters 105
Filtering a Query 106
Just a Filter 107
A Query as a Filter 107
Validating Queries 108
Understanding Errors 108
Understanding Queries 109

**8. Sorting and Relevance. . . . . 111**
Sorting 111
Sorting by Field Values 112
Multilevel Sorting 113
Sorting on Multivalue Fields 113
String Sorting and Multifields 114
What Is Relevance? 115
Understanding the Score 116
Understanding Why a Document Matched 119
Fielddata 119

**9. Distributed Search Execution. . . . . 121**
Query Phase 122
Fetch Phase 123
Search Options 125
preference 125

timeout 126
routing 126
search_type 127
scan and scroll 127

**10. Index Management. . . . . 131**
Creating an Index 131
Deleting an Index 132
Index Settings 132
Configuring Analyzers 133
Custom Analyzers 134
Creating a Custom Analyzer 135
Types and Mappings 137
How Lucene Sees Documents 137
How Types Are Implemented 138
Avoiding Type Gotchas 138
The Root Object 140
Properties 140
Metadata: _source Field 141
Metadata: _all Field 142
Metadata: Document Identity 144
Dynamic Mapping 145
Customizing Dynamic Mapping 147
date_detection 147
dynamic_templates 148
Default Mapping 149
Reindexing Your Data 150
Index Aliases and Zero Downtime 151

**11. Inside a Shard. . . . . 153**
Making Text Searchable 154
Immutability 155
Dynamically Updatable Indices 155
Deletes and Updates 158
Near Real-Time Search 159
refresh API 160
Making Changes Persistent 161
flush API 165
Segment Merging 166

optimize API 168

## Part II. Search in Depth

**12. Structured Search. . . . . 173**
Finding Exact Values 173
term Filter with Numbers 174
term Filter with Text 175
Internal Filter Operation 178
Combining Filters 179
Bool Filter 179
Nesting Boolean Filters 181
Finding Multiple Exact Values 182
Contains, but Does Not Equal 183
Equals Exactly 184
Ranges 185
Ranges on Dates 186
Ranges on Strings 187
Dealing with Null Values 187
exists Filter 188
missing Filter 190
exists/missing on Objects 191
All About Caching 192
Independent Filter Caching 192
Controlling Caching 193
Filter Order 194

**13. Full-Text Search. . . . . 197**
Term-Based Versus Full-Text 197
The match Query 199
Index Some Data 199
A Single-Word Query 200
Multiword Queries 201
Improving Precision 202
Controlling Precision 203
Combining Queries 204
Score Calculation 205
Controlling Precision 205
How match Uses bool 206
Boosting Query Clauses 207
Controlling Analysis 209

Default Analyzers 211
Configuring Analyzers in Practice 213
Relevance Is Broken! 214

**14. Multifield Search. . . . . 217**
Multiple Query Strings 217
Prioritizing Clauses 218
Single Query String 219
Know Your Data 220
Best Fields 221
dis_max Query 222
Tuning Best Fields Queries 223
tie_breaker 224
multi_match Query 225
Using Wildcards in Field Names 226
Boosting Individual Fields 227
Most Fields 227
Multifield Mapping 228
Cross-fields Entity Search 231
A Naive Approach 231
Problems with the most_fields Approach 232
Field-Centric Queries 232
Problem 1: Matching the Same Word in Multiple Fields 233
Problem 2: Trimming the Long Tail 233
Problem 3: Term Frequencies 234
Solution 235
Custom _all Fields 235
cross-fields Queries 236
Per-Field Boosting 238
Exact-Value Fields 239

**15. Proximity Matching. . . . . 241**
Phrase Matching 242
Term Positions 242
What Is a Phrase 243
Mixing It Up 244
Multivalue Fields 245
Closer Is Better 246
Proximity for Relevance 247
Improving Performance 249
Rescoring Results 249
Finding Associated Words 250

Producing Shingles 251
Multifields 252
Searching for Shingles 253
Performance 255

**16. Partial Matching. . . . . 257**
Postcodes and Structured Data 258
prefix Query 259
wildcard and regexp Queries 260
Query-Time Search-as-You-Type 262
Index-Time Optimizations 264
Ngrams for Partial Matching 264
Index-Time Search-as-You-Type 265
Preparing the Index 265
Querying the Field 267
Edge n-grams and Postcodes 270
Ngrams for Compound Words 271

**17. Controlling Relevance. . . . . 275**
Theory Behind Relevance Scoring 275
Boolean Model 276
Term Frequency/Inverse Document Frequency (TF/IDF) 276
Vector Space Model 279
Lucene's Practical Scoring Function 282
Query Normalization Factor 283
Query Coordination 284
Index-Time Field-Level Boosting 286
Query-Time Boosting 286
Boosting an Index 287
t.getBoost() 288
Manipulating Relevance with Query Structure 288
Not Quite Not 289
boosting Query 290
Ignoring TF/IDF 291
constant_score Query 291
function_score Query 293
Boosting by Popularity 294
modifier 296
factor 298
boost_mode 299
max_boost 301
Boosting Filtered Subsets 301

filter Versus query 302
functions 303
score_mode 303
Random Scoring 303
The Closer, The Better 305
Understanding the price Clause 308
Scoring with Scripts 308
Pluggable Similarity Algorithms 310
Okapi BM25 310
Changing Similarities 313
Configuring BM25 314
Relevance Tuning Is the Last 10% 315

## Part III. Dealing with Human Language

**18. Getting Started with Languages. . . . . 319**
Using Language Analyzers 320
Configuring Language Analyzers 321
Pitfalls of Mixing Languages 323
At Index Time 323
At Query Time 324
Identifying Language 324
One Language per Document 325
Foreign Words 326
One Language per Field 327
Mixed-Language Fields 329
Split into Separate Fields 329
Analyze Multiple Times 329
Use n-grams 330

**19. Identifying Words. . . . . 333**
standard Analyzer 333
standard Tokenizer 334
Installing the ICU Plug-in 335
icu_tokenizer 335
Tidying Up Input Text 337
Tokenizing HTML 337
Tidying Up Punctuation 338

**20. Normalizing Tokens. . . . . 341**
In That Case 341

You Have an Accent 342
Retaining Meaning 343
Living in a Unicode World 346
Unicode Case Folding 347
Unicode Character Folding 349
Sorting and Collations 350
Case-Insensitive Sorting 351
Differences Between Languages 353
Unicode Collation Algorithm 353
Unicode Sorting 354
Specifying a Language 355
Customizing Collations 358

**21. Reducing Words to Their Root Form. . . . . 359**
Algorithmic Stemmers 360
Using an Algorithmic Stemmer 361
Dictionary Stemmers 363
Hunspell Stemmer 364
Installing a Dictionary 365
Per-Language Settings 365
Creating a Hunspell Token Filter 366
Hunspell Dictionary Format 367
Choosing a Stemmer 369
Stemmer Performance 370
Stemmer Quality 370
Stemmer Degree 370
Making a Choice 371
Controlling Stemming 371
Preventing Stemming 371
Customizing Stemming 372
Stemming in situ 373
Is Stemming in situ a Good Idea 375

**22. Stopwords: Performance Versus Precision. . . . . 377**
Pros and Cons of Stopwords 378
Using Stopwords 379
Stopwords and the Standard Analyzer 379
Maintaining Positions 380
Specifying Stopwords 380
Using the stop Token Filter 381
Updating Stopwords 383
Stopwords and Performance 383

and Operator 383
minimum_should_match 384
Divide and Conquer 385
Controlling Precision 386
Only High-Frequency Terms 387
More Control with Common Terms 388
Stopwords and Phrase Queries 388
Positions Data 389
Index Options 389
Stopwords 390
common_grams Token Filter 391
At Index Time 392
Unigram Queries 393
Bigram Phrase Queries 393
Two-Word Phrases 394
Stopwords and Relevance 394

**23. Synonyms. . . . . 395**
Using Synonyms 396
Formatting Synonyms 397
Expand or contract 398
Simple Expansion 398
Simple Contraction 399
Genre Expansion 400
Synonyms and The Analysis Chain 401
Case-Sensitive Synonyms 401
Multiword Synonyms and Phrase Queries 402
Use Simple Contraction for Phrase Queries 404
Synonyms and the query_string Query 405
Symbol Synonyms 405

**24. Typoes and Mispelings. . . . . 409**
Fuzziness 409
Fuzzy Query 410
Improving Performance 411
Fuzzy match Query 412
Scoring Fuzziness 413
Phonetic Matching 413

**Part IV. Aggregations**

**25. High-Level Concepts. . . . . 419**
Buckets 420
Metrics 420
Combining the Two 420

**26. Aggregation Test-Drive. . . . . 423**
Adding a Metric to the Mix 426
Buckets Inside Buckets 427
One Final Modification 429

**27. Building Bar Charts. . . . . 433**

**28. Looking at Time. . . . . 437**
Returning Empty Buckets 439
Extended Example 441
The Sky's the Limit 443

**29. Scoping Aggregations. . . . . 445**

**30. Filtering Queries and Aggregations. . . . . 449**
Filtered Query 449
Filter Bucket 450
Post Filter 451
Recap 452

**31. Sorting Multivalue Buckets. . . . . 453**
Intrinsic Sorts 453
Sorting by a Metric 454
Sorting Based on "Deep" Metrics 455

**32. Approximate Aggregations. . . . . 457**
Finding Distinct Counts 458
Understanding the Trade-offs 460
Optimizing for Speed 461
Calculating Percentiles 462
Percentile Metric 464
Percentile Ranks 467
Understanding the Trade-offs 469

**33. Significant Terms. . . . . 471**
significant_terms Demo 472
Recommending Based on Popularity 474

Recommending Based on Statistics 478

**34. Controlling Memory Use and Latency. . . . . 481**
Fielddata 481
Aggregations and Analysis 483
High-Cardinality Memory Implications 486
Limiting Memory Usage 487
Fielddata Size 488
Monitoring fielddata 489
Circuit Breaker 490
Fielddata Filtering 491
Doc Values 493
Enabling Doc Values 494
Preloading Fielddata 494
Eagerly Loading Fielddata 495
Global Ordinals 496
Index Warmers 498
Preventing Combinatorial Explosions 500
Depth-First Versus Breadth-First 502

**35. Closing Thoughts. . . . . 507**

## Part V. Geolocation

**36. Geo-Points. . . . . 511**
Lat/Lon Formats 511
Filtering by Geo-Point 512
geo_bounding_box Filter 513
Optimizing Bounding Boxes 514
geo_distance Filter 515
Faster Geo-Distance Calculations 516
geo_distance_range Filter 517
Caching geo-filters 517
Reducing Memory Usage 519
Sorting by Distance 520
Scoring by Distance 522

**37. Geohashes. . . . . 523**
Mapping Geohashes 524
geohash_cell Filter 525

**38. Geo-aggregations. . . . . 527**
geo_distance Aggregation 527
geohash_grid Aggregation 530
geo_bounds Aggregation 532

**39. Geo-shapes. . . . . 535**
Mapping geo-shapes 536
precision 536
distance_error_pct 537
Indexing geo-shapes 537
Querying geo-shapes 538
Querying with Indexed Shapes 540
Geo-shape Filters and Caching 541

## Part VI. Modeling Your Data

**40. Handling Relationships. . . . . 545**
Application-side Joins 546
Denormalizing Your Data 548
Field Collapsing 549
Denormalization and Concurrency 552
Renaming Files and Directories 555
Solving Concurrency Issues 555
Global Locking 556
Document Locking 557
Tree Locking 558

**41. Nested Objects. . . . . 561**
Nested Object Mapping 563
Querying a Nested Object 564
Sorting by Nested Fields 565
Nested Aggregations 567
reverse_nested Aggregation 568
When to Use Nested Objects 570

**42. Parent-Child Relationship. . . . . 571**
Parent-Child Mapping 572
Indexing Parents and Children 572
Finding Parents by Their Children 573
min_children and max_children 575
Finding Children by Their Parents 575

Children Aggregation 576
Grandparents and Grandchildren 577
Practical Considerations 579
Memory Use 579
Global Ordinals and Latency 580
Multigenerations and Concluding Thoughts 580

**43. Designing for Scale. . . . . 583**
The Unit of Scale 583
Shard Overallocation 585
Kagillion Shards 586
Capacity Planning 587
Replica Shards 588
Balancing Load with Replicas 589
Multiple Indices 590
Time-Based Data 592
Index per Time Frame 592
Index Templates 593
Retiring Data 594
Migrate Old Indices 595
Optimize Indices 595
Closing Old Indices 596
Archiving Old Indices 596
User-Based Data 597
Shared Index 597
Faking Index per User with Aliases 600
One Big User 601
Scale Is Not Infinite 602

## Part VII. Administration, Monitoring, and Deployment

**44. Monitoring. . . . . 607**
Marvel for Monitoring 607
Cluster Health 608
Drilling Deeper: Finding Problematic Indices 609
Blocking for Status Changes 611
Monitoring Individual Nodes 612
indices Section 613
OS and Process Sections 616
JVM Section 617
Threadpool Section 620

FS and Network Sections 622
Circuit Breaker 622
Cluster Stats 623
Index Stats 623
Pending Tasks 624
cat API 626

**45. Production Deployment. . . . . 631**
Hardware 631
Memory 631
CPUs 632
Disks 632
Network 633
General Considerations 633
Java Virtual Machine 634
Transport Client Versus Node Client 634
Configuration Management 635
Important Configuration Changes 635
Assign Names 636
Paths 636
Minimum Master Nodes 637
Recovery Settings 638
Prefer Unicast over Multicast 639
Don't Touch These Settings! 640
Garbage Collector 640
Threadpools 641
Heap: Sizing and Swapping 641
Give Half Your Memory to Lucene 642
Don't Cross 32 GB! 642
Swapping Is the Death of Performance 644
File Descriptors and MMap 645
Revisit This List Before Production 646

**46. Post-Deployment. . . . . 647**
Changing Settings Dynamically 647
Logging 648
Slowlog 648
Indexing Performance Tips 649
Test Performance Scientifically 650
Using and Sizing Bulk Requests 650
Storage 651
Segments and Merging 651

Other 653
Rolling Restarts 654
Backing Up Your Cluster 655
Creating the Repository 655
Snapshotting All Open Indices 656
Snapshotting Particular Indices 657
Listing Information About Snapshots 657
Deleting Snapshots 658
Monitoring Snapshot Progress 658
Canceling a Snapshot 661
Restoring from a Snapshot 661
Monitoring Restore Operations 662
Canceling a Restore 663
Clusters Are Living, Breathing Creatures 664

**Index** . . . . . **665**

# CHAPTER 1
# You Know, for Search…

Elasticsearch is an open-source search engine built on top of Apache Lucene™, a full-text search-engine library. Lucene is arguably the most advanced, high-performance, and fully featured search engine library in existence today—both open source and proprietary.

But Lucene is just a library. To leverage its power, you need to work in Java and to integrate Lucene directly with your application. Worse, you will likely require a degree in information retrieval to understand how it works. Lucene is *very* complex.

Elasticsearch is also written in Java and uses Lucene internally for all of its indexing and searching, but it aims to make full-text search easy by hiding the complexities of Lucene behind a simple, coherent, RESTful API.

However, Elasticsearch is much more than just Lucene and much more than "just" full-text search. It can also be described as follows:

- A distributed real-time document store where *every field* is indexed and searchable
- A distributed search engine with real-time analytics
- Capable of scaling to hundreds of servers and petabytes of structured and unstructured data

And it packages up all this functionality into a standalone server that your application can talk to via a simple RESTful API, using a web client from your favorite programming language, or even from the command line.

It is easy to get started with Elasticsearch. It ships with sensible defaults and hides complicated search theory away from beginners. It *just works*, right out of the box. With minimal understanding, you can soon become productive.

Elasticsearch can be downloaded, used, and modified free of charge. It is available under the Apache 2 license, one of the most flexible open source licenses available.

As your knowledge grows, you can leverage more of Elasticsearch's advanced features. The entire engine is configurable and flexible. Pick and choose from the advanced features to tailor Elasticsearch to your problem domain.

### The Mists of Time

Many years ago, a newly married unemployed developer called Shay Banon followed his wife to London, where she was studying to be a chef. While looking for gainful employment, he started playing with an early version of Lucene, with the intent of building his wife a recipe search engine.

Working directly with Lucene can be tricky, so Shay started work on an abstraction layer to make it easier for Java programmers to add search to their applications. He released this as his first open source project, called Compass.

Later Shay took a job working in a high-performance, distributed environment with in-memory data grids. The need for a high-performance, real-time, distributed search engine was obvious, and he decided to rewrite the Compass libraries as a standalone server called Elasticsearch.

The first public release came out in February 2010. Since then, Elasticsearch has become one of the most popular projects on GitHub with commits from over 300 contributors. A company has formed around Elasticsearch to provide commercial support and to develop new features, but Elasticsearch is, and forever will be, open source and available to all.

Shay's wife is still waiting for the recipe search…

## Installing Elasticsearch

The easiest way to understand what Elasticsearch can do for you is to play with it, so let's get started!

The only requirement for installing Elasticsearch is a recent version of Java. Preferably, you should install the latest version of the official Java from *www.java.com*.

You can download the latest version of Elasticsearch from *elasticsearch.org/download*.

```
curl -L -O http://download.elasticsearch.org/PATH/TO/VERSION.zip ❶
unzip elasticsearch-$VERSION.zip
cd  elasticsearch-$VERSION
```

❶ Fill in the URL for the latest version available on *elasticsearch.org/download*.

When installing Elasticsearch in production, you can use the method described previously, or the Debian or RPM packages provided on the downloads page. You can also use the officially supported Puppet module or Chef cookbook.

## Installing Marvel

Marvel is a management and monitoring tool for Elasticsearch, which is free for development use. It comes with an interactive console called Sense, which makes it easy to talk to Elasticsearch directly from your browser.

Many of the code examples in the online version of this book include a View in Sense link. When clicked, it will open up a working example of the code in the Sense console. You do not have to install Marvel, but it will make this book much more interactive by allowing you to experiment with the code samples on your local Elasticsearch cluster.

Marvel is available as a plug-in. To download and install it, run this command in the Elasticsearch directory:

```
./bin/plugin -i elasticsearch/marvel/latest
```

You probably don't want Marvel to monitor your local cluster, so you can disable data collection with this command:

```
echo 'marvel.agent.enabled: false' >> ./config/elasticsearch.yml
```

# Running Elasticsearch

Elasticsearch is now ready to run. You can start it up in the foreground with this:

```
./bin/elasticsearch
```

Add `-d` if you want to run it in the background as a daemon.

Test it out by opening another terminal window and running the following:

```
curl 'http://localhost:9200/?pretty'
```

You should see a response like this:

```
{
   "status": 200,
   "name": "Shrunken Bones",
   "version": {
      "number": "1.4.0",
      "lucene_version": "4.10"
   },
   "tagline": "You Know, for Search"
}
```

This means that your Elasticsearch *cluster* is up and running, and we can start experimenting with it.

A *node* is a running instance of Elasticsearch. A *cluster* is a group of nodes with the same `cluster.name` that are working together to share data and to provide failover and scale, although a single node can form a cluster all by itself.

You should change the default `cluster.name` to something appropriate to you, like your own name, to stop your nodes from trying to join another cluster on the same network with the same name!

You can do this by editing the `elasticsearch.yml` file in the `config/` directory and then restarting Elasticsearch. When Elasticsearch is running in the foreground, you can stop it by pressing Ctrl-C; otherwise, you can shut it down with the `shutdown` API:

```
curl -XPOST 'http://localhost:9200/_shutdown'
```

## Viewing Marvel and Sense

If you installed the Marvel management and monitoring tool, you can view it in a web browser by visiting *http://localhost:9200/_plugin/marvel/*.

You can reach the *Sense* developer console either by clicking the "Marvel dashboards" drop-down in Marvel, or by visiting *http://localhost:9200/_plugin/marvel/sense/*.

# Talking to Elasticsearch

How you talk to Elasticsearch depends on whether you are using Java.

## Java API

If you are using Java, Elasticsearch comes with two built-in clients that you can use in your code:

*Node client*
: The node client joins a local cluster as a *non data node*. In other words, it doesn't hold any data itself, but it knows what data lives on which node in the cluster, and can forward requests directly to the correct node.

*Transport client*
: The lighter-weight transport client can be used to send requests to a remote cluster. It doesn't join the cluster itself, but simply forwards requests to a node in the cluster.

Both Java clients talk to the cluster over port *9300*, using the native Elasticsearch *transport* protocol. The nodes in the cluster also communicate with each other over port 9300. If this port is not open, your nodes will not be able to form a cluster.

The Java client must be from the same version of Elasticsearch as the nodes; otherwise, they may not be able to understand each other.

More information about the Java clients can be found in the Java API section of the Guide.

## RESTful API with JSON over HTTP

All other languages can communicate with Elasticsearch over port *9200* using a RESTful API, accessible with your favorite web client. In fact, as you have seen, you can even talk to Elasticsearch from the command line by using the `curl` command.

Elasticsearch provides official clients for several languages—Groovy, JavaScript, .NET, PHP, Perl, Python, and Ruby—and there are numerous community-provided clients and integrations, all of which can be found in the Guide.

A request to Elasticsearch consists of the same parts as any HTTP request:

```
curl -X<VERB> '<PROTOCOL>://<HOST>/<PATH>?<QUERY_STRING>' -d '<BODY>'
```

The parts marked with `< >` above are:

`VERB`
: The appropriate HTTP *method* or *verb*: `GET`, `POST`, `PUT`, `HEAD`, or `DELETE`.

`PROTOCOL`
: Either `http` or `https` (if you have an `https` proxy in front of Elasticsearch.)

`HOST`
: The hostname of any node in your Elasticsearch cluster, or `localhost` for a node on your local machine.

`PORT`
: The port running the Elasticsearch HTTP service, which defaults to `9200`.

`QUERY_STRING`
: Any optional query-string parameters (for example `?pretty` will *pretty-print* the JSON response to make it easier to read.)

`BODY`
: A JSON-encoded request body (if the request needs one.)

For instance, to count the number of documents in the cluster, we could use this:

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

Elasticsearch returns an HTTP status code like `200 OK` and (except for `HEAD` requests) a JSON-encoded response body. The preceding `curl` request would respond with a JSON body like the following:

```
{
    "count" : 0,
    "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    }
}
```

We don't see the HTTP headers in the response because we didn't ask `curl` to display them. To see the headers, use the `curl` command with the `-i` switch:

```
curl -i -XGET 'localhost:9200/'
```

For the rest of the book, we will show these `curl` examples using a shorthand format that leaves out all the bits that are the same in every request, like the hostname and port, and the `curl` command itself. Instead of showing a full request like

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

we will show it in this shorthand format:

```
GET /_count
{
    "query": {
        "match_all": {}
    }
}
```

In fact, this is the same format that is used by the Sense console that we installed with Marvel. If in the online version of this book, you can open and run this code example in Sense by clicking the View in Sense link above.

# Document Oriented

Objects in an application are seldom just a simple list of keys and values. More often than not, they are complex data structures that may contain dates, geo locations, other objects, or arrays of values.

Sooner or later you're going to want to store these objects in a database. Trying to do this with the rows and columns of a relational database is the equivalent of trying to squeeze your rich, expressive objects into a very big spreadsheet: you have to flatten the object to fit the table schema—usually one field per column—and then have to reconstruct it every time you retrieve it.

Elasticsearch is *document oriented*, meaning that it stores entire objects or *documents*. It not only stores them, but also *indexes* the contents of each document in order to make them searchable. In Elasticsearch, you index, search, sort, and filter documents —not rows of columnar data. This is a fundamentally different way of thinking about data and is one of the reasons Elasticsearch can perform complex full-text search.

## JSON

Elasticsearch uses JavaScript Object Notation, or *JSON*, as the serialization format for documents. JSON serialization is supported by most programming languages, and has become the standard format used by the NoSQL movement. It is simple, concise, and easy to read.

Consider this JSON document, which represents a user object:

```
{
    "email":      "john@smith.com",
    "first_name": "John",
    "last_name":  "Smith",
    "info": {
        "bio":         "Eco-warrior and defender of the weak",
        "age":         25,
        "interests": [ "dolphins", "whales" ]
    },
    "join_date": "2014/05/01"
}
```

Although the original `user` object was complex, the structure and meaning of the object has been retained in the JSON version. Converting an object to JSON for indexing in Elasticsearch is much simpler than the equivalent process for a flat table structure.

Almost all languages have modules that will convert arbitrary data structures or objects into JSON for you, but the details are specific to each language. Look for modules that handle JSON *serialization* or *marshalling*. The official Elasticsearch clients all handle conversion to and from JSON for you automatically.

# Finding Your Feet

To give you a feel for what is possible in Elasticsearch and how easy it is to use, let's start by walking through a simple tutorial that covers basic concepts such as indexing, search, and aggregations.

We'll introduce some new terminology and basic concepts along the way, but it is OK if you don't understand everything immediately. We'll cover all the concepts introduced here in *much* greater depth throughout the rest of the book.

So, sit back and enjoy a whirlwind tour of what Elasticsearch is capable of.

## Let's Build an Employee Directory

We happen to work for *Megacorp*, and as part of HR's new *"We love our drones!"* initiative, we have been tasked with creating an employee directory. The directory is supposed to foster employer empathy and real-time, synergistic, dynamic collaboration, so it has a few business requirements:

- Enable data to contain multi value tags, numbers, and full text.
- Retrieve the full details of any employee.
- Allow structured search, such as finding employees over the age of 30.
- Allow simple full-text search and more-complex *phrase* searches.
- Return highlighted search *snippets* from the text in the matching documents.
- Enable management to build analytic dashboards over the data.

# Indexing Employee Documents

The first order of business is storing employee data. This will take the form of an *employee document*: a single document represents a single employee. The act of storing data in Elasticsearch is called *indexing*, but before we can index a document, we need to decide *where* to store it.

In Elasticsearch, a document belongs to a *type*, and those types live inside an *index*. You can draw some (rough) parallels to a traditional relational database:

```
Relational DB  ⇒ Databases ⇒ Tables ⇒ Rows      ⇒ Columns
Elasticsearch  ⇒ Indices   ⇒ Types  ⇒ Documents ⇒ Fields
```

An Elasticsearch cluster can contain multiple *indices* (databases), which in turn contain multiple *types* (tables). These types hold multiple *documents* (rows), and each document has multiple *fields* (columns).

### Index Versus Index Versus Index

You may already have noticed that the word *index* is overloaded with several meanings in the context of Elasticsearch. A little clarification is necessary:

*Index (noun)*
: As explained previously, an *index* is like a *database* in a traditional relational database. It is the place to store related documents. The plural of *index* is *indices* or *indexes*.

*Index (verb)*
: *To index a document* is to store a document in an *index (noun)* so that it can be retrieved and queried. It is much like the `INSERT` keyword in SQL except that, if the document already exists, the new document would replace the old.

*Inverted index*
: Relational databases add an *index*, such as a B-tree index, to specific columns in order to improve the speed of data retrieval. Elasticsearch and Lucene use a structure called an *inverted index* for exactly the same purpose.

  By default, every field in a document is *indexed* (has an inverted index) and thus is searchable. A field without an inverted index is not searchable. We discuss inverted indexes in more detail in "Inverted Index" on page 81.

So for our employee directory, we are going to do the following:

- Index a *document* per employee, which contains all the details of a single employee.
- Each document will be of *type* `employee`.
- That type will live in the `megacorp` *index*.
- That index will reside within our Elasticsearch cluster.

In practice, this is easy (even though it looks like a lot of steps). We can perform all of those actions in a single command:

```
PUT /megacorp/employee/1
{
    "first_name" : "John",
    "last_name" :  "Smith",
```

```
    "age" :        25,
    "about" :      "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

Notice that the path `/megacorp/employee/1` contains three pieces of information:

`megacorp`
: The index name

`employee`
: The type name

`1`
: The ID of this particular employee

The request body—the JSON document—contains all the information about this employee. His name is John Smith, he's 25, and enjoys rock climbing.

Simple! There was no need to perform any administrative tasks first, like creating an index or specifying the type of data that each field contains. We could just index a document directly. Elasticsearch ships with defaults for everything, so all the necessary administration tasks were taken care of in the background, using default values.

Before moving on, let's add a few more employees to the directory:

```
PUT /megacorp/employee/2
{
    "first_name" :  "Jane",
    "last_name" :   "Smith",
    "age" :         32,
    "about" :       "I like to collect rock albums",
    "interests":  [ "music" ]
}

PUT /megacorp/employee/3
{
    "first_name" :  "Douglas",
    "last_name" :   "Fir",
    "age" :         35,
    "about":        "I like to build cabinets",
    "interests":  [ "forestry" ]
}
```

# Retrieving a Document

Now that we have some data stored in Elasticsearch, we can get to work on the business requirements for this application. The first requirement is the ability to retrieve individual employee data.

This is easy in Elasticsearch. We simply execute an HTTP `GET` request and specify the *address* of the document—the index, type, and ID. Using those three pieces of information, we can return the original JSON document:

```
GET /megacorp/employee/1
```

And the response contains some metadata about the document, and John Smith's original JSON document as the `_source` field:

```
{
  "_index" :   "megacorp",
  "_type" :    "employee",
  "_id" :      "1",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "first_name" :  "John",
      "last_name" :   "Smith",
      "age" :         25,
      "about" :       "I love to go rock climbing",
      "interests":  [ "sports", "music" ]
  }
}
```

In the same way that we changed the HTTP verb from `PUT` to `GET` in order to retrieve the document, we could use the `DELETE` verb to delete the document, and the `HEAD` verb to check whether the document exists. To replace an existing document with an updated version, we just `PUT` it again.

# Search Lite

A `GET` is fairly simple—you get back the document that you ask for. Let's try something a little more advanced, like a simple search!

The first search we will try is the simplest search possible. We will search for all employees, with this request:

```
GET /megacorp/employee/_search
```

You can see that we're still using index `megacorp` and type `employee`, but instead of specifying a document ID, we now use the `_search` endpoint. The response includes all three of our documents in the `hits` array. By default, a search will return the top 10 results.

```
{
   "took":      6,
   "timed_out": false,
   "_shards": { ... },
   "hits": {
```

```
      "total":      3,
      "max_score":  1,
      "hits": [
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "3",
            "_score":         1,
            "_source": {
               "first_name":  "Douglas",
               "last_name":   "Fir",
               "age":         35,
               "about":       "I like to build cabinets",
               "interests": [ "forestry" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "1",
            "_score":         1,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "2",
            "_score":         1,
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

The response not only tells us which documents matched, but also includes the whole document itself: all the information that we need in order to display the search results to the user.

Next, let's try searching for employees who have "Smith" in their last name. To do this, we'll use a *lightweight* search method that is easy to use from the command line. This method is often referred to as a *query-string* search, since we pass the search as a URL query-string parameter:

```
GET /megacorp/employee/_search?q=last_name:Smith
```

We use the same `_search` endpoint in the path, and we add the query itself in the `q=` parameter. The results that come back show all Smiths:

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

# Search with Query DSL

Query-string search is handy for ad hoc searches from the command line, but it has its limitations (see "Search *Lite*" on page 76). Elasticsearch provides a rich, flexible, query language called the *query DSL*, which allows us to build much more complicated, robust queries.

The *domain-specific language* (DSL) is specified using a JSON request body. We can represent the previous search for all Smiths like so:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "last_name" : "Smith"
        }
    }
}
```

This will return the same results as the previous query. You can see that a number of things have changed. For one, we are no longer using *query-string* parameters, but instead a request body. This request body is built with JSON, and uses a `match` query (one of several types of queries, which we will learn about later).

## More-Complicated Searches

Let's make the search a little more complicated. We still want to find all employees with a last name of Smith, but we want only employees who are older than 30. Our query will change a little to accommodate a *filter*, which allows us to execute structured searches efficiently:

```
GET /megacorp/employee/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "range" : {
                    "age" : { "gt" : 30 } ❶
                }
            },
            "query" : {
                "match" : {
                    "last_name" : "smith" ❷
                }
            }
        }
    }
}
```

❶ This portion of the query is a `range` *filter*, which will find all ages older than 30—`gt` stands for *greater than*.

❷ This portion of the query is the same `match` *query* that we used before.

Don't worry about the syntax too much for now; we will cover it in great detail later. Just recognize that we've added a *filter* that performs a range search, and reused the same `match` query as before. Now our results show only one employee who happens to be 32 and is named Jane Smith:

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

# Full-Text Search

The searches so far have been simple: single names, filtered by age. Let's try a more advanced, full-text search—a task that traditional databases would really struggle with.

We are going to search for all employees who enjoy rock climbing:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "about" : "rock climbing"
        }
    }
}
```

You can see that we use the same `match` query as before to search the `about` field for "rock climbing." We get back two matching documents:

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.16273327,
      "hits": [
         {
            ...
            "_score":         0.16273327, ❶
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
```

```
            "age":          25,
            "about":        "I love to go rock climbing",
            "interests": [ "sports", "music" ]
         }
      },
      {
         ...
         "_score":          0.016878016, ❶
         "_source": {
            "first_name":   "Jane",
            "last_name":    "Smith",
            "age":          32,
            "about":        "I like to collect rock albums",
            "interests": [ "music" ]
         }
      }
   ]
  }
}
```

❶ The relevance scores

By default, Elasticsearch sorts matching results by their relevance score, that is, by how well each document matches the query. The first and highest-scoring result is obvious: John Smith's `about` field clearly says "rock climbing" in it.

But why did Jane Smith come back as a result? The reason her document was returned is because the word "rock" was mentioned in her `about` field. Because only "rock" was mentioned, and not "climbing," her `_score` is lower than John's.

This is a good example of how Elasticsearch can search *within* full-text fields and return the most relevant results first. This concept of *relevance* is important to Elasticsearch, and is a concept that is completely foreign to traditional relational databases, in which a record either matches or it doesn't.

# Phrase Search

Finding individual words in a field is all well and good, but sometimes you want to match exact sequences of words or *phrases*. For instance, we could perform a query that will match only employee records that contain both "rock" *and* "climbing" *and* that display the words are next to each other in the phrase "rock climbing."

To do this, we use a slight variation of the `match` query called the `match_phrase` query:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
```

```
            "about" : "rock climbing"
        }
    }
}
```

This, to no surprise, returns only John Smith's document:

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         }
      ]
   }
}
```

# Highlighting Our Searches

Many applications like to *highlight* snippets of text from each search result so the user can see *why* the document matched the query. Retrieving highlighted fragments is easy in Elasticsearch.

Let's rerun our previous query, but add a new `highlight` parameter:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    },
    "highlight": {
        "fields" : {
            "about" : {}
        }
    }
}
```

When we run this query, the same hit is returned as before, but now we get a new section in the response called `highlight`. This contains a snippet of text from the `about` field with the matching words wrapped in `<em></em>` HTML tags:

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            },
            "highlight": {
               "about": [
                  "I love to go <em>rock</em> <em>climbing</em>" ❶
               ]
            }
         }
      ]
   }
}
```

❶ The highlighted fragment from the original text

You can read more about the highlighting of search snippets in the highlighting reference documentation.

# Analytics

Finally, we come to our last business requirement: allow managers to run analytics over the employee directory. Elasticsearch has functionality called *aggregations*, which allow you to generate sophisticated analytics over your data. It is similar to `GROUP BY` in SQL, but much more powerful.

For example, let's find the most popular interests enjoyed by our employees:

```
GET /megacorp/employee/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
```

```
    }
}
```

Ignore the syntax for now and just look at the results:

```
{
   ...
   "hits": { ... },
   "aggregations": {
      "all_interests": {
         "buckets": [
            {
               "key":       "music",
               "doc_count": 2
            },
            {
               "key":       "forestry",
               "doc_count": 1
            },
            {
               "key":       "sports",
               "doc_count": 1
            }
         ]
      }
   }
}
```

We can see that two employees are interested in music, one in forestry, and one in sports. These aggregations are not precalculated; they are generated on the fly from the documents that match the current query. If we want to know the popular interests of people called Smith, we can just add the appropriate query into the mix:

```
GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
        "field": "interests"
      }
    }
  }
}
```

The `all_interests` aggregation has changed to include only documents matching our query:

```
...
"all_interests": {
   "buckets": [
      {
         "key": "music",
         "doc_count": 2
      },
      {
         "key": "sports",
         "doc_count": 1
      }
   ]
}
```

Aggregations allow hierarchical rollups too. For example, let's find the average age of employees who share a particular interest:

```
GET /megacorp/employee/_search
{
    "aggs" : {
        "all_interests" : {
            "terms" : { "field" : "interests" },
            "aggs" : {
                "avg_age" : {
                    "avg" : { "field" : "age" }
                }
            }
        }
    }
}
```

The aggregations that we get back are a bit more complicated, but still fairly easy to understand:

```
...
"all_interests": {
   "buckets": [
      {
         "key": "music",
         "doc_count": 2,
         "avg_age": {
            "value": 28.5
         }
      },
      {
         "key": "forestry",
         "doc_count": 1,
         "avg_age": {
            "value": 35
         }
      },
      {
         "key": "sports",
```

```
            "doc_count": 1,
            "avg_age": {
               "value": 25
            }
         }
      ]
   }
```

The output is basically an enriched version of the first aggregation we ran. We still have a list of interests and their counts, but now each interest has an additional `avg_age`, which shows the average age for all employees having that interest.

Even if you don't understand the syntax yet, you can easily see how complex aggregations and groupings can be accomplished using this feature. The sky is the limit as to what kind of data you can extract!

## Tutorial Conclusion

Hopefully, this little tutorial was a good demonstration about what is possible in Elasticsearch. It is really just scratching the surface, and many features—such as suggestions, geolocation, percolation, fuzzy and partial matching—were omitted to keep the tutorial short. But it did highlight just how easy it is to start building advanced search functionality. No configuration was needed—just add data and start searching!

It's likely that the syntax left you confused in places, and you may have questions about how to tweak and tune various aspects. That's fine! The rest of the book dives into each of these issues in detail, giving you a solid understanding of how Elasticsearch works.

## Distributed Nature

At the beginning of this chapter, we said that Elasticsearch can scale out to hundreds (or even thousands) of servers and handle petabytes of data. While our tutorial gave examples of how to use Elasticsearch, it didn't touch on the mechanics at all. Elasticsearch is distributed by nature, and it is designed to hide the complexity that comes with being distributed.

The distributed aspect of Elasticsearch is largely transparent. Nothing in the tutorial required you to know about distributed systems, sharding, cluster discovery, or dozens of other distributed concepts. It happily ran the tutorial on a single node living inside your laptop, but if you were to run the tutorial on a cluster containing 100 nodes, everything would work in exactly the same way.

Elasticsearch tries hard to hide the complexity of distributed systems. Here are some of the operations happening automatically under the hood:

- Partitioning your documents into different containers or *shards*, which can be stored on a single node or on multiple nodes
- Balancing these shards across the nodes in your cluster to spread the indexing and search load
- Duplicating each shard to provide redundant copies of your data, to prevent data loss in case of hardware failure
- Routing requests from any node in the cluster to the nodes that hold the data you're interested in
- Seamlessly integrating new nodes as your cluster grows or redistributing shards to recover from node loss

As you read through this book, you'll encounter supplemental chapters about the distributed nature of Elasticsearch. These chapters will teach you about how the cluster scales and deals with failover (Chapter 2), handles document storage (Chapter 4), executes distributed search (Chapter 9), and what a shard is and how it works (Chapter 11).

These chapters are not required reading—you can use Elasticsearch without understanding these internals—but they will provide insight that will make your knowledge of Elasticsearch more complete. Feel free to skim them and revisit at a later point when you need a more complete understanding.

## Next Steps

By now you should have a taste of what you can do with Elasticsearch, and how easy it is to get started. Elasticsearch tries hard to work out of the box with minimal knowledge and configuration. The best way to learn Elasticsearch is by jumping in: just start indexing and searching!

However, the more you know about Elasticsearch, the more productive you can become. The more you can tell Elasticsearch about the domain-specific elements of your application, the more you can fine-tune the output.

The rest of this book will help you move from novice to expert. Each chapter explains the essentials, but also includes expert-level tips. If you're just getting started, these tips are probably not immediately relevant to you; Elasticsearch has sensible defaults and will generally do the right thing without any interference. You can always revisit these chapters later, when you are looking to improve performance by shaving off any wasted milliseconds.