Elasticsearch Blueprints

Elasticsearch is a distributed search server similar to Apache Solr with a focus on large datasets, schemaless setup, and high availability. Utilizing the Apache Lucene library (also used in Apache Solr), Elasticsearch enables powerful full-text search, as well as autocomplete "morelikeithis" search, multilingual functionality, and an extensive search query DSL.

This book starts with the creation of a Google-like web search service, enabling you to generate your own search results. You will then learn how an e-commerce website can be built using Elasticsearch. We will discuss various approaches in getting relevant content up the results, such as relevancy based on how well a query matched the text, time-based recent documents, geographically nearer items, and other frequently used approaches.

Finally, the book will cover various geocapabilities of Elasticsearch to make your searches similar to real-world scenarios.

Who this book is written for

If you are a data enthusiast and would like to explore and specialize on search technologies based on Elasticsearch, this is the right book for you. A compelling case-to-case mapping of features and implementation of Elasticsearch to solve many real-world use cases makes this book the right choice to start and specialize on Elasticsearch.

What you will learn from this book

- Build a simple scalable server for effective searching in Elasticsearch
- Design a scalable e-commerce search solution to generate accurate search results using various filters such as filters based on date range and price range
- Improve the relevancy and scoring of your searches
- Manage real-world, complex data using various techniques, including parent-child search and searching questions based on the criteria of questions and answers
- Use the excellent data crunching and aggregation capability of Elasticsearch to analyze your data
- Generate real-time visualizations of your data using compelling visualization techniques, such as time graphs, pie charts, and stacked graphs
- Enhance the quality of your search and widen the scope of matches using various analyzer techniques, such as lower casing, stemming, and synonym matching
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Google-like Web Search'
- A synopsis of the book’s content
- More information on Elasticsearch Blueprints
Vineeth Mohan is an architect and developer. He currently works as the CTO at Factweavers Technologies and is also an Elasticsearch-certified trainer.

He loves to spend time studying emerging technologies and applications related to data analytics, data visualizations, machine learning, natural language processing, and developments in search analytics. He began coding during his high school days, which later ignited his interest in computer science, and he pursued engineering at Model Engineering College, Cochin. He was recruited by the search giant Yahoo! during his college days. After 2 years of work at Yahoo! on various big data projects, he joined a start-up that dealt with search and analytics. Finally, he started his own big data consulting company, Factweavers.

Under his leadership and technical expertise, Factweavers is one of the early adopters of Elasticsearch and has been engaged with projects related to end-to-end big data solutions and analytics for the last few years.

There, he got the opportunity to learn various big-data-based technologies, such as Hadoop, and high-performance data ingress systems and storage. Later, he moved to a start-up in his hometown, where he chose Elasticsearch as the primary search and analytic engine for the project assigned to him.

Later in 2014, he founded Factweavers Technologies along with Jalaluddeen; it is consultancy that aims at providing Elasticsearch-based solutions. He is also an Elasticsearch-certified corporate trainer who conducts trainings in India. Till date, he has worked on numerous projects that are mostly based on Elasticsearch and has trained numerous multinationals on Elasticsearch.
Preface

Elasticsearch is a distributed search server similar to Apache Solr with a focus on large datasets, schemaless setup, and high availability. Utilizing the Apache Lucene library (also used in Apache Solr), Elasticsearch enables powerful full-text searches, autocomplete, the "morelikethis" search, multilingual functionality, as well as an extensive search query DSL.

Elasticsearch's schemafree architecture provides developers with built-in flexibility as well as ease of setup. This architecture allows Elasticsearch to index and search unstructured content, making it perfectly suited for both small projects and large big data warehouses—even with petabytes of unstructured data.

This book will enable you to utilize the amazing features of Elasticsearch and build projects to simplify operations on even large datasets. This book starts with the creation of a Google-like web search service, enabling you to generate your own search results. You will then learn how an e-commerce website can be built using Elasticsearch, which will help users search and narrow down the set of products they are interested in. You will explore the most important part of a search—relevancy—based on the various parameters, such as relevance, document collection relevance, user usage pattern, geographic nearness, and document relevance to select the top results.

Next, you will discover how Elasticsearch manages relational content for even complex real-world data. You will then learn the capabilities of Elasticsearch as a strong analytic search platform, which coupled with some visualization techniques can produce real-time data visualization. You will also discover how to improve your search quality and widen the scope of matches using various analyzer techniques. Finally, this book will cover the various geo capabilities of Elasticsearch to make your searches similar to real-world scenarios.
What this book covers

Chapter 1, Google-like Web Search, takes you along the course of building a simple scalable search server. You will learn how to create an index and add some documents to it and you will try out some essential features, such as highlighting and pagination of results. Also, it will cover topics such as setting an analyzer for our text; applying filters to eliminate unwanted characters, such as HTML tags; and so on.

Chapter 2, Building Your Own E-Commerce Solution, covers how to design a scalable e-commerce search solution to generate accurate search results using various filters, such as date-range based and prize-range based filters.

Chapter 3, Relevancy and Scoring, unleashes the power and flexibility of Elasticsearch that will help you implement your own scoring logic.

Chapter 4, Managing Relational Content, covers how to use the document linking or relational features of Elasticsearch.

Chapter 5, Analytics Using Elasticsearch, covers the capability and usage of Elasticsearch in the analytics area with a few use case scenarios.

Chapter 6, Improving the Search Experience, helps you learn how to improve the search quality of a text search. This includes the description of various analyzers and a detailed description of how to mix and match them.

Chapter 7, Spicing Up a Search Using Geo, explores how to use geo information to get the best out of search and scoring.

Chapter 8, Handling Time-based Data, explains the difficulties we face when we use normal indexing in Elasticsearch.
Google-like Web Search

Text search problems are one of the key and common use cases for web-based applications. Developers over the world have been keen to bring an open source solution to this problem. Hence, the Lucene revolution happened. **Lucene** is the heart of most of the search engines that you see today. It basically accepts the text that is to be searched, stores it in an easy searchable form or data structure (inverted index), and then accepts various types of search queries and returns a set of matching results. After the first search revolution, came the second one. Many server-based search solutions, such as Apache SOLR, were built on top of Lucene and marked the second phase of the search revolution. Here, a powerful wrapper was made to interface web users that wanted to index and search text of Lucene. Many powerful tools, notably SOLR, were developed at this stage of revolution. Some of these search frameworks were able to provide document database features too. Then, the next phase of the search revolution came, which is still on-going. The design goal of this phase is provide scaling solutions for the existing stack. **Elasticsearch** is a search and analytic engine that provides a powerful wrapper to Lucene along with an inbuilt document database and provisions various scaling solutions. The document database is also implemented using Lucene. Though competitors of Elasticsearch have some more advanced feature sets, those tools lack the simplicity and the wide range of scalability solutions Elasticsearch offers. Hence, we can see that Elasticsearch is the farthest point to which the search revolution has reached and is the future of text search.

This chapter takes you along the course to build a simple scalable search server. We will see how to create an index and add some documents to it and try out some essential features such as highlighting and pagination of results. Also, we will cover topics such as how to set an analyzer for our text and how to apply filters to eliminate unwanted characters such as HTML tags, and so on.
Here are the important topics that we will cover in this chapter:

- Deploying Elasticsearch
- Concept of the head UI shards and replicas
- Index - type mapping
- Analyzers, filters, and tokenizers
- The head UI

Let's start and explore Elasticsearch in detail.

**Deploying Elasticsearch**

First, let's download and install the following tools:

- **cURL**: cURL is an open source command-line tool available in both Windows and Unix. It is widely used to communicate with web interfaces. Since all communication to Elasticsearch can be done through standard REST protocols, we will use cURL throughout the book to communicate with Elasticsearch. The official website of cURL is [http://curl.haxx.se/download.html](http://curl.haxx.se/download.html).

- **Elasticsearch**: You need to install Elasticsearch from its official site [http://www.elasticsearch.org/](http://www.elasticsearch.org/). When this book was written, the latest version of Elasticsearch available was 1.0.0, so I would recommend that you use the same version. The only dependency of Elasticsearch is Java 1.6 or its higher versions. Once you make sure that you have Java installed, download the Elasticsearch ZIP file.

First, let's download Elasticsearch:

1. Unzip and place the files in a folder.
2. Next, let's install the Elasticsearch-head plugin. Head is the standard web frontend of the Elasticsearch server. Most of the Elasticsearch operations can be done via a head plugin. To install head, run the following command from the folder where Elasticsearch is installed:

   ```bash
   bin/plugin -install mobz/elasticsearch-head # (Linux users)
   bin\plugin -install mobz/elasticsearch-head # (Windows users)
   ```

3. You should see a new folder in the `plugins` directory. Open a console and type the following to start Elasticsearch:

   ```bash
   bin/elasticsearch   #(Linux users)
   bin\elasticsearch.bat # (Windows users)
   ```
4. The `-d` command is used to run Elasticsearch in the background rather than the foreground. By running the application in the foreground, we can track the changes taking place in it through the logs spitted in the console. The default behavior is to run in the foreground.

One of the basic design goals of Elasticsearch is its high configurability clubbed with its optimal default configurations that get you started seamlessly. So, all you have to do is start Elasticsearch. You don't have to learn any complex configuration concepts at least to get started. So our search server is up and running now.

To see the frontend of your Elasticsearch server, you can visit http://localhost:9200/_plugin/head/.

**Communicating with the Elasticsearch server**

cURL will be our tool of choice that we will use to communicate with Elasticsearch. Elasticsearch follows a REST-like protocol for its exposed web API. Some of its features are as follows:

- **PUT**: The HTTP method **PUT** is used to send configurations to Elasticsearch.
- **POST**: The HTTP method **POST** is used to create new documents or to perform a search operation. While successful indexing of documents is done using **POST**, Elasticsearch provides you with a unique ID that points to the index file.
- **GET**: The HTTP method **GET** is used to retrieve an already indexed document. Each document has a unique ID called a **doc ID** (short form for document's ID). When we index a document using **POST**, it provides a document ID, which can be used to retrieve the original document.
- **DELETE**: The HTTP method **DELETE** is used to delete documents from the Elasticsearch index. Deletion can be performed based on a search query or directly using the document ID.

To specify the HTTP method in cURL, you can use the `-X` option, for example, `curl -X POST http://localhost/`. JSON is the data format used to communicate with Elasticsearch. To specify the data in cURL, we can specify it in the following forms:

- **A command line**: You can use the `-d` option to specify the JSON to be sent in the command line itself, for example:
  ```bash
  -d '{ "time": "12-10-2010"}"
  ```
• **A file:** If the JSON is too long or inconvenient to be mentioned in a command line, you can specify it in a file or ask cURL to pick the JSON up from the file. You need to use the same `-d` option with a `@` symbol just before the filename, for example:

    curl -X POST 'http://localhost:9200/news/public/' -d @file

---

**Shards and replicas**

The concept of **sharding** is introduced in Elasticsearch to provide horizontal scaling. Scaling, as you know, is to increase the capacity of the search engine, both the index size and the query rate (query per second) capacity. Let's say an application can store up to 1,000 feeds and gives reasonable performance. Now, we need to increase the performance of this application to 2,000 feeds. This is where we look for scaling solutions. There are two types of scaling solutions:

- **Vertical scaling:** Here, we add hardware resources, such as more main memory, more CPU cores, or RAID disks to increase the capacity of the application.

- **Horizontal scaling:** Here, we add more machines to the system. As in our example, we bring in one more machines and give both the machines 1,000 feeds each. The result is computed by merging the results from both the machines. As both the processes take place in parallel, they won't eat up more time or bandwidth.

Guess what! Elasticsearch can be scaled both horizontally and vertically. You can increase its main memory to increase its performance and you can simply add a new machine to increase its capacity. Horizontal scaling is implemented using the concept of sharding in Elasticsearch. Since Elasticsearch is a distributed system, we need to address our data safety/availability concerns. Using replicas we achieve this. When one replica (size 1) is defined for a cluster with more than one machine, two copies of the entire feed become available in the distributed system. This means that even if a single machine goes down, we won't lose data and at the same time. The load would be distributed somewhere else. One important point to mention here is that the default number of shards and replicas are generally sufficient and also, we have the provision to change the replica number later on.

This is how we create an index and pass the number of shards and replicas:

    curl -X PUT "localhost:9200/news" -d '{
    "settings": {
    "index": {
    "number_of_shards": 2,
    }'
Chapter 1

```
"number_of_replicas": 1
}
}
'}

A few things to be noted here are:

- Adding more primary shards will increase the write throughout the index
- Adding more replicas will increase the durability of the index and the read throughout, at the cost of disk space

**Index-type mapping**

An index is a grouping logic where feeds of the same type are encapsulated together. A type is a sub grouping logic under index. To create a type under index, you need to decide on a type name. As in our case, we take the index name as news and the type name as public. We created the index in the previous step and now we need to define the data types of the fields that our data hold in the type mapping section.

Check out the sample given next. Here, the date data type takes the time format to be yyyy/MM/dd HH:mm:ss by default:

```
curl -X PUT "localhost:9200/news/public/_mapping" -d '{
"public" : {
 "properties" : {
 "Title" : { "type" : "string" },
 "Content": { "type" : "string" },
 "DOP": { "type" : "date" }
 }
 }
'}
```

Once we apply mapping, certain aspects of it such as new field definitions can be updated. However, we can't update certain other aspects such as changing the type of a field or changing the assigned analyzer. So, we now know how to create an index and add necessary mappings to the index we created. There is another important thing that you must take care of while indexing your data, that is, the analysis of our data. I guess you already know the importance of analysis. In simple terms, analysis is the breaking down of text into an elementary form called tokens. This tokenization is a must and has to be given serious consideration. Elasticsearch has many built-in analyzers that do this job for you. At the same time, you are free to deploy your own custom analyzers as well if the built-in analyzers do not serve your purpose. Let's see analysis in detail and how we can define analyzers for fields.
Setting the analyzer

Analyzers constitute an important part of indexing. To understand what analyzers do, let's consider three documents:

- **Document1 (tokens):** { This, is, easy }
- **Document2 (tokens):** { This, is, fast }
- **Document3 (tokens):** { This, is, easy, and, fast }

Here, terms such as *This, is,* as well as *and* are not relevant keywords. The chances of someone wanting to search for such words are very less, as these words don't contribute to the facts or context of the document. Hence, it's safe to avoid these words while indexing or rather you should avoid making these words searchable.

So, the tokenization would be as follows:

- **Document1 (tokens):** { easy }
- **Document2 (tokens):** { fast }
- **Document3 (tokens):** { easy, fast }

Words such as *the, or,* as well as *and* are referred to as stop words. In most cases, these are for grammatical support and the chances that someone will search based on these words are slim. Also, the analysis and removal of stop words is very much language dependent. The process of selecting/transforming the searchable tokens from a document while indexing is called **analyzing.** The module that facilitates this is called an analyzer. The analyzer we just discussed is a stop word analyzer. By applying the right analyzer, you can minimize the number of searchable tokens and hence get better performance results.

There are three stages through which you can perform an analysis:

- **Character filters:** Filtering is done at character level before processing for tokens. A typical example of this is an HTML character filter. We might give an HTML to be indexed to Elasticsearch. In such instances, we can provide the HTML CHAR filter to do the work.
- **Tokenizers:** The logic to break down text into tokens is depicted in this state. A typical example of this is whitespace tokenizers. Here, text is broken down into tokens by splitting the text based on the white space occurrence.
- **Token filters:** On top of the previous process, we apply a token filter. In this stage, we filter tokens to match our requirement. The length token filter is a typical token filter. A token filter of type **length** removes words which are too long or too short for the stream.
Here is a flowchart that depicts this process:

![Flowchart]

It should be noted that any number of such components can be incorporated in each stage. A combination of these components is called an analyzer. To create an analyzer out of the existing components, all we need to do is add the configuration to our Elasticsearch configuration file.
Types of character filters
The following are the different types of character filters:

- **HTML stripper**: This strips the HTML tags out of the text.
- **Mapping char filter**: Here, you can ask Elasticsearch to convert a set of characters or strings to another set of characters or strings. The options are as follows:

  "mappings" : ["ph=>f", "qu=>q"]

Types of tokenizers
The following are different types of tokenizers:

- **The whitespace tokenizer**: A tokenizer of this type whitespace divides text at whitespace.
- **The shingle tokenizer**: There are instances where you want to search for text with two consecutive words, such as Latin America. In conventional searches, Latin would be a token and America would be a token, so you won't be able to boil down to the text that has these words next to each other. In the shingle tokenizer, \( n \) number of tokens are grouped into a single token. Token generation for a 2Gram tokenizer would be as follows:

  "Latin America is a great place to go in summer"
  => { "Latin America" ,"America is" , "is a" , "a great" ,
      "great place" , "place to" , "to go" , "go in" ,
      "in summer" }

- **The lowercase tokenizer**: This converts text into lowercase, thereby decreasing the index size.

Types of token filters
The following are the different types of token filters:

- **The stop word token filter**: A set of words are recognized as stop words. This includes words like "is", "the", as well as "and" that don't add facts to the statement, but support the statement grammatically. A stop word token filter removes the stop words and hence helps to conduct more meaningful and efficient searches.
The length token filter: With this, we can filter out tokens that have length greater than a configured value.

The stemmer token filter: Stemming is an interesting concept. There are words such as "learn", "learning", "learnt", and so on that refer to the same word, but then are in different tenses. Here, we only need to index the actual word "learn" for any of its tenses. This is what a stemmer token filter does. It translates different tenses of the same word to the actual word.

Creating your own analyzer

Now, let's create our own analyzer and apply it on an index. I want to make an analyzer that strips out HTML tags before indexing. Also, there should not be any differentiation between lowercase and uppercase while searching. In short, the search is case insensitive. We are not interested in searching words such as "is" and "the", which are stop words. Also, we are not interested in words that have more than 900 characters. The following are the settings that you need to paste in the config/Elasticsearch.yml file to create this analyzer:

```yaml
index:
  analysis:
    analyzer:
      myCustomAnalyzer:
        tokenizer: smallLetter
        filter: [lowercase, stopWord]
        char_filter: [html_strip]
        tokenizer:
          smallLetter:
            type: standard
            max_token_length: 900
        filter:
          stopWord:
            type: stop
          stopwords: ["are", "the", "is"]
```

Here, I named my analyzer myCustomAnalyzer. By adding the character filter html_strip, all HTML tags are removed out of the stream. A filter called stopWord is created, where we define the stop words. If we don't mention the stop words, those are taken from the default set. The smallLetter tokenizer removes all the words that have more than 900 characters.
Readymade analyzers

A combination of character filters, token filters, and tokenizers is called an analyzer. You can make your own analyzer using these building blocks, but then, there are readymade analyzers that work well in most of the use cases. A Snowball Analyzer is an analyzer of the type snowball that uses the standard tokenizer with the standard filter, lowercase filter, stop filter, and snowball filter, which is a stemming filter.

Here is how you can pass the analyzer setting to Elasticsearch:

```bash
curl -X PUT "http://localhost:9200/wiki" -d '{
  "index": {
    "number_of_shards": 4,
    "number_of_replicas": 1,
    "analysis": {
      "analyzer": {
        "content": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": ["lowercase", "stop", "kstem"],
          "char_filter": ["html_strip"]
        }
      }
    }
  }
}
'
```

Having understood how we can create an index and define field mapping with the analyzers, we shall go ahead and index some Wikipedia documents. For quick demonstration purpose, I have created a simple Python script to make some JSON files. I am trying to create corresponding JSON files for the wiki pages for the following countries:

- China
- India
- Japan
- The United States
- France
Here is the script written in Python if you want to use it. This takes as input two command-line arguments: the first one is the title of the page and the second is the link:

```python
import urllib2
import json
import sys

link = sys.argv[2]
htmlObj = { "link" : link ,
            "Author" : "anonymous" ,
            "timestamp" : "09-02-2014 14:16:00",
            "Title" : sys.argv[1]
}
response = urllib2.urlopen(link)
htmlObj['html'] = response.read()
print json.dumps(htmlObj , indent=4)
```

Let's assume the name of the Python file is `json_generator.py`. The following is how we execute it:

```
```

Now, we have a JSON file called `France.json` that has a sample data we are looking for.

I assume that you generated JSON files for each country that we mentioned. As seen earlier, indexing a document once it is created is simple. Using the script shown next, I created the index and defined the mappings:

```bash
curl -X PUT "http://localhost:9200/wiki" -d '{
  "index" : {
    "number_of_shards" : 4,
    "number_of_replicas" : 1 ,
    "analysis":{
      "analyzer":{
        "content" : {
          "type" : "custom",
          "tokenizer" : "standard",
          "filter" : ["lowercase", "stop", "kstem"],
          "char_filter" : ["html_strip"]
```
Once this is done, documents can be indexed like this. I assume that you have the file India.json. You can index it as:

curl -X POST 'http://localhost:9200/wiki/articles/' -d @India.json

Index all the documents likewise.

**Using phrase query to search**

We added some documents to the index that we created. Now, let’s examine some ways to query our data. Elasticsearch provides many types of queries to query our indexed documents. Of all the ones available, the simple query string query is a great place to start. The main advantage of this query is that it will never throw an exception. Also, a simple query string query discards the invalid parts of the query.

It mostly covers what is expected from most of the search engines. It takes OR of all the terms present in the query text, though we can change this behavior to AND. Also, it recognizes all Boolean keywords in the query text and performs the search accordingly. For details, you can look through [http://lucene.apache.org/core/2_9_4/queryparsersyntax.html](http://lucene.apache.org/core/2_9_4/queryparsersyntax.html).
To query an Elasticsearch index, we must create a JSON query. A simple JSON query is shown here:

```json
{
    "query": {
        "simple_query_string": {
            "query": "sms",
            "fields": [ "_all" ]
        }
    }
}
```

The screenshot of how a query is passed and the response is received in the head UI is shown as follows:

The explanation of the field's result is as follows:

- **took**: This is the time taken by Elasticsearch in milliseconds to perform the search on the index.
- **hits**: This array contains the records of the first 10 documents that matched.
- **_id**: This is a unique ID that refers to that document.
**Using the highlighting feature**

When we searched for a record, what we got was its actual data or `_source`. However, this information is not what we actually need in search results. Instead, we want to extract the text out of the content, which helps the users to better understand the context where the text was matched in the document. For example, say the user searched for the word `cochin`, he would like to check whether the document speaks about the city Cochin or the cochin bank in Japan. Seeing other words around the word `cochin` will further help the user to judge whether that is the document he/she is searching for. Elasticsearch provides you with fragments of text on request for the highlighted text. Each fragment has the matched text and some words around it. As there can be any number of matched queries in the same document, you would be provided an array of fragments per document, where each fragment would contain the context of the matched query.

Here is how we ask Elasticsearch to provide the highlighted text:

```json
{
    "query" : {...},
    "highlight" : {
        "fields" : {
            "Content" : {}
        }
    }
}
```

Under fields, you need to specify which all fields' highlighted text is required by you. In this example, we require the `Content` field.

Now, let's see another awesome feature that Elasticsearch offers. You would have noticed in Google search that the matched text in the highlighted fragments is shown in bold. Elasticsearch provides support for this as follows:

```json
{
    "query" : {...},
    "highlight" : {
        "pre_tags" : ["<b>"],
        "post_tags" : ["</b>"]
    }
}
```
Pagination

While searching, users can't view all the results at once. They like to see one batch at a time. Usually, a single batch contains 10 matched documents, as in Google search results, where each page contains 10 search results. This also gives us an advantage over the search engine as it need not send all the results back at once. The following is how we use pagination in Elasticsearch. Let's say that we are interested in seeing only five results at a time, then to get the first page, we have to use the following parameters:

- `size` = 5 (defaults to 10).
- `from` = 0, 5, 10, 15, 20 (defaults to 0). This depends on the page number you need.

Also, it should be noted that the total number of pages can be calculated from `count/_size`. Sample query for the page 5 of the search result where we show 5 results at a time:

```json
{
  "from": 4,
  "size": 5,
  "query": {...}
}
```

This is how the complete query looks, which enables pagination and highlighting:

```json
{
  "from": 0,
  "size": 10,
  "query": {
    "simple_query_string": {
      "query": "china",
      "fields": [
        "_all"
      ]
    }
  }
}
```
The head UI explained

When you open the head page, you see a UI that lists all the indexes and all the information related to it. Also, by looking at the tabs to the left, you know how well your cluster is doing, as shown in the following figure:

Now, take the Browser tab in the head UI. You will see all the feeds you index here. Note that it shows only the first 10 indexed feeds.
Now, on selecting one of your feeds, a nice model window appear, showing you the following view:

In this chapter, we looked at how we can deploy Elasticsearch. We had a quick look at of how to set an analyzer and index some documents. Then, we attempted to search for a document we indexed. We will look at how pagination and highlighting work in later sections of this book.
Summary
Kick starting Elasticsearch is much easier than any other open source projects. It ships with the best possible configurations, which make the process of starting this easy, and it ships with the most optimistic settings for performance. Hence, the initial learning curve on the user side is reduced. We went through a getting started that was easy; and discussed some of the architectural choices, which make this application truly distributed.

Though Elasticsearch head is a good tool to interact with Elasticsearch. There are other choices, such as Sense (packed with Elasticsearch Marvel), KOPF, and so on, which can also be used for the same purpose. There is a wide variety of ways in which we can use analyzers to improve a user's search experience. A separate chapter is dedicated to this in this book.

In the next chapter, you will learn how you can effectively use Elasticsearch to build an e-commerce application. Elasticsearch is a natural fit to build an e-commerce application. Search over structured and unstructured data, pagination, scoring, aggregation, filtering, and highlighting makes Elasticsearch an ideal backend for e-commerce-related applications.
Where to buy this book
You can buy Elasticsearch Blueprints from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.