Elasticsearch Server
Third Edition

Elasticsearch is a very fast and scalable open source search engine, designed with distribution and cloud in mind, complete with all the goodies that Apache Lucene has to offer. Elasticsearch’s schema-free architecture allows developers to index and search unstructured content, making it perfectly suited for both small projects and big data warehouses, even those with petabytes of unstructured data.

This book will guide you through the world of the most commonly used Elasticsearch server functionalities. You’ll start off by getting an understanding of the basics of Elasticsearch and its data indexing functionality. Next, you will see the querying capabilities of Elasticsearch, followed by a thorough explanation of scoring and search relevance. After this, you will explore the aggregation and data analysis capabilities of Elasticsearch and will learn how cluster administration and scaling can be used to boost your application’s performance. You’ll find out how to use the friendly REST APIs and how to tune Elasticsearch to make the most of it. By the end of this book, you will be able to create amazing search solutions in line with your project’s specifications.

Who this book is written for
If you are a competent developer and want to learn about the great and exciting world of Elasticsearch, then this book is for you. No prior knowledge of Java or Apache Lucene is needed.

What you will learn from this book
- Configure, create, and retrieve data from your indices
- Use an Elasticsearch query DSL to create a wide range of queries
- Discover the highlighting and geographical search features offered by Elasticsearch
- Find out how to index data that is not flat or data that has a relationship
- Exploit a prospective search to search for queries rather than documents
- Use the aggregations framework to get more from your data and improve your client’s search experience
- Monitor your cluster state and health using the Elasticsearch API as well as third-party monitoring solutions

In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'Getting Started with Elasticsearch Cluster'
- A synopsis of the book’s content
- More information on *Elasticsearch Server Third Edition*
Rafał Kuć is a software engineer, trainer, speaker and consultant. He is working as a consultant and software engineer at Sematext Group Inc. where he concentrates on open source technologies such as Apache Lucene, Solr, and Elasticsearch. He has more than 14 years of experience in various software domains—from banking software to e-commerce products. He is mainly focused on Java; however, he is open to every tool and programming language that might help him to achieve his goals easily and quickly. Rafał is also one of the founders of the solr.pl site, where he tries to share his knowledge and help people solve their Solr and Lucene problems. He is also a speaker at various conferences around the world such as Lucene Eurocon, Berlin Buzzwords, ApacheCon, Lucene/Solr Revolution, Velocity, and DevOps Days.

Rafal began his journey with Lucene in 2002; however, 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 that was it. He started working with Elasticsearch in the middle of 2010. At present, Lucene, Solr, Elasticsearch, and information retrieval are his main areas of interest.

Marek Rogoziński is a software architect and consultant with more than 10 years of experience. His specialization concerns solutions based on open source search engines, such as Solr and Elasticsearch, and the software stack for big data analytics including Hadoop, Hbase, and Twitter Storm.

He is also a cofounder of the solr.pl site, which publishes information and tutorials about Solr and Lucene libraries. He is the coauthor of ElasticSearch Server and its second edition, and the first and second editions of Mastering ElasticSearch, all published by Packt Publishing.

He is currently the chief technology officer and lead architect at ZenCard, a company that processes and analyzes large quantities of payment transactions in real time, allowing automatic and anonymous identification of retail customers on all retailer channels (m-commerce/e-commerce/brick&m mortar) and giving retailers a customer retention and loyalty tool.
Welcome to Elasticsearch Server, Third Edition. This is the third instalment of the book dedicated to yet another major release of Elasticsearch—this time version 2.2. In the third edition, we have decided to go on a similar route that we took when we wrote the second edition of the book. We not only updated the content to match the new version of Elasticsearch, but also restructured the book by removing and adding new sections and chapters. We read the suggestions we got from you—the readers of the book, and we carefully tried to incorporate the suggestions and comments received since the release of the first and second editions.

While reading this book, you will be taken on a journey to the wonderful world of full-text search provided by the Elasticsearch server. We will start with a general introduction to Elasticsearch, which covers how to start and run Elasticsearch, its basic concepts, and how to index and search your data in the most basic way. This book will also discuss the query language, so called Query DSL, that allows you to create complicated queries and filter returned results. In addition to all of this, you'll see how you can use the aggregation framework to calculate aggregated data based on the results returned by your queries. We will implement the autocomplete functionality together and learn how to use Elasticsearch spatial capabilities and prospective search.

Finally, this book will show you Elasticsearch's administration API capabilities with features such as shard placement control, cluster handling, and more, ending with a dedicated chapter that will discuss Elasticsearch's preparation for small and large deployments—both ones that concentrate on indexing and also ones that concentrate on indexing.
Preface

**What this book covers**

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

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

*Chapter 3, Searching Your Data*, introduces the full-text search capabilities of Elasticsearch by discussing how to query it, how the querying process works, and what types of basic and compound queries are available. In addition to this, we will show how to use position-aware queries in Elasticsearch.

*Chapter 4, Extending Your Query Knowledge*, shows how to efficiently narrow down your search results by using filters, how highlighting works, how to sort your results, and how query rewrite works.

*Chapter 5, Extending Your Index Structure*, shows how to index more complex data structures. We learn how to index tree-like data types, how to index data with relationships between documents, and how to modify index structure.

*Chapter 6, Make Your Search Better*, covers Apache Lucene scoring and how to influence it in Elasticsearch, the scripting capabilities of Elasticsearch, and its language analysis capabilities.

*Chapter 7, Aggregations for Data Analysis*, introduces you to the great world of data analysis by showing you how to use the Elasticsearch aggregation framework. We will discuss all types of aggregations—metrics, buckets, and the new pipeline aggregations that have been introduced in Elasticsearch.

*Chapter 8, Beyond Full-text Searching*, discusses non full-text search-related functionalities such as percolator—reversed search, and the geo-spatial capabilities of Elasticsearch. This chapter also discusses suggesters, which allow us to build a spellchecking functionality and an efficient autocomplete mechanism, and we will show how to handle deep-paging efficiently.

*Chapter 9, Elasticsearch Cluster in Detail*, discusses nodes discovery mechanism, recovery and gateway Elasticsearch modules, templates, caches, and settings update API.

*Chapter 10, Administrating Your Cluster*, covers the Elasticsearch backup functionality, rebalancing, and shards moving. In addition to this, you will learn how to use the warm up functionality, use the Cat API, and work with aliases.
Chapter 11, Scaling by Example, is dedicated to scaling and tuning. We will start with hardware preparations and considerations and a single Elasticsearch node-related tuning. We will go through cluster setup and vertical scaling, ending the chapter with high querying and indexing use cases and cluster monitoring.
Welcome to the wonderful world of Elasticsearch—a great full text search and analytics engine. It doesn't matter if you are new to Elasticsearch and full text searches in general, or if you already have some experience in this. We hope that, by reading this book, you'll be able to learn and extend your knowledge of Elasticsearch. As this book is also dedicated to beginners, we decided to start with a short introduction to full text searches in general, and after that, a brief overview of Elasticsearch.

Please remember that Elasticsearch is a rapidly changing of software. Not only are features added, but the Elasticsearch core functionality is also constantly evolving and changing. We try to keep up with these changes, and because of this we are giving you the third edition of the book dedicated to Elasticsearch 2.x.

The first thing we need to do with Elasticsearch is install and configure it. With many applications, you start with the installation and configuration and usually forget the importance of these steps. We will try to guide you through these steps so that it becomes easier to remember. In addition to this, we will show you the simplest way to index and retrieve data without going into too much detail. The first chapter will take you on a quick ride through Elasticsearch and the full text search world. By the end of this chapter, you will have learned the following topics:

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

Full text searching
Back in the days when full text searching was a term known to a small percentage of engineers, most of us used SQL databases to perform search operations. Using SQL databases to search for the data stored in them was okay to some extent. Such a search wasn’t fast, especially on large amounts of data. Even now, small applications are usually good with a standard LIKE %phrase% search in a SQL database. However, as we go deeper and deeper, we start to see the limits of such an approach—a lack of scalability, not enough flexibility, and a lack of language analysis. Of course, there are additional modules that extend SQL databases with full text search capabilities, but they are still limited compared to dedicated full text search libraries and search engines such as Elasticsearch. Some of those reasons led to the creation of Apache Lucene (http://lucene.apache.org/), a library written completely in Java (http://java.com/en/), which is very fast, light, and provides language analysis for a large number of languages spoken throughout the world.

The Lucene glossary and architecture
Before going into the details of the analysis process, we would like to introduce you to the glossary and overall architecture of Apache Lucene. We decided that this information is crucial for understanding how Elasticsearch works, and even though the book is not about Apache Lucene, knowing the foundation of the Elasticsearch analytics and indexing engine is vital to fully understand how this great search engine works.

The basic concepts of the mentioned library are as follows:

• **Document**: This is the main data carrier used during indexing and searching, comprising one or more fields that contain the data we put in and get from Lucene.

• **Field**: This is a section of the document, which is built of two parts: the name and the value.

• **Term**: This is a unit of search representing a word from the text.

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

- Elasticsearch Server (document 1)
- Mastering Elasticsearch Second Edition (document 2)
- Apache Solr Cookbook Third Edition (document 3)

A very simplified visualization of the Lucene inverted index could look as follows:

<table>
<thead>
<tr>
<th>Term</th>
<th>Count</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>1</td>
<td>&lt;3&gt;</td>
</tr>
<tr>
<td>cookbook</td>
<td>1</td>
<td>&lt;3&gt;</td>
</tr>
<tr>
<td>edition</td>
<td>2</td>
<td>&lt;2&gt;, &lt;3&gt;</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>2</td>
<td>&lt;1&gt;, &lt;2&gt;</td>
</tr>
<tr>
<td>mastering</td>
<td>1</td>
<td>&lt;2&gt;</td>
</tr>
<tr>
<td>second</td>
<td>1</td>
<td>&lt;2&gt;</td>
</tr>
<tr>
<td>server</td>
<td>1</td>
<td>&lt;1&gt;</td>
</tr>
<tr>
<td>solr</td>
<td>1</td>
<td>&lt;3&gt;</td>
</tr>
<tr>
<td>third</td>
<td>1</td>
<td>&lt;3&gt;</td>
</tr>
</tbody>
</table>

Each term points to the number of documents it is present in. For example, the term edition is present twice in the second and third documents. Such a structure allows for very efficient and fast search operations in term-based queries (but not exclusively). Because the occurrences of the term are connected to the terms themselves, Lucene can use information about the term occurrences to perform fast and precise scoring information by giving each document a value that represents how well each of the returned documents matched the query.

Of course, the actual index created by Lucene is much more complicated and advanced because of additional files that include information such as term vectors (per document inverted index), doc values (column oriented field information), stored fields (the original and not the analyzed value of the field), and so on. However, all you need to know for now is how the data is organized and not what exactly is stored.
Each index is divided into multiple write-once and read-many-time structures called segments. Each segment is a miniature Apache Lucene index on its own. When indexing, after a single segment is written to the disk it can't be updated, or we should rather say it can't be fully updated; documents can't be removed from it, they can only be marked as deleted in a separate file. The reason that Lucene doesn't allow segments to be updated is the nature of the inverted index. After the fields are analyzed and put into the inverted index, there is no easy way of building the original document structure. When deleting, Lucene would have to delete the information from the segment, which translates to updating all the information within the inverted index itself.

Because of the fact that segments are write-once structures Lucene is able to merge segments together in a process called segment merging. During indexing, if Lucene thinks that there are too many segments falling into the same criterion, a new and bigger segment will be created—one that will have data from the other segments. During that process, Lucene will try to remove deleted data and get back the space needed to hold information about those documents. Segment merging is a demanding operation both in terms of the I/O and CPU. What we have to remember for now is that searching with one large segment is faster than searching with multiple smaller ones holding the same data. That's because, in general, searching translates to just matching the query terms to the ones that are indexed. You can imagine how searching through multiple small segments and merging those results will be slower than having a single segment preparing the results.

**Input data analysis**

The transformation of a document that comes to Lucene and is processed and put into the inverted index format is called *indexation*. One of the things Lucene has to do during this is data analysis. You may want some of your fields to be processed by a language analyzer so that words such as car and cars are treated as the same by your index. On the other hand, you may want other fields to be divided only on the white space character or be only lowercased.

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

A tokenizer in Lucene is used to split the text into tokens, which are basically the terms with additional information such as its position in the original text and its length. The results of the tokenizer's work is called a token stream, where the tokens are put one by one and are ready to be processed by the filters.
Apart from the tokenizer, the Lucene analyzer is built of zero or more token filters that are used to process tokens in the token stream. Some examples of filters are as follows:

- **Lowercase filter**: Makes all the tokens lowercased
- **Synonyms filter**: Changes one token to another on the basis of synonym rules
- **Language stemming filters**: Responsible for reducing tokens (actually, the text part that they provide) into their root or base forms called the stem (https://en.wikipedia.org/wiki/Word_stem)

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

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

### Indexing and querying

You may wonder how all the information we've described so far affects indexing and querying when using Lucene and all the software that is built on top of it. During indexing, Lucene will use an analyzer of your choice to process the contents of your document; of course, different analyzers can be used for different fields, so the name field of your document can be analyzed differently compared to the summary field.

For example, the name field may only be tokenized on whitespaces and lowercased, so that exact matches are done and the summary field is stemmed in addition to that. We can also decide to not analyze the fields at all—we have full control over the analysis process.

During a query, your query text can be analyzed as well. However, you can also choose not to analyze your queries. This is crucial to remember because some Elasticsearch queries are analyzed and some are not. For example, prefix and term queries are not analyzed, and match queries are analyzed (we will get to that in Chapter 3, Searching Your Data). Having queries that are analyzed and not analyzed is very useful; sometimes, you may want to query a field that is not analyzed, while sometimes you may want to have a full text search analysis. For example, if we search for the **LightRed** term and the query is being analyzed by the standard analyzer, then the terms that would be searched are light and red. If we use a query type that has not been analyzed, then we will explicitly search for the **LightRed** term. We may not want to analyze the content of the query if we are only interested in exact matches.
What you should remember about indexing and querying analysis is that the index should match the query term. If they don't match, Lucene won't return the desired documents. For example, if you use stemming and lowercasing during indexing, you need to ensure that the terms in the query are also lowercased and stemmed, or your queries won't return any results at all. For example, let's get back to our LightRed term that we analyzed during indexing; we have it as two terms in the index: light and red. If we run a LightRed query against that data and don't analyze it, we won't get the document in the results—the query term does not match the indexed terms. It is important to keep the token filters in the same order during indexing and query time analysis so that the terms resulting from such an analysis are the same.

**Scoring and query relevance**

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

If you want to read more about the Apache Lucene TF-IDF scoring formula, please visit Apache Lucene Javadocs for the TFIDF. The similarity class is available at [http://lucene.apache.org/core/5_4_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html](http://lucene.apache.org/core/5_4_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html).

**The basics of Elasticsearch**

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

In the next few pages, we will get you through the basic concepts of Elasticsearch. You can skip this section if you are already familiar with Elasticsearch architecture. However, if you are not familiar with Elasticsearch, we strongly advise you to read this section. We will refer to the key words used in this section in the rest of the book, and understanding those concepts is crucial to fully utilize Elasticsearch.

Index

An index is the logical place where Elasticsearch stores the data. Each index can be spread onto multiple Elasticsearch nodes and is divided into one or more smaller pieces called shards that are physically placed on the hard drives. If you are coming from the relational database world, you can think of an index like a table. However, the index structure is prepared for fast and efficient full text searching and, in particular, does not store original values. That structure is called an inverted index (https://en.wikipedia.org/wiki/Inverted_index).

If you know MongoDB, you can think of the Elasticsearch index as a collection in MongoDB. If you are familiar with CouchDB, you can think about an index as you would about the CouchDB database. Elasticsearch can hold many indices located on one machine or spread them over multiple servers. As we have already said, every index is built of one or more shards, and each shard can have many replicas.

Document

The main entity stored in Elasticsearch is a document. A document can have multiple fields, each having its own type and treated differently. Using the analogy to relational databases, a document is a row of data in a database table. When you compare an Elasticsearch document to a MongoDB document, you will see that both can have different structures. The thing to keep in mind when it comes to Elasticsearch is that fields that are common to multiple types in the same index need to have the same type. This means that all the documents with a field called title need to have the same data type for it, for example, string.

Documents consist of fields, and each field may occur several times in a single document (such a field is called multivalued). Each field has a type (text, number, date, and so on). The field types can also be complex—a field can contain other subdocuments or arrays. The field type is important to Elasticsearch because type determines how various operations such as analysis or sorting are performed. Fortunately, this can be determined automatically (however, we still suggest using mappings; take a look at what follows).
Getting Started with Elasticsearch Cluster

Unlike the relational databases, documents don't need to have a fixed structure—every document may have a different set of fields, and in addition to this, fields don't have to be known during application development. Of course, one can force a document structure with the use of schema. From the client's point of view, a document is a JSON object (see more about the JSON format at https://en.wikipedia.org/wiki/JSON). Each document is stored in one index and has its own unique identifier, which can be generated automatically by Elasticsearch, and document type. The thing to remember is that the document identifier needs to be unique inside an index and should be for a given type. This means that, in a single index, two documents can have the same unique identifier if they are not of the same type.

**Document type**

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

**Mapping**

In the section about the basics of full text searching (the *Full text searching* section), we wrote about the process of analysis—the preparation of the input text for indexing and searching done by the underlying Apache Lucene library. Every field of the document must be properly analyzed depending on its type. For example, a different analysis chain is required for the numeric fields (numbers shouldn't be sorted alphabetically) and for the text fetched from web pages (for example, the first step would require you to omit the HTML tags as it is useless information). To be able to properly analyze at indexing and querying time, Elasticsearch stores the information about the fields of the documents in so-called mappings. Every document type has its own mapping, even if we don't explicitly define it.
Key concepts of the Elasticsearch infrastructure

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

Nodes and clusters

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

Shards

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

Replicas

In order to increase query throughput or achieve high availability, shard replicas can be used. A replica is just an exact copy of the shard, and each shard can have zero or more replicas. In other words, Elasticsearch can have many identical shards and one of them is automatically chosen as a place where the operations that change the index are directed. This special shard is called a primary shard, and the others are called replica shards. When the primary shard is lost (for example, a server holding the shard data is unavailable), the cluster will promote the replica to be the new primary shard.
Gateway
The cluster state is held by the gateway, which stores the cluster state and indexed data across full cluster restarts. By default, every node has this information stored locally; it is synchronized among nodes. We will discuss the gateway module in The gateway and recovery modules section of Chapter 9, Elasticsearch Cluster, in detail.

Indexing and searching
You may wonder how you can tie all the indices, shards, and replicas together in a single environment. Theoretically, it would be very difficult to fetch data from the cluster when you have to know where your document is: on which server, and in which shard. Even more difficult would be searching when one query can return documents from different shards placed on different nodes in the whole cluster. In fact, this is a complicated problem; fortunately, we don't have to care about this at all—it is handled automatically by Elasticsearch. Let's look at the following diagram:
When you send a new document to the cluster, you specify a target index and send it to any of the nodes. The node knows how many shards the target index has and is able to determine which shard should be used to store your document. Elasticsearch can alter this behavior; we will talk about this in the Introduction to routing section in Chapter 2, Indexing Your Data. The important information that you have to remember for now is that Elasticsearch calculates the shard in which the document should be placed using the unique identifier of the document—this is one of the reasons each document needs a unique identifier. After the indexing request is sent to a node, that node forwards the document to the target node, which hosts the relevant shard.

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

![Diagram of searching request execution]

When you try to fetch a document by its identifier, the node you send the query to uses the same routing algorithm to determine the shard and the node holding the document and again forwards the request, fetches the result, and sends the result to you. On the other hand, the querying process is a more complicated one. The node receiving the query forwards it to all the nodes holding the shards that belong to a given index and asks for minimum information about the documents that match the query (the identifier and score are matched by default), unless routing is used, when the query will go directly to a single shard only. This is called the scatter phase. After receiving this information, the aggregator node (the node that receives the client request) sorts the results and sends a second request to get the documents that are needed to build the results list (all the other information apart from the document identifier and score). This is called the gather phase. After this phase is executed, the results are returned to the client.
Now the question arises: what is the replica's role in the previously described process? While indexing, replicas are only used as an additional place to store the data. When executing a query, by default, Elasticsearch will try to balance the load among the shard and its replicas so that they are evenly stressed. Also, remember that we can change this behavior; we will discuss this in the Understanding the querying process section in Chapter 3, Searching Your Data.

**Installing and configuring your cluster**

Installing and running Elasticsearch even in production environments is very easy nowadays, compared to how it was in the days of Elasticsearch 0.20.x. From a system that is not ready to one with Elasticsearch, there are only a few steps that one needs to go. We will explore these steps in the following section:

**Installing Java**

Elasticsearch is a Java application and to use it we need to make sure that the Java SE environment is installed properly. Elasticsearch requires Java Version 7 or later to run. You can download it from [http://www.oracle.com/technetwork/java/javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html). You can also use OpenJDK ([http://openjdk.java.net/](http://openjdk.java.net/)) if you wish. You can, of course, use Java Version 7, but it is not supported by Oracle anymore, at least without commercial support. For example, you can't expect new, patched versions of Java 7 to be released. Because of this, we strongly suggest that you install Java 8, especially given that Java 9 seems to be right around the corner with the general availability planned to be released in September 2016.

**Installing Elasticsearch**

To install Elasticsearch you just need to go to [https://www.elastic.co/downloads/elasticsearch](https://www.elastic.co/downloads/elasticsearch), choose the last stable version of Elasticsearch, download it, and unpack it. That's it! The installation is complete.

At the time of writing, we used a snapshot of Elasticsearch 2.2. This means that we've skipped describing some properties that were marked as deprecated and are or will be removed in the future versions of Elasticsearch.
The main interface to communicate with Elasticsearch is based on the HTTP protocol and REST. This means that you can even use a web browser for some basic queries and requests, but for anything more sophisticated you’ll need to use additional software, such as the cURL command. If you use the Linux or OS X command, the cURL package should already be available. If you use Windows, you can download the package from http://curl.haxx.se/download.html.

Running Elasticsearch

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

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

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

```json
{
   "name" : "Blob",
   "cluster_name" : "elasticsearch",
   "version" : {
      "number" : "2.2.0",
      "build_hash" : "5b1dd1cf5a1957682d84228a569e124fedf8e325",
      "build_timestamp" : "2016-01-13T18:12:26Z",
      "build_snapshot" : true,
      "lucene_version" : "5.4.0"
   },
   "tagline" : "You Know, for Search"
}
```

The output is structured as a JavaScript Object Notation (JSON) object. If you are not familiar with JSON, please take a minute and read the article available at https://en.wikipedia.org/wiki/JSON.
Elasticsearch is smart. If the default port is not available, the engine binds to the next free port. You can find information about this on the console during booting as follows:

```
[2016-01-13 20:04:49,953] [INFO ] [http ] [Blob] publish_address {127.0.0.1:9201}, bound_addresses {{fe80::1}:9200}, {{::1}:9200}, {127.0.0.1:9201}
```

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

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

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

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

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

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

```
[2016-01-13 20:07:58,561] [INFO ] [cluster.service ] [Big Man] detected_master {Blob}{5QPh00RUQraeLHAIrb4Jw}{127.0.0.1}{127.0.0.1:9300}, added {{Blob}{5QPh00RUQraeLHAIrb4Jw}{127.0.0.1}{127.0.0.1:9300},}, reason: zen-disco-receive(from master [{Blob}{5QPh00RUQraeLHAIrb4Jw}{127.0.0.1}{127.0.0.1:9300}])
```

This means that our second instance (named Big Man) discovered the previously running instance (named Blob). Here, Elasticsearch automatically formed a new two-node cluster. Starting from Elasticsearch 2.0, this will only work with nodes running on the same physical machine—because Elasticsearch 2.0 no longer supports multicast. To allow your cluster to form, you need to inform Elasticsearch about the nodes that should be contacted initially using the `discovery.zen.ping.unicast.hosts` array in `elasticsearch.yml`. For example, like this:

```
discovery.zen.ping.unicast.hosts: ["192.168.2.1", "192.168.2.2"]
```
Shutting down Elasticsearch

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

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

The previous versions of Elasticsearch exposed a dedicated shutdown API but, in 2.0, this option has been removed because of security reasons.

The directory layout

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

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin</td>
<td>The scripts needed to run Elasticsearch instances and for plugin management</td>
</tr>
<tr>
<td>Config</td>
<td>The directory where configuration files are located</td>
</tr>
<tr>
<td>Lib</td>
<td>The libraries used by Elasticsearch</td>
</tr>
<tr>
<td>Modules</td>
<td>The plugins bundled with Elasticsearch</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>The directory used by Elasticsearch to store all the data</td>
</tr>
<tr>
<td>Logs</td>
<td>The files with information about events and errors</td>
</tr>
<tr>
<td>Plugins</td>
<td>The location to store the installed plugins</td>
</tr>
<tr>
<td>Work</td>
<td>The temporary files used by Elasticsearch</td>
</tr>
</tbody>
</table>
Configuring Elasticsearch

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

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

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

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

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

Remember that most of the parameters that have been set in the elasticsearch.yml file can be overwritten with the use of the Elasticsearch REST API. We will talk about this API in The update settings API section of Chapter 9, Elasticsearch Cluster in Detail.
The second file (logging.yml) defines how much information is written to system logs, defines the log files, and creates new files periodically. Changes in this file are usually required only when you need to adapt to monitoring or backup solutions or during system debugging; however, if you want to have a more detailed logging, you need to adjust it accordingly.

Let's leave the configuration files for now and look at the base for all the applications—the operating system. Tuning your operating system is one of the key points to ensure that your Elasticsearch instance will work well. During indexing, especially when having many shards and replicas, Elasticsearch will create many files; so, the system cannot limit the open file descriptors to less than 32,000. For Linux servers, this can usually be changed in /etc/security/limits.conf and the current value can be displayed using the ulimit command. If you end up reaching the limit, Elasticsearch will not be able to create new files; so merging will fail, indexing may fail, and new indices will not be created.

On Microsoft Windows platforms, the default limit is more than 16 million handles per process, which should be more than enough. You can read more about file handles on the Microsoft Windows platform at https://blogs.technet.microsoft.com/markerussinovich/2009/09/29/pushing-the-limits-of-windows-handles/.

The next set of settings is connected to the Java Virtual Machine (JVM) heap memory limit for a single Elasticsearch instance. For small deployments, the default memory limit (1,024 MB) will be sufficient, but for large ones it will not be enough. If you spot entries that indicate OutOfMemoryError exceptions in a log file, set the ES_HEAP_SIZE variable to a value greater than 1024. When choosing the right amount of memory size to be given to the JVM, remember that, in general, no more than 50 percent of your total system memory should be given. However, as with all the rules, there are exceptions. We will discuss this in greater detail later, but you should always monitor your JVM heap usage and adjust it when needed.
The system-specific installation and configuration

Although downloading an archive with Elasticsearch and unpacking it works and is convenient for testing, there are dedicated methods for Linux operating systems that give you several advantages when you do production deployment. In production deployments, the Elasticsearch service should be run automatically with a system boot; we should have dedicated start and stop scripts, unified paths, and so on. Elasticsearch supports installation packages for various Linux distributions that we can use. Let's see how this works.

Installing Elasticsearch on Linux

The other way to install Elasticsearch on a Linux operating system is to use packages such as RPM or DEB, depending on your Linux distribution and the supported package type. This way we can automatically adapt to system directory layout; for example, configuration and logs will go into their standard places in the /etc/ or /var/log directories. But this is not the only thing. When using packages, Elasticsearch will also install startup scripts and make our life easier. What's more, we will be able to upgrade Elasticsearch easily by running a single command from the command line. Of course, the mentioned packages can be found at the same URL address as we mentioned previously when we talked about installing Elasticsearch from zip or tar.gz packages: https://www.elastic.co/downloads/elasticsearch. Elasticsearch can also be installed from remote repositories via standard distribution tools such as apt-get or yum.

Before installing Elasticsearch, make sure that you have a proper version of Java Virtual Machine installed.

Installing Elasticsearch using RPM packages

When using a Linux distribution that supports RPM packages such as Fedora Linux, (https://getfedora.org/) Elasticsearch installation is very easy. After downloading the RPM package, we just need to run the following command as root:

```
yum elasticsearch-2.2.0.noarch.rpm
```

Alternatively, you can add the remote repository and install Elasticsearch from it (this command needs to be run as root as well):

```
rpm --import https://packages.elastic.co/GPG-KEY-elasticsearch
```
This command adds the GPG key and allows the system to verify that the fetched package really comes from Elasticsearch developers. In the second step, we need to create the repository definition in the /etc/yum.repos.d/elasticsearch.repo file. We need to add the following entries to this file:

```plaintext
[elasticsearch-2.2]
name=Elasticsearch repository for 2.2.x packages
baseurl=http://packages.elastic.co/elasticsearch/2.x/centos
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1
```

Now it's time to install the Elasticsearch server, which is as simple as running the following command (again, don't forget to run it as root):

```
yum install elasticsearch
```

Elasticsearch will be automatically downloaded, verified, and installed.

**Installing Elasticsearch using the DEB package**

When using a Linux distribution that supports DEB packages (such as Debian), installing Elasticsearch is again very easy. After downloading the DEB package, all you need to do is run the following command:

```
sudo dpkg -i elasticsearch-2.2.0.deb
```

It is as simple as that. Another way, which is similar to what we did with RPM packages, is by creating a new packages source and installing Elasticsearch from the remote repository. The first step is to add the public GPG key used for package verification. We can do that using the following command:

```
wget -qO - https://packages.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
```

The second step is by adding the DEB package location. We need to add the following line to the /etc/apt/sources.list file:

```
deb http://packages.elastic.co/elasticsearch/2.2/debian stable main
```

This defines the source for the Elasticsearch packages. The last step is updating the list of remote packages and installing Elasticsearch using the following command:

```
sudo apt-get update && sudo apt-get install elasticsearch
```
Elasticsearch configuration file localization

When using packages to install Elasticsearch, the configuration files are in slightly different directories than the default conf directory. After the installation, the configuration files should be stored in the following location:

- `/etc/sysconfig/elasticsearch` or `/etc/default/elasticsearch`:
  A file with the configuration of the Elasticsearch process as a user to run as, directories for logs, data and memory settings
- `/etc/elasticsearch/`:
  A directory for the Elasticsearch configuration files, such as the elasticsearch.yml file

Configuring Elasticsearch as a system service on Linux

If everything goes well, you can run Elasticsearch using the following command:

```
/bin/systemctl start elasticsearch.service
```

If you want Elasticsearch to start automatically every time the operating system starts, you can set up Elasticsearch as a system service by running the following command:

```
/bin/systemctl enable elasticsearch.service
```

Elasticsearch as a system service on Windows

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

```
service.bat install
```

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

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

```
service.bat
```

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

```
service.bat start
```
Manipulating data with the REST API

Elasticsearch exposes a very rich REST API that can be used to search through the data, index the data, and control Elasticsearch behavior. You can imagine that using the REST API allows you to get a single document, index or update a document, get the information on Elasticsearch current state, create or delete indices, or force Elasticsearch to move around shards of your indices. Of course, these are only examples that show what you can expect from the Elasticsearch REST API. For now, we will concentrate on using the create, retrieve, update, delete (CRUD) part of the Elasticsearch API (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete), which allows us to use Elasticsearch in a fashion similar to how we would use any other NoSQL (https://en.wikipedia.org/wiki/NoSQL) data store.

Understanding the REST API

If you've never used an application exposing the REST API, you may be surprised how easy it is to use such applications and remember how to use them. In REST-like architectures, every request is directed to a concrete object indicated by a path in the address. For example, let's assume that our hypothetical application exposes the /books REST end-point as a reference to the list of books. In such case, a call to /books/1 could be a reference to a concrete book with the identifier 1. You can think of it as a data-oriented model of an API. Of course, we can nest the paths—for example, a path such as /books/1/chapters could return the list of chapters of our book with identifier 1 and a path such as /books/1/chapters/6 could be a reference to the sixth chapter in that particular book.

We talked about paths, but when using the HTTP protocol, (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) we have some additional verbs (such as POST, GET, PUT, and so on.) that we can use to define system behavior in addition to paths. So if we would like to retrieve the book with identifier 1, we would use the GET request method with the /books/1 path. However, we would use the PUT request method with the same path to create a book record with the identifier or one, the POST request method to alter the record, DELETE to remove that entry, and the HEAD request method to get basic information about the data referenced by the path.

Now, let's look at example HTTP requests that are sent to real Elasticsearch REST API endpoints, so the preceding hypothetical information will be turned into something real:

GET http://localhost:9200/: This retrieves basic information about Elasticsearch, such as the version, the name of the node that the command has been sent to, the name of the cluster that node is connected to, the Apache Lucene version, and so on.
GET http://localhost:9200/_cluster/state/nodes/ This retrieves information about all the nodes in the cluster, such as their identifiers, names, transport addresses with ports, and additional node attributes for each node.

DELETE http://localhost:9200/books/book/123: This deletes a document that is indexed in the books index, with the book type and an identifier of 123.

We now know what REST means and we can start concentrating on Elasticsearch to see how we can store, retrieve, alter, and delete the data from its indices. If you would like to read more about REST, please refer to http://en.wikipedia.org/wiki/Representational_state_transfer.

Storing data in Elasticsearch

In Elasticsearch, every document is represented by three attributes—the index, the type, and the identifier. Each document must be indexed into a single index, needs to have its type correspond to the document structure, and is described by the identifier. These three attributes allows us to identify any document in Elasticsearch and needs to be provided when the document is physically written to the underlying Apache Lucene index. Having the knowledge, we are now ready to create our first Elasticsearch document.

Creating a new document

We will start learning the Elasticsearch REST API by indexing one document. Let's imagine that we are building a CMS system (http://en.wikipedia.org/wiki/Content_management_system) that will provide the functionality of a blogging platform for our internal users. We will have different types of documents in our indices, but the most important ones are the articles that will be published and are readable by users.

Because we talk to Elasticsearch using JSON notation and Elasticsearch responds to us again using JSON, our example document could look as follows:

```json
{
    "id": "1",
    "title": "New version of Elasticsearch released!",
    "content": "Version 2.2 released today!",
    "priority": 10,
    "tags": ["announce", "elasticsearch", "release"]
}
```
As you can see in the preceding code snippet, the JSON document is built with a set of fields, where each field can have a different format. In our example, we have a set of text fields (id, title, and content), we have a number (the priority field), and an array of text values (the tags field). We will show documents that are more complicated in the next examples.

One of the changes introduced in Elasticsearch 2.0 has been that field names can't contain the dot character. Such field names were possible in older versions of Elasticsearch, but could result in serialization errors in certain cases and thus Elasticsearch creators decided to remove that possibility.

One thing to remember is that by default Elasticsearch works as a schema-less data store. This means that it can try to guess the type of the field in a document sent to Elasticsearch. It will try to use numeric types for the values that are not enclosed in quotation marks and strings for data enclosed in quotation marks. It will try to guess the date and index them in dedicated fields and so on. This is possible because the JSON format is semi-typed. Internally, when the first document with a new field is sent to Elasticsearch, it will be processed and mappings will be written (we will talk more about mappings in the Mappings configuration section of Chapter 2, Indexing Your Data).

Let’s now index our document and make it available for retrieval and searching. We will index our articles to an index called blog under a type named article. We will also give our document an identifier of 1, as this is our first document. To index our example document, we will execute the following command:

```
curl -XPUT 'http://localhost:9200/blog/article/1' -d '{"title": "New version of Elasticsearch released!", "content": "Version 2.2 released today!", "priority": 10, "tags": ["announce", "elasticsearch", "release"]}
```
Note a new option to the curl command, the -d parameter. The value of this option is the text that will be used as a request payload—a request body. This way, we can send additional information such as the document definition. Also, note that the unique identifier is placed in the URL and not in the body. If you omit this identifier (while using the HTTP PUT request), the indexing request will return the following error:

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

If everything worked correctly, Elasticsearch will return a JSON response informing us about the status of the indexing operation. This response should be similar to the following one:

```json
{
    "_index":"blog",
    "_type":"article",
    "_id":"1",
    "_version":1,
    "_shards":{
        "total":2,
        "successful":1,
        "failed":0
    },
    "created":true
}
```

In the preceding response, Elasticsearch included information about the status of the operation, index, type, identifier, and version. We can also see information about the shards that took part in the operation—all of them, the ones that were successful and the ones that failed.

**Automatic identifier creation**

In the previous example, we specified the document identifier manually when we were sending the document to Elasticsearch. However, there are use cases when we don't have an identifier for our documents—for example, when handling logs as our data. In such cases, we would like some application to create the identifier for us and Elasticsearch can be such an application. Of course, generating document identifiers doesn't make sense when your document already has them, such as data in a relational database. In such cases, you may want to update the documents; in this case, automatic identifier generation is not the best idea. However, when we are in need of such functionality, instead of using the HTTP PUT method we can use POST and omit the identifier in the REST API path. So if we would like Elasticsearch to generate the identifier in the previous example, we would send a command like this:

```
curl -XPOST 'http://localhost:9200/blog/article/' -d '{"title": "New version of Elasticsearch released!", "content": "Version 2.2 released today!", "priority": 10, "tags": ["announce", "elasticsearch", "release"] }'
```
We've used the HTTP POST method instead of PUT and we've omitted the identifier. The response produced by Elasticsearch in such a case would be as follows:

```
{
  "_index":"blog",
  "_type":"article",
  "_id":"AU1y-s6w2WzST_RhTvCJ",
  "_version":1,
  "_shards":{
    "total":2,
    "successful":1,
    "failed":0,
    "created":true
  }
}
```

As you can see, the response returned by Elasticsearch is almost the same as in the previous example, with a minor difference—the _id field is returned. Now, instead of the 1 value, we have a value of AU1y-s6w2WzST_RhTvCJ, which is the identifier Elasticsearch generated for our document.

## Retrieving documents

We now have two documents indexed into our Elasticsearch instance—one using an explicit identifier and one using a generated identifier. Let's now try to retrieve one of the documents using its unique identifier. To do this, we will need information about the index the document is indexed in, what type it has, and of course what identifier it has. For example, to get the document from the blog index with the article type and the identifier of 1, we would run the following HTTP GET request:

```
curl -XGET 'localhost:9200/blog/article/1?pretty'
```

The additional URI property called pretty tells Elasticsearch to include new line characters and additional white spaces in response to make the output easier to read for users.

Elasticsearch will return a response similar to the following:

```
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "1",
  "_version" : 1,
  "found" : true,
  "_source": {
    "title" : "New version of Elasticsearch released!",
    "content" : "Version 2.2 released today!"
  }
}
```
Getting Started with Elasticsearch Cluster

As you can see in the preceding response, Elasticsearch returned the _source field, which is the original document sent to Elasticsearch and a few additional fields that tell us about the document, such as the index, type, identifier, document version, and of course information as to whether the document was found or not (the found property).

If we try to retrieve a document that is not present in the index, such as the one with the 12345 identifier, we get a response like this:

```json
{
    "_index": "blog",
    "_type": "article",
    "_id": "12345",
    "found": false
}
```

As you can see, this time the value of the found property was set to false and there was no _source field because the document has not been retrieved.

### Updating documents

Updating documents in the index is a more complicated task compared to indexing. When the document is indexed and Elasticsearch flushes the document to a disk, it creates segments—an immutable structure that is written once and read many times. This is done because the inverted index created by Apache Lucene is currently impossible to update (at least most of its parts). To update a document, Elasticsearch internally first fetches the document using the GET request, modifies its _source field, removes the old document, and indexes a new document using the updated content. The content update is done using scripts in Elasticsearch (we will talk more about scripting in Elasticsearch in the Scripting capabilities of Elasticsearch section in Chapter 6, Make Your Search Better).

Please note that the following document update examples require you to put the script.inline: on property into your elasticsearch.yml configuration file. This is needed because inline scripting is disabled in Elasticsearch for security reasons. The other way to handle updates is to store the script content in the file in the Elasticsearch configuration directory, but we will talk about that in the Scripting capabilities of Elasticsearch section in Chapter 6, Make Your Search Better.
Let's now try to update our document with identifier 1 by modifying its content field to contain the This is the updated document sentence. To do this, we need to run a POST HTTP request on the document path using the _update REST end-point. Our request to modify the document would look as follows:

```
curl -XPOST 'http://localhost:9200/blog/article/1/_update' -d '{
    "script" : "ctx._source.content = new_content",
    "params" : {
        "new_content" : "This is the updated document"
    }
}
''
```

As you can see, we've sent the request to the /blog/article/1/_update REST end-point. In the request body, we've provided two parameters—the update script in the script property and the parameters of the script. The script is very simple; it takes the _source field and modifies the content field by setting its value to the value of the new_content parameter. The params property contains all the script parameters.

For the preceding update command execution, Elasticsearch would return the following response:

```
{
    "_index":"blog","_type":"article","_id":"1","_version":2,"_shards":{
    "total":2,"successful":1,"failed":0}
}
```

The thing to look at in the preceding response is the _version field. Right now, the version is 2, which means that the document has been updated (or re-indexed) once. Basically, each update makes Elasticsearch update the _version field.

We could also update the document using the doc section and providing the changed field, for example:

```
curl -XPOST 'http://localhost:9200/blog/article/1/_update' -d '{
    "doc" : {
        "content" : "This is the updated document"
    }
}
''
```

We now retrieve the document using the following command:

```
curl -XGET 'http://localhost:9200/blog/article/1?pretty'
```
And we get the following response from Elasticsearch:

```
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "1",
  "_version" : 2,
  "found" : true,
  "_source" : {
    "title" : "New version of Elasticsearch released!",
    "content" : "This is the updated document",
    "priority" : 10,
    "tags" : [ "announce", "elasticsearch", "release" ]
  }
}
```

As you can see, the document has been updated properly.

The thing to remember when using the update API of Elasticsearch is that the `_source` field needs to be present because this is the field that Elasticsearch uses to retrieve the original document content from the index. By default, that field is enabled and Elasticsearch uses it to store the original document.

### Dealing with non-existing documents

The nice thing when it comes to document updates, which we would like to mention as it can come in handy when using Elasticsearch Update API, is that we can define what Elasticsearch should do when the document we try to update is not present.

For example, let’s try incrementing the priority field value for a non-existing document with identifier 2:

```
curl -XPOST 'http://localhost:9200/blog/article/2/_update' -d '{
  "script" : "ctx._source.priority += 1"
}
```

The response returned by Elasticsearch would look more or less as follows:

```
{"error":{"root_cause":[{"type":"document_missing_exception","reason":"[article][2]: document missing","shard":"2","index":"blog"}],"type":"document_missing_exception","reason":"[article][2]: document missing","shard":"2","index":"blog"},"status":404}
```
As you can imagine, the document has not been updated because it doesn't exist. So now, let's modify our request to include the upsert section in our request body that will tell Elasticsearch what to do when the document is not present. The new command would look as follows:

```
curl -XPOST 'http://localhost:9200/blog/article/2/_update' -d '{
    "script" : "ctx._source.priority += 1",
    "upsert" : {
        "title" : "Empty document",
        "priority" : 0,
        "tags" : ["empty"]
    }
}
```

With the modified request, a new document would be indexed; if we retrieve it using the GET API, it will look as follows:

```
{
    "_index" : "blog",
    "_type" : "article",
    "_id" : "2",
    "_version" : 1,
    "found" : true,
    "source" : {
        "title" : "Empty document",
        "priority" : 0,
        "tags" : ["empty"]
    }
}
```

As you can see, the fields from the upsert section of our update request were taken by Elasticsearch and used as document fields.

### Adding partial documents

In addition to what we already wrote about the update API, Elasticsearch is also capable of merging partial documents from the update request to already existing documents or indexing new documents using information about the request, similar to what we saw seen with the upsert section.
Let's imagine that we would like to update our initial document and add a new field called `count` to it (setting it to 1 initially). We would also like to index the document under the specified identifier if the document is not present. We can do this by running the following command:

```bash
curl -XPOST 'http://localhost:9200/blog/article/1/_update' -d '{
  "doc" : {
    "count" : 1,
  },
  "doc_as_upsert" : true
}
```

We specified the new field in the `doc` section and we said that we want the `doc` section to be treated as the `upsert` section when the document is not present (with the `doc_as_upsert` property set to `true`).

If we now retrieve that document, we see the following response:

```json
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "1",
  "_version" : 3,
  "found" : true,
  "source" : {
    "title" : "New version of Elasticsearch released!",
    "content" : "This is the updated document",
    "priority" : 10,
    "tags" : [ "announce", "elasticsearch", "release" ],
    "count" : 1
  }
}
```

For a full reference on document updates, please refer to the official Elasticsearch documentation on the Update API, which is available at https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-update.html.
Chapter 1

Deleting documents
Now that we know how to index documents, update them, and retrieve them, it is time to learn about how we can delete them. Deleting a document from an Elasticsearch index is very similar to retrieving it, but with one major difference—instead of using the HTTP GET method, we have to use HTTP DELETE one.

For example, if we would like to delete the document indexed in the blog index under the article type and with an identifier of 1, we would run the following command:

curl -XDELETE 'localhost:9200/blog/article/1'

The response from Elasticsearch indicates that the document has been deleted and should look as follows:

```json
{
    "found":true,
    "_index":"blog",
    "_type":"article",
    "_id":"1",
    "_version":4,
    "_shards":{
        "total":2,
        "successful":1,
        "failed":0
    }
}
```

Of course, this is not the only thing when it comes to deleting. We can also remove all the documents of a given type. For example, if we would like to delete the entire blog index, we should just omit the identifier and the type, so the command would look like this:

curl -XDELETE 'localhost:9200/blog'

The preceding command would result in the deletion of the blog index.
Versioning

Finally, there is one last thing that we would like to talk about when it comes to data manipulation in Elasticsearch — the great feature of versioning. As you may have already noticed, Elasticsearch increments the document version when it does updates to it. We can leverage this functionality and use optimistic locking (http://en.wikipedia.org/wiki/Optimistic_concurrency_control), and avoid conflicts and overwrites when multiple processes or threads access the same document concurrently. You can assume that your indexing application may want to try to update the document, while the user would like to update the document while doing some manual work. The question that arises is: Which document should be the correct one — the one updated by the indexing application, the one updated by the user, or the merged document of the changes? What if the changes are conflicting? To handle such cases, we can use versioning.

Usage example

Let's index a new document to our blog index — one with an identifier of 10, and let's index its second version soon after we do that. The commands that do this look as follows:

```
curl -XPUT 'localhost:9200/blog/article/10' -d '{"title":"Test document"}'
curl -XPUT 'localhost:9200/blog/article/10' -d '{"title":"Updated test document"}'
```

Because we've indexed the document with the same identifier, it should have a version 2 (you can check it using the GET request).

Now, let's try deleting the document we've just indexed but let's specify a version property equal to 1. By doing this, we tell Elasticsearch that we are interested in deleting the document with the provided version. Because the document is a different version now, Elasticsearch shouldn't allow indexing with version 1. Let's check if what we say is true. The command we will use to send the delete request looks as follows:

```
curl -XDELETE 'localhost:9200/blog/article/10?version=1'
```

The response generated by Elasticsearch should be similar to the following one:

```
{
  "error" : {
    "root_cause" : [ {
      "type" : "version_conflict_engine_exception",
      "reason" : "[article][10]: version conflict, current [2], provided [1]",
      "shard" : 1,
```

[32]
As you can see, the delete operation was not successful—the versions didn't match. If we set the version property to 2, the delete operation would be successful:

curl -XDELETE 'localhost:9200/blog/article/10?version=2&pretty'

The response this time will look as follows:

```
{
    "found" : true,
    "_index" : "blog",
    "_type" : "article",
    "_id" : "10",
    "_version" : 3,
    "_shards" : {
        "total" : 2,
        "successful" : 1,
        "failed" : 0
    }
}
```

This time the delete operation has been successful because the provided version was proper.

**Versioning from external systems**

The very good thing about Elasticsearch versioning capabilities is that we can provide the version of the document that we would like Elasticsearch to use. This allows us to provide versions from external data systems that are our primary data stores. To do this, we need to provide an additional parameter during indexing—version_type=external and, of course, the version itself. For example, if we would like our document to have the 12345 version, we could send a request like this:

```
curl -XPUT 'localhost:9200/blog/article/20?version=12345&version_type=external' -d '{"title":"Test document"}'
```
The response returned by Elasticsearch is as follows:

```json
{
  "_index": "blog",
  "_type": "article",
  "_id": "20",
  "_version": 12345,
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "created": true
}
```

We just need to remember that, when using `version_type=external`, we need to provide the version in cases where we index the document. In cases where we would like to change the document and use optimistic locking, we need to provide a version parameter equal to, or higher than, the version present in the document.

### Searching with the URI request query

Before getting into the wonderful world of the Elasticsearch query language, we would like to introduce you to the simple but pretty flexible URI request search, which allows us to use a simple Elasticsearch query combined with the Lucene query language. Of course, we will extend our search knowledge using Elasticsearch in Chapter 3, Searching Your Data, but for now we will stick to the simplest approach.

### Sample data

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

```bash
curl -XPOST 'localhost:9200/books/es/1' -d '{"title":"Elasticsearch Server", "published": 2013}'
curl -XPOST 'localhost:9200/books/es/3' -d '{"title":"Mastering Elasticsearch", "published": 2013}'
curl -XPOST 'localhost:9200/books/solr/1' -d '{"title":"Apache Solr 4 Cookbook", "published": 2012}"
```

Running the preceding commands will create the book's index with two types: es and solr. The title and published fields will be indexed and thus, searchable.

**URI search**

All queries in Elasticsearch are sent to the _search endpoint. You can search a single index or multiple indices, and you can restrict your search to a given document type or multiple types. For example, in order to search our book's index, we will run the following command:

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

The results returned by Elasticsearch will include all the documents from our book's index (because no query has been specified) and should look similar to the following:

```json
{
  "took": 3,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 6,
    "max_score": 1.0,
    "hits": [ {
      "_index": "books",
      "_type": "es",
      "_id": "2",
      "_score": 1.0,
      "_source": {
        "title": "Elasticsearch Server Second Edition",
        "published": 2014
      }
    }, {
      "_index": "books",
      "_type": "es",
      "_id": "4",
      "_score": 1.0,
      "_source": {
        "title": "Mastering Elasticsearch Second Edition",
        "published": 2014
      }
    }
  ]
}
```
Getting Started with Elasticsearch Cluster

"published" : 2015
}

}, {
  "_index" : "books",
  "_type" : "solr",
  "_id" : "2",
  "_score" : 1.0,
  "_source" : {
    "title" : "Solr Cookbook Third Edition",
    "published" : 2015
  }
}, {
  "_index" : "books",
  "_type" : "es",
  "_id" : "1",
  "_score" : 1.0,
  "_source" : {
    "title" : "Elasticsearch Server",
    "published" : 2013
  }
}, {
  "_index" : "books",
  "_type" : "solr",
  "_id" : "1",
  "_score" : 1.0,
  "_source" : {
    "title" : "Apache Solr 4 Cookbook",
    "published" : 2012
  }
}, {
  "_index" : "books",
  "_type" : "es",
  "_id" : "3",
  "_score" : 1.0,
  "_source" : {
    "title" : "Mastering Elasticsearch",
    "published" : 2013
  }
}]
]
As you can see, the response has a header that tells you the total time of the query and the shards used in the query process. In addition to this, we have documents matching the query—the top 10 documents by default. Each document is described by the index, type, identifier, score, and the source of the document, which is the original document sent to Elasticsearch.

We can also run queries against many indices. For example, if we had another index called clients, we could also run a single query against these two indices as follows:

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

We can also run queries against all the data in Elasticsearch by omitting the index names completely or setting the queries to _all:

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

In a similar manner, we can also choose the types we want to use during searching. For example, if we want to search only in the es type in the book's index, we run a command as follows:

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

Please remember that, in order to search for a given type, we need to specify the index or multiple indices. Elasticsearch allows us to have quite a rich semantics when it comes to choosing index names. If you are interested, please refer to https://www.elastic.co/guide/en/elasticsearch/reference/current/multi-index.html; however, there is one thing we would like to point out. When running a query against multiple indices, it may happen that some of them do not exist or are closed. In such cases, the ignore_unavailable property comes in handy. When set to true, it tells Elasticsearch to ignore unavailable or closed indices.

For example, let's try running the following query:

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

The response would be similar to the following one:

```json
{
   "error": {
      "root_cause": [
         {
            "type": "index_missing_exception",
            "reason": "no such index",
            "index": "non_existing"
         }
      ],
      "type": "index_missing_exception"
   }
}
```
Now let’s check what will happen if we add the `ignore_unavailable=true` to our request and execute the following command:

```
curl -XGET 'localhost:9200/books,non_existing/_search?pretty&ignore_unavailable=true'
```

In this case, Elasticsearch would return the results without any error.

### Elasticsearch query response

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

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

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

```
{
    "took" : 37,
    "timed_out" : false,
    "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    },
    "hits" : {
        "total" : 4,
        "max_score" : 0.625,
        "hits" : [ {
            "_index" : "books",
            "_type" : "es",
            "_id" : "1",
            "score" : 0.625,
            "_source" : {
                "title" : "Elasticsearch Server",
                "published" : 2013
            }
        }, {
            "_index" : "books",
            "_type" : "es",
            "_id" : "2",
            "score" : 0.625,
            "_source" : {
                "title" : "Elasticsearch Server",
                "published" : 2013
            }
        }, {
            "_index" : "books",
            "_type" : "es",
            "_id" : "3",
            "score" : 0.625,
            "_source" : {
                "title" : "Elasticsearch Server",
                "published" : 2013
            }
        }, {
            "_index" : "books",
            "_type" : "es",
            "_id" : "4",
            "score" : 0.625,
            "_source" : {
                "title" : "Elasticsearch Server",
                "published" : 2013
            }
        }
    ]
}
```
The first section of the response gives us information about how much time the request took (the took property is specified in milliseconds), whether it was timed out (the timed_out property), and information about the shards that were queried during the request execution—the number of queried shards (the total property of the _shards object), the number of shards that returned the results successfully (the successful property of the _shards object), and the number of failed shards (the failed property of the _shards object). The query may also time out if it is executed for a longer period than we want. (We can specify the maximum query execution time using the timeout parameter.) The failed shard means that something went wrong with that shard or it was not available during the search execution.
Of course, the mentioned information can be useful, but usually, we are interested in the results that are returned in the hits object. We have the total number of documents returned by the query (in the total property) and the maximum score calculated (in the max_score property). Finally, we have the hits array that contains the returned documents. In our case, each returned document contains its index name (the _index property), the type (the _type property), the identifier (the _id property), the score (the _score property), and the _source field (usually, this is the JSON object sent for indexing.

**Query analysis**

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

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

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

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

The response will be as follows:

```
{
  "tokens": [{
    "token": "elasticsearch",
    "start_offset": 0,
    "end_offset": 13,
    "type": "<ALPHANUM>",
    "position": 0
  }, {
    "token": "server",
    "start_offset": 14,
```
You can see that Elasticsearch has divided the text into two terms—the first one has a token value of `elasticsearch` and the second one has a token value of the server.

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

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

The response of the request will look as follows:

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

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

**URI query string parameters**

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

```sh
```

Please remember to enclose the URL of the request using the ` '` characters because, on Linux-based systems, the `&` character will be analyzed by the Linux shell.
The query
The q parameter allows us to specify the query that we want our documents to match. It allows us to specify the query using the Lucene query syntax described in the Lucene query syntax section later in this chapter. For example, a simple query would look like this: q=title:elasticsearch.

The default search field
Using the df parameter, we can specify the default search field that should be used when no field indicator is used in the q parameter. By default, the _all field will be used. (This is the field that Elasticsearch uses to copy the content of all the other fields. We will discuss this in greater depth in Chapter 2, Indexing Your Data). An example of the df parameter value can be df=title.

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

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

Query explanation
If we set the explain parameter to true, Elasticsearch will include additional explain information with each document in the result—such as the shard from which the document was fetched and the detailed information about the scoring calculation (we will talk more about it in the Understanding the explain information section in Chapter 6, Make Your Search Better). Also remember not to fetch the explain information during normal search queries because it requires additional resources and adds performance degradation to the queries. For example, a query that includes explain information could look as follows:

```
curl -XGET 'localhost:9200/books/_search?pretty&explain=true&q=title:solr'
```
The results returned by Elasticsearch for the preceding query would be as follows:

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 2,
    "max_score": 0.70273256,
    "hits": [ {
      "_shard": 2,
      "_node": "v5iRsht9SOWVzu-GY-YH1A",
      "_index": "books",
      "_type": "solr",
      "_id": "2",
      "_score": 0.70273256,
      "_source": { 
        "title": "Solr Cookbook Third Edition",
        "published": 2015
      },
      "_explanation": { 
        "value": 0.70273256,
        "description": "weight(title:solr in 0) [PerFieldSimilarity], result of:",
        "details": [ { 
          "value": 0.70273256,
          "description": "fieldWeight in 0, product of:",
          "details": [ { 
            "value": 1.0,
            "description": "tf(freq=1.0), with freq of:",
            "details": [ { 
              "value": 1.0,
              "description": "termFreq=1.0",
              "details": [ ]
            } ]
          } ]
        }, { 
          "value": 1.4054651,
          "description": "idf(docFreq=1, maxDocs=3)",
          "details": [ ]
        } ]
      }
    },
    { 
      "value": 1.4054651,
      "description": "idf(docFreq=1, maxDocs=3)",
      "details": [ ]
    }
  }
}
```
"_shard" : 3,
"_node" : "v5iRsht9SOWVzu-GY-YHLA",
"_index" : "books",
"_type" : "solr",
"_id" : "1",
"_score" : 0.5,
"_source" : {
  "title" : "Apache Solr 4 Cookbook",
  "published" : 2012
},
"_explanation" : {
  "value" : 0.5,
  "description" : "weight(title: solr in 1)
  [PerFieldSimilarity], result of:",
  "details" : [ {
    "value" : 0.5,
    "description" : "fieldWeight in 1, product of:",
    "details" : [ {
      "value" : 1.0,
      "description" : "tf(freq = 1.0), with freq of:",
      "details" : [ {
        "value" : 1.0,
        "description" : "termFreq = 1.0",
        "details" : [ ]
      }]
    }, { 
      "value" : 1.0,
      "description" : "idf(docFreq = 1, maxDocs = 2)",
      "details" : [ ]
    }, { 
      "value" : 0.5,
      "description" : "fieldNorm(doc = 1)",
      "details" : [ ]
    }]
  }]
}]}
The fields returned
By default, for each document returned, Elasticsearch will include the index name, the type name, the document identifier, score, and the _source field. We can modify this behavior by adding the fields parameter and specifying a comma-separated list of field names. The field will be retrieved from the stored fields (if they exist; we will discuss them in Chapter 2, Indexing Your Data) or from the internal _source field. By default, the value of the fields parameter is _source. An example is: fields=title,priority.

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

Sorting the results
Using the sort parameter, we can specify custom sorting. The default behavior of Elasticsearch is to sort the returned documents in descending order of the value of the _score field. If we want to sort our documents differently, we need to specify the sort parameter. For example, adding sort=published:desc will sort the documents in descending order of published field. By adding the sort=published:asc parameter, we will tell Elasticsearch to sort the documents on the basis of the published field in ascending order.

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

The search timeout
By default, Elasticsearch doesn't have timeout for queries, but you may want your queries to timeout after a certain amount of time (for example, 5 seconds). Elasticsearch allows you to do this by exposing the timeout parameter. When the timeout parameter is specified, the query will be executed up to a given timeout value and the results that were gathered up to that point will be returned. To specify a timeout of 5 seconds, you will have to add the timeout=5s parameter to your query.
The results window
Elasticsearch allows you to specify the results window (the range of documents in the results list that should be returned). We have two parameters that allow us to specify the results window: `size` and `from`. The `size` parameter defaults to 10 and defines the maximum number of results returned. The `from` parameter defaults to 0 and specifies from which document the results should be returned. In order to return five documents starting from the 11th one, we will add the following parameters to the query: `size=5&from=10`.

Limiting per-shard results
Elasticsearch allows us to specify the maximum number of documents that should be fetched from each shard using `terminate_after` property and specifying the maximum number of documents. For example, if we want to get no more than 100 documents from each shard, we can add `terminate_after=100` to our URI request.

Ignoring unavailable indices
When running queries against multiple indices, it is handy to tell Elasticsearch that we don't care about the indices that are not available. By default, Elasticsearch will throw an error if one of the indices is not available, but we can change this by simply adding the `ignore_unavailable=true` parameter to our URI request.

The search type
The URI query allows us to specify the search type using the `search_type` parameter, which defaults to `query_then_fetch`. Two values that we can use here are: `dfs_query_then_fetch` and `query_then_fetch`. The rest of the search types available in older Elasticsearch versions are now deprecated or removed. We'll learn more about search types in the Understanding the querying process section of Chapter 3, Searching Your Data.

Lowercasing term expansion
Some queries, such as the prefix query, use query expansion. We will discuss this in the Query rewrite section in Chapter 4, Extending Your Querying Knowledge. We are allowed to define whether the expanded terms should be lowercased or not using the `lowercase_expanded_terms` property. By default, the `lowercase_expanded_terms` property is set to `true`, which means that the expanded terms will be lowercased.
Chapter 1

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

If you want to see all the parameters exposed by Elasticsearch as the URI request parameters, please refer to the official documentation available at: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-uri-request.html.

Lucene query syntax
We thought that it would be good to know a bit more about what syntax can be used in the q parameter passed in the URI query. Some of the queries in Elasticsearch (such as the one currently being discussed) support the Lucene query parser syntax—the language that allows you to construct queries. Let's take a look at it and discuss some basic features.

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

```
title:book
```

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

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

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

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

```
+title:book -description:cat
```
We can also group multiple terms with parentheses, as shown in the following query:

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

We can also boost parts of the query (this increases their importance for the scoring algorithm — the higher the boost, the more important the query part is) with the `^` operator and the boost value after it, as shown in the following query:

```
title:book^4
```

These are the basics of the Lucene query language and should allow you to use Elasticsearch and construct queries without any problems. However, if you are interested in the Lucene query syntax and you would like to explore that in depth, please refer to the official documentation of the query parser available at [http://lucene.apache.org/core/5_4_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html](http://lucene.apache.org/core/5_4_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html).

**Summary**

In this chapter, we learned what full text search is and the contribution Apache Lucene makes to this. In addition to this, we are now familiar with the basic concepts of Elasticsearch and its top-level architecture. We used the Elasticsearch REST API not only to index data, but also to update, retrieve, and finally delete it. We've learned what versioning is and how we can use it for optimistic locking in Elasticsearch. Finally, we searched our data using the simple URI query.

In the next chapter, we'll focus on indexing our data. We will see how Elasticsearch indexing works and what the role of primary shards and replicas is. We'll see how Elasticsearch handles data that it doesn't know and how to create our own mappings — the JSON structure that describes the structure of our index. We'll also learn how to use batch indexing to speed up the indexing process and what additional information can be stored along with our index to help us achieve our goal. In addition, we will discuss what an index segment is, what segment merging is, and how to tune a segment. Finally, we'll see how routing works in Elasticsearch and what options we have when it comes to both indexing and querying routing.
Where to buy this book

You can buy Elasticsearch Server Third Edition 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.