Mastering Elasticsearch
Second Edition

Elasticsearch is a modern, fast, distributed, scalable, fault tolerant, and open source search and analytics engine. Elasticsearch leverages the capabilities of Apache Lucene, providing a new level of control over how you can index and search even huge sets of data.

This book covers intermediate and advanced functionalities of Elasticsearch and walks you through its internals including caches, the Apache Lucene library, and its monitoring capabilities. You’ll learn about practical usage of Elasticsearch configuration parameters and how to use the monitoring API.

With this book, you’ll delve into Elasticsearch’s query rewrite, query template, bulk operation, document grouping, and function score queries. You will also learn how to improve user search experience, index distribution, segment statistics, and merging. By the end of the book, you will be able to enhance Elasticsearch’s performance and create your own Elasticsearch plugins.

Who this book is written for
This book is for Elasticsearch users who want to extend their knowledge and develop new skills. Prior knowledge of the Query DSL and data indexing is expected.

What you will learn from this book
- Understand Apache Lucene and Elasticsearch’s design and architecture
- Use and configure different scoring models to alter the default scoring mechanism
- Choose the appropriate amount of shards and replicas for your deployment
- Improve user search experience by utilizing Elasticsearch functionality
- Control segment merging and learn why Elasticsearch uses merging
- Develop custom Elasticsearch plugins and cover detailed examples of how to extend Elasticsearch by writing your own plugins
- Apply your knowledge to create scalable, efficient, and fault tolerant clusters and monitor your cluster by using and understanding the Elasticsearch API

Rafał Kuć, Marek Rogoziński

Further your knowledge of the Elasticsearch server by learning more about its internals, querying, and data handling.

In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 'Introduction to Elasticsearch'
- A synopsis of the book’s content
- More information on Mastering Elasticsearch Second Edition

About the Authors

Rafał Kuć is a born team leader and software developer. Currently, he is working as a consultant and a software engineer at Sematext Group, Inc., where he concentrates on open source technologies, such as Apache Lucene, Solr, Elasticsearch, and the Hadoop stack. He has more than 13 years of experience in various software branches—from banking software to e-commerce products. He is mainly focused on Java but is open to every tool and programming language that will make the achievement of his goal easier and faster. Rafał is also one of the founders of the solr.pl website, where he tries to share his knowledge and help people with their problems related to Solr and Lucene. He is also a speaker at various conferences around the world, such as Lucene Eurocon, Berlin Buzzwords, ApacheCon, Lucene Revolution, and DevOps Days.

He began his journey with Lucene in 2002, but 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 came Solr, and that was it. He started working with Elasticsearch in the middle of 2010. Currently, Lucene, Solr, Elasticsearch, and information retrieval are his main points of interest.

Acknowledgments

With Marek, we were thinking about writing an update to Mastering Elasticsearch, Packt Publishing. It was not a book for everyone, but the first version didn't put enough emphasis on that—we were treating Mastering Elasticsearch as an update to Elasticsearch Server. The same goes with Mastering Elasticsearch Second Edition. The book you are holding in your hands was written as an extension to Elasticsearch Server Second Edition, Packt Publishing, and should be treated as a continuation to that book. Because of such an approach, we could concentrate on topics such as choosing the right queries, scaling Elasticsearch, extensive scoring descriptions with examples, internals of filtering, new aggregations, comparison to documents' relations handling, and so on. Hopefully, after reading this book, you'll be able to easily get all the details about Elasticsearch and the underlying Apache Lucene architecture; this will let you get the desired knowledge easier and faster.

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

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

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

Once again, thank you.

Marek Rogoźniński is a software architect and consultant with over 10 years of experience. He specializes in solutions based on open source search engines, such as Solr and Elasticsearch, and software stack for Big Data analytics, including Hadoop, Hbase, and Twitter Storm.

He is also a cofounder of the solr.pl website, which publishes information and tutorials about Solr and Lucene libraries. He is the coauthor of Mastering ElasticSearch, ElasticSearch Server, and Elasticsearch Server Second Edition, both published by Packt Publishing.

Currently, he holds the position of chief technology officer and lead architect at ZenCard, a company processing and analyzing large amounts of payment transactions in real time, allowing automatic and anonymous identification of retail customers on all retailer channels (m-commerce / e-commerce / brick and mortar) and giving retailers a customer retention and loyalty tool.
Acknowledgments

This is our fourth book about Elasticsearch and, again, I am fascinated by how quickly Elasticsearch is evolving. We always have to find the balance between describing features marked as experimental or work in progress, and we have to take the risk that the final code might behave differently or even ignore some of the interesting features. The second edition of this book has quite a large number of rewrites and covers some new features; however, this comes at the cost of the removal of some information that was less useful for readers. With this book, we've tried to introduce some additional topics connected to Elasticsearch. However, the whole ecosystem and the ELK stack (Elasticsearch, Logstash, and Kibana) or Hadoop integration deserves a dedicated book.

Now, it is time to say thank you.

Thanks to all the people who created Elasticsearch, Lucene, and all the libraries and modules published around these projects or used by these projects.

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

Last but not least, thanks to all the friends who stood by me and understood my constant lack of time.
Welcome to the world of Elasticsearch and *Mastering Elasticsearch Second Edition*. While reading the book, you'll be taken through different topics—all connected to Elasticsearch. Please remember though that this book is not meant for beginners and we really treat the book as a follow-up or second part of *Elasticsearch Server Second Edition*. There is a lot of new content in the book and, sometimes, you can refer to the content of *Elasticsearch Server Second Edition* within this book.

Throughout the book, we will discuss different topics related to Elasticsearch and Lucene. We start with an introduction to the world of Lucene and Elasticsearch to introduce you to the world of queries provided by Elasticsearch, where we discuss different topics related to queries, such as filtering and which query to choose in a particular situation. Of course, querying is not all and, because of that, the book you are holding in your hands provides information on newly introduced aggregations and features that will help you give meaning to the data you have indexed in Elasticsearch indices, and provide a better search experience for your users.

Even though, for most users, querying and data analysis are the most interesting parts of Elasticsearch, they are not all that we need to discuss. Because of this, the book tries to bring you additional information when it comes to index architecture such as choosing the right number of shards and replicas, adjusting the shard allocation behavior, and so on. We will also get into the places where Elasticsearch meets Lucene, and we will discuss topics such as different scoring algorithms, choosing the right store mechanism, what the differences between them are, and why choosing the proper one matters.

Last, but not least, we touch on the administration part of Elasticsearch by discussing discovery and recovery modules, and the human-friendly Cat API, which allows us to very quickly get relevant administrative information in a form that most humans should be able to read without parsing JSON responses. We also talk about and use tribe nodes, giving us possibilities of creating federated searches across many nodes.

Because of the title of the book, we couldn't omit performance-related topics, and we decided to dedicate a whole chapter to it. We talk about doc values and the improvements they bring, how garbage collector works, and what to do when it does not work as we expect. Finally, we talk about Elasticsearch scaling and how to prepare it for high indexing and querying use cases.

Just as with the first edition of the book, we decided to end the book with the development of Elasticsearch plugins, showing you how to set up the Apache Maven project and develop two types of plugins—custom REST action and custom analysis.
If you think that you are interested in these topics after reading about them, we think this is a book for you and, hopefully, you will like the book after reading the last words of the summary in Chapter 9, Developing Elasticsearch Plugins.

**What This Book Covers**

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

*Chapter 2, Power User Query DSL*, describes how the Apache Lucene scoring works, why Elasticsearch rewrites queries, what query templates are, and how we can use them. In addition to that, it explains the usage of filters and which query should be used in a particular use case.

*Chapter 3, Not Only Full Text Search*, describes queries rescoring, multimatching control, and different types of aggregations that will help you with data analysis—significant terms aggregation and top terms aggregation that allow us to group documents with a certain criteria. In addition to that, it discusses relationship handling in Elasticsearch and extends your knowledge about scripting in Elasticsearch.

*Chapter 4, Improving the User Search Experience*, covers user search experience improvements. It introduces you to the world of Suggesters, which allows you to correct user query spelling mistakes and build efficient autocomplete mechanisms. In addition to that, you'll see how to improve query relevance by using different queries and the Elasticsearch functionality with a real-life example.

*Chapter 5, The Index Distribution Architecture*, covers techniques for choosing the right amount of shards and replicas, how routing works, how shard allocation works, and how to alter its behavior. In addition to that, we discuss what query execution preference is and how it allows us to choose where the queries are going to be executed.

*Chapter 6, Low-level Index Control*, describes how to alter the Apache Lucene scoring and how to choose an alternative scoring algorithm. It also covers NRT searching and indexing and transaction log usage, and allows you to understand segment merging and tune it for your use case. At the end of the chapter, you will also find information about Elasticsearch caching and request breakers aiming to prevent out-of-memory situations.

*Chapter 7, Elasticsearch Administration*, describes what the discovery, gateway, and recovery modules are, how to configure them, and why you should bother. We also describe what the Cat API is, how to back up and restore your data to different cloud services (such as Amazon AWS or Microsoft Azure), and how to use tribe nodes—Elasticsearch federated search.
Chapter 8, *Improving Performance*, covers Elasticsearch performance-related topics ranging from using doc values to help with field data cache memory usage through the JVM garbage collector work, and queries benchmarking to scaling Elasticsearch and preparing it for high indexing and querying scenarios.

Chapter 9, *Developing Elasticsearch Plugins*, covers Elasticsearch plugins' development by showing and describing in depth how to write your own REST action and language analysis plugin.
Before going further into the book, we would like to emphasize that we are treating this book as an extension to the *Elasticsearch Server Second Edition* book we’ve written, also published by Packt Publishing. Of course, we start with a brief introduction to both Apache Lucene and Elasticsearch, but this book is not for a person who doesn’t know Elasticsearch at all. We treat *Mastering Elasticsearch* as a book that will systematize your knowledge about Elasticsearch and extend it by showing some examples of how to leverage your knowledge in certain situations. If you are looking for a book that will help you start your journey into the world of Elasticsearch, please take a look at *Elasticsearch Server Second Edition* mentioned previously.

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

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

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

Getting familiar with Lucene

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

Overall architecture

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

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

Apache Lucene writes all the information to the structure called **inverted** index. It is a data structure that maps the terms in the index to the documents, not the other way round like the relational database does. 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 can look. For example, let's assume that we have the documents with only title field to be indexed and they look like the following:

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

So, the index (in a very simple way) could be visualized as shown in the following figure:

<table>
<thead>
<tr>
<th>Term</th>
<th>Count</th>
<th>Docs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>&lt;3&gt;</td>
</tr>
<tr>
<td>Apache</td>
<td>1</td>
<td>&lt;3&gt;</td>
</tr>
<tr>
<td>Cooking</td>
<td>1</td>
<td>&lt;3&gt;</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>2</td>
<td>&lt;2&gt;</td>
</tr>
<tr>
<td>Mastering</td>
<td>1</td>
<td>&lt;1&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>
</tbody>
</table>

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

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

However, multiple segments can be merged together in a process called segments merge. After forcing, segments are merged, or after Lucene decides it is time for merging to be performed, segments are merged together by Lucene to create larger ones. This can be I/O demanding; however, it is needed to clean up some information because during that time some information that is not needed anymore is deleted, for example the deleted documents. In addition to this, searching with the use of one larger segment is faster than searching against multiple smaller ones holding the same data. However, once again, remember that segments merging is an I/O demanding operation and you shouldn't force merging, just configure your merge policy carefully.
If you want to know what files are building the segments and what information is stored inside them, please take a look at Apache Lucene documentation available at http://lucene.apache.org/core/4_10_3/core/org/apache/lucene/codecs/lucene410/package-summary.html.

Getting deeper into Lucene index
Of course, the actual index created by Lucene is much more complicated and advanced, and consists of more than the terms their counts and documents in which they are present. We would like to tell you about a few of those additional index pieces because even though they are internal, it is usually good to know about them as they can be very handy.

Norms
A norm is a factor associated with each indexed document and stores normalization factors used to compute the score relative to the query. Norms are computed on the basis of index time boosts and are indexed along with the documents. With the use of norms, Lucene is able to provide an index time-boosting functionality at the cost of a certain amount of additional space needed for norms indexation and some amount of additional memory.

Term vectors
Term vectors are small inverted indices per document. They consist of pairs—a term and its frequency—and can optionally include information about term position. By default, Lucene and Elasticsearch don't enable term vectors indexing, but some functionality such as the fast vector highlighting requires them to be present.

Posting formats
With the release of Lucene 4.0, the library introduced the so-called codec architecture, giving developers control over how the index files are written onto the disk. One of the parts of the index is the posting format, which stores fields, terms, documents, terms positions and offsets, and, finally, the payloads (a byte array stored at an arbitrary position in Lucene index, which can contain any information we want). Lucene contains different posting formats for different purposes, for example one that is optimized for high cardinality fields like the unique identifier.
Doc values
As we already mentioned, Lucene index is the so-called inverted index. However, for certain features, such as faceting or aggregations, such architecture is not the best one. The mentioned functionality operates on the document level and not the term level and because Elasticsearch needs to uninvert the index before calculations can be done. Because of that, doc values were introduced and additional structure used for faceting, sorting and aggregations. The doc values store uninverted data for a field they are turned on for. Both Lucene and Elasticsearch allow us to configure the implementation used to store them, giving us the possibility of memory-based doc values, disk-based doc values, and a combination of the two.

Analyzing your data
Of course, the question arises of how the data passed in the documents is transformed into the inverted index and how the query text is changed into terms to allow searching. The process of transforming this data is called analysis.

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

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

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

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

Filters are processed one after another, so we have almost unlimited analysis possibilities with adding multiple filters one after another.
The last thing is the character mappings, which is used before tokenizer and is responsible for processing text before any analysis is done. One of the examples of character mapper is the HTML tags removal process.

**Indexing and querying**

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

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

What you should remember about indexing and querying analysis is that the index should be matched by the query term. If they don't match, Lucene won't return the desired documents. For example, if you are using stemming and lowercasing during indexing, you need to be sure that the terms in the query are also lowercased and stemmed, or your queries will return no results at all.

**Lucene query language**

Some of the query types provided by Elasticsearch support Apache Lucene query parser syntax. Because of this, it is crucial to understand the Lucene query language.

**Understanding the basics**

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

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

- **AND**: It means that the given two terms (left and right operand) need to match in order for the clause to be matched. For example, we would run a query, such as apache AND lucene, to match documents with both apache and lucene terms in a document field.
• **OR:** It means that any of the given terms may match in order for the clause to be matched. For example, we would run a query, such as apache OR lucene, to match documents with apache or lucene (or both) terms in a document field.

• **NOT:** It means that in order for the document to be considered a match, the term appearing after the **NOT** operator must not match. For example, we would run a query lucene **NOT** Elasticsearch to match documents that contain lucene term, but not the Elasticsearch term in the document field.

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

• **+** : It means that the given term needs to be matched in order for the document to be considered as a match. For example, in order to find documents that match lucene term and may match apache term, we would run a query such as +lucene apache.

• **-** : It means that the given term can’t be matched in order for the document to be considered a match. For example, in order to find a document with lucene term, but not Elasticsearch term, we would run a query such as +lucene -Elasticsearch.

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

In addition to all these, there is one more thing: you can use parenthesis to group clauses together for example, with something like the following query:

```
Elasticsearch AND (mastering OR book)
```

**Querying fields**

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

```
title:Elasticsearch
```

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

```
title:(+Elasticsearch +"mastering book")
```
The previous query can also be expressed in the following way:

```
+title:Elasticsearch +title:"mastering book"
```

**Term modifiers**

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

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

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

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

```
title:"mastering Elasticsearch"
```

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

We can also use boosting to increase our term importance by using the `^` character and providing a float number. Boosts lower than one would result in decreasing the document importance. Boosts higher than one will result in increasing the importance. The default boost value is 1. Please refer to the *Default Apache Lucene scoring explained* section in Chapter 2, *Power User Query DSL*, for further information on what boosting is and how it is taken into consideration during document scoring.
In addition to all these, we can use square and curly brackets to allow range searching. For example, if we would like to run a range search on a numeric field, we could run the following query:

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

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

In case of string-based fields, we also can run a range query, for example name: [Adam TO Adria].

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

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

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

If you would like your range bound from one side and not bound by the other, for example querying for documents with a price higher than 10.00, we would run the following query:

```
price: [10.00 TO *]
```

**Handling special characters**

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

**Introducing Elasticsearch**

Although we've said that we expect the reader to be familiar with Elasticsearch, we would really like you to fully understand Elasticsearch; therefore, we've decided to include a short introduction to the concepts of this great search engine.

As you probably know, Elasticsearch is production-ready software to build search and analysis-oriented applications. It was originally started by Shay Banon and published in February 2010. Since then, it has rapidly gained popularity just within a few years and has become an important alternative to other open source and commercial solutions. It is one of the most downloaded open source projects.
Basic concepts

There are a few concepts that come with Elasticsearch and their understanding is crucial to fully understand how Elasticsearch works and operates.

Index

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

Document

Document is the main entity in the Elasticsearch world (and also in the Lucene world). At the end, all use cases of using Elasticsearch can be brought at a point where it is all about searching for documents and analyzing them. Document consists of fields, and each field is identified by its name and can contain one or multiple values. Each document may have a different set of fields; there is no schema or imposed structure—this is because Elasticsearch documents are in fact Lucene ones. From the client point of view, Elasticsearch document is a JSON object (see more on the JSON format at http://en.wikipedia.org/wiki/JSON).

Type

Each document in Elasticsearch has its type defined. This allows us to store various document types in one index and have different mappings for different document types. If you would like to compare it to an SQL world, a type in Elasticsearch is something similar to a database table.

Mapping

As already mentioned in the Introducing Apache Lucene section, all documents are analyzed before being indexed. We can configure how the input text is divided into tokens, which tokens should be filtered out, or what additional processing, such as removing HTML tags, is needed. This is where mapping comes into play—it holds all the information about the analysis chain. Besides the fact that Elasticsearch can automatically discover field type by looking at its value, in most cases we will want to configure the mappings ourselves to avoid unpleasant surprises.
Node

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

Elasticsearch nodes can serve different purposes. Of course, Elasticsearch is designed to index and search our data, so the first type of node is the data node. Such nodes hold the data and search on them. The second type of node is the master node—a node that works as a supervisor of the cluster controlling other nodes' work. The third node type is the tribe node, which is new and was introduced in Elasticsearch 1.0. The tribe node can join multiple clusters and thus act as a bridge between them, allowing us to execute almost all Elasticsearch functionalities on multiple clusters just like we would be using a single cluster.

Cluster

Cluster is a set of Elasticsearch nodes that work together. The distributed nature of Elasticsearch allows us to easily handle data that is too large for a single node to handle (both in terms of handling queries and documents). By using multi-node clusters, we can also achieve uninterrupted work of our application, even if several machines (nodes) are not available due to outage or administration tasks such as upgrade. Elasticsearch provides clustering almost seamlessly. In our opinion, this is one of the major advantages over competition; setting up a cluster in the Elasticsearch world is really easy.

Shard

As we said previously, clustering allows us to store information volumes that exceed abilities of a single server (but it is not the only need for clustering). To achieve this requirement, Elasticsearch spreads data to several physical Lucene indices. Those Lucene indices are called shards, and the process of dividing the index is called sharding. Elasticsearch can do this automatically and all the parts of the index (shards) are visible to the user as one big index. Note that besides this automation, it is crucial to tune this mechanism for particular use cases because the number of shard index is built or configured during index creation and cannot be changed without creating a new index and re-indexing the whole data.
Replica

Sharding allows us to push more data into Elasticsearch that is possible for a single node to handle. Replicas can help us in situations where the load increases and a single node is not able to handle all the requests. The idea is simple—create an additional copy of a shard, which can be used for queries just as original, primary shard. Note that we get safety for free. If the server with the primary shard is gone, Elasticsearch will take one of the available replicas of that shard and promote it to the leader, so the service work is not interrupted. Replicas can be added and removed at any time, so you can adjust their numbers when needed. Of course, the content of the replica is updated in real time and is done automatically by Elasticsearch.

Key concepts behind Elasticsearch architecture

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

- Reasonable default values that allow users to start using Elasticsearch just after installing it, without any additional tuning. This includes built-in discovery (for example, field types) and auto-configuration.
- Working in distributed mode by default. Nodes assume that they are or will be a part of the cluster.
- Peer-to-peer architecture without single point of failure (SPOF). Nodes automatically connect to other machines in the cluster for data interchange and mutual monitoring. This covers automatic replication of shards.
- Easily scalable both in terms of capacity and the amount of data by adding new nodes to the cluster.
- Elasticsearch does not impose restrictions on data organization in the index. This allows users to adjust to the existing data model. As we noted in type description, Elasticsearch supports multiple data types in a single index, and adjustment to the business model includes handling relationships between documents (although, this functionality is rather limited).
- Near Real Time (NRT) searching and versioning. Because of the distributed nature of Elasticsearch, it is impossible to avoid delays and temporary differences between data located on the different nodes. Elasticsearch tries to reduce these issues and provide additional mechanisms as versioning.
Workings of Elasticsearch

The following section will include information on key Elasticsearch features, such as bootstrap, failure detection, data indexing, querying, and so on.

The startup process

When Elasticsearch node starts, it uses the discovery module to find the other nodes in the same cluster (the key here is the cluster name defined in the configuration) and connect to them. By default the multicast request is broadcast to the network to find other Elasticsearch nodes with the same cluster name. You can see the process illustrated in the following figure:

![Diagram of Elasticsearch startup process]

In the preceding figure, the cluster, one of the nodes that is master eligible is elected as master node (by default all nodes are master eligible). This node is responsible for managing the cluster state and the process of assigning shards to nodes in reaction to changes in cluster topology.

Note that a master node in Elasticsearch has no importance from the user perspective, which is different from other systems available (such as the databases). In practice, you do not need to know which node is a master node; all operations can be sent to any node, and internally Elasticsearch will do all the magic. If necessary, any node can send sub-queries in parallel to other nodes and merge responses to return the full response to the user. All of this is done without accessing the master node (nodes operates in peer-to-peer architecture).
The master node reads the cluster state and, if necessary, goes into the recovery process. During this state, it checks which shards are available and decides which shards will be the primary shards. After this, the whole cluster enters into a yellow state.

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

**Failure detection**

During normal cluster work, the master node monitors all the available nodes and checks whether they are working. If any of them are not available for the configured amount of time, the node is treated as broken and the process of handling failure starts. For example, this may mean rebalancing of shards, choosing new leaders, and so on. As another example, for every primary shard that is present on the failed nodes, a new primary shard should be elected from the remaining replicas of this shard. The whole process of placing new shards and replicas can (and usually should) be configured to match our needs. More information about it can be found in Chapter 7, *Elasticsearch Administration*.

Just to illustrate how it works, let's take an example of a three nodes cluster. One of the nodes is the master node, and all of the nodes can hold data. The master node will send the ping requests to other nodes and wait for the response. If the response doesn't come (actually how many ping requests may fail depends on the configuration), such a node will be removed from the cluster. The same goes in the opposite way—each node will ping the master node to see whether it is working.
Communicating with Elasticsearch

We talked about how Elasticsearch is built, but, after all, the most important part for us is how to feed it with data and how to build queries. In order to do that, Elasticsearch exposes a sophisticated Application Program Interface (API). In general, it wouldn't be a surprise if we would say that every feature of Elasticsearch has an API. The primary API is REST based (see http://en.wikipedia.org/wiki/Representational_state_transfer) and is easy to integrate with practically any system that can send HTTP requests.

Elasticsearch assumes that data is sent in the URL or in the request body as a JSON document (see http://en.wikipedia.org/wiki/JSON). If you use Java or language based on Java Virtual Machine (JVM), you should look at the Java API, which, in addition to everything that is offered by the REST API, has built-in cluster discovery. It is worth mentioning that the Java API is also internally used by Elasticsearch itself to do all the node-to-node communication. Because of this, the Java API exposes all the features available through the REST API calls.
Indexing data

There are a few ways to send data to Elasticsearch. The easiest way is using the index API, which allows sending a single document to a particular index. For example, by using the `curl` tool (see [http://curl.haxx.se/](http://curl.haxx.se/)). An example command that would create a new document would look as follows:

```
```

The second way allows us to send many documents using the bulk API and the UDP bulk API. The difference between these methods is the connection type. Common bulk command sends documents by HTTP protocol and UDP bulk sends this using connection less datagram protocol. This is faster but not so reliable. The last method uses plugins, called rivers, but let's not discuss them as the rivers will be removed in future versions of Elasticsearch.

One very important thing to remember is that the indexing will always be first executed at the primary shard, not on the replica. If the indexing request is sent to a node that doesn't have the correct shard or contains a replica, it will be forwarded to the primary shard. Then, the leader will send the indexing request to all the replicas, wait for their acknowledgement (this can be controlled), and finalize the indexation if the requirements were met (like the replica quorum being updated).

The following illustration shows the process we just discussed:
Querying data

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

- Use various query types including simple term query, phrase, range, Boolean, fuzzy, span, wildcard, spatial, and function queries for human readable scoring control
- Build complex queries by combining the simple queries together
- Filter documents, throwing away ones that do not match selected criteria without influencing the scoring, which is very efficient when it comes to performance
- Find documents similar to a given document
- Find suggestions and corrections of a given phrase
- Build dynamic navigation and calculate statistics using aggregations
- Use prospective search and find queries matching a given document

When talking about querying, the important thing is that query is not a simple, single-stage process. In general, the process can be divided into two phases: the scatter phase and the gather phase. The scatter phase is about querying all the relevant shards of your index. The gather phase is about gathering the results from the relevant shards, combining them, sorting, processing, and returning to the client. The following illustration shows that process:
You can control the scatter and gather phases by specifying the search type to one of the six values currently exposed by Elasticsearch. We've talked about query scope in our previous book *Elasticsearch Server Second Edition* by Packt Publishing.

**The story**

As we said in the beginning of this chapter, we treat the book you are holding in your hands as a continuation of the *Elasticsearch Server Second Edition* book. Because of this, we would like to continue the story that we've used in that book. In general, we assume that we are implementing and running an online book store, as simple as that.

The mappings for our library index look like the following:

```json
{
   "book" : {
      "_index" : {
         "enabled" : true
      },
      "_id" : {
         "index" : "not_analyzed",
         "store" : "yes"
      },
      "properties" : {
         "author" : {
            "type" : "string"
         },
         "characters" : {
            "type" : "string"
         },
         "copies" : {
            "type" : "long",
            "ignore_malformed" : false
         },
         "otitle" : {
            "type" : "string"
         },
         "tags" : {
            "type" : "string",
            "index" : "not_analyzed"
         },
         "title" : {
            "type" : "string"
         }
   }
}
```
The mappings can be found in the library.json file provided with the book.

The data that we will use is provided with the book in the books.json file. The example documents from that file look like the following:

```
{ "index": {"_index": "library", "_type": "book", "_id": "1"} }
{ "title": "All Quiet on the Western Front","otitle": "Im Westen nichts Neues","author": "Erich Maria Remarque","year": 1929,"characters": ["Paul Bäumer", "Albert Kropp", "Haie Westhus", "Fredrich Müller", "Stanislaus Katczinsky", "Tjaden"],"tags": ["novel"],"copies": 1, "available": true,  
  "section" : 3 }
{ "index": {"_index": "library", "_type": "book", "_id": "2"} }
{ "title": "Catch-22","author": "Joseph Heller","year": 1961,"characters": 
```

Introduction to Elasticsearch

[John Yossarian, Captain Aardvark, Chaplain Tappman, Colonel Cathcart, Doctor Daneeka], "tags": ["novel"], "copies": 6, "available": false, "section": 1

{"index": {"_index": "library", "_type": "book", "_id": "3"}}
{"title": "The Complete Sherlock Holmes", "author": "Arthur Conan Doyle", "year": 1936, "characters": ["Sherlock Holmes", "Dr. Watson", "G. Lestrade"], "tags": [], "copies": 0, "available": false, "section": 12}

{"index": {"_index": "library", "_type": "book", "_id": "4"}}
{"title": "Crime and Punishment", "otitle": "Преступление и наказание", "author": "Fyodor Dostoevsky", "year": 1886, "characters": ["Raskolnikov", "Sofia Semyonovna Marmeladova"], "tags": [], "copies": 0, "available": true}

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

To create the index using the provided mappings and to index the data, we would run the following commands:
curl -XPOST 'localhost:9200/library'
curl -XPUT 'localhost:9200/library/book/_mapping' -d @library.json
curl -s -XPOST 'localhost:9200/_bulk' --data-binary @books.json

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

In the next chapter, you'll learn about the default scoring formula Apache Lucene uses, what the query rewrite process is, and how it works. In addition to that, we'll discuss some of the Elasticsearch functionality, such as query templates, filters, and how they affect performance, what we can do with that, and how we can choose the right query to get the job done.
Where to buy this book
Click here for ordering and shipping details.