

Elasticsearch IN ACTION

SECOND EDITION

Madhusudhan Konda



MANNING



MEAP Edition
Manning Early Access Program
Elasticsearch in Action
Second Edition
Version 11

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

I am absolutely delighted and thankful for your purchase of this MEAP of *Elasticsearch in Action, Second Edition*.

Search in the new normal, the current digital world, is omnipresent. Elasticsearch is a multi-faceted popular search engine with a ton of advanced features and many bells and whistles. A humongous task was bestowed upon me to create a new *In Action* book on a complicated technology that is ever-changing at a rapid rate.

As with any new technology, mastery is associated with a sharp learning curve. Elasticsearch, an enterprise distributed search and analytics engine, is no exception. Its countless advanced features including multi-language analyzers, geospatial and time-series storage, anomaly detection using machine learning algorithms, graph analytics, auto-complete and fuzzy searching, root cause analysis using log stores, rich visualizations of data, complex aggregations, and many more, make Elasticsearch a must-have tool for most organizations and businesses.

Architecting, designing, and developing an enterprise search engine using Elasticsearch requires a solid understanding of the product. It requires a thorough deep dive into the search engine, architecture, its APIs, and its moving parts. You need someone to hold your hand, explain the concepts and fundamentals, walk you through the examples, and dissect the complex topics when learning a new technology—especially one that's a seemingly simple yet complex beast under the shiny surface like Elasticsearch. And this is exactly my aim!

This book is for anyone who is interested in learning Elasticsearch in a practical, tried-and-tested manner. It delves into high-level architecture, explains the dichotomy of the product, walks through the search and analytics capabilities using various APIs, provides the infrastructure patterns, code snippets at a low level, and much more. This book is a must-have for developers, DevOps, and architects working in the search space. My goal is to make this book easy to read with tons of hands-on examples.

To make the most of this book, please check out the accompanying working examples on GitHub (<https://github.com/madhusudhankonda/elasticsearch-in-action/wiki>). You can spin up a server and try out the examples while reading the explanations in detail. Each of the chapters on GitHub will also have some exercises to practice, too. Beginner and intermediate level engineers should find it a page-turner and try out the code snippets to solidify their understanding of the technology with ease and comfort.

Please be sure to post any questions, comments, or suggestions you have in the [liveBook discussion forum](#).

I sincerely hope you enjoy reading this book as much as I'm enjoying writing it!

—Madhusudhan Konda

brief contents

- 1 Overview*
- 2 Getting started*
- 3 Architecture*
- 4 Mapping*
- 5 Working with documents*
- 6 Indexing operations*
- 7 Text analysis*
- 8 Introducing search*
- 9 Term-level search*
- 10 Full-text search*
- 11 Compound queries*
- 12 Advanced search*
- 13 Aggregations*
- 14 Administration*

1

Overview

This chapter covers

- Setting the scene for modern search engines
- Introducing Elasticsearch, a search and analytics engine
- Looking at its functional and technical features
- Understanding Elasticsearch's core areas, use cases, and its prominent features
- Overview of Elastic Stack: Beats, Logstash, Elasticsearch, and Kibana

We expect Google to return our search results in a flash. Further, we want the results to be relevant, meaningful, and exactly what we are looking for. When I start searching in that simple-looking search bar, I expect Google to autocomplete my search, correct my spellings, and suggest some keywords. While we may take it for granted, the mechanics, science, and brains behind that tiny search bar are unfathomable. Most of us are pretty happy navigating to one of the top links on the first page. In fact, research suggests that 80% of people won't even bother to visit the second page. The beauty is that the results are so close that we don't give a second thought to suspect Google's results.

Search engines have benefited from years of talented brains working on sophisticated algorithms and powerful data structures. Advanced processors and other hardware has also made incredible strides. While the results can appear in a fraction of a second, a ton of complex actions are performed behind the scenes, including but not limited to applying complicated search and matching algorithms, abundant computing resources, ranking schemes, and whatnot! The good thing is that we, the end-users, do not need to be bothered with the mechanics of how these search engines work behind the scenes.

In this chapter, we will look at the search space in general and skim over the evolution of search from a traditional database-backed search to the current modern search engines and their many convenient features. Along the way, we will introduce ourselves to Elasticsearch, the ultra-fast open source search engine. We will look at its features, use cases, and customer adoption.

1.1 What makes a good search engine?

Recently, my family adopted a puppy, Milly (that's her, pictured!). As we are the first-time owners of a pup, we started searching for dog food online. I browsed through my preferred (popular) grocer, but to my disappointment, the search results (see figure 1.1) were not appealing. You can see "poop bags" in the list of results (ranked on the top) among other non-relevant products. There weren't any filters, no dropdowns, no price range knobs; it's just a plain page with the search results.



Sponsored	Sponsored	Sponsored	
Edgard & Cooper Hypoallergenic, Grain Free Dry Food for Adult Dogs with Venison ... ★★★★★ (2) £5.45 £7.79 / kg Add	Edgard & Cooper Grain Free Dry Food for Adult Dogs with Free-Run Chicken 0.7kg ★★★★★ (2) £5.45 £7.79 / kg Add	Cesar Classics Loaf Wet Adult 1+ Dog Food Tray Mixed 8x150g ★★★★★ (1) £5.60 £4.67 / kg Add	Sainsbury's Dog Poop Bags x50 ★★★★★ (15) £1.15 £1.15 / ea Add

Figure 1.1 A search for dog food showing irrelevant results from a supermarket.

Another grocer showed a pet harness and a baby's lunch box (see figure 1.2):

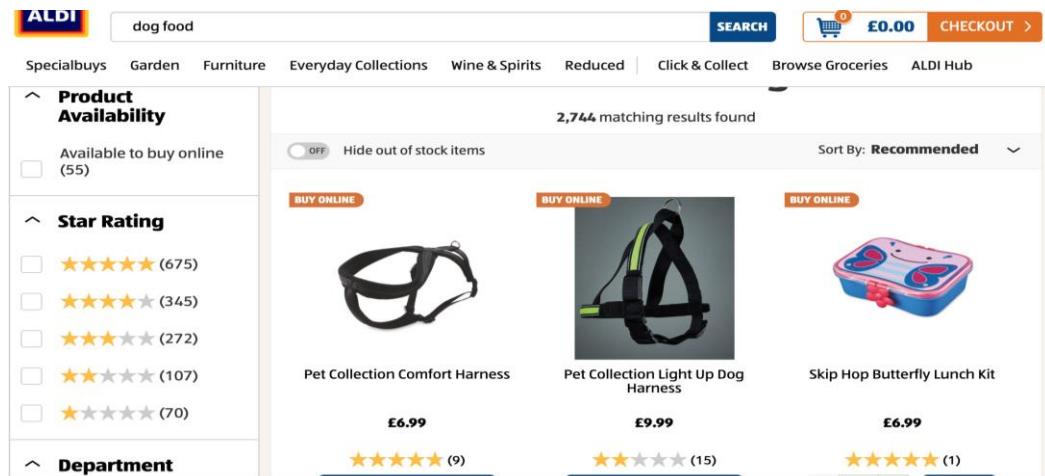


Figure 1.2 Another search for dog food and another supermarket showing absurd results.

The search engines behind these supermarkets didn't provide me suggestions while I typed, nor did they correct my input when I misspelled *dog food* as *dig food*. Some of the results weren't ranked relevant (though this can be forgiven as not all search engines are expected to produce relevant results). And one grocer brought back 2400 results!

Not pleased with the result from the grocer, I headed over to Amazon's online shop. The second I started typing *dog*, I could see the dropdown with all the possible suggestions (figure 1.3).

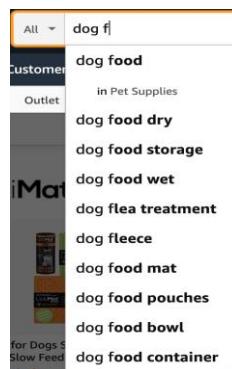


Figure 1.3: Amazon's search engine providing suggestions for my search

The initial search from Amazon brings back the relevant results, which are enabled by default using the *Sort by Featured* option. Conveniently, we can also change the sort order (low-to-high or vice versa, etc). Once the initial search is carried out, we can also drill down into other categories

by selecting the department, average customer review, price range, and so on. I even tried the wrong spelling too: *dig food*. Amazon asked me if I meant *dog food* as figure 1.4 shows.



Did you mean *dog food*

Figure 1.4: Did-you-mean feature from a modern search engine

In the current digital world, search is undoubtedly a first-class citizen, which organizations are adopting without a second thought. They understand the business value and the varied use cases that a search engine offers. In the next section, we will explore the exponential growth of search engines and how technology has enabled creating cutting edge search solutions.

1.2 Search is the new normal

With the exponential growth of data (terabytes to petabytes to exabytes), the need for searching for a tool that would enable successful searches in a needle-in-the-haystack situation is imperative. What was once touted as a simple search is now becoming a necessary functionality for most organization's survival tool chest. Organizations are expected to provide a search function by default so their customers can click in a search bar or navigate through a multitude of search drill downs to find what they need in a jiffy.

It is increasingly difficult to find websites and applications without the humble magnifying glass search bar in one form or another. Providing a full-fledged search consisting of all the advanced functionality is a competitive advantage. Basic searches used to be supported by plain old databases. Today, modern engines provide the advanced functionality required to be competitive. Quite often when dealing with search engines, you'd come across data and search variants: structured and unstructured data and respective searches. Let's look at them briefly in the next section.

1.2.1 Structured vs unstructured (full-text) search

Data predominantly comes in two flavours: structured and unstructured data. The structured data, for example like dates, numbers, booleans, is very organized, has a definite shape and fits the predefined data type patterns and hence is easily searchable. The unstructured data, like blog posts, social media, audio/video clips, articles and so on, has no appropriate model and cannot be easily searchable. The unstructured data is the bread and butter of most modern search engines.

Queries against structured data return results in exact matches. That is, we are concerned about finding the documents that match the search criteria - but not how well they are matched. This kind of search results are binary, that is, either you have a result or none. As we do not search how well the documents match but just if the documents match, no relevancy scores are attached to these results. The traditional database search is more of this sort. For example, fetch all the flights that were cancelled in the last month, weekly bestseller books, activity of a logged in user and so on.

On the other hand, full-text (unstructured) queries will try to find results that are relevant to the query. That is, Elasticsearch will find all the documents that are most suited for the query. For example, if a user searches for “vaccine”, keyword, the search engine will not only retrieve documents related to vaccination, it could also throw in relevant documents such as inoculations, jabs and any “vaccine” related. Elasticsearch employs a similarity algorithm to generate a relevance score for full-text queries. The score is a positive floating-point number attached to the results, with the highest scored document indicating more relevant to the query criteria.

1.2.2 Search supported by a database

A “good old” search was mostly based on traditional relational databases. Older search engines might have been based on layered architectures implemented in multi-tiered applications. The queries written in SQL using clauses like where and like probably provided the foundation for search. These solutions are not necessarily performant and efficient for searching through full text to provide modern search functionality.

While databases support full-text searching (queries against some free text like a blog post, movie review, research articles, etc), they may struggle to provide efficient search in a near-real-time on heavy loads. The distributed nature of the search engine like Elasticsearch provides an instant scalability feature that most databases may not have been designed for. A search engine developed with a backing database (with no database’s full-text search capabilities enabled) may not be able to provide the *relevant* search results for the queries, let alone cope with volumetric growth, and serve results in real time.

FULL-TEXT SEARCHING WITH DATABASES: Relational databases like Oracle and MySQL support full-text search functionality, albeit with fewer functionalities than a modern full-text search engine like Elasticsearch. These are fundamentally different in storing and retrieving data, so one must make sure to prototype their requirements before jumping either way. Usually if the schemas are not going to change or data loads are low and if you already have a database engine with full-text search capabilities, perhaps starting your first stride with full-text on database might make sense.

1.2.3 Full-text search engines

To satisfy full-text search requirements along with other advanced functions, search engines came into existence with an approach different from that of a relational database query. The data undergoes an analysis phase in a search engine before it gets stored for later retrieval. This upfront analysis helps answer queries with ease. Elasticsearch is one such modern full-text search engine.

Shay Banon, founder of Elastic, developed a search product called Compass (<http://www.compass-project.org>) in early 2000. It was based on an open source search engine library called Apache Lucene (<https://lucene.apache.org>). Lucene is Doug Cutting’s full-text search library written in Java. Because it’s a library, one needs to import and integrate it with an application using its APIs. Compass, and other search engines use Lucene to provide a generalized search engine service, so that you don’t have to integrate Lucene from scratch into your own application. Shay eventually decided to abandon Compass and focus on ElasticSearch, because it had more potential.

APACHE SOLR: Apache Solr is an open source search engine built on Apache Lucene in 2004. As a strong competitor to Elasticsearch, Solr has a thriving user community and I must say closer to opensource than Elasticsearch (Elastic moved from Apache to Elastic Licence and Server Side Public Licence (SSPL) in early 2021). Both Solr and Elasticsearch excel in full-text searching, however Elasticsearch may have an edge when it comes to analytics. While both these products compete in almost all functionality from head to toe, Solr has been a favourite for large static datasets working in a big data ecosystem. Obviously one has to run through prototypes and analysis to pick a product, the general trend is that most projects looking up for integrating with a search engine for the first time may look towards Elasticsearch due to its relatively no-hurdle startup time. One must carry a detailed comparison for the use cases they intend to use the search engine before adopting and embracing them.

The burst of big data coupled with technological innovations paved the way to modern search engines. Modern search engines are highly effective and better suited to searching and providing relevance. Regardless of whether the data is structured, semi-structured, or unstructured (more than three quarters of the data is assumed to be unstructured data), modern search engines can easily query it and return the results with relevance.

As search becomes the new normal, modern search engines are trying hard to entertain the ever-growing requirements by donning new hats each day. Cheap hardware combined with the explosion of data is leading to the emergence of these modern search beasts. Let's consider these present-day search engines in further detail in the next section.

OPENSEARCH: Elastic changed their licencing policy in 2021, which applies to Elasticsearch release versions 7.11 and above. The licencing has been moved from open source to a dual licence under an Elastic Licence and a Server Side Public Licence (SSPL). This licence allows the community to use the product for free as expected. Managed service providers, however, cannot provide the products as services anymore. (There is a spat between Elastic and AWS on this move with AWS creating a forked version of Elasticsearch - called Open Distro for Elasticsearch - offering it as a managed service.)

As Elastic moved from the open source licensing model to the SSPL model, a new product called OpenSearch (<https://opensearch.org>) was born in order to fill the gaping hole left by the new licensing agreement. The base code for OpenSearch was created from the open source Elasticsearch and Kibana version 7.10.2. The product's first GA version 1.0 was released in July 2021. Watch out for OpenSearch becoming a competitor to Elasticsearch in the search engine space.

1.3 Modern search engines

A good modern search engine should provide

- First-class support for full-text (unstructured) and structured data
- Type-ahead suggestions, auto-corrections, and did-you-mean recommendations
- Forgiveness for users' spelling mistakes
- Search capabilities on geolocations and GeoPoints
- Easy scalability, either up or down, based on the fluctuating demands
- Blazing performance, speedy indexing and search capabilities
- Architecture that's a high-availability and fault-tolerant distributed system
- Support for machine learning functionalities

In this section, we will briefly discuss the high-level features of a modern search engine.

1.3.1 Functional features

A modern search engine is not just fetching the matching results but producing accurate and perhaps relevant results. They should forgive users' spellings by enabling spell checks (fuzzy search), support type-ahead suggestions, provide auto-completions as well as derive and provide did-you-mean recommendations, and all this at runtime! Search engines must also respond precisely to geolocation-based queries.

1.3.2 Technical features

Setting the functional features aside, there is a long list of technical aspects that modern search engines endeavor to meet to make sure the experience of a user's search is smooth, effective, and performant. The foremost one is exhibiting extraordinary performance, both in consuming the data as well as retrieving the data as a result of a search.

To cope with an unexpected rise in demand for services, search engines should autoscale elastically. This could be as simple as adding an additional server to scale horizontally to the existing server farm. Additionally, the data must survive system crashes and be highly available, becoming fully distributed and resilient in nature. Search engines should build redundancy of data into the equation by default. Furthermore, replicating and backing up data should be an automated task without any (or minimal) manual interventions.

NOTE HORIZONTAL VS VERTICAL SCALING There are two schools when it comes to scaling systems - horizontal and vertical scaling - also called scaling out and scaling up respectively. In horizontal scaling, we add additional machines to a cluster of existing servers to scale out so the data is re-balanced and distributed by the system's underlying machinery. On the other hand, in a vertical scaling scenario, we beef up the existing servers by adding more memory and CPUs to scale up the servers. Each of these methods have merits and demerits, we will cover them in the context of Elasticsearch in a dedicated chapter on Administration later in the book.

1.3.3 Search is for everyone

Search is not just for the end customers. Internal applications, support IT staff, analysts, product owners, managers, and many others can use search and analytical tools to explore their data

further. A lot of organizations amass tons of data over time and this data is useless if it's not analyzed for gathering intelligence from it.

1.3.4 Analytics

While we are talking about searches, there is another side of the coin—the *analytics*. Searching is all about finding the minuscule from an universe of data. Analytics on the other hand buckets the data into various groups, visualizing and drawing the intelligence out of it and finding statistical observations about your data. Analytical functions might answer questions such as the peak load on our system, the average age of users interested in buying our last-minute deals, top news articles, trending topics and so on. Most modern search engines are equipped with this functionality out of the box, albeit at various levels. Elasticsearch, as we learn in this book, is big on both search and analytics.

1.3.5 Integrated machine learning

Finally, as search engines hold the data, they can be put to use to predict certain events that may happen in the future by “learning” the data using sophisticated machine learning models. Collecting the server loads over a period of time, for example, could lead the machine learning models to predict perhaps the peak or abnormalities in the data. Models can let the organization know when to ramp up the servers or forecast the rate of errors based on the trends analyzed on back-dated and/or real-time data.

Every modern search engine is unique in what they offer. There are a handful of these offering basic search to advanced search and analytics capabilities as well as machine learning and other helpful features. One such product is Elasticsearch. Elasticsearch is a search and analytics engine with superior capabilities on offer. In the next section, we'll go through an introduction to Elasticsearch, its features, and when and why you would want to utilize the toolset that it offers.

1.4 Introducing Elasticsearch

Elasticsearch is an open source search and analytics engine. Developed in Java it is an ultra-fast, highly available search engine built on the popular full-text library, Apache Lucene (<https://lucene.apache.org>). Elasticsearch wraps around the powerful functionality of Lucene by providing a distributed system with RESTful interfaces. Lucene is the powerhouse of Elasticsearch, and Kibana is the administrative UI from the Elastic folks used to manage and work with Elasticsearch. We will work with Kibana's code editor (DevTools) throughout this book.

Elasticsearch lets you search your data with modern search capabilities. The full-text searching is where Elasticsearch as a modern search engine excels. Elasticsearch can retrieve the relevant documents for a user's search criteria at an awesome speed. You can search for exact terms too, like keywords, dates, or a range of numbers or dates. Elasticsearch is packed with top-notch features such as relevancy, did-you-mean suggestions, auto-completion, fuzzy and geospatial searching, highlighting, and more.

In addition to being a frontrunner in providing near real-time search capabilities, Elasticsearch also stands tall in statistical aggregations on big data. Of course, one must consider the use case before jumping into embracing the product as not all the features show the merit of performance at scale. Out of the box, it also boasts commendable features such as application performance

monitoring, predictive analytics and outlier detection, security threat monitoring and detection, and other features.

Elasticsearch focuses on finding a deeper meaning to the data that's been collected. It can aggregate data, create statistical calculations, and find intelligence within the data. One can create rich visualizations and dashboards and share them with others using Kibana tooling. Elasticsearch can help find averages, sums, means and modes as well as complex analytics including bucketing data like histograms and other analytical functions.

Furthermore, Elasticsearch runs supervised and unsupervised machine learning algorithms on your data. Models help to detect anomalies, find outliers as well as forecast events. In a supervised learning mode, we can provide training sets so the model learns and makes predictions.

Though not a core feature of a search engine, Elasticsearch comes with the capability to observe applications and their health by monitoring performance metrics such as memory and CPU cycles of the web servers in a network. It lets you sieve through millions of web server logs to find or debug application issues. Elasticsearch invests time and resources in building security solutions too, for example, alerting security threats, IP filtering, endpoint prevention, and more.

1.4.1 Core areas

Elastic, the company behind the Elasticsearch, has been positioning itself predominantly in three core areas as figure 1.5 shows.



Elastic Enterprise Search

Workplace, website, and app search



Elastic Observability

Unified logs, metrics, and APM data



Elastic Security

SIEM, endpoint, and threat hunting

Fig 1.5 Core application areas of Elastic, the company behind Elasticsearch

ELASTIC ENTERPRISE SEARCH

Whether letting the users search across varied content providers (like Slack, Confluence, Google Drive, and others) or enabling search capabilities for your applications, apps, and websites, the Enterprise Enterprise Search suite helps build the models and customized engine. We will work through the latter case using Elasticsearch as a search server for our applications in the Chapter X Advanced Search.

ELASTIC OBSERVABILITY

Using the tools in the observability space, the state of applications, servers, racks, and modules can all be monitored, logged, tracked, and alerted. We can leverage the Elastic tools to perform application management and monitoring on a large scale. We will learn about the management and monitoring of the engine in Chapter X: Management and Monitoring.

ELASTIC SECURITY

Elastic enters the realm of security by enabling threat detection and prevention and providing advanced features such as the capability of removing malware at the source, encryption at rest, and others. As a Security Information and Event Management (SIEM) tool, Elastic is positioning itself to protect organizations with its advanced security toolkits.

1.4.2 Elastic Stack

While Elasticsearch is the core of the search engine, a handful of Elastic products complement the search engine. The suite of products is called Elastic Stack, which includes Kibana, Logstash, Beats, and Elasticsearch. It was formally called ELK Stack but renamed Elastic Stack after Beats was introduced into the product suite in recent years.

The combination of these four products helps build an enterprise application by integrating, consuming, processing, analysing, searching, and storing various data sets from disparate sources. As demonstrated in figure 1.6, Beats and Logstash bring the data into Elasticsearch, while Kibana is the visual UI that works on that data.

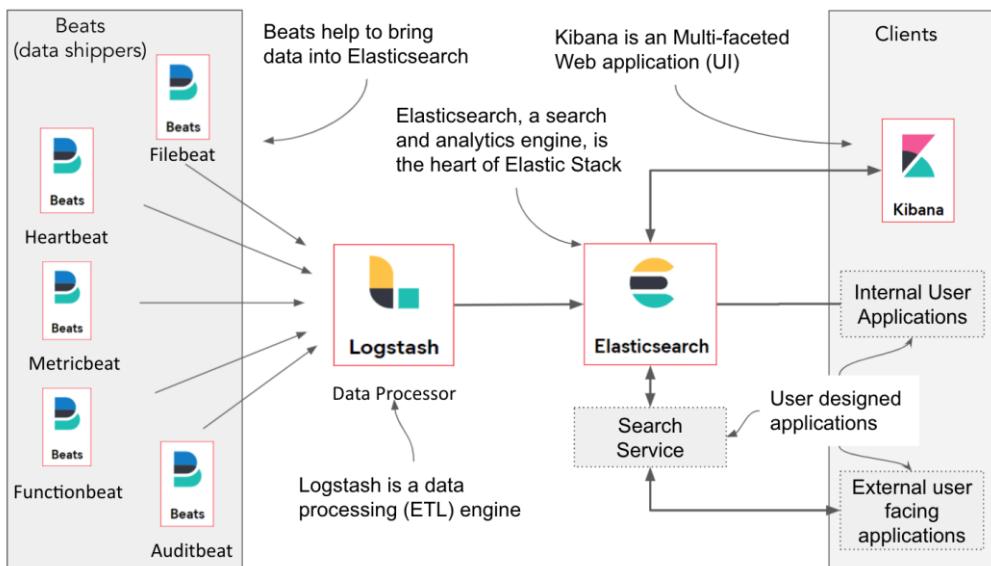


Figure 1.6: Elastic Stack ecosystem, consisting of Beats, Logstash, Elasticsearch and Kibana

BEATS

Beats are data shippers; they load data from various external systems and pump it into Elasticsearch. There are various types of beats available out of the box. These include Filebeat, Metricbeat, Heartbeat, etc., and each one performs a specific data consumption task. These are single-purpose components, for example, Filebeats are designed for file-based transports and Metricbeat for machine and the operating system's vital memory and CPU information. The beats'

agents are installed on the servers so they can consume the data from their source and then send it over to their destination.

LOGSTASH

Logstash is an open source data processing engine. It extracts data originating from multiple sources, processes it, and sends it to a variety of target destinations. During the processing of the data, it transforms and enriches the data. It supports a myriad of sources and destinations including file, HTTP, JMS, Kafka, Amazon S3, Twitter, and dozens of others. It promotes a pipeline architecture, and every event that goes through the pipeline gets parsed as per the pre-configured rules, thus creating a real-time pipeline for data ingestion.

KIBANA

Kibana is a multipurpose web console that provides a host of options, from executing queries to developing dashboards, graphs and charts visualizations to drilldowns and aggregations. However, one can use any REST client to talk to Elasticsearch for invoking the APIs too, not necessarily Kibana. For example, we can invoke APIs using cURL, Postman or native language clients.

1.4.3 Popular adoption

There is a long list of organizations that use Elasticsearch for various purposes from searching to business analysis, log analytics, security alert monitoring, application management, as well as a document store. A few of these organizations include

- *Uber*—Uber powers its rider and event prediction using Elasticsearch. It does so by storing millions of events and searching through them as well as analyzing the data at a near real-time rate. Uber predicts the demand based on location, time, day, and other variables including taking past data into consideration. This helps Uber deliver the ride pretty quickly.
- *Netflix*—Netflix adopted Elastic Stack to provide customer insights to its internal teams. It also uses Elasticsearch for log event analysis to support debugging, alerting, and managing its internal services. The email campaigns and customer operations are all backed by the Elasticsearch engine. Next time you receive an email from Netflix mentioning a newly added movie or a TV series, keep in mind that the campaign analytics behind that simple email were all supported by Elasticsearch.
- *PayPal*—PayPal embraced Elasticsearch as a search engine to allow customers the capability to store and search through their transactions. They have implemented transaction search features along with analytics for use by merchants, end customers, and developers.
- *eBay*—Similarly, the online e-commerce company eBay adopted Elasticsearch a while back to support full-text searching to end users. As a user, we are directly using Elasticsearch when searching through their inventory. They also use the Elastic Stack for analytics, log monitoring, and storing the transactional data in a document store.
- *Walmart*—Walmart utilizes Elasticsearch for analysing its data to identify customer purchasing patterns, trends, and performance metrics. It heavily depends on the Elastic Stack for a few other cases including holiday sales predictions and analytics.
- *Tinder*—If you are a Tinder user, you swipe left or right to look for your love, and you may find love in a fraction of a second (technically, a bi-directional matching algorithm is applied). Tinder and Elasticsearch go hand in hand. It is deployed for various use cases

including geosearches, profile matching, event prediction, and analytical features.

- *Pfizer*—Powered by Elasticsearch. Pfizer uses the Elastic suite for indexing scientific and research data, audit reports, and near real-time metrics data. It uses Elastic Stack heavily for its scientific data cloud project, building a scientific data lake.
- *GitHub*—GitHub, a popular code repository for developers, indexes its 8 million (and counting) code repositories, consisting of over 2 billion documents with Elasticsearch to enable a powerful search experience for its users.
- *Stack Overflow*—Stack Overflow uses Elasticsearch for providing developers with quick and relevant answers.
- *Medium*—Medium (a popular blog platform) uses the Elastic Stack for serving reader queries in a near real-time mode.

There are other organizations, such as LinkedIn, Facebook, and investment banks like Goldman Sachs, Barclays, and Credit Suisse, that use Elasticsearch in one shape or another. Elasticsearch is a multi-faceted server application that can be employed for various use cases from analyzing application logs, searching, or threat detection to analytics and creating visualizations. Let's look at the dominant use cases in the next section.

1.4.4 Elasticsearch use cases

Pinpointing Elasticsearch to a particular use case or a domain is difficult. It is omnipresent in many areas: from search to analytics to machine learning jobs. It is widely used across a multitude of industries, ranging from finance, defence, transport, government, retail, cloud, entertainment, space, and more. Let's take a high-level glance at how Elasticsearch can be used in an organization.

As a search engine

Elasticsearch has become the go-to technology for its full-text search capabilities. The product is not just limited to full-text searching, but can also be used for structured data and geolocation-based searches too. Broadly speaking, customers use Elasticsearch in three domains: App Search, Enterprise Search, and Site Search.

Application Search is where Elasticsearch serves as a backbone for providing the search and analytical capabilities for applications. A search service backed up by Elasticsearch can be designed as a microservice that serves the application's search requirements, such as searching for customers, orders, invoices, emails, and so on.

The data is scattered across many data stores, applications, and databases in organizations. For example, it is usual to find an organization integrated with confluence, intranet spaces, Slack, emails, databases, cloud drives (iCloud drive, Google Drive, etc), and others. Collating and searching through the vast amount of data with integrations to varied sources is a challenge for organizations. This is where Elasticsearch can be employed for *Enterprise Search* and data organization.

If you have an online business website amassing data, providing search is something of a bare necessity for attracting the customers and keeping them happy. The *Site Search* is a Software-as-a-service (SaaS) offering from Elastic, once enabled, crawls through the given site pages fetching the data and building indices backed by Elasticsearch. Once the crawling and indexing is complete, the site can be integrated easily with the search facility. The Site Search module also helps create a

search bar and the code snippet related to this search bar. The administrator of the website can copy the snippet of the generated code onto their homepage to enable a search bar instantly, thus making the website fully functional with integrated search.

BUSINESS ANALYTICS

Organizations capture tons of data from various sources and that data holds the key to survival and success at times. Elasticsearch can help extract trends, statistics, and metrics from this data from time to time, thus enabling the organization with knowledge about its operations, sales, turnover, profits, and many other features at its fingertips for timely management.

SECURITY ANALYTICS, THREAT, AND FRAUD DETECTION

Data security is any organization's nightmare. Elasticsearch's security analytics will help organizations analyse every bit of information - be it application or network or endpoint or cloud. Once analysed, it can provide insights into threats and vulnerabilities. It can let the organization hunt for malware and ransomware thus alleviating the risk of falling prey to hackers.

LOGGING AND APPLICATION MONITORING

Applications spit out a lot of data in the form of application logs and metrics. These logs provide insights into the health of the application. With the advent of the cloud and the microservices world, the logs are scattered across the services and meaningful analysis is a cumbersome affair. Elasticsearch is your friend here. One of the popular use cases for Elasticsearch is indexing the logs and analyzing them for application errors and debugging purposes.

Embracing Elasticsearch (or Elastic Stack) is easy but not necessarily a right choice for every use case. Let's briefly go over the issues we may encounter and the use cases that Elasticsearch is a wrong choice for in the next section.

1.5 Prominent features of Elasticsearch

The functional features like full-text searching, auto-completion, suggestions, fuzzy and geosearching, coupled with technical functions such as high availability, autoscaling, and blazingly fast response times places Elasticsearch as the forerunner in the search engine race.

1.5.1 Multifaceted text searching

Organizations amass unstructured data, data that may not have an explicit mapping or metadata or a shape such as legal documents, research papers, news articles, emails, confluence pages, word documents, PDFs, and so on.

For example, a media organization produces and collects thousands of newsworthy articles every day, and over the years, these add up to millions of documents. The data in these documents is unstructured. A bank searching through employees emails for compliance purposes actually works on unstructured (email) data. Driverless cars, mobile devices, smart TVs, cameras, and all high-tech gadgets emit data in real time. And while the data is somewhat structured, it can fall into the unstructured bracket too.

Elasticsearch, the search engine based on the NoSQL model, can search through this unstructured data efficiently. Larger volume of data is not a constraint for Elasticsearch when it comes to finding results.

Elasticsearch is pretty good in serving the queries against structured data, data of precise nature like timestamps, dates, quantities, ranges, and so on. With structured data, queries compare terms with the data. The results returned are because all of the expected conditions have been met.

For example, searching for *flights from March 1 through March 5* yields only those flights available during the start and end dates. Similarly, if we want to find bestsellers across all books, perhaps we can run a query like *books that sold a million copies*. In both cases, results return a range that they fall in. Either they are in that range or not: nothing like the “may be” case. We can create complex queries around such data and expect accurate results. Elasticsearch supports almost a dozen term-level queries (queries on structured data are called term-level queries in Elasticsearch).

We are all humans and, of course, we may make mistakes when asking a search engine what we are after. Say you are in a hurry and searched for *Kava Book* on an e-commerce site selling technical books powered by Elasticsearch. The search engine forgives, however, and it asks you if you mean *Java Book* instead. This type of search is called *fuzzy search*, a search where spelling mistakes are forgiven. This feature is quite useful for many organizations, especially those that gather a copious amount of unstructured data.

Say, for example, you wake up late on a Sunday morning and are looking forward to a sumptuous brunch. You hurry to your mobile to fetch restaurants nearby using an app. If that app is powered by Elasticsearch, then chances are geoqueries were fired at Elasticsearch, which returns all the restaurants in and around a 5 kilometer radius, along with filtering criteria for fine-tuning the search.

As another example, you may have received that dreaded call at 2 a.m. in the morning from your support team mentioning that your mission-critical application was crumbling down. You have no choice but to login to find out the root cause and avoid catastrophic failure. But, as the system was already integrated with the Elasticsearch ecosystem, analyzing the incoming logs (most likely millions of log events accumulated in real time over the last few days) reveals an error that may be the cause of the application's breakdown. With the aid of visualization tools and APIs provided by Elasticsearch, you can search, analyze, and simultaneously fix the error.

There are a multitude of search features that Elasticsearch furnishes. We will go over these in more detail in the upcoming chapters, but in the next section, let's flip the coin and learn about another important offering from Elasticsearch: analytics.

1.5.2 Analytics and aggregations

Analytics enables organizations to find insights into their data and helps them to derive statistics from it. Analytics is taking a step back and looking at the data from a high level to draw conclusions about it. It helps organizations understand customers, evaluate the performance of their products, and forecast sales as well as answering a wide range of questions such as application performance over a period of time, security threats, and more. Elasticsearch provides first-class support for analytics.

With Elasticsearch, we can generate metrics such as averages, sum, min and max values, and others using *metrics aggregations*. These are single-valued measures that come in handy for most basic situations like providing the number of blog posts, setting a best seller tag against a category of books, finding out the min and max range of an employee salary band, and so on.

The *bucket aggregations* collect the data in buckets (hence the name, bucket aggregations), splitting the data into ranges. For example, “access denied” errors per day for the past week, coffee vs tea sales figures during weekends, average score of a student and so on. We can represent these analytics using histograms, bar charts, and other appealing visualizations with Kibana’s dashboard features.

While metric and bucket aggregations work on the document data to produce the aggregated output, *pipeline aggregations* work on the output of other aggregations to produce new aggregations. They are very useful to compute moving averages, derivatives and others.

1.5.3 RESTful over HTTP

Talking to Elasticsearch is simple. Elasticsearch boasts a rich set of consistent and congruent APIs, exposed as RESTful endpoints over HTTP. Any client who can talk REST can access the server without a second thought. Elasticsearch supports clients written in any of the popular programming languages like Java, .Net, JavaScript, Python, Ruby, Go, Perl, and others. This standardized access makes Elasticsearch suitable for integrating and working with legacy, n-tier, or microservice applications without having to re-engineer the existing application state.

1.5.4 Schema-free engine

Elasticsearch is a schema-free engine. Elasticsearch stores data as JavaScript Object Notation (JSON) documents. It serializes these documents and stores them for later retrieval. We can index the data without having to define any schema, which is indeed an appealing feature.

Elasticsearch guesses the schema (if not predefined) based on our input document’s data. While in most cases, the mapping may be correct, there is a chance the data could be interpreted incorrectly, which leads to issues in searching and sorting. Since we know the shape of our data, we would create the schema beforehand to save ourselves wasting time when searches don’t return expected results.

It is time to wrap up this chapter. This chapter laid the groundwork for using Elasticsearch by introducing us to its search capabilities and how searching has become an integral part of numerous applications. In the next chapter, we will install, configure, and run Elasticsearch and Kibana, and we will play with it by indexing a few documents and running search queries and analytics. Stay tuned.

1.6 Summary

- Search is the new normal and is the most sought after functionality for organizations. It is a competitive advantage enabler.
- Search engines built using relational databases as the backend used to serve our search purposes, but were never near fulfilling the full-fledged search functionality as found in modern search engines.
- Modern search engines provide multi-faceted, full-text search capabilities and are expected to provide multifold benefits from basic search to advanced search and analytical functions, all with a split-second performance. They are also expected to handle data in terabytes to petabytes and then scale if needed.
- Elasticsearch is an open source search and analytics engine built over Apache Lucene. It is a highly available server-side application developed in Java.
- Because Elasticsearch was designed as a programming-language agnostic product, the mode of communication with the server is over HTTP, using rich RESTful APIs. These APIs receive and send data in JSON format.
- Elastic Stack is a suite of products composed of Beats, Logstash, Elasticsearch, and Kibana. Beats are single-purpose data shippers; Logstash is a data-processing ETL (extract, transform, load) engine; Kibana is the administrative UI tool; Elasticsearch is the heart and soul of this stack.
- Elastic Stack enables your company or organization to position itself in three core areas: search, observability, and security.
- Elasticsearch has become popular over the last few years due to its features like structured/unstructured search and analytics capabilities, a rich set of RESTful APIs, its schema-free nature, and performance, high availability, and scalability characteristics.

2

Getting started

This chapter covers

- Indexing sample documents with Elasticsearch
- Retrieving, deleting, and updating documents
- Searching with basic to advanced queries
- Running aggregations on data

This chapter is all about experiencing a taste of Elasticsearch. Elasticsearch is a Java binary that is available to download from the Elastic company's website. Once the server is installed and up and running, we will need to plug in our business data so the engine can analyze that data and persist. Once the engine is primed with this data, we can execute search queries to fetch results.

Although any client capable of invoking REST calls can talk to Elasticsearch, we'll use Kibana as our preferred client throughout this book. Kibana is a web application from the same company that developed Elasticsearch. It is an opulent visual editor that comes with all the bells and whistles, not simply searching for data. With Kibana you get abundant capabilities such as advanced analytical and statistical functions, rich visualizations and dashboards, machine learning models, and more. As Elasticsearch exposes all its functionality via RESTful APIs, we can construct our queries using these APIs in Kibana editor and communicate with the server over HTTP.

Once we get the required software installed and running, we will index a few documents and retrieve them using the document APIs. We will execute search queries by invoking the search APIs and analyze the results. The queries include basic search criteria such as pattern matching, searching prefixes and phrases, correcting spelling mistakes, finding results for a range, searching multiple fields, and so on. We will briefly touch advanced queries using multiple clauses in a search. While performing these actions, we will also learn about sorting the data, paginating the results, highlighting, and other cool things along the way.

Finally, we'll zoom out and analyze the data by executing two types of aggregations: metric and bucket. With these aggregation types, we'll use queries to fetch some metrics such as average, sum, minimum and maximum values, etc..

Because we need both Elasticsearch and Kibana up and running to execute the samples, this chapter will work on the assumption that you have already completed the setup. If you still need to do that, follow the instructions in the *Appendix A: Installing and running Elasticsearch and Kibana* to download and install your software and bring up the Elasticsearch server and Kibana UI.

NOTE INSTALLATION FLAVOURS Installing Elasticsearch and Kibana can come in multiple flavours - right from downloading the binaries and uncompressed and installing them on to your local machine in a traditional way to using package manager, docker or even cloud. Choose appropriate flavour of installation for development to get started

Once you are set up with the applications running, let's get started with Elasticsearch.

NOTE COPY THE FULL CODE TO YOUR KIBANA To make the coding exercises easy, I've created a [ch1_getting_started.txt](#) file under the kibana_scripts folder at the root of the repository. Copy the contents of this file to your Kibana as-is. You should be able to work through examples by executing the individual code snippets alongside following the chapter's contents.

2.1 Priming with data

A search engine can't work on thin air! It needs data as its input so it can produce results as output when queried. We need to put our data into Elasticsearch, which is the first step in priming the engine. But before we start storing the data in Elasticsearch, let's get to know the sample application that we will work with in this chapter.

For our examples in this chapter, we need to have a basic understanding of the problem domain and the data model. Let's assume we are building an online bookstore, obviously, we are not architecting the whole application: we are only interested in the data model part for our discussion. We'll go over the details of this fictitious bookstore in the next section as a prerequisite for our objective of working with Elasticsearch.

2.1.1 An online bookstore

To demonstrate the Elasticsearch features, let's use a fictional book store that sells technical books online. For this, all we want to do is to create an inventory of books and write some queries to search through them.

NOTE: The code presented in this chapter is available in the [GitHub](#) repository. Follow the instructions listed in the repository to index the data.

The data model for our bookstore application is simple. We only have a `book` as our entity with a few properties such as `title`, `author`, and so on as described in table 2.1. We do not need to complicate things by creating elaborate entities here. Instead, we'll focus on the goal of getting hands-on experience with Elasticsearch.

Table 2.1 Data model for a book represented in a tabular format

Field	Explanation	Example
title	The title of a book	"Effective Java"
author	The author of the book	"Joshua Bloch"
release_date	Data of release	01-06-2001
amazon_rating	Average rating on Amazon	4.7
best_seller	Flag that qualifies the book as a best seller	true
prices	Inner object with individual prices in three currencies	<pre>"prices":{ "usd":9.95, "gbp":7.95, "eur":8.95 }</pre>

As mentioned, Elasticsearch is a document data store, expecting the documents to be presented in JSON format. Because we need to store our books in Elasticsearch, we must model our entities as JSON-based documents. Let's represent a book in a JSON document as demonstrated in figure 2.1.



Figure 2.1 A JSON representation of a `book` entity for our Elasticsearch data store

As figure 2.1 shows, the JSON format represents data in a simple name-value pair. For our example, the book's title (name) is *Effective Java* and its author (as a value) is Joshua Bloch. We can add additional fields (including nested objects) as you can see for the `prices` object shown in the figure 2.1.

Now that we have an idea of our book store and its data model, it is time to start populating Elasticsearch with a set of books to create an inventory of books. Let's do this in the next section.

2.1.2 Indexing documents

To work with the server, we need to get the client's data indexed into Elasticsearch. There are few ways we can bring the data into Elasticsearch: for example, creating an adapter to pipe the data from a relational database, extracting from a file system, or streaming events from a real-time source. Whichever choice is your data source, we use RESTful APIs to load the data into Elasticsearch.

Any REST-based client (cURL, Postman, Advanced REST client, HTTP module for JavaScript/NodeJS, etc.) can help us talk to Elasticsearch via the API. Fortunately, Elastic has a product that does exactly this (and more): Kibana. Kibana is a web application with a rich user interface, allowing users to index, query, visualize, and work with data. *This is our preferred option as we will use Kibana extensively in this book.*

NOTE RESTFUL ACCESS The mode of communication with Elasticsearch is via JSON-based RESTful APIs.

In the current digital world, it is highly unlikely to find a programming language that doesn't support accessing RESTful services. In fact, designing Elasticsearch with APIs exposed as JSON-based RESTful endpoints was a smart choice because this will enable programming-language agnostic adoption.

Now we know how to model our data, the next step is to get it indexed. To get the data into Elasticsearch we use Document APIs, discussed in the next section.

2.1.3 Document APIs

We work with Elasticsearch's document APIs for creating, deleting, updating, and retrieving documents. To index a document, we need to use an HTTP PUT or POST (more on POST later) on an endpoint. Figure 2.2 shows the syntax of a full URL format for an HTTP PUT method.

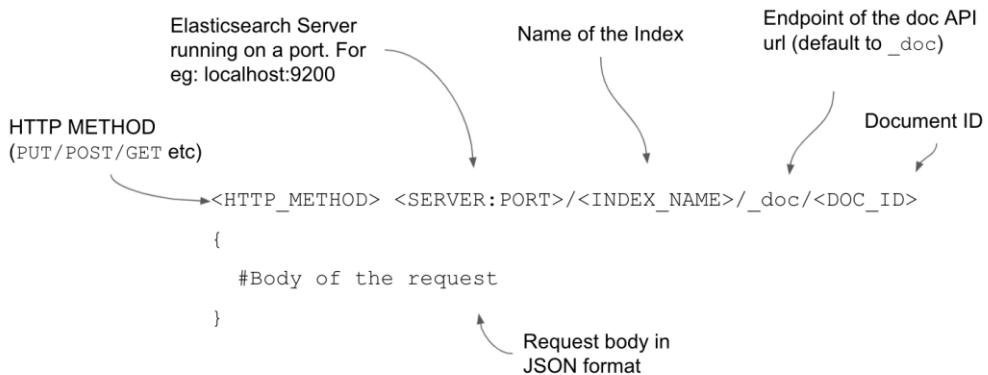


Figure 2.2 Elasticsearch URL invocation endpoint using an HTTP method

The URL is composed of few elements:

- An HTTP action such as PUT/GET/POST
- Server's hostname and port
- Index name
- A document API's endpoint (`_doc`)
- A document's ID
- A request body

The Elasticsearch API accepts a JSON document as the request body so the book we want to index should accompany this request. For example, the following code snippet will index a book document with an ID 99 to a `books` index (listing 2.1):

Listing 2.1: Indexing a book document into books index

```
PUT books/_doc/1 #A The books is the index and document's ID is 1
{ #B The body of the request consists of JSON data
  "title": "Effective Java",
  "author": "Joshua Bloch",
  "release_date": "2001-06-01",
  "amazon_rating": 4.7,
  "best_seller": true,
  "prices": {
    "usd": 9.95,
    "gbp": 7.95,
    "eur": 8.95
  }
}
```

2.1.4 Using cURL

We can use cURL to work with Elasticsearch too, though we will prefer Kibana over cURL in this book. Let me show you the full cURL command as demonstrated in the figure 2.3.

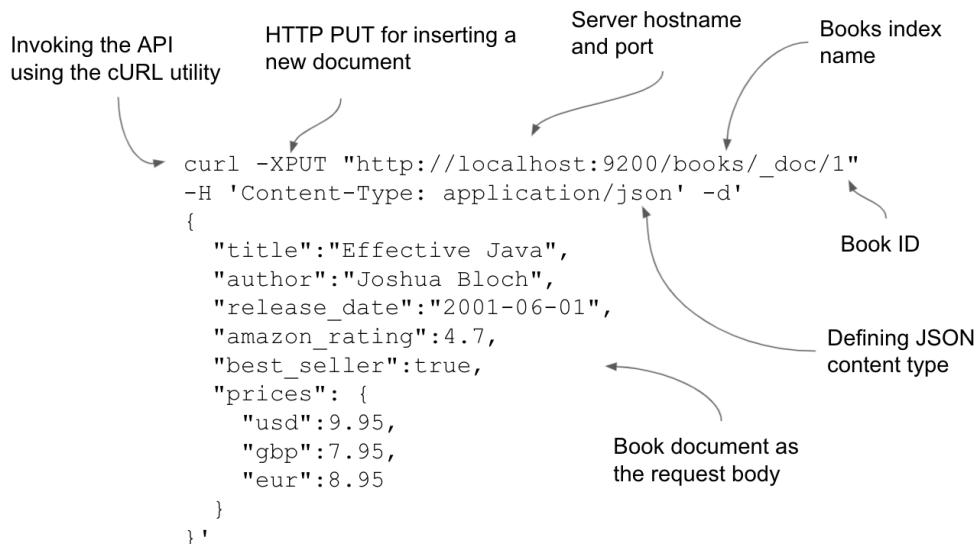


Figure 2.3 Elasticsearch URL invocation endpoint using cURL

As you can see, cURL expects us to provide some request parameters such as the content type, document, and so on. Because the cURL commands are terminal commands (command-line invocation), preparing the request is cumbersome and, at times, erroneous. Fortunately, Kibana lets us drop the server details, content types, and other parameters, so the invocation transforms into something like that shown in figure 2.4.

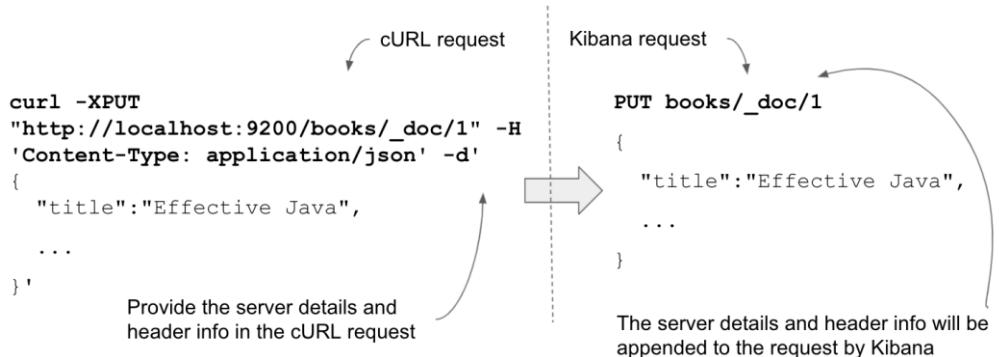


Figure 2.4 Transitioning from a cURL command to Kibana's request command

As I mentioned, we are going to stick with Kibana to interact with Elasticsearch in this book.

2.1.5 Indexing our first document

To use Kibana to index our first document, we'll go to Kibana's DevTools application for executing queries. Because we'll spend a lot of time on the DevTools page, by the end of this book, you'll be quite familiar with this page; I can tell!

Go to the Kibana dashboard at <http://localhost:5601>, and on the top-left corner, you will find a main menu. Under this menu, you will find some links and sublinks. For our purposes now, navigate to the Dev Tools link under Management as figure 2.5 shows.

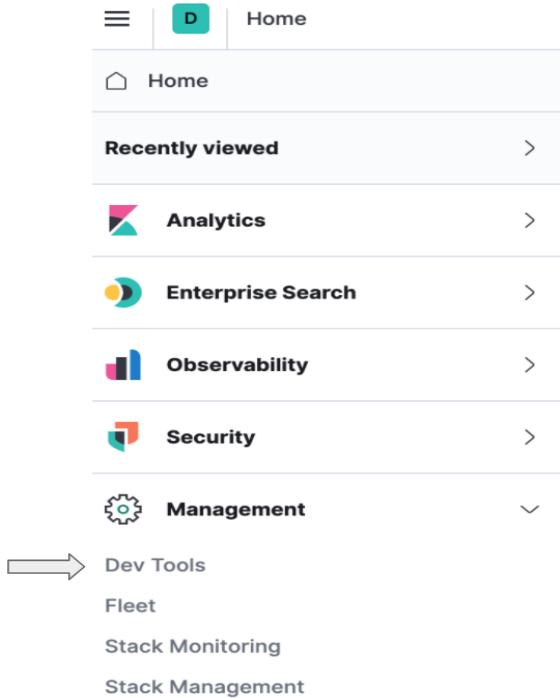


Figure 2.5 Accessing the DevTools navigation page

As this could be your first time accessing this code editor page, let me explain the components you'll see here. When you navigate to this page (see figure 2.6), the code editor opens, showing two panes. On the left pane, you write the code with the special syntax provided by Elasticsearch. Once you write the code snippet, you can invoke the URL in the snippet by clicking the Execute button in the middle of the page (a green triangle).

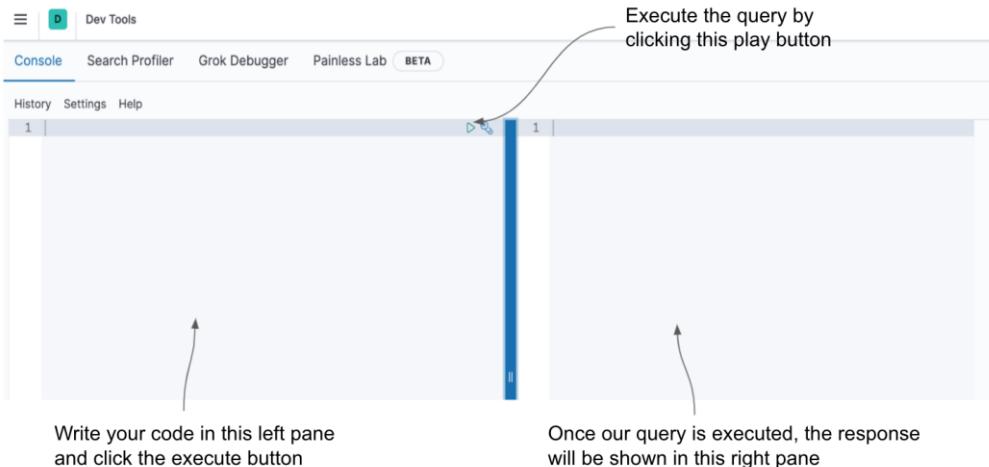


Figure 2.6 Kibana's DevTools code editor

Kibana must be connected to an Elasticsearch instance (the instance details are defined in the Kibana's configuration file) before it's ready for action. Kibana uses the server's details to wrap the code snippet with the appropriate Elasticsearch URLs and send it to the server for execution. To index our document for Elasticsearch (the code is given in listing 2.1 earlier), let's create a code snippet. Figure 2.7 shows the indexing request and a response.

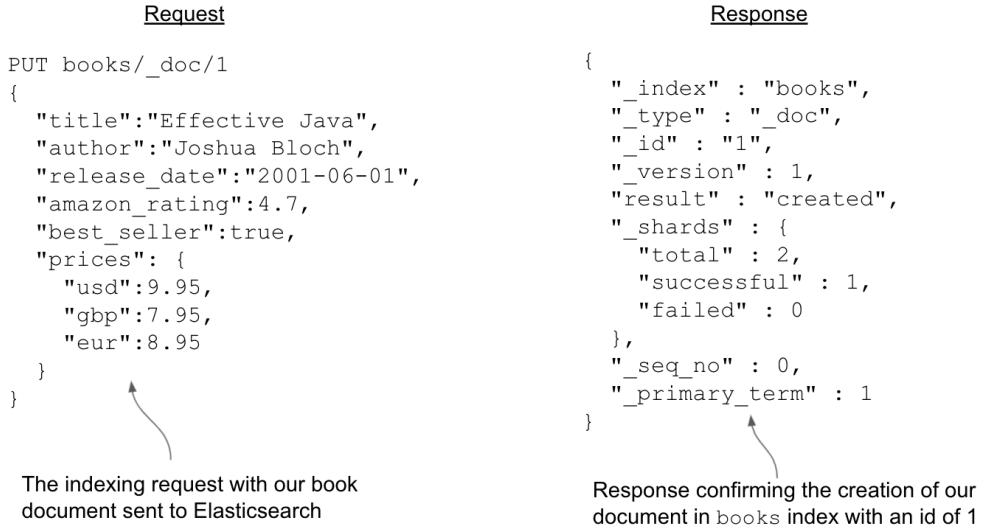


Figure 2.7 Indexing a document in Kibana (left) and the response from Elasticsearch (right)

Once the code is ready, click the “green triangle play” button (figure 2.6). Kibana sends this request to the Elasticsearch server once invoked. After receiving the request, Elasticsearch processes the request (figure 2.7), it stores the message and sends the response back to the client (Kibana). You can view the response on the right-hand side panel in the editor.

The response is a JSON document. In figure 2.7, the `result` property indicates that the document was created successfully. The response has some additional metadata (such as the index, ID, and document version), which are self-explanatory. We will discuss the constituents of the requests and responses in detail in the upcoming chapters, but let me explain the request and response flow on a high level to help you out a bit here. The overall flow process is represented visually in the figure 2.8 here:

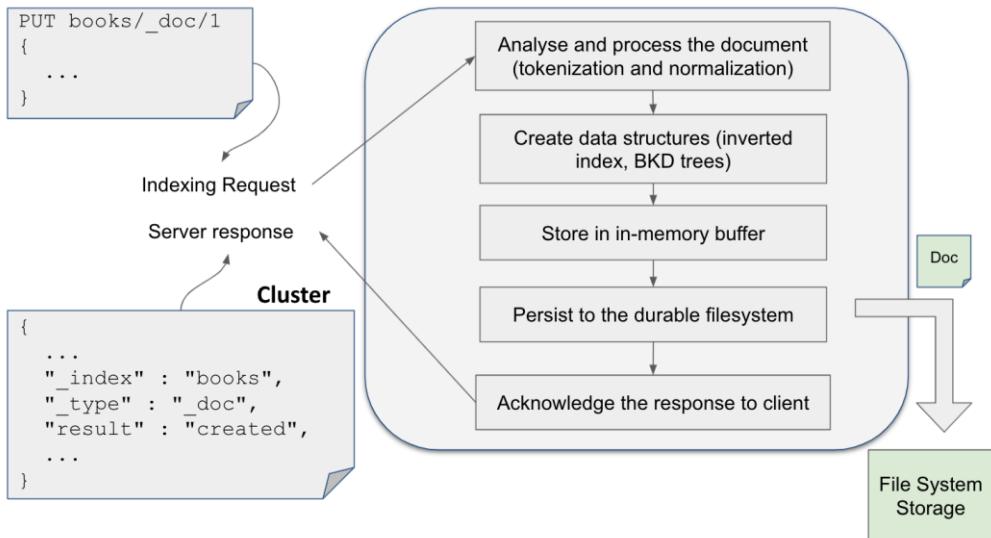


Figure 2.8 Elasticsearch's request and response flow at a high level

The flow steps are:

- Kibana posts the request to the Elasticsearch server with the required input parameters.
- On receiving the request, the server
 - Analyzes the document data and, for speedier access, stores it in the inverted index (a high-performing data structure, which is the heart and soul of the search engine (we will learn more about it in the later chapters)).
 - Creates a new index (we did not create the index upfront) and stores the documents. The server also creates required mappings and data schemas.
 - Sends the response back to the client.
- Kibana receives the response and displays it in the right panel (figure 2.6) for our consumption.

Awesome! We've indexed our first document for Elasticsearch! This is similar to inserting a record into a table in a relational database.

2.1.6 Constituents of the request

The request we sent (`PUT books/_doc/1`) needs a bit of distilling. There are five parts to this request, and I'll quickly go over each of these here:

- *PUT method*—PUT is a HTTP verb (also known as a method) that indicates we are sending a request to the server to create a resource (a book document in this case). Elasticsearch uses the HTTP protocol for its RESTful API invocations, so we expect PUT, POST, GET, DELETE, and other standard HTTP methods in the request URL syntax.
- *The books index*—The `books` in our URL is called an *index*, which is a bucket for collecting all book documents. It is similar to a table in a relational database. Only book documents are stored in this `books` index.
- *The _doc endpoint*—The `_doc` in our URL is the endpoint. This is a constant part of the path that's associated with the operation being performed. In earlier versions of Elasticsearch (version < 7.0), the `_doc`'s place used to be filled up by a document's mapping type. The mapping types were deprecated and `_doc` came to replace the document's mapping types as a generic constant endpoint path in the URL.
- *Document ID*—The number 1 in our URL represents the document's ID. It is like a primary key for a record in a database. We use this identifier to retrieve the document.
- *Request body*—The body of the request is the JSON representation of the book data. Elasticsearch supports nested objects sent over in JSON's nested objects format.

NOTE DOCUMENT TYPES AND THE _DOC ENDPOINT Prior to the 7.x version for Elasticsearch, an index could hold multiple types of entities (for example, a `books` index could hold not just `books` but also `book_reviews`, `book_sales`, `bookshops`, and so on). Having all types of documents in a single index led to complications. Field mappings were shared across multiple types, thus leading to errors as well as data sparsity. To avoid issues with types and their management, Elastic decided to remove the types altogether. In earlier versions, the invocation URL with a type would look something like this: `<index_name>/<type>/<id>` (or, for example, `books/book/1`). The types were deprecated in version 7.x. Now we are expected to have an index dedicated to one and only one type: `_doc` is an endpoint etched into the URL. We will learn about removal of types in chapter 5 on document operations.

The gist is that we used a HTTP PUT method to index a document with an ID of 1 into the `books` index for Elasticsearch. When indexing our first document, did you notice we didn't create a schema at all? Instead, we indexed the document by invoking the API, right?

Unlike in a relational database, Elasticsearch doesn't ask us to create the schema beforehand (its schema-less feature) and is happy to derive the schema from the first document that it indexes. It also creates the index (the `books` index to be precise). The bottom line is that Elasticsearch doesn't like to get in our way during development. That said, we must follow the best practice of creating our own schemas beforehand in production environments.

In this section, we successfully indexed a document. Let's follow the same process and index a couple of documents.

2.1.7 Indexing more documents

For the upcoming examples to work, we need a few more documents to index. Let's head over to Kibana's code editor and write some code for two more documents and get them indexed. Figure 2.9 shows the code for the document ID 2 and ID 3. Execute these queries individually to have two more documents indexed for your engine.

```
PUT books/_doc/2
{
  "title": "Core Java Volume I - Fundamentals",
  "author": "Cay S. Horstmann",
  "release_date": "2018-08-27",
  "amazon_rating": 4.8,
  "best_seller": true,
  "prices": {
    "usd": 19.95,
    "gbp": 17.95,
    "eur": 18.95
  }
}

PUT books/_doc/3
{
  "title": "Java: A Beginner's Guide",
  "author": "Herbert Schildt",
  "release_date": "2018-11-20",
  "amazon_rating": 4.2,
  "best_seller": true,
  "prices": {
    "usd": 19.99,
    "gbp": 19.99,
    "eur": 19.99
  }
}
```

Indexing a document with ID 2 Indexing a document with ID 3

Figure 2.9 Indexing two more documents using document API

NOTE MORE DOCUMENTS CODE ON GITHUB While I had presented the two documents side by side in the above figure (figure 2.9), it was for brevity purposes. The full code is available on the book's accompanying GitHub repository

2.2 Retrieving Data

We primed our server, albeit with a very limited set of documents. It is time to spring into action to check how we can retrieve, search and aggregate on these documents. From The next section onwards, we will get hands-on by executing queries and retrieving data.

Let's start with a basic requirement of finding out the total number of books we have in stock (the `books` index in this case). Elasticsearch exposes a `_count` API to satisfy this requirement. It is pretty straightforward as we'll see in the next section.

2.2.1 Counting all documents

Knowing the total number of documents in an index is a requirement; one that's fulfilled by the `_count` API. Invoking the `_count` endpoint on the `books` index results in the number of documents persisted in that index, as shown in the listing 2.2:

Listing 2.2: Counting all documents using _count API

```
GET books/_count #A Counting the number of books
```

This should return a response as shown in the figure 2.10:

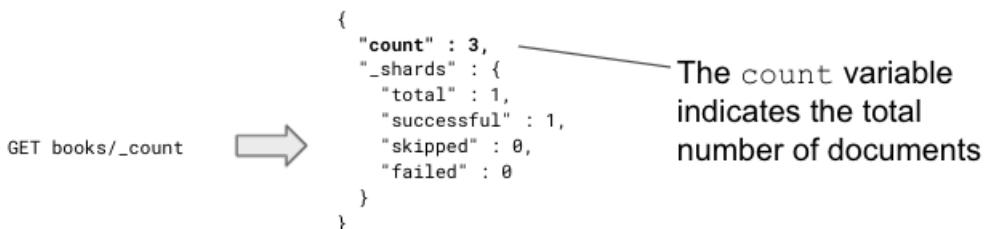


Figure 2.10 The JSON response for a `_count` API invocation

The highlighted variable `count` indicates the total number of documents we have in the `books` index. You can use the same API to fetch from multiple indices in one go too, as the code listing 2.3 indicates:

Listing 2.3: Fetching the number of all documents from multiple indices

```
GET books,movies/_count
```

This call should return a total of all documents from multiple indices, in this case `books` and `movies`. However, as we did not create the `movies` index yet, the system will throw an `index_not_found_exception` indicating `movies` index doesn't exist. Make sure you create and index a few documents into `movies` index beforehand (code is available on the GitHub repo for this exercise).

We can fetch the number of documents in all indices, too, by issuing a `GET _count` call. This will return all the documents available including system and hidden indices.

In the next section, we will use some document APIs to retrieve the documents.

2.2.2 Retrieving documents

Just as we deposit money in our bank account, we indexed our documents into Elasticsearch in the last section. We now need to access these documents (like we fetch money from our bank account), which is what we are going to find out how to do in this section.

Every document that gets indexed has unique identifiers, some provided by us and others generated by the server. Operations to retrieve our documents depend on the document IDs. If we have those, we can go ahead and fetch the documents. If we don't have the IDs, we must use a `_search` API. These are the operations that we'll see in this section:

- Getting a single document given an identifier
- Getting multiple documents given a set of identifiers (IDs)
- Getting all documents in one go (no IDs, no search criteria)

We can fetch a single document from Elasticsearch by invoking a document API call given an ID and use another API query (called `ids` query) to fetch multiple documents in one go. We also go over the `_search` API to retrieve all documents in one go.

RETRIEVING A SINGLE DOCUMENT

Similar to fetching a record from a database using a primary key, retrieving the document from Elasticsearch is straightforward, as long as we have the document's ID. To do that, we issue a GET command in the Kibana console with the API URL and document ID. The generic format for retrieving the documents with an identifier is

```
GET <index>/_doc/<id>
```

Let's try fetching our documents as we already know their identifiers. I will not show the Kibana UI screenshots everytime we invoke or write a query; instead, the query is presented to you as is in the text format.

To retrieve a single document, issue a GET command providing the ID as 1 as shown in the listing 2.4 here:

Listing 2.4 Retrieving an individual document by its ID

```
GET books/_doc/1
```

If this command is executed successfully, you will see a response on the right-hand side panel of the code editor, as figure 2.11 demonstrates:

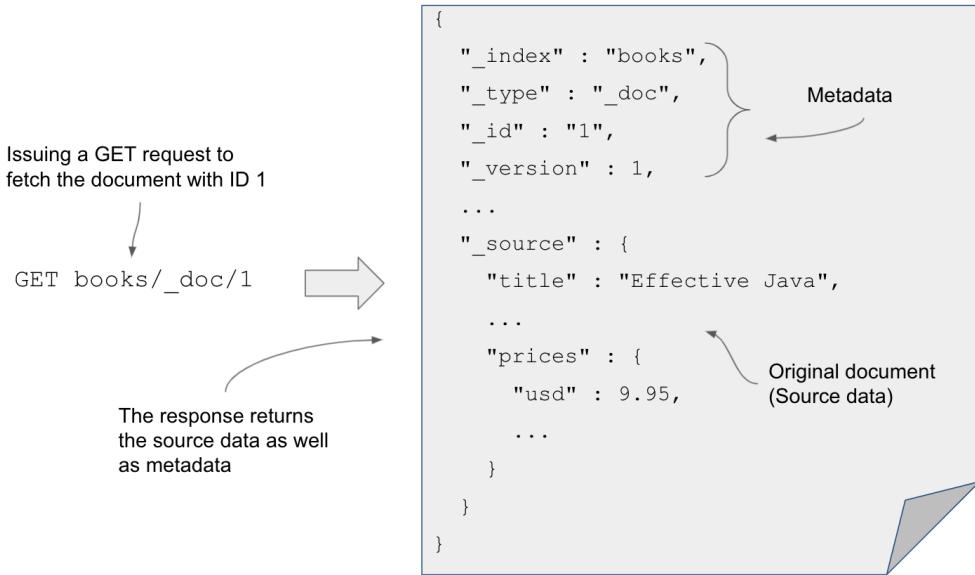


Figure 2.11 Fetching a book document by an ID

The response has two pieces of information: the original document under the `_source` tag and the metadata of this document (such as `index`, `id`, `found`, `version` etc.). That's the most basic way to retrieve a document.

CALLOUTS: To fetch the original source document (no metadata), issue the command:

`GET books/_source/1` (replacing `_doc` with `_source` endpoint).

RETRIEVING MULTIPLE DOCUMENTS BY IDs

When we need to retrieve a set of documents given a set of identifiers, we can use an `ids` query. The query fetches the documents given a set of document IDs. It's a much simpler way to fetch them in one go, given a list of document IDs.

One thing to note here, though, is that unlike other queries that use document APIs to fetch documents, the `ids` query uses a search API, specifically a `_search` endpoint. Let's see that in action.

NOTE QUERY DSL Elasticsearch provides a domain specific language (DSL) for writing queries, commonly called Query DSL. It's a simple JSON based query writing language, extensively used to write queries in Kibana. All the queries you see here in this chapter (and the rest of the book) are based on Query DSL.

We create a GET request on our index with the `_search` endpoint, passing the IDs wrapped in a `query` object, as the listing 2.5 shows here:

Listing 2.5: Fetching multiple documents using `ids` query

```
GET books/_search
{
  "query": {
    "ids": {
      "values": [1,2,3]
    }
  }
}
```

The request body in our query in listing 2.5 is constructed using a `query` object, which has an inner object named `ids`. The document IDs are the values provided as an array.

The response indicates the successful hits (results), as shown in the figure 2.12 here, returning all three documents with those three IDs:



Figure 2.12: Retrieving documents given a set of IDs using an `ids` query, which invokes a `_search` endpoint.

NOTE SUPPRESSING THE ORIGINAL SOURCE DATA IN THE RESPONSE Sometimes we can stop clogging the network and wasting the bandwidth by avoiding retrieving the source data that is not required in the first place. For example, imagine if our source document has 200 fields and all may not be relevant to retrieve. We can control what fields are sent to the client and which are not by disabling the feature. We can also set the flag `_source": false` on the request to completely suppress sending the source data in the response.

Of course, this way of retrieving the documents is cumbersome. Imagine, instead, you want to retrieve 1,000 documents with those many IDs! Well, fortunately, we don't need to use an `ids` query. Instead, we can utilize other forms of search features that we will go over later in the book.

So far, we've looked at a document API that fetches the documents using IDs. This is not exactly a search feature but a data retrieval. Elasticsearch provides a myriad of search features to fetch results based on various conditions and criteria. We have dedicated several chapters on search, covering basic to advanced, in the later part of the book. But as a curious developer, we want to see high-level search queries in action, don't we? Let's discuss the set of queries based on search functions in the next section.

RETRIEVING ALL DOCUMENTS

In the previous section, we already worked with the basic `_search` endpoint. Using the same syntax, we can write a query to fetch all the books from our index, as the listing 2.x shows:

Listing 2.6: Retrieving all documents in one go from the books index

```
GET books/_search
```

The response, as demonstrated in figure 2.13, returns three documents in our `books` index as expected.

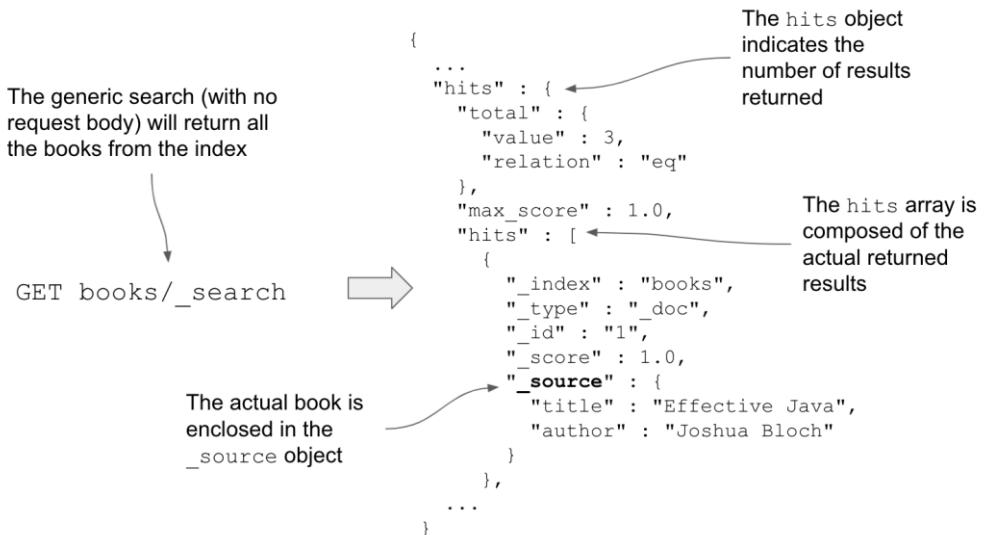


Figure 2.13 Retrieving all documents using the search API

Don't worry too much if you are baffled with the query syntax or the response fields. We will cover the semantics in due course.

NOTE GENERIC SEARCH IS A SHORT FORM OF MATCH_ALL QUERY The query `GET books/_search` you saw previously is a short form of a special query named `match_all` query. Usually, a request body is added to the URL with a query clause, as shown here:

```
GET books/_search
{
  "query": {
    "match_all": {}
  }
}
```

So far, we've seen queries that aren't really exhibiting the full power of Elasticsearch, other than just fetching data without much intelligence. Occasionally, we may want to issue a query with criteria (for example, fetching books by a particular author with an overall rating higher than 4.5). The real power of Elasticsearch is hidden in the flexibility and depth of search functionality. We will, albeit at a high level, look at this in the next few sections.

2.2.3 Full text queries

It's important to be able to find documents that meet specific criteria, once you have a number of them indexed and want to find specific material. Let's look at two use cases that search books for an author and a title.

SEARCHING A BOOK WRITTEN BY A SPECIFIC AUTHOR

Say, for example, our readers visiting our bookstore want to find all the titles authored by Joshua. We can construct the query using the same `_search` API, but this time, let's use a `match` query (listing 2.7) with the criteria to search on an `author` field:

Listing 2.7 Querying for books authored by a specific author

```
GET books/_search
{
  "query": {
    "match": {
      "author": "Joshua"
    }
  }
}
```

By now, you should be familiar with the query URL syntax (`GET books/_search`) from the previous section (retrieving all documents), so let's not spend any more time on this. The request body, however, looks interesting. In the request body, we created a `query` object defining a `match` query. In this query clause, we ask Elasticsearch to *match all the documents authored by Joshua across all our books in stock*. Read the query as if you are reading prose in plain English and try to digest the essence of it.

Once the query is sent to the server, it analyses the query, matches it with its internal data structures (inverted indexes), fetches the document(s) from the file store and returns them to the client. In this example, the server found the single book authored by Joshua and returned it to us, evidenced by the code snippet in figure 2.14.

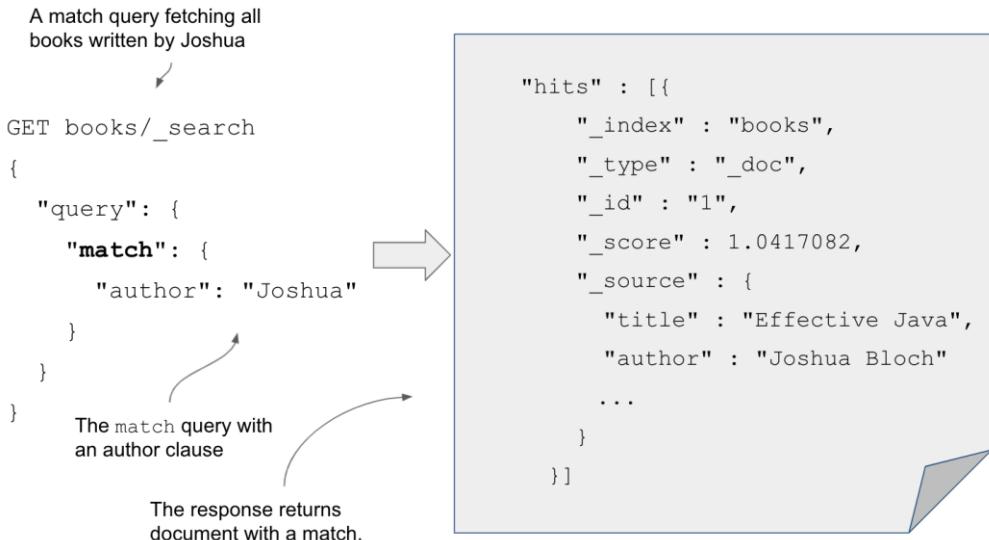


Figure 2.14 Fetching books authored by Joshua

NOTE PREFIX QUERY You can rewrite the above query with varied combination searches, for example, search for lowercase or mixed case name, search on lastname and so on:

```

//successful queries
"author":"JoShua"
"author":"joshua" (or "JOSHUA")
"author":"Bloch"
  
```

All these queries will succeed however, searching for a shortened name will fail:

```

//query will fail for searching "josh"
"author":"josh"
  
```

To return such regex typed queries we use **prefix query**. You can change the **match query** to **prefix query** as shown below to fetch a shortened name:

```

GET books/_search
{
  "query": {
    "prefix": {
      "author": "josh"
    }
  }
}
  
```

However, the author's value must be lowercase as the **prefix query** is a term level query.

If we search for a full name like "Joshua Bloch", we will get the books returned as expected. However, if we tweak the query with "Joshua Doe", what do you expect? We don't have any books written by Joshua Doe, so shouldn't return any results right? That's not the case, we will still get the books returned written by Joshua Bloch, although there is no Joshua Doe in our author's list. The reason for this is that the engine is searching all *books written by Joshua OR Doe*. The OR operator is used implicitly in this case. Let's see how we can change the implicit OR operator to AND to get an exact match in the next section.

SEARCHING A BOOK WITH AN EXACT AUTHOR

Now that we know how to search for books written by an author's first name or last name, let's see how we search for books with an *exact* full name? We can use the `match` query again for this purpose. Let's take an example of a search query: `author: Joshua Schildt` (mixing up Joshua's first name and Herbert's last name). Obviously we know there's no book written by that fictitious author. Let's quickly modify the query as shown in the listing 2.8:

Listing 2.8: Searching for books written by fictitious author

```
GET books/_search
{
  "query": {
    "match": {
      "author": "Joshua Schildt" #A Searching for a fictitious author
    }
  }
}
```

When you run this query you'll get two books, one by Joshua Bloch and another one by Herbert Schildt. However, this is not correct. We want books written by "Joshua Schildt", don't it? By default the engine is implicitly adding an OR operator when creating a search query with multiple words: so it would become books authored by "Joshua" OR "Schildt".

To change this logic and add AND gate, we should define a parameter called `operator` explicitly setting it to AND, as the listing 2.9 below shows. There's a slight change to the query in that we need to add an object consisting of the query and operator to the `author` object (unlike in the listing 2.7 where we simply provided the query value to the field)

Listing 2.9: Amending the query with an AND operator to fetch exact matches

```
GET books/_search
{
  "query": {
    "match": {
      "author": { #A The author field is now having inner properties defined
        "query": "Joshua Schildt", #B provide your query here
        "operator": "AND" #C The AND operator (default is OR)
      }
    }
  }
}
```

Now that the query is amended with an AND clause, executing the query will yield no results (no books written by "Joshua Schildt"). Going with the same logic, if we wish to fetch book with an exact title, say "Effective Java", the figure 2.15 demonstrates the code:

```
GET books/_search
{
  "query": {
    "match": {
      "title": {
        "query": "Effective Java",
        "operator": "and"
      }
    }
  }
}

A match query fetching a specific title. The and operator searches for a title with both words in the title ("Effective and Java")
```

Figure 2.15 Fetching an exact match for a title using the `and` operator

As the code snippet in figure 2.15 demonstrates, we are searching for books with the exact title *Effective Java*. We would've got a list of all the books that have either of the words *Effective OR Java* in the `title` field had we not changed the operator. By providing an `and` operator to join both words, the query finds a book(s) that has both search words in the `title` field.

Before we execute a bit more involved search queries, there's one little thing we need to do: index more documents. We only have three documents so far, but it would be helpful to have more for any meaningful queries. Elasticsearch provides a convenient `bulk` API (`_bulk`) to index documents in batch. Let's do a short detour to index a few more documents, this time using a `_bulk` API.

2.2.4 Indexing more documents using the `_bulk` API

We need to add a few more documents to our store as we start warming up to a variety of search queries. We can follow the same process of using document APIs to index a few more documents as we did in section 2.1.2. However, as you can imagine, loading a lot of documents individually is a cumbersome process.

Fortunately, there's a handy `_bulk` API that lets us index documents all in one go. We can use either Kibana or cURL to execute the `_bulk` API when indexing multiple documents, albeit both methods have differences in the data format. Because we discuss the `_bulk` API at length in

Chapter 5 on Document Operations, here we will briefly go over some highlights. We will also use the datasets provided with this book, which are available at [GitHub](#).

NOTE BULK OPERATION WILL OVERRIDING THE EXISTING DATA The `_bulk` operation will index 10 books into the existing index (`books`) that was created at the beginning of the chapter. The new book documents have additional fields such as `price`, `rating` and others. Hence, the object structure was enriched with additional properties and differs from the earlier one. You can do this if you wish to stop overriding the existing index:

- Modify the index name in the bulk data file, as shown here (the new index is `books_new`):

```
{"index": {"_index": "books_new", "_id": "1"}}
```

- Alternatively, you can remove the `_index` field completely and add that to the URL:

```
POST books_new/_bulk
{"index": {"_id": "1"}}
```

Go over to the book's GitHub repository and copy the contents of the `books-kibana-dataset.txt` file and paste it into Kibana's DevTools. Figure 2.16 shows a portion of the file's content.

```

1 POST _bulk
2 {"index": {"_index": "books", "_id": "1"}}
3 {"title": "Core Java Volume I - Fundamentals", "author": "Cay S. Horstmann", "edition": 11, "synopsis": "Java reference book that offers a detailed explanation of various features of Core Java, including exception handling, interfaces, and lambda expressions. Significant highlights of the book include simple language, conciseness, and detailed examples.", "amazon_rating": 4.6, "release_date": "2018-08-27", "tags": ["Programming Languages, Java Programming"]}
4 {"index": {"_index": "books", "_id": "2"}}
5 {"title": "Effective Java", "author": "Joshua Bloch", "edition": 3, "synopsis": "A must-have book for every Java programmer and Java aspirant, Effective Java makes up for an excellent complementary read with other Java books or learning material. The book offers 78 best practices to follow for making the code better.", "amazon_rating": 4.7, "release_date": "2017-12-27", "tags": ["Object Oriented Software Design"]}

```

Figure 2.16 Bulk indexing documents for Elasticsearch using the `_bulk` endpoint in Kibana.

Once your query is executed, you should receive an acknowledgement indicating that all the documents were successfully indexed.

NOTE WEIRD LOOKING BULK INDEXING DOCUMENT FORMATS If you closely look at the documents that get loaded using `_bulk`, you will notice some weird syntax. Every two lines correspond to one document like this:

- The first line is the metadata about the record, which includes the operation (`index`, `delete`, `update`) we are about to execute (`index` in this case), the document ID, and the index where it's going to go.
- The second line is the actual document itself.

We will revisit this format a bit later in the book when we learn about document operations.

Now that we have a few more documents indexed, we can get back on the track and experiment with a few more search features. We'll begin with searching for a word across multiple fields as discussed in the next section.

2.2.5 Searching across multiple fields

When a customer searches for something in a search bar, the search doesn't necessarily restrict to just one field. For example, we want to search all the documents where the word `Java` appears, not just in the `title` field, but also in other fields like `synopsis`, `tags`, and so on. This is where we enable a *multi-field* search. We use a `multi_match` query, which searches the criteria across multiple fields.

Let's see an example where we create a query to search for `Java` in two fields, `title` and `synopsis`. Similar to a `match` query we saw earlier, Elasticsearch provides the `multi_match` query that will serve our purpose. We need to provide the search words in the inner `query` object, along with the fields that we are interested in. This is demonstrated in the listing 2.10:

Listing 2.10: Searching across multiple fields using `multi_match` query

```
GET books/_search
{
  "query": {
    "multi_match": { #A Multi match query that searches across multiple fields
      "query": "Java", #B The search words
      "fields": ["title","synopsis"] #C Searching across two fields
    }
  }
}
```

Interestingly, this search returns all books (we now have 10 books in our dataset). The following code snippet shows the JSON response (for brevity, I've reduced the list to just two books).

```
{
  ...
  "hits" : [
    ...
    "_source" : {
      "title" : "Effective Java",
      "synopsis": "A must-have book for every Java ...",
      ...
    },
    ...
    "_source" : {
      "title" : "Head First Java",
      "synopsis": "The most important selling points of Head First Java ..."
      ...
    }
    ...
  ]
}
```

As expected, we've searched across multiple fields and got our results. But, say, we want to bump up the priority of a result based on a field. For example, if `Java` is found in the `title` field, boost that search result twice while keeping the other documents at a normal priority. This is where we use the (naturally) boosting feature (users may feel happy to see the documents where the search word is found in the `title` at the top of the list).

Relevancy scores

In listing 2.3, you may have noticed numerical scoring values on the documents expressed as a `_score` attribute attached to the individual results. The `_score` is a positive floating point number indicating how relevant the resultant document is to the query. The first document returned had the highest score, while the last one scored the least. This is the relevancy score, a score that indicates how well the documents matched the query. The higher the score, the greater the match. Elasticsearch has an algorithm called Okapi Best Match 25 (BM25), which is an enhanced Term Frequency/Inverse Document Frequency similarity algorithm that calculates the relevancy scores for each of the results and sorts them in that order when presenting those to the client. As an exercise, I encourage you to execute various queries on the `books` dataset we have so you can play around with the query and `_score`.

Keep note the `_score` value of each of these results in the listing 2.x. We will revisit this and compare the same score after boosting the results. For now, let's see how we can boost results in action.

BOOSTING RESULTS

We want to give a higher priority (relevance) to certain fields when issuing a query against multiple fields. This way, even if the user doesn't specify what explicitly, we can provide the best results. Elasticsearch lets us bump up or boost priority for certain fields in our queries by providing the boost factor next to the field. That is, if we need bump up `title`'s field by factor three, we set the boost on the field as `title^3` (field followed by caret symbol and a boost factor)

This is demonstrated in the listing 2.11, where `title^3` is being bumped up by three times:

Listing 2.11 A multi_match query that boosts a field's importance

```
GET books/_search
{
  "query": {
    "multi_match": { #A We are searching through multiple fields
      "query": "Java",
      "fields": ["title^3","synopsis"]#B Caret followed by the boost number
    }
  }
}
```

Listing 2.12 shows the results, where the `title` field's score is weighted up. This means, out of all the documents, the one with the highest score appears on top of the list. Basically we bumped up the document to a higher position by boosting its score. If you look at the `_score` for the results, you'll see that these are changed from the previous query.

Listing 2.12 Results of the boosted query (the scores are boosted)

```
{
  ...
  "hits" : [
    ...
    {
      "_score" : 1.0061301,#A the score was boosted by a factor of three
      "_source" : {
        "title" : "Effective Java",
        ...
      },
      ...
      "_score" : 0.90180784,
      "_source" : {
        "title" : "Head First Java",
        ...
      },
      ...
    }
  ]
}
```

When you compare the score, the result for the book *Effective Java* is 0.33537668 before boosting, but the score rose to 1.0061301 after boosting the `title` field.

We may wish to search on a phrase, for example searching “how is the weather in London this morning” or “recipe for potato mash” etc. We use another type of query named `match_phrase` query for searching a long list of words as a phrase - coming up next.

2.2.6 Search on a phrase

At times we wish to search for a sequence of words, exactly in that order, like finding out all books that have a phrase:“must-have book for every Java programmer” amongst synopsis fields in our books. We can write a `match_phrase` query for this purpose, as the listing 2.13 demonstrates:

Listing 2.13: Searching for books with an exact phrase

```
GET books/_search
{
  "query": {
    "match_phrase": {#A The match_phrase query expects a sequence of words
      "synopsis": "must-have book for every Java programmer"#B Our phrase
    }
  }
}
```

This query will search for that sequence of words in a synopsis field across all our books and returning "Effective Java" book, as the snippet shows:

```
"hits" : [{  
  "_score" : 7.300332,  
  "_source" : {  
    "title" : "Effective Java",  
    "synopsis" : "A must-have book for every Java programmer and Java ...",  
  }}]
```

The highlighted portion in the code snippet above proves that the query has indeed successfully grabbed the book we are looking for. We may take a little detour to learn about highlighting the results.

HIGHLIGHTING THE RESULTS

While we are here, let me show you how we can highlight a portion of text in the returned document that matches our original query. For example, when you search for a word or a phrase on a blog site, usually the site will show the matching text with some sort of highlighting with colors or shades. The figure 2.17 here shows highlighting in action - we are searching for "match phrase" words on Elasticsearch's documentation site.

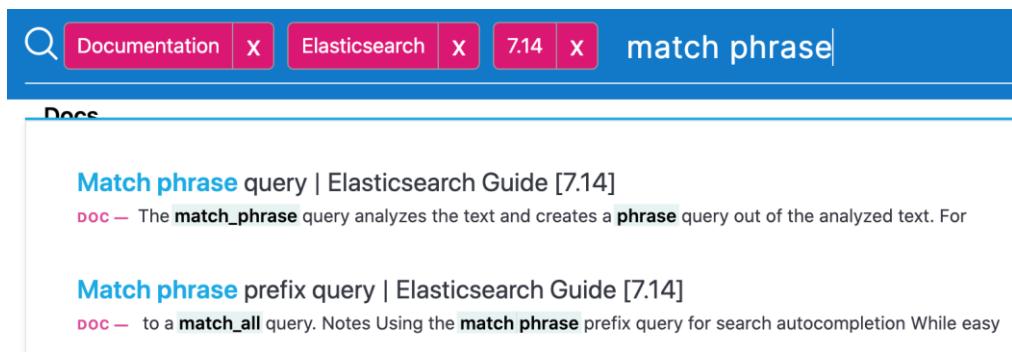


Figure 2.17: Highlighting the matches with a visual representation of colours/shadows

We can perform the same effect for our results using a convenient feature called *highlighting*. To do this, we modify our search query providing a `highlight` object in the request body at the same level as the `query` object. This is shown in the listing 2.14:

Listing 2.14 Match Phrase query with highlights

```
GET books/_search
{
  "query": {
    "match_phrase": {
      "synopsis": "must-have book for every Java programmer"
    }
  },
  "highlight": {#A The highlight object at the same level as query object
    "fields": {# B mention which fields we wish to have highlights
      "synopsis": {}
    }
  }
}
```

In the `highlight` object, we can set the fields that we wish to get the highlight applied to. For example, in the code (listing 2.14) above, we are letting the engine know to set the highlights on the `synopsis` object. The final outcome would something like this, the matches highlighted with a `html` markup tag (`em`) indicating the words are emphasised:

```
"hits" : [
  "_source" : {
    ...
    "title" : "Effective Java",
    "synopsis" : "A must-have book for every Java
  },
  "highlight" : {
    "synopsis" : [
      "A <em>must</em>-<em>have</em> <em>book</em> <em>for</em> <em>every</em> <em>Java</em>
      <em>programmer</em> and Java aspirant.."]{}}
]
```

Let's come back to our match prefix query. We know we can rely on `match_prefix` for exact phrase search. But what happens if we miss out one or two words out of the phrase. For example, does the query work if we ask it to search for "*must-have book for every Java programmer*" (dropping the word `for` from the phrase)? You can surely re-run the query (it's available on GitHub) and find out that the query doesn't yield results. To fix this, Elasticsearch provides a parameter for `match_prefix` query called `slop`, discussed in the next section:

PHRASES WITH MISSING WORDS

The `match_prefix` query expects a full phrase: a phrase without any missing words in between. However it is not always the case that we have a phrase without missing words - for example, instead of searching for "Elasticsearch in action", users may search for "Elasticsearch action". To honour this, we set `match_phrase` with a `slop` parameter. A `slop` expects a positive integer indicating how many words that the phrase is missing when searching. The query in the listing 2.15 here has been set with a `slop` of 1, indicating one word is missing in the phrase that's been searched:

Listing 2.15: Matching a phrase with missing words (using slop)

```
GET books/_search
{
  "query": {
    "match_phrase": {
      "synopsis": {
        "query": "must-have book every Java programmer", #A missing "for" word
        "slop": 1 #B The slop is set to 1, indicating one word is missing
      }
    }
  }
}
```

Executing this query will return our book as expected. You can experiment with other values of `slop` with the books dataset we have in the repository.

MATCHING PHRASES WITH A PREFIX

There is a variant to `match_phrase` query called `match_phrase_prefix`, which, in addition to matching a phrase as per `match_phrase` query, will also match the last word in the search criteria as a prefix. For example, we write a query shown in the listing 2.16 to search all books with words "Java co" in the title field:

Listing 2.16: Query to fetch all books with a title having "Java co"

```
GET books/_search
{
  "query": {
    "match_phrase_prefix": {
      "title": "Java co"
    }
  }
}
```

This query will search for all books that have a title like **Java concurrency**, **Java collections**, **Java computing** and so on. In this example, the second word is basically a prefix (wildcard) searching anything starting with co (concurrency, collections, computing etc).

NOTE UPDATE YOUR CODE The books dataset we've indexed using bulk API don't have the books on *Java Collections* or *Java Computing*, so the above query will return just *Java Concurrency in Practice* book. You can however, index new documents with just a `title` like the following to prove the query:

```
PUT books/_doc/99
{
  "title": "Java Collections Deep Dive"
}
PUT books/_doc/100
{
  "title": "Java Computing World"
}
```

Once you've indexed these ad hoc queries, re-run the above `match_phrase_prefix` query.

I have, however, updated the `kibana_scripts/ch1_getting_started.txt` with these ad hoc documents.

Sometimes we might enter incorrect spellings as our search criteria. Search engines are forgiving; they return results in spite of spelling errors. Modern search engines embrace fuzziness and, hence, provide the functionality to support the user's spelling mistakes. There's a special query, a `fuzzy` query, in Elasticsearch that corrects misspellings when searching for something, and it's coming up next.

2.2.7 Fuzzy queries

We all make spelling mistakes when searching the web. Elasticsearch provides a fuzzy query for correcting these spell mistakes in your search to some extent. The fuzzy query is based on the Levenshtein edit distance algorithm, which we will learn in a later chapter.

In the following listing 2.17, we use a `fuzzy` query to search for a keyword, `kava`. We are deliberately making a spelling mistake here as our intention was to search for `Java`.

Listing 2.17 A fuzzy query searching with a one letter spelling mistake

```
GET books/_search
{
  "query": {
    "fuzzy": {#A Fuzzy query to support spelling mistakes
      "title": {
        "value": "kava",#B The incorrectly spelt criteria
        "fuzziness": 1 #C Fuzziness 1 indicates one letter forgiveness
      }
    }
  }
}
```

We set `fuzziness` as `1` because we expect a single letter change (`k -> j`) is required to match the subject. The following code snippet shows that this `fuzzy` query returns eight of the books available in the index as all these books have `Java` in the title.

```
{
  ...
  "hits" : {
    "total" : {
      "value" : 8,
      "relation" : "eq"
    }
  ...
}
}
```

Fuzziness and Levenshtein's edit distance .

Fuzzy query searches for terms that are similar to the query by employing something called Levenshtein's edit distance. In our example, we need to replace a single letter (`k -> j`) to find out the matching documents that match with the query. The edit distance is a mechanism to change one word into another by making single character changes. It is not just changing a word, but inserting and deleting the characters to match the words is also part of the fuzziness function. For example, “kava” can be changed to “java” by replacing a single letter.

So far, we've looked at searching on unstructured (full-text) data. There are a plethora of full-text queries, we will learn about them in dedicated chapters later on in the book. There's another type of queries that we can look at - the term queries. Elasticsearch provides term queries for structured data like numericals, dates, IP address, and so on. We look at searching structured data in the next few sections.

2.2.8 Term-level queries

Elasticsearch creates a separate form of queries, known as term-level queries, to support querying structured data. Numbers, dates, range, IP addresses, etc., belong to a structured text category. Elasticsearch treats the structured and unstructured data in different ways: the unstructured (full-text) data gets analyzed, while the structured fields are stored as is.

Let's revisit our book document (listing 2.18) and cast an eye on the `edition`, `amazon_rating` and `release_date` (highlighted in bold) fields:

Listing 2.18: A sample book document

```
{
  "title" : "Effective Java",
  "author" : "Joshua Bloch",
  "synopsis" : "A must-have book for every Java programmer and Java, ...",
  "edition" : 3,
  "amazon_rating" : 4.7,
  "release_date" : "2017-12-27",
  ...
}
```

When we index the document for the first time, as we did not create the index upfront, Elasticsearch will deduce the schema by analysing the values of the fields. The `edition` field, for example, is represented as a numeric field (non-text field), hence is derived as a `long` data type by

the engine. Going by the similar logic, the `amazon_rating` field is deduced as a `float` data type due to the field having decimal values. Finally, the `release_date` is deduced as a date data type because the value is represented in ISO8601 date format (`yyyy-MM-dd`), inferring it as a date field. All these three fields are categorised as non-text fields, meaning that the values are not tokenized (split into tokens) and normalized (adding synonyms, root words etc) but stored as they are.

The set of queries that are performed on structured text are called term-level queries. They produce a binary output: fetch the result if the query matches with the criteria. They will not consider how well the documents match (relevant) instead they concentrate on whether the query has a match or not. As relevancy is not considered, the term level queries do not produce a relevancy score.

Let's look at a few term-level queries briefly in the next few sections.

FETCHING A PARTICULAR EDITION BOOK (TERM QUERY)

A `term` query is used to fetch exact matches for a value provided in the search criteria. For example, if we want to fetch all the third edition books, we can write the `term` query as the listing 2.19 shows:

Listing 2.19: Fetching third edition books

```
GET books/_search
{
  "_source": ["title","edition"], #A Only two fields are returned
  "query": {
    "term": { #B Declare the query as a term level query
      "edition": { #C Provide the field and the value as search criteria
        "value": 3
      }
    }
  }
}
```

This query returns all third edition books (we only have one book - *Effective Java*), given a snippet here:

```
"hits" : [
  ...
  "_score" : 1.0,
  "_source" : {
    "title" : "Effective Java",
    "edition" : 3,
    ...
  }
]
```

If you look carefully at the result, as I mentioned earlier already, the score is given as 1.0 by default because the term level queries are not concerned with relevancy.

THE RANGE QUERY

A `range` query fetches results that match a range: for example, fetch the flights between 1 a.m. to 1 p.m. or find the list of teenagers (aged between 14 and 19), and so on. The `range` query can be

applied to dates, numericals, and other attributes, making it a powerful companion when searching for range data.

Here, going with our book examples, we can use a `range` query to fetch all the books with an `amazon_rating` greater than 4.5 and less than 5 stars. The listing 2.20 demonstrates this:

Listing 2.20 Range query to fetch books that rate between 4.5 and 5 stars

```
GET books/_search
{
  "query": {
    "range": { #A Range query declaration
      "amazon_rating": {#B Mention the range to match
        "gte": 4.5,#C gte - greater than or equal to
        "lte": 5 #D lte - less than or equal to
      }
    }
  }
}
```

The above range query will fetch three books (we have three books whose ratings are above 4.5) as you can see in the output (I've omitted the output for brevity).

There are a handful of term-level queries such as terms, IDs, exists, prefix and others. We will go over them in more detail in search related chapters in the later part of the book.

So far, we've gone through the queries that may have been helpful when fetching the results based on some basic criteria: match on title, search a word in multiple fields, find the top-rated sellers, and so on. But, in reality, the queries can be as complex as you can imagine: for example, fetching first edition books authored by Joshua with a rating greater than 4.5 and most certainly published after 2015. Fortunately, Elasticsearch has advanced query types that we can put to use for complex criteria search in the form of *compound queries*. Let's look at an example of a compound query in the next section.

2.3 Compound queries

Compound queries in Elasticsearch provide us with a mechanism to create sophisticated search queries. Compound queries combine individual queries, called leaf queries (the ones we've seen so far), to build powerful and robust queries providing us the capability to cater to complex scenarios.

There are a handful of compound queries:

- Boolean (`bool`) query,
- Constant score (`constant_score`) query
- Function (`function_score`) score,
- Boosting (`boosting`) query
- Disjunction max (`dis_max`) query

Of all the queries, the `bool` query is the most commonly used, hence we find the `bool` query in action here.

2.3.1 Boolean (`bool`) query

A boolean, commonly called a `bool` query, is used to create a sophisticated query logic by combining other queries based on boolean conditions. A `bool` query expects the search to be built using a set of four clauses: `must`, `must_not`, `should`, and `filter`. The listing 2.21 provides the format of the `bool` query:

Listing 2.21 The format of a bool query with expected clauses

```
GET books/_search
{
  "query": {
    "bool": {#A A bool query is a combination of conditional boolean clauses
      "must": [{ }],#B The criteria must match with the documents
      "must_not": [{ }],#C The criteria must-not match (no score contribution)
      "should": [{ }],#D The query should match
      "filter": [{ }]]#E The query must match (no score contribution)
    }
  }
}
```

As you can see, the `bool` query expects one or more of these clauses (`must`, `must_not`, `should`, and `filter`) to define the criteria. More than one criteria can be expressed with a combination of these clauses. Let's briefly look at these clauses, listed in the table 2.2 here:

Table 2.2: The list of boolean (`bool`) query clauses

Clause	Explanation
must	The <code>must</code> clause houses the query where the search criteria <code>must</code> be expected to match within the documents. The positive match contributes to bumping up the relevancy score. We can build the <code>must</code> clause with as many leaf queries as possible
must_not	In a <code>must_not</code> clause, the criteria <code>must not</code> match with the documents. This clause will not contribute to the score (it is run in a filter context execution context)
should	It is not mandatory that criteria defined in the <code>should</code> clause is expected to match. However if it matches, the relevancy score is bumped up.
filter	In the <code>filter</code> clause, the criteria <code>must</code> match with the documents, similar to the <code>must</code> clause. The only difference is that the score is irrelevant in the <code>filter</code> clause. (it is run in a filter context execution context)

We'll see it in action at a high level here, but we'll discuss these queries in a dedicated chapter X on Compound Queries later in the book.

Let's say our requirement is to search for books:

- authored by Joshua
- having a rating greater than 4.7
- published after 2015

We need to employ a `bool` query for combining these criteria with the help of some of the four clauses to make this query whole. In the following sections, we'll construct the compound `bool` query for the above search criteria. Rather than introducing you to the full query in one go (which might be overwhelming), we'll break the query down into the individual search criteria before putting it all together. To begin, the next section introduces the `must` clause to fetch the authors.

2.3.2 The must (`must`) clause

We are looking to find all the books authored by Joshua, hence we can create a `bool` query with a `must` clause. Inside the `must` clause, we write a `match` query searching for books written by Joshua (we've looked at `match` queries earlier in section 2.2.3). The listing 2.22 demonstrates this code:

Listing 2.22 The bool query with a must clause for matching

```
GET books/_search
{
  "query": {
    "bool": { #A A boolean query
      "must": [{# A must clause - the documents must match to the criteria
        "match": {#A One of the queries - a match query
          "author": "Joshua Bloch"
        }
      }]
    }
  }
}
```

Notice that the `bool` query is enclosed in the `query` object. It has a `must` clause that, in turn, takes multiple queries as an array, here, matching all books written by Joshua Bloch. There should be two books (*Effective Java* and *Java Concurrency in Practice*) returned by this query, indicating that these are the only two books by Joshua Bloch in our store of documents.

What does it mean when we say that the `must` clause accepts a set of multiple queries as an array? It means we can add additional queries in the `must` clause to make it much more sophisticated. As an example for curious, here's a code (listing 2.23) with a `must` clause having two leaf queries: one a `match` query searching for the `author` and another query which is a `match_phrase` query, searching for a phrase:

Listing 2.23: The must clause with multiple leaf queries

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [{ "#A Must query with two leaf queries
        "match": { "#B A match query finding books authored by Joshua
          "author": "Joshua Bloch"
        }
      },
      {
        "match_phrase": { "#C A second query searching for a phrase
          "synopsis": "best Java programming books"
        }
      }]
    }
  }
}
```

Let's move on to add another type of clause, the `must_not` clause, which creates a negation criteria.

2.3.3 The must not (must_not) clause

Let's improve our criteria. We won't fetch Joshua Bloch's books if the ratings are below 4.7. We use a `must_not` clause for this with a `range` query that sets the rating to less than (`lt`) 4.7. The following listing demonstrates the `must_not` clause, along with the `must` clause from the original query in listing 2.24.

Listing 2.24 A bool query with must and must not clauses in action

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [{ "match": { "author": "Joshua" } }], #A Must clause
      "must_not": [{ "range": { "amazon_rating": { "lt": 4.7} }}] #B A must_not clause with a
      range query
    }
  }
}
```

This query results (see the code snippet below) in only one book, *Effective Java*, as the other book (*Java Concurrency in Practice*) is dropped from the list because it does not fit the `must_not` criteria (its rating is 4.3, which is less than the prerequisite of 4.7):

```
"hits" : [
  ...
  "_score" : 1.9459882,
  "_source" : {
    "title" : "Effective Java",
    ...
  }
]
```

In addition to searching for the books written by Joshua with rating no less than 4.7, shall we add a condition that if the books match a tag (say `tag = "Software"`), bump up the relevancy score? For this we will need to use the `should` clause, discussed in the next section.

CALLOUTS: The scores you see here in this section are indicative purposes only.

2.3.4 The `should` (`should`) clause

The `should` clause behaves like an `OR` operator. That is, the search words are matched against the `should` query, and if they match, the relevancy score is bumped up. If the words do not match, the query will not fail as it does not consider the clause. The `should` clause is more about increasing the relevancy score but not affecting the results.

In our earlier example, the document's score was **1.945982**. We can increase the score of the returned document(s) if it has a `Software` tag attached to it. This is what the `should` clause will do.

The listing 2.25 adds the `should` clause to the `bool` query. It tries to match the search text if the documents have `Software` tags attached to them.

Listing 2.25 A `should` query increases the relevancy score when a match is found

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [{"match": {"author": "Joshua"}}, {"must_not": [{"range": {"amazon_rating": {"lt": 4.7}}}]}, {"should": [{"match": {"tags": "Software"}]}]}
    }
  }
```

This query returns the result (omitting the results) with `_score` : 2.267993. The score was **1.945982** before the `should` clause was applied and is now 2.267993.

If you are curious, change the query with a non-matching word (`tags` equals `"movies"`, for example) and retry the query. The query will not fail, but the score will be the same, which proves that the `should` query affects the scoring only.

Going with the flow, the final clause is the `filter` clause that works exactly like the `must` clause except it doesn't affect the score. Let's see it in action in the next section.

2.3.5 The `filter` (`filter`) clause

Let's improvise our query a bit more, this time we'll filter out books published before 2015, i.e, we don't want any books appearing in our resultset which are published before 2015. We use a `filter` clause for this purpose, and any results that don't match the filter criteria are dropped. The following query (listing 2.26) adds the `filter` clause with a `release_date` clause:

Listing 2.26 A should clause increases the relevancy score when a match is found

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [{"match": {"author": "Joshua"}}, {"range": {"amazon_rating": {"lt": 4.7}}}],
      "must_not": [{"range": {"amazon_rating": {"gt": 4.7}}}],
      "should": [{"match": {"tags": "Software"}}, {"range": {"release_date": {"gte": "2015-01-01"}}}]
    }
  }
}
```

This query returns only one book, *Effective Java*, as this is the only book in our index that matches all three clauses in the `bool` query. Notice the unchanged score in your output - it is still 2.267993. As we learned earlier, the `filter` clause will run in a filter context, meaning the scores aren't affected.

Finally, we also wanted to bring Joshua's third edition books, right? We simply update our filter to have a term query, as shown in the listing 2.27:

Listing 2.27 The bool query with the additional filter on edition field

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [{"match": {"author": "Joshua"}}, {"range": {"amazon_rating": {"lt": 4.7}}}],
      "must_not": [{"range": {"amazon_rating": {"gt": 4.7}}}],
      "should": [{"match": {"tags": "Software"}}, {"range": {"release_date": {"gte": "2015-01-01"}}, {"term": {"edition": 3}}}]
    }
  }
}
```

The `bool` query is a Swiss army knife search tool. We'll dedicate a lot of time discussing various ways, options, and tips in the chapter on advanced searches to enhance our toolkit.

So far, we looked at searching through the data. The other side of the coin is analytics. While searching helps us find a needle in the haystack, aggregations, on the other hand, help zoom out to establish a summary of our data, like the total number of server errors in the last one hour, average book sales in the third quarter, classifying the movies by their gross earnings and so on. We will also look at aggregations and other functions in the coming chapters dedicated to aggregations, but here we'll check out some examples.

2.4 Aggregations

Analytics enables organizations to find insights into the data. So far, we've looked at searching for the documents from a given corpus of documents. Analytics is looking at the big picture and analyzing the data from a very high level to draw conclusions about it. We use aggregation APIs to provide analytics in Elasticsearch. Aggregations fall into three categories:

- *Metric aggregations*—Simple aggregations like sum, min, max, and average fall into this category of aggregations. They provide an aggregate value across a set of document data
- *Bucket aggregations*—Bucket aggregations help collect data into buckets, segregated by intervals like days, age groups, etc. These help us build histograms, pie charts and other visualizations.
- *Pipeline aggregations*—Pipeline aggregations work on the output from the other aggregations.

We'll go over a few examples on metrics and bucket aggregations in the coming pages. You will find more in-depth coverage with examples of all these categories in Chapter X on Aggregations.

Similar to the endpoint used for searching, we use `_search` endpoint for aggregations too. However, we use a new object called `aggs` (short for aggregations) in the request in place of the `query` object we've used so far. To reveal the true power of Elasticsearch, we can combine both - search and aggregations - in a single query.

To demonstrate the aggregations effectively, we need to switch gears and index a new set of data, COVID-related data for the top 10 countries. The [covid-26march2021.txt](#) file is available on the GitHub repository but here's a snippet to index using the `_bulk` API (listing 2.28):

Listing 2.28 Indexing COVID related data for aggregation exercises (using `_bulk` API)

```
POST covid/_bulk
{"index":{}}
{"country":"USA","date":"2021-03-26","deaths":561142,"recovered":23275268}
{"index":{}}
{"country":"Brazil","date":"2021-03-26","deaths":307326,"recovered":10824095}
...
```

The `_bulk` API will index 10 documents into a newly created `covid` index. As we aren't concerned with the ID of each of these documents, we've let the system generate a random one for each of them - hence you see an empty index name and ID in the index action of the API: `{"index":{}}`. Now that we have a set of documents in our COVID index, let's carry out some basic aggregation tasks, starting with metric aggregations.

2.4.1 Metrics

Metric aggregations are the simple aggregations that one would use often in our daily lives: for example, what is the average height of the students across a class? What's the minimum hedge trade? What's the gross earnings of a movie? And so on. Elasticsearch provides quite a few such metrics, most of which are self-explanatory.

Before we go to pick up the metrics in action, let's quickly go over the syntax for aggregations. Elasticsearch, similar to the search APIs provide support for aggregations using Query DSL, an example shown in the figure 2.18 below:

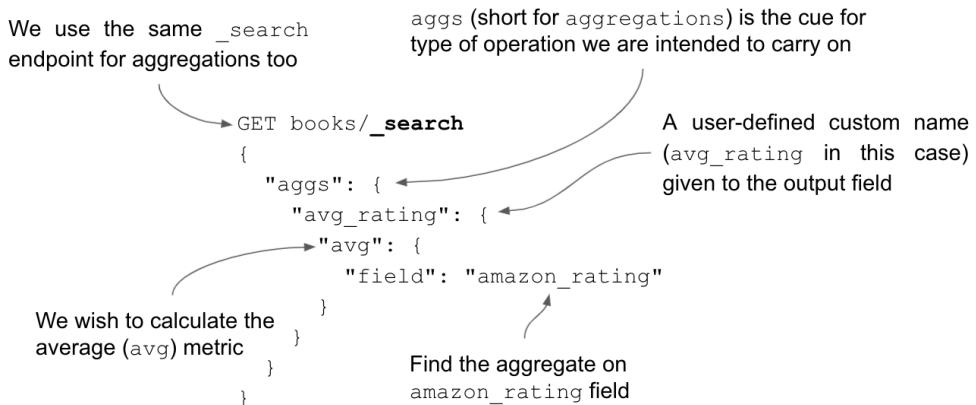


Figure 2.18: Query DSL syntax for finding average rating aggregation

As the figure 2.18 demonstrates, the aggregation queries are no different from search queries - they both are written using the same Query DSL syntax. The notable point is that the `aggs` (short for aggregations) is the root level object with an `avg` (for average) metric defined on an `amazon_rating` field. Once this query is executed, we will

Let's jump right into action executing some metric aggregations.

FINDING OUT THE TOTAL NUMBER OF CRITICAL PATIENTS (SUM METRIC)

Going back to our COVID data, let's suppose we want to find the total number of critical patients across all the top 10 countries. For that we would use a `sum` metric as described in the code listing 2.29:

Listing 2.29: Fetching the total number of critical patients

```

GET covid/_search
{
  "aggs": { #A Writing an aggregation query
    "critical_patients": { #B User defined query output name
      "sum": {#C The sum metric - sum of all the critical patients
        "field": "critical" #D The field on which the aggregation is applied
      }
    }
  }
}
  
```

With this snippet, we create an aggregation-type query, the "`aggs`" is the cue for Elasticsearch, indicating that this is an aggregation task. The "`sum`" is the type of aggregation we intend to carry out. Here we are asking the engine to find the total number of critical patients by adding them up for each of the countries. The response would be something like shown in the snippet here:

```
"aggregations" : {
  "critical_patients" : {
    "value" : 88090.0
  }
}
```

As the response indicates, the sum total of all critical patients is returned under our report's name (`critical_patients`) in the response. Note the response will consist of all documents returned if not asked explicitly to suppress them. We can set `size=0` as the root level to stop the response containing documents, as the snippet here indicates:

```
GET covid/_search
{
  "size": 0,
  "aggs": {
    ...
  }
}
```

Now that we know how the aggregations for a metric work, let's swiftly go over a couple of them in the next section.

USING OTHER METRICS

In the similar vein, if we want to find the maximum number of deaths across all countries in our COVID data, we use a `max` aggregation as demonstrated in the query listing 2.30:

Listing 2.30: Using `max` metric

```
GET covid/_search
{
  "size": 0,
  "aggs": {
    "max_deaths": {
      "max": {
        "field": "deaths"
      }
    }
  }
}
```

This query returns the the highest number of deaths among the 10 countries we have in our data set:

```
"aggregations" : {
  "max_deaths" : {
    "value" : 561142.0
  }
}
```

Similarly, we can find the minimum (`min`), average (`avg`), and others too. But there's one statistical function that returns all these basic metrics in one go: the `stats` metric, demonstrated in the listing 2.31 here:

Listing 2.31: All cores statistics in one go using stats metric

```
GET covid/_search
{
  "size": 0,
  "aggs": {
    "all_stats": {
      "stats": { #A stats query returns all five core metrics in one go
        "field": "deaths"
      }
    }
  }
}
```

This stats query returns, in one go, count, average, maximum, minimum, and others. Here's the snippet of the response from our initial stats query:

```
"aggregations" : {
  "all_stats" : {
    "count" : 20,
    "min" : 30772.0,
    "max" : 561142.0,
    "avg" : 163689.1,
    "sum" : 3273782.0
  }
}
```

If you are curious, swap the stats query with an extended_stats and check the result. You would see a lot more stats like variance, standard deviation, and others. I'll leave it to you to experiment with (the code is available on GitHub). Now that we know what metric aggregations are all about, let's briefly look at another type of aggregations - the bucketing aggregations.

2.4.2 Bucketing

Bucketing is all about segregating data into various groups or so-called buckets. For example, we can add these groups to our buckets: a survey of adult groups according to their age bracket (20–, 31–40, 41–50) or movies according to the review ratings or number of new houses constructed per month, etc.

Elasticsearch provides at least two dozen of these aggregations out of the box, each having its own bucketing strategy. What's more is that you can nest the aggregations under main buckets. Let's look at a couple of bucketing aggregations in action.

HISTOGRAM BUCKETS

The histogram bucketing aggregation creates a list of buckets on a numerical value by going over all the documents. For example, if we want to categorize countries by the number of the critical patients in buckets of 2500, we can write the query as shown in the listing 2.32:

Listing 2.32: Fetching the countries by number of critical patients in buckets of 2500

```
GET covid/_search
{
  "size": 0,
  "aggs": {
    "critical_patients_as_histogram": {#A The user-defined name of the report
      "histogram": {#B The type of the bucketing aggregation - histogram
        "field": "critical",#C The field the aggregation applied on
        "interval": 2500#D The bucket interval
      }
    }
  }
}
```

The response will be something like the following, where each bucket has a key and a value:

```
"aggregations" : {
  "critical_patients_as_histogram" : {
    "buckets" : [
      {
        "key" : 0.0,
        "doc_count" : 8
      },
      {
        "key" : 2500.0,
        "doc_count" : 6
      },
      {
        "key" : 5000.0,
        "doc_count" : 0
      },
      {
        "key" : 7500.0,
        "doc_count" : 6
      }
    ]
  }
}
```

The first bucket has four documents (countries), which lists the number of critical patients of up to 2500. The second bucket has three countries with a number of critical patients between 2500 to 5000 and so on.

RANGE BUCKETS

The `range` bucketing defines a set of buckets based on predefined ranges. For example, say we want to segregate the number of COVID casualties by country (casualties up to 60000, 60000–70000, 70000–80000, 80000–120000). We can define those ranges using this type as this example demonstrates:

Listing 2.33: Casualties by custom ranges using range bucketing

```
GET covid/_search
{
  "size": 0,
  "aggs": {
    "range_countries": {
      "range": { #A The range bucketing aggregation
        "field": "deaths", #B Field on which we apply the agg
        "ranges": [ #C Define the custom ranges
          {"to": 60000},
          {"from": 60000,"to": 70000},
          {"from": 70000,"to": 80000},
          {"from": 80000,"to": 120000}
        ]
      }
    }
  }
}
```

In the code listing 2.33, we defined a `range` aggregation bucket type with a set of custom ranges. Once the query is executed, the resultant bucket (the snippet shown below) shows the keys with the custom bucket range and values as the number of documents for that range:

```
"aggregations" : {
  "range_countries" : {
    "buckets" : [
      {
        "key" : "*-60000.0",
        "to" : 60000.0,
        "doc_count" : 2
      },
      {
        "key" : "60000.0-70000.0",
        "from" : 60000.0,
        "to" : 70000.0,
        "doc_count" : 0
      },
      {
        "key" : "70000.0-80000.0",
        "from" : 70000.0,
        "to" : 80000.0,
        "doc_count" : 4
      },
      {
        "key" : "80000.0-120000.0",
        "from" : 80000.0,
        "to" : 120000.0,
        "doc_count" : 6
      }
    ]
  }
}
```

The results indicate that there were two countries with casualties upto 60000, and 6 countries having a casualties between 80000 and 120000 and so on.

There are a rich set of aggregations that we can carry on our data using out-of-the-box statistical functions. We will run through these all in chapter X on Aggregations but until then, hold tight.

We've barely scratched the surface of what Elasticsearch can offer in this chapter. There is a lot of functionality that we will be exploring in the coming chapters. For now it's time to call it a wrap.

In the next chapter, we will go over the architecture of Elasticsearch, the mechanics of search, its moving parts, and much more.

2.5 Summary

In this chapter, we learned :

- Elasticsearch provides a set of document APIs for indexing data, and Kibana's DevTools console helps with writing the indexing queries for persistence.
- To retrieve a document using a single document API, we issue a GET command on the index with an ID (GET <index_name>/_doc/<ID>)
- To retrieve multiple documents if the document identifiers are available, you can use an `ids` query.
- Elasticsearch exposes a wide set of search APIs, ranging from basic queries to advanced.
 - Full-text queries search through the unstructured data to find the relevant documents.
 - Term queries on the other hand, sieve through structured data like numbers and dates searching for the documents for matches.
- Compound queries allow for compiling leaf queries to prepare for a more advanced set of queries. The `bool` query, one of the compound queries, provides a mechanism to create an advanced query with multiple clauses (for example, `must`, `must_not`, `should` and `filter`). These clauses help us create a sophisticated query.
- Although a search looks for matching documents with a given criteria, analytics, on the other hand, lets us aggregate data by providing statistical functions.
- Metrics aggregations fetch the common aggregations like `max`, `min`, `sum`, `avg`.
- Bucketing aggregations ensures that the documents are segregated into various buckets based on certain criteria.

3

Architecture

This chapter covers

- High-level architecture and Elasticsearch's building blocks
- Search and indexing mechanics
- How an inverted index works
- Relevancy and similarity algorithms
- Routing algorithm

In the last chapter, we played with some fundamental ElasticSearch features; we indexed some documents, executed search queries, walked through analytical functions, and more. We briefly played with the server without knowing much about its internals. The good news is that we don't need to break a sweat to get started with Elasticsearch.

Elasticsearch, like any other search engine, requires deep dives in order to become a master of the technology. That said, the product is designed to work out of the box with intuitive APIs and tools, and you can make use of the software without much prerequisite knowledge. Before we get carried away with the easy-to-use, hands-on aspects of Elasticsearch, it would benefit us in the long run if we gain an understanding of the high-level architecture, the inner workings of the server, and the dichotomy of its moving parts.

Getting a grip on the server is highly recommended for an engineer wishing to use the Elasticsearch effectively and efficiently. We may need to debug why the results are not what is expected for some queries. We may be bestowed with the task to find out the reasons for the degradation of the performance or memory issues arising on the exponential growth of the indexed data. Or, the business requirements may call for a custom language analyzer to integrate the application with another country's language. As engineers, we are expected to be able to fine tune the queries or tweak the administrative features or spin up an Elasticsearch multi-cluster farm for a business need and so on. And to gain such knowledge, one must master the technology by understanding its internal workings, its bolts and bearings, which is what this chapter is all about.

In this chapter, we'll talk through the building blocks that make up the inner workings of Elasticsearch, and we'll gain a better understanding of how the searching and indexing process works behind the scenes. We'll learn about fundamentals that underlie the search engine, such as the concept of an inverted index, relevancy, and text analysis. Finally, we'll explore the clustering and distributed nature of the Elasticsearch server. First, let's look at how the Elasticsearch engine works from a high level.

3.1 A 10,000 foot overview

Elasticsearch is a server-side application, capable of running on anything from personal PCs to a farm of computers serving gigabytes to petabytes of data. It was developed using the Java programming language with Apache Lucene under the hood.

Apache Lucene (a high performance, full-text searching library developed in Java) is well-known for its powerful searching and indexing features. However, Lucene is not a complete application that you can simply download, install, and work with. As it is a library, you are expected to integrate your application with it via the programming interfaces. Elasticsearch does exactly that: it wraps Lucene as its core full-text search library, building a distributed and scalable server-side application. Elasticsearch built a programming language-agnostic application with Lucene as the centerpiece for serving full-text searches.

Elasticsearch is more than just a full-text search engine, however. It has grown into a popular search engine with aggregations and analytics that serve various use cases (for example, application monitoring, log data analyses, web app search functions, security event capturing, machine learning, and more). It is a performant, scalable modern search engine with distributability, and speed as its primary goals.

Elasticsearch is nothing without data: the data that goes into Elasticsearch and the data that comes out. Let's spend some time looking at how Elasticsearch treats this data.

3.1.1 Data in

Elasticsearch needs data before providing us with the required answers to the queries. The data can be indexed into Elasticsearch from multiple sources and in various ways: extracting from a database, copying files from file systems, or even loading from other systems including real-time streaming systems and so on. In figure 3.1, the data is ingested into Elasticsearch from multiple sources such as database, file system and an event source.

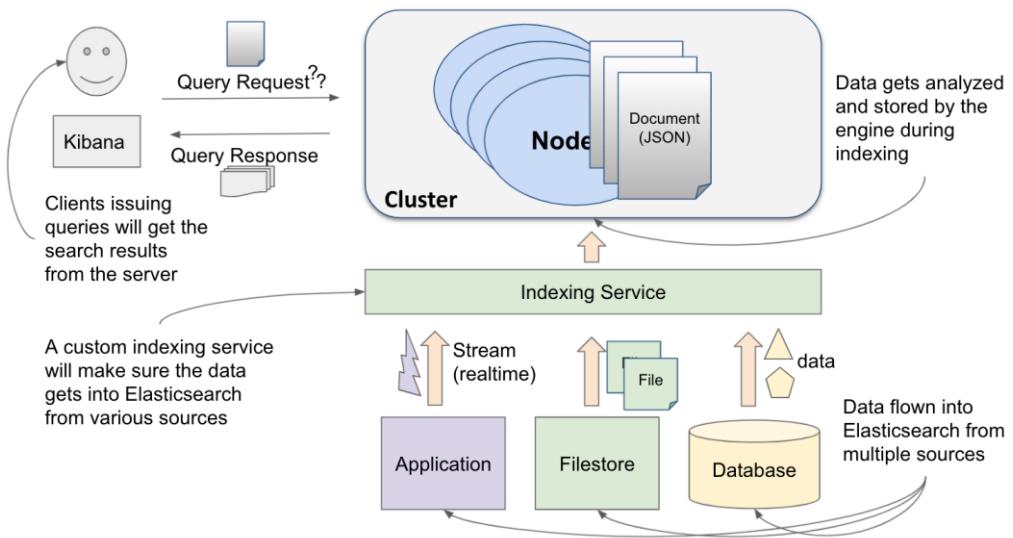


Figure 3.1 Priming Elasticsearch with data

In figure 3.1, we ingest the data into Elasticsearch via three data sources:

- *Database*—Applications usually store data in a database as an authoritative system of record. We can prime Elasticsearch with the data that's been fetched from the database either on a batch basis or near real time. As the data shape in a database might not be exactly what Elasticsearch expects, one can expect an extract, transform, load (ETL) tool (such as Logstash) to transform and enrich the data before indexing into Elasticsearch.
- *Filestore*—Applications can spit out tons of log data on a periodic basis. The data can be held in cloud locations like AWS S3. We can use tools such as Filebeat or Logstash to dump the data into Elasticsearch for search, debug, and analysis purposes.
- *Applications*—Some applications like Twitter or pricing/quotes engines emit events as streams of data. We can use Elastic Stack components like Logstash or build real-time streaming libraries that publish data to Elasticsearch, which is then used for search functionality.

Elasticsearch expects the data to be ingested from a persistent storage or in real time for it to be analyzed and processed internally. The analysis process in the form of search and analytics helps to retrieve the data efficiently. In a typical setup, organizations usually use Extract-Transform-Load (ETL) tools to transport the data into Elasticsearch. The expectation is that the ingested data will be quickly searchable once it makes it through the Elasticsearch store.

NOTE Code for this chapter is available in the book's GitHub repository:
<https://github.com/madhusudhankonda/elasticsearch-in-action/wiki/Ch-3:-Architecture>.

3.1.2 Processing the data

The basic unit of information is represented by a JSON document in Elasticsearch. For example, we can create a news article as a JSON document for a magazine as the following listing 3.1 shows.

Listing 3.1 A typical news article represented as JSON document

```
{
  "title": "Is Remote Working the New Norm?",
  "author": "John Doe",
  "synopsis": "Covid changed lives. It changed the way we work...",
  "publish_date": "2021-01-01",
  "number_of_words": 3500
}
```

COLLECTING THE DATA

Similar to storing data into a relational database, we need to persist (store) the data in Elasticsearch. To house the data, Elasticsearch creates a set of *buckets* based on each type of data. The news articles, for instance, will be housed in a bucket called *news*, which is a name we chose. In Elasticsearch lingo, we call this bucket an *index*. All the documents of a particular type (say, cars, trades, or students) end up in their own index. An index is a logical collection of documents.

NOTE INDEX IN ELASTICSEARCH IS WHAT TABLE IN A DATABASE In a relational database, we define a table to hold our records. Extending this concept, the index in Elasticsearch is equivalent to a table in a database, except that the index is a logical collection of documents, i.e., it is backed up by shards (we will touch base on shards shortly).

Data comprises various types: dates, numbers, strings, booleans, IP addresses, locations, and so on. Elasticsearch lets us index the documents by supporting rich data types. By a process called Mapping, the JSON data types are converted into the Elasticsearch appropriate data types. Mapping is a schema definition where we let Elasticsearch know how to handle our data fields. For example, the `title`, and `synopsis` fields in the news article document (listing 3.1) are text fields, while the `published_date` is a date field and `number_of_words` is a numeric field.

During the process of indexing the data, Elasticsearch analyzes the incoming data field-by-field. It analyses each field using advanced algorithms based on the mapping definitions. It then stores these fields in efficient data structures such that the data is in a form to be searched and analyzed for our purposes. The full-text fields undergo an additional process called *text analysis*, which is the heart and soul of a modern search engine like Elasticsearch. The text analysis (discussed in the next section) is a crucial step in getting our raw data into a form that enables Elasticsearch to retrieve the data efficiently while supporting a myriad of queries.

ANALYZING THE DATA

The data represented as text is analysed during the text analysis phase. The text is broken down into words (called *tokens*) using a set of rules. Fundamentally, two processes happen during the analysis process: tokenization and normalization.

Tokenization is the process of breaking the text into tokens based on a set of rules. As demonstrated in Table 3.1, the text in the `synopsis` field is broken down into individual words (tokens) separated by a whitespace delimiter:

Table 3.1 The untokenized vs tokenized synopsis field

Untokenized String (Before Tokenization)	Covid changed our lives. It changed the way we work..
Tokenized String (After Tokenization)	[Covid,changed,our,lives,it,changed,the,way,we,work]

If we search for *Covid* and *work*, for example, we'd expect the document to match our search criteria. It would be impossible to match the search criteria if the text is dumped as is without being converted to tokens.

While we've separated the words based on whitespaces in the previous example, there are a few variants of tokenizers available in Elasticsearch. We can extract tokens based on digits, nonletters, stop words, and others; we are not simply limited to whitespace. Say, for example, we want to search our data in numerous ways. Going with the same example, users could use any of the following search text:

- How did Corona changed our way of working
- How did Covid effect work
- Covid effects on working lives

During tokenization, Elasticsearch simply persists the tokens (individual words) but doesn't enhance them. That is, we could assume that the user is searching for *Covid* when they keyed in *Corona*. Unless we have enriched tokens, there's no easy way to answer the search text. That's exactly why we have another process called normalization that works on these tokens.

Normalization helps build a rich user experience by creating additional data around the tokens. It is a process of reducing (stemming) these tokens to root words or creating synonyms for the tokens. For example

- The `lives` token can be stemmed to create alternate words like `life`.
- The `covid` token can be stemmed to produce `corona`, `coronavirus`, and `sars`]

Normalization can also help build a list of synonyms for the tokens, again enriching the user's search experience. For instance, a list of synonyms for *work* might be *working*, *office work*, *employment*, or *job*].

Along with the tokens, the root words, synonyms, and others are all stored in an advanced data structure called an *inverted index*. Let's take another example of a single word, *vaccine*. It can be analyzed and stored in the inverted indexes against several root words or synonyms like *vaccination*, *vaccinated*, *immunization*, *immune*, *inoculation*, *booster jab*, and so forth.

When a user searches for *immunization*, for example, documents with *vaccine* or related words can pop up too as these words are related to each other. Should you search *where can I get immunization for covid* on Google, there is a high likelihood that your return results will be "vaccine" related. Google returned the results as expected for me (see figure 3.2).

The screenshot shows a Google search results page. The search query is "where can i get immunization for covid". The top result is a snippet from the NHS website titled "Coronavirus (COVID-19) vaccine - NHS". The snippet includes a red "COVID-19 alert" icon, a summary of the NHS offering the vaccine to high-risk individuals, and links to "Coronavirus vaccine research", "Register with a GP", and "Why vaccination is safe and effective". Below the snippet is a section titled "Common questions" with four expandable dropdowns: "Who is eligible to get the COVID-19 vaccine?", "Can I get the Covid vaccine if I had Covid?", "What is the schedule for Covid vaccines?", and "How many people are vaccinated for Covid?". At the bottom of the snippet, there is a note: "For informational purposes only. Consult your local medical authority for health advice." and a "Feedback" link.

Figure 3.2 The search results (as expected) from Google

We will learn about this analysis phase in a dedicated chapter on text analysis later in the book. But once the data has been analyzed, it is sent to a particular data node for persistence. Usually, the text analysis and persistence activities along with data replication and others are all carried out in a fraction of a second, so the data is ready for consumption in under a second after it is indexed (a Service Level Agreement (SLA) Elasticsearch vouches for). The next step after data processing in Elasticsearch is data retrieval in the form of search and analytics, which we discuss in the next section.

3.1.3 Data out

Finally, the data is collected, analyzed, and stored in Elasticsearch for it to be retrieved via search and analytical queries. Searches fetch matching data for a specified query while analytics gathers the data to form summarized statistics. Of course we can also retrieve individual documents, not a part of the search, if we have the document identifiers handy.

When a search query is issued, if the field is a full-text field, it undergoes an analysis phase similar to what was performed during the indexing of that field. That is, the query is tokenized and

normalized as per the same analyzers associated with those fields. The respective tokens are searched and matched in the inverted index and results based on the matches are relayed back to the client. If a field is set with a French analyzer, for example, the same analyzer is used during the search phase. This guarantees that the words that were indexed and inserted into inverted indexes are also matched while searching.

As with any application, Elasticsearch has a bunch of building blocks that make up the server. Let's discuss those in the next section as we will extensively work with them throughout this book.

3.2 The building blocks

In the last chapter, we indexed some sample documents and carried out some searches on them. However, we pushed the discussion on the components like indices, documents, shards and replicas for a later time. And that time is here. There are a handful of components that make up Elasticsearch; for example, documents, indices, shards, and replicas. These are the building blocks that make up the search engine. We must understand them in detail and learn their significance in shaping up the search engine.

3.2.1 Document

A *document* is the basic unit of information that gets indexed by Elasticsearch for storage. Each of the document's individual fields are analyzed for faster searches and analytics. Elasticsearch expects the documents in JSON formats. JSON is a simple human-readable data format that has gained popularity in the last few years. It represents data as a key-value pair. When we communicate with Elasticsearch over RESTful APIs, we send the queries as a JSON object to Elasticsearch. Elasticsearch, in turn, serializes these JSON documents and stores them in its distributed document store once analyzed.

PARSING THE DOCUMENT DATA

Our data, represented as JSON documents, gets parsed by Elasticsearch during the indexing process. For example, the figure 3.3 represents a JSON document for a `student` object:

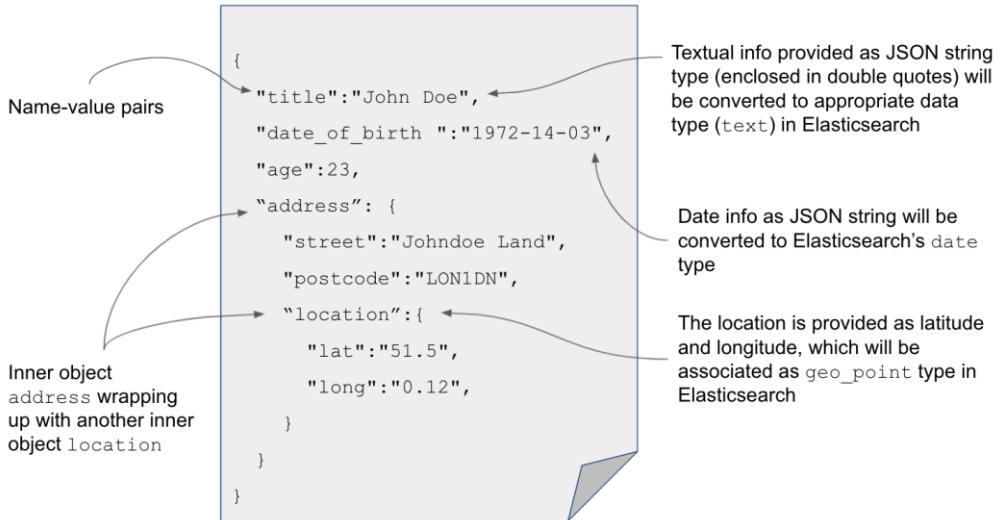


Figure 3.3 A student document represented in JSON format.

We describe the attributes of the student using the name-value pairs. The name fields are always strings in quotes while the values conform to JSON's data types (int, string, boolean, and so on). The inner objects can be represented as nested objects, which JSON supports, as the `address` field for the student document shows. Note that JSON doesn't have a date type defined but the industry practice is to provide the date and time data as a string (preferably in ISO 8601 format: `yyyy-MM-dd` or with time component `yyyy-MM-ddTHh:mm:ss`). Elasticsearch parses the stringified date information to extract the data based on the schema definition.

Elasticsearch uses the JSON parser to unmarshal the data into the appropriate types based on mapping rules available on the index. Each index will have a set of mapping rules that Elasticsearch will apply when a document is being indexed or a search query is executed. We will learn more about mapping in the coming chapters.

Elasticsearch reads the values from the JSON document and, accordingly, converts them into its own specific data types for analysis and storing. Just as JSON supports the nesting of the data (a top-level object consisting of a next-level object wrapped up again with another object underneath), Elasticsearch supports nested data structures too.

RELATIONAL DATABASE ANALOGY

If you have some understanding of relational databases, the analogy demonstrated in the figure 3.4 may help. Let's compare Elasticsearch to a database. The document in figure 3.3 is equivalent to a record in the table of a relational database shown in figure 3.4.

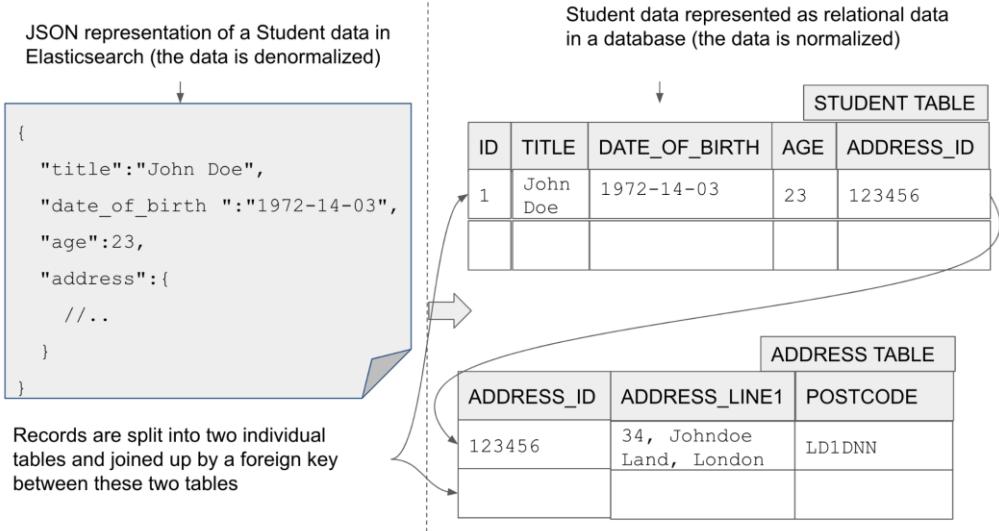


Figure 3.4 A JSON document vs. a relational database table structure

The database structures are relational. The `STUDENT` and `ADDRESS` tables are related by the foreign key `ADDRESS_ID`, which is represented in two different tables. Elasticsearch doesn't have the concept of relationships, so the whole student document gets stored into a single index. The inner objects ("address" field in figure 3.4) sits in the same index as the main fields too.

NOTE RELATIONSHIPS IN ELASTICSEARCH The data in Elasticsearch is denormalized to assist speedy search and retrieval, unlike in a relational database where the data gets normalized in various forms. While you can create parent-child relationships to some extent in Elasticsearch, it does come with potential bottlenecks and performance degradations. If your data is expected to be relational, perhaps Elasticsearch may not be the right solution.

Just as we can insert multiple records in a table, we can index several JSON documents into Elasticsearch. However, unlike in a relational database where we must create the table schema up front, Elasticsearch is just the opposite (like other NoSQL databases). It lets you insert documents without a predefined schema, a *schema-less* feature that comes quite handy during testing and development, but can also be a real problem in production environments.

DOCUMENT OPERATION APIs

As a standard across Elasticsearch, indexing documents is expected to use clearly defined document APIs. There are two types of document APIs: those that work on a single document and those that work on multiple documents in one go (or batch). You can index or retrieve documents one by one using the single document APIs or batch them up using multi-document APIs (exposed as RESTful APIs over HTTP). These are briefly described here:

- *Single document APIs*—Perform actions on individual documents, one by one. More like CRUD-related operations (create, read, update, and delete), these APIs let us get, index, delete, and update documents.
- *Multiple document APIs*—Work with multiple documents in one go. These allow us to delete and update multiple documents with a single query, index in bulk, and reindex data from source index to target index.

Each of these APIs have a specific usage, as you would learn in detail in Chapter 5: Working with Documents.

While we are on the subject of documents, there's one thing that comes up so often: the document types. Though the document types are deprecated and removed completely from version 8.x, they would eventually come on your radar when working with Elasticsearch and confuse you, especially with older (version 5.x or less) versions of Elasticsearch. So, let's take a moment and understand what they are and their current state before jumping on to learn indices.

3.2.2 Removal of types

The data we persist has a specific shape: a movie document has properties related to a movie, data for a car has properties related to cars, or an employee document has data relevant to the context of employment and business. We are expected to index these JSON documents into respective buckets or collections: movie documents consisting of movie data need to be held in an index named `movies`, for example. So, in essence we index a document of type `Movie` into `movies` index, `Car` into `cars` index and so on. That is, we can say the movie document that we indexed into Elasticsearch has a type called `Movie`. Likewise, all car documents fall under the `Car` type, and employee documents the `Employee` type, and so on.

Prior to version 5.x, Elasticsearch allowed users to index multiple types of documents in a single index. That is, a `car` index can technically have `Cars`, `PeformanceCars`, `CarItems`, `Carsales`, `DealerShowRooms`, `UsedCars`, or even other types like `Customers`, `Orders` and so on. While this sounds like a good plan to hold all car-related models in one place, there is a limitation. In a database, the columns in a table are independent of each other. Unfortunately, that's not the case for Elasticsearch. Document's fields, though they may be in various types, exist in the same index. That means a field in one type having a `text` data type cannot have a different type, say `date`, in another type. This is because of the way Lucene maintains the field types in an index. As Lucene manages fields on an index level, there is no flexibility to declare two fields of different data types in the same index.

Beginning with version 6.0, a single type per index was introduced, meaning the `cars` index is expected to hold just `car` documents. When indexing a car document, the index name followed by the type is expected (for example, `PUT cars/car/1` indexes a car document with ID 1 and puts it into a `cars` index. However, APIs were upgraded from 7.0.0: you are advised to use `_doc` as the endpoint going forward. The type of the document is replaced by a default document type `_doc`, which later on has become a permanent fixture in the url as an endpoint. Hence, the URL becomes `PUT cars/_doc/1`. As figure 3.5 shows, Elasticsearch allows us to use the types, but it throws a warning (in the figure on the right), advising us to use typeless endpoints.

The url consists of the type (car) of the document too (which is deprecated and be removed in version 8.0)

```

PUT cars/car/1
{
  "make": "Toyota",
  "model": "Avensis"
}

> V7.0: The explicit type is replaced with an endpoint named _doc (not document type)

```

```

PUT cars/_doc/1
{
  ...
}

```

#! [types removal] Specifying types in document index requests is deprecated, use the typeless endpoints instead
`(/{index})/_doc/{id}, (/{index})/_doc, or (/{index})/_create/{id}.`

```

{
  "_index" : "cars",
  "_type" : "car",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}

```

While using the type upto 7.x is allowed (you will receive a warning as shown here), it is advisable to drop the type completely

Figure 3.5 Removing document types from Elasticsearch version 8

Unless there's a strong reason, always model your data in such a way that each index has one particular data shape. When we index a document, we create a one-to-one mapping between the document and the index. That is, one index can only have one document type. Throughout the book, we follow this principle.

Now that we know our data will be presented as a JSON document to get stored in Elasticsearch, the next logical step is to find out where these documents get stored? Just as a table in a database that hosts all the records, there is a special *bucket* (or a *collection*) to hold all the documents of a particular shape in Elasticsearch. It is called an index, with details in the next section.

3.2.3 Index

We need a container to host our documents in the store. For this, Elasticsearch creates an index as a logical collection of documents. Just as we keep our documents in a filing cabinet, Elasticsearch keeps the data documents in an index, except the index is not a physical storage place; it is just a "logical" grouping. It is composed (or backed up by) of what we call shards. As the figure 3.6 demonstrates, we have a `cars` index composed of three shards on three nodes (node is an instance of Elasticsearch), one shard per node. In addition to having three shards, the index is also declared to have two replicas per shard, both hosted on other two nodes. That leads to a question: what is a shard?

Shards are the physical instances of Apache Lucene, the workhorses behind the scenes in getting our data in and out of storage. In other words, shards take care of the physical storage and retrieval of our data. Any index created by default is set to be backed up by a single shard and a

replica. Of course the index's shards and replicas can be configured and customized based on data size needs.

Shards can be categorized into primary and replicas. Primary shards are the ones that hold the original documents while replica shards (or simply replicas), as the name suggests, are copies of primary shards. We can have one or more replicas for each shard. You can have no replicas too, the design is unsuitable for production environments. In real-world production environments, once would create multiple replicas for each shard. Replicas hold data copies thus increasing the redundancy in the system; thus helping in speeding up search queries. We'll learn about shards and replicas in the next section, hang tight.

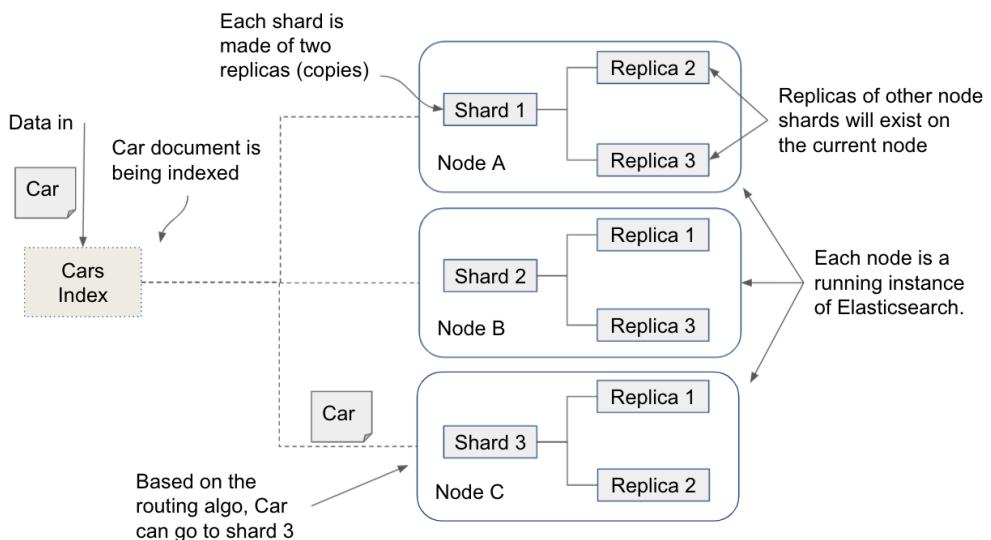


Figure 3.6 An index designed with one shard and one replica distributed across three nodes.

An index technically can hold any number of documents (but as we discussed earlier, only one document type). It is therefore advisable to find an optimum size for your own needs. Each index can either exist on a single node or distributed across multiple nodes in the cluster. As per the design we have in figure 3.6 (one index with one shard and two replicas), every document we index will be stored as three copies: one in a shard and two copies on the replicas.

NOTE Optimal index sizing must be carried out before setting up the index. The size depends on the data we may have to serve today as well as estimated size for the future. Replicas create additional copies of data, so make sure you create a capacity considering this. For example, if you have to hold 2 terabytes (TB) of data and have an index strategy of 1 shard and 2 replicas. This amounts to 6 TBs per node (1 shard = 2 TB, 2 replicas = $2 \times 2 = 4$ TB). We'll look at the sizing of an index later in the book (Chapter X : Administration of Elasticsearch).

Every index has some properties like mappings, settings, and aliases. Mapping is a process of defining schema definitions, while settings let us configure the shards and replicas. Aliases are alternate names given to a single or a set of indices. There are few settings such as changing the number of replicas, for example, that can be changed dynamically. However, some properties, like the number of shards, cannot be changed when the index is in operation. We should ideally create templates for indexes so that the creation of any new index will derive the configuration from these templates.

We use REST APIs to work on indexes, for example, creating and deleting indexes, changing their settings, closing and opening them, re-indexing the data, and other operations. We will work through these APIs extensively in Chapter X: Indexing Operations.

3.2.4 Data streams

We have been working on indices (such as `movies`, `movie_reviews` etc) which will hold and collect data over time. If the data gets huge, we could add additional indices to copy (or move) data across to accommodate. The expectation is that this type of data doesn't need to be rolled over into newer indices periodically, like hourly, daily or monthly. Keeping this in the back of your mind, let's look at a different type of data - time series data.

As the name indicates, the time-series data is time sensitive and time dependent. Take an example of logs that are generated from an apache web server, shown in the figure 3.7:

```
83.149.9.216 - - [17/May/2015:10:05:03 +0000] "GET /presentations/logstash-monitorama-2013/images/kibana-search.png HTTP/1.1" 200 203023
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:43 +0000] "GET /presentations/logstash-monitorama-2013/images/kibana-dashboard1.png HTTP/1.1" 200 17177
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:47 +0000] "GET /presentations/logstash-monitorama-2013/plugin/highlight/highlight.js HTTP/1.1" 200 26185
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:12 +0000] "GET /presentations/logstash-monitorama-2013/plugin/zoom-js/zoom.js HTTP/1.1" 200 7697
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:07 +0000] "GET /presentations/logstash-monitorama-2013/plugin/notes/notes.js HTTP/1.1" 200 2892
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:34 +0000] "GET /presentations/logstash-monitorama-2013/images/sad-medic.png HTTP/1.1" 200 430406
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:57 +0000] "GET /presentations/logstash-monitorama-2013/css/fonts/Roboto-Bold.ttf HTTP/1.1" 200 38720
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/32.0.1700.77 Safari/537.36"
```

Figure 3.7: Sample apache web server log file

The logs are continuously logged to a current day's log file. For each of the log statements, a timestamp is associated with it. At midnight, the file will be backed up with a date stamp and a new file will be created for the brand new day. The log framework will initiate the rollover automatically during the day cutover.

If we wish to hold the log data in Elasticsearch, we need to rethink the strategy of indexing the data that changes/rolls over periodically into indices. Surely, we can write an index-rollover script that could potentially rollover the indices at midnight every day. But there's more to this than just rolling over the data. For example, we also need to take care of directing the search requests against a single *mother* index rather than multiple rolling indices. We will be creating an alias for this purpose, discussed later in the book.

ALIAS: Alias is an alternate name set against a single or a set of multiple indices. The ideal way to search against multiple indices is by creating an alias (we will learn about aliases in Ch 6: Indexing Operations) pointing to multiple indices. When we search against an alias, we are essentially searching against all the indices that were backed up by this alias.

This leads us to an important concept called *data streams*, discussed in the next section in detail.

TIME SERIES DATA

Data streams accommodate time series data in Elasticsearch - they let us hold the data in multiple indices but allow access as a single resource for search and analytical related queries. As discussed earlier, the data that is tagged to a date or time axis such as logs, automated car's events, pollution levels in a city etc, is expected to be hosted in timed indices. These indices on a high level are called data streams. Behind the scenes, each of the data streams has a set of indices for each of the time points. These indices are auto generated by Elasticsearch and hidden.

The figure 3.8 shown demonstrates an example data stream for ecommerce order logs generated and captured daily. It also shows us how the order data stream is composed of auto generated hidden indices per day. The data stream itself is nothing more than an alias for the time-series (rolling) hidden indices behind the scenes. While the search/read requests are spanned across all the data stream's backing hidden indices, the indexing requests will be only directed to the new (current) index.

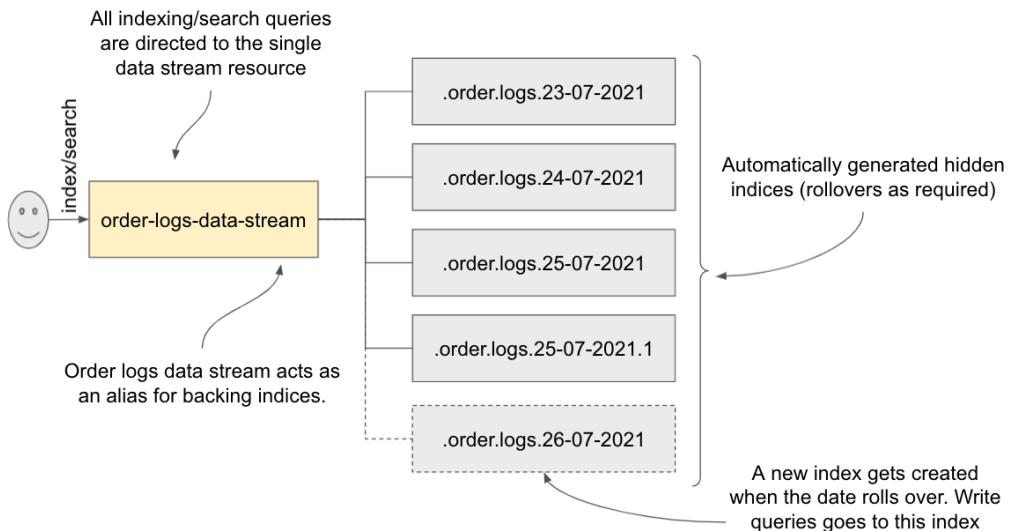


Figure 3.8 A data stream consists of automatically generated hidden indices

Data streams are created using a matching indexing template. Templates are the blueprints consisting of settings and configuration values when creating resources like indices. Indices created

from a template inherit the settings defined in the template. We will look at developing data streams with indexing templates in chapter on Indexing Operations.

In the last couple of sections, we briefly learned that an index and data streams are distributed across shards and replicas. We also went over shards, albeit briefly. It is time to dig deeper a bit about shards and replicas.

3.2.5 Shards and replicas

Shards are the software components holding data, creating the supported data structures (like inverted index), managing queries, and analysing the data in Elasticsearch. They are instances of Apache Lucene, allocated to an index during the index creation. During the process of indexing, the document travels through to the shard. Shards creates immutable file segments to hold the document on to a durable file system.

Lucene is a high-performance engine for indexing documents efficiently. A lot goes behind the scenes when indexing a document and Lucene does this highly efficiently. For example, the documents are initially copied into an in-memory buffer on the shard, then written to writable segments before merging and finalizing them to the underlying file system store. The figure 3.9 demonstrates the inner workings of a Lucene engine during indexing:

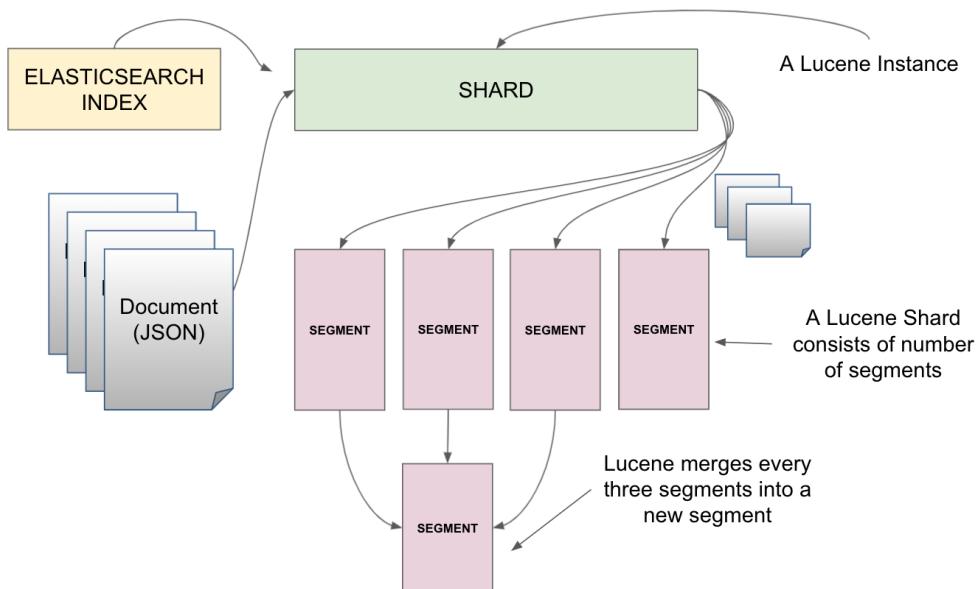


Figure 3.9 Lucene's mechanism for indexing documents

The shards are distributed across the cluster for availability and failover. On the other hand, replicas allow redundancy in the system. Once an index is in operation, the reallocation of shards cannot be carried through as this invalidates the existing data in an index.

As duplicate copies of shards, the replica shards serve the redundancy and high availability in an application. By serving the read requests, replicas enable distributing the read load during peak times. The replica of a respective shard is not co-located on the same node as the shard as it defeats the purpose of redundancy. For example, if the node crashes, you would lose all the data from the shard and its replica if they were co-located. Hence the shards and their respective replicas are distributed across different nodes in the cluster.

NOTE CLUSTER AND A NODE Cluster is a collection of nodes. A node is an instance of an Elasticsearch server. When we start an Elasticsearch server on our machine, for example, we are essentially creating a node. This node will join a cluster by default. The cluster, as it has just this node, is called a single node cluster. Starting more instances of the server will join this cluster provided the settings are correct.

DISTRIBUTION OF SHARDS AND REPLICAS

Each shard is expected to hold a certain amount of data. As we know, the data is spread across multiple shards on multiple nodes. Let's check how the shards get distributed when we start new nodes or how these diminish when we lose nodes.

Say, we have created an index `virus_mutations` for holding the Corona virus mutations' data. According to our strategy, this index will be catered by three shards (you will understand the mechanics of creating an index with a certain number of shards and replicas in Chapter 6 on Index Management but for now, let's continue the discussion on how shards get distributed.)

When we started our first node (Node A), not all shards would've been created for the index. This usually happens when the server is just starting up. Elasticsearch highlights this state of cluster as RED, indicating that the system is unhealthy. This is demonstrated in the figure 3.10 here:

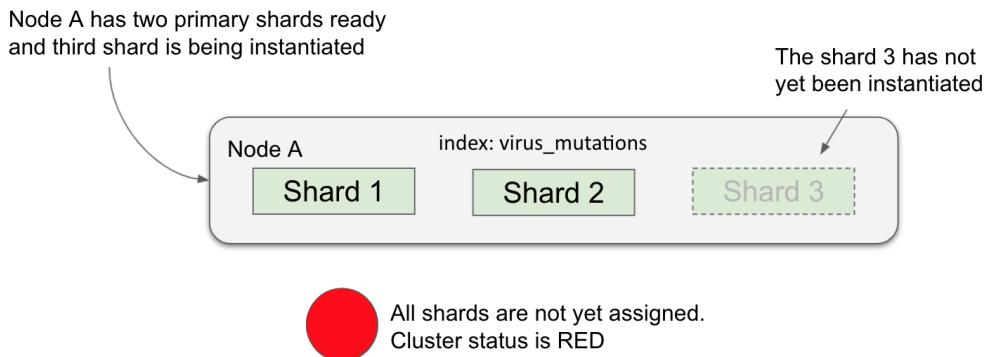


Figure 3.10: Engine is not ready showing RED status as shards are not yet fully instantiated

Once the Node A has come up, based on the settings, three shards are created on this node for the `virus_mutations` index as figure 3.11 illustrates. The Node A joins a newly created single-node cluster by default. The indexing and searching operations can begin immediately as we have all

three shards created successfully. There are no replicas created yet, however. Replicas, as you know, are the data copies and used for backup. Creating them on the same node wouldn't be the right thing to do (if this node crashes, the replica will be lost too).

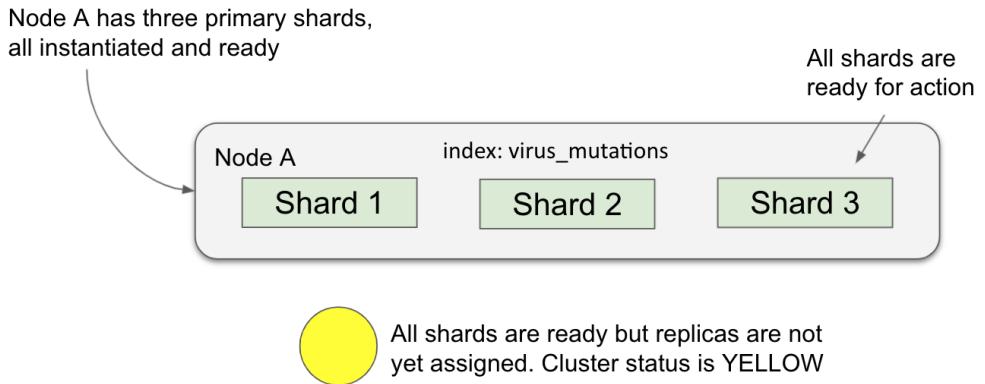


Figure 3.11 A single node with three shards joining a single-node cluster

As the replicas weren't instantiated yet, there is a high chance that we may lose the data should anything happen to this Node A. And due to this risk, the health of the cluster is set to YELLOW status.

We know that all these shards are on a single node, and for whatever the reason, if this single node crashes, we will lose everything. To avoid data loss, we decided to start a second node to join the existing cluster. Once the new node (Node B) is created and added to the cluster, Elasticsearch distributes the original three shards as follows:

- The shard 2 and shard 3 are removed from Node A
- The shard 2 and shard 3 are then moved to Node B

This move distributes our data across the cluster by adjusting the shards as figure 3.12 shows.

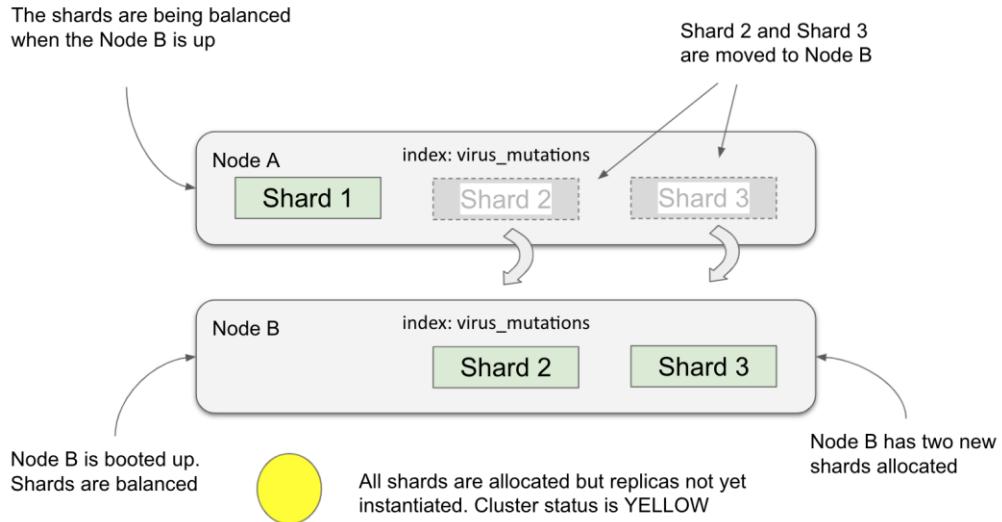


Figure 3.12 Shards were balanced on to the new node but replicas aren't assigned yet (yellow status).

As soon as the shards were created and distributed after adding the additional node, the cluster status turned yellow.

NOTE CLUSTER HEALTH ENDPOINT Elasticsearch exposes the cluster's health via an endpoint: `GET _cluster/health`. This endpoint fetches the cluster's details, especially the cluster name, status of the cluster, number of shards and replicas and others.

When the new node springs into life, there's one more thing that comes into play, in addition to distributing the shards: instantiation of replicas. A clone(s) of each shard is created and data is copied to these from the respective shards. Note that the replicas will not be sitting on the same node as the primary shard. Replica 1 is a copy of shard 1, but it is created and available on node B. Similarly, replica 2 and replica 3 are copies of shard 2 and shard 3, respectively, which are on node B, but the replicas are available on node A. Figure 3.13 demonstrates this:

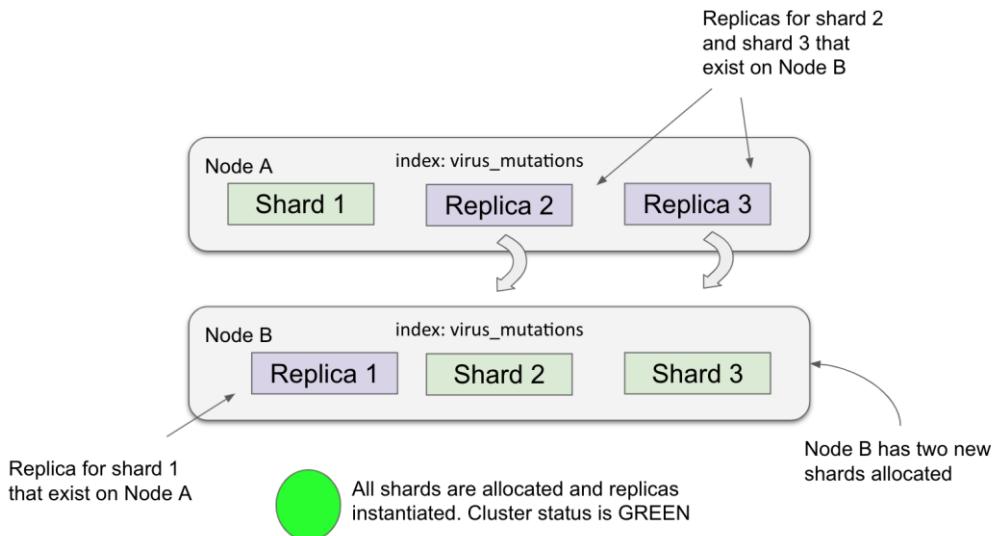


Figure 3.13 Happy days! All shards and replicas are allocated.

Both primary and replica shards are all assigned and ready, hence the cluster status is now turned to green.

HEALTH STATUS OF A CLUSTER

Elasticsearch creates a simple traffic light system based on indicators to let us know the health of the cluster at any given time. There are three states for a cluster, and the explanation follows the visual diagram, illustrated in the figure 3.14.

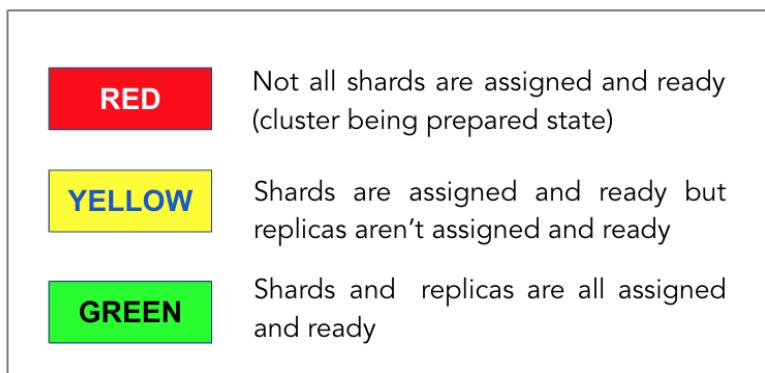


Figure 3.14: Health of shards using a traffic light signal board

Let's understand what these colours indicate:

- **Red**—This state indicates that the shards are yet to be assigned and, hence, not all the data is available for querying. This usually happens when the cluster is starting up, during which time, the shards are in a transient state.
- **Yellow**—This state indicates that replicas are yet to be assigned, but all the shards were assigned and in action. This would be more likely to occur when the nodes hosting the replicas may have crashed or are just coming up.
- **Green**—This is the happy state when all the shards and replicas are assigned and serving as expected.

Elasticsearch exposes cluster APIs to fetch the cluster-related information, including the cluster health. We can use a `GET _cluster/health` endpoint to fetch the health indicator as figure 3.15 demonstrates the result of such a call.

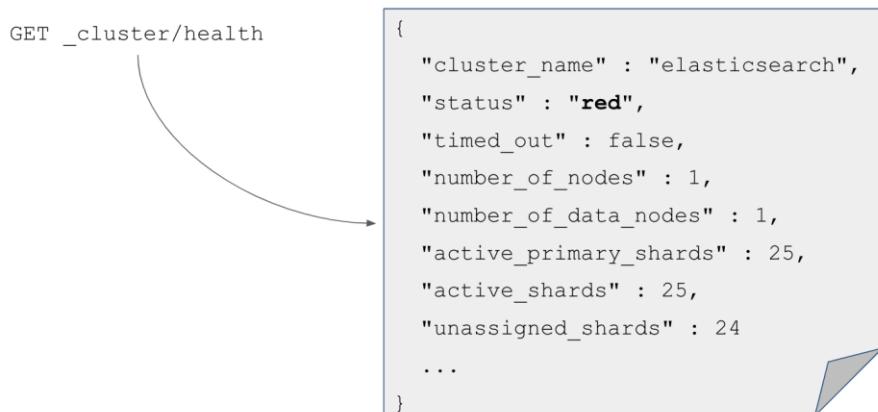


Figure 3.15 Fetching the status of a cluster by invoking the cluster's health endpoint

REBALANCING SHARDS

Of course, there's always a risk of hardware failures. In our example, what happens if Node A crashes? If Node A disappears, Elasticsearch re-adjusts the shards by promoting replica 1 to shard 1 because replica 1 is a copy of shard 1. Figure 3.16 depicts this arrangement.

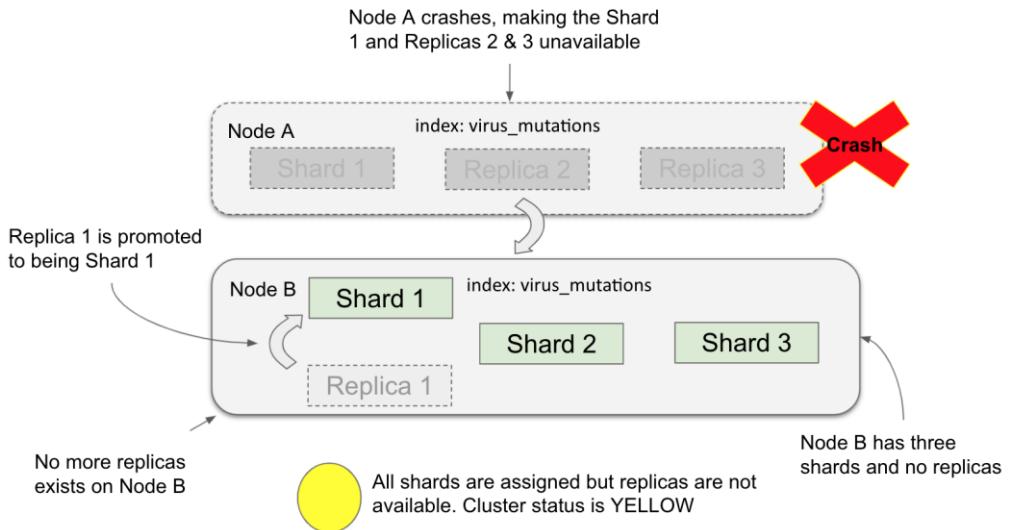


Figure 3.16 Replicas were lost (or promoted to a shard) when a node crashed.

Now the Node B is the only node in the cluster and has three shards. As the replicas aren't around, the cluster's status is set to yellow. Once the DevOps brings up the Node A, shards will get rebalanced and resassigned, and replicas get instantiated so the system tries to attain a healthy green state. Notable point is that Elasticsearch will manage such disasters and live with minimal resources if needed to avoid downtime, all behind the scenes without us having to worry about any of that operational headaches.

SHARD SIZING

A common question that gets asked often is about the sizing of the shards. There is no one-size-fits-all here. According to the organization's current data requirements and future needs, one must carry out (with due diligence) sizing trials to get a conclusive result. The industry's best practice is to size an individual shard with no more than 50 GB. But I have seen shards going up to 400 GB in size too. In fact, GitHub's indices are spread across 128 shards with 120 GB each. My personal advice is to keep them between 25 GB and 40 GB keeping the node's heap memory in mind. If we know that a movies index holds up to 500 GB data at some point, it is advised to keep this data distributed among 10 to 20 shards.

There is also one more parameter to consider for sizing the shards: the heap memory. As we know, the nodes have a limited set of computing resources such as memory and disk space. Each Elasticsearch instance can be tweaked to use heap memory based on the available memory. My advice is to host up to 20 shards per GB of heap memory. By default, Elasticsearch is instantiated with 1 GB memory, but the setting can be changed by editing the `jvm.options` file in the config directory of the installation. Tweak the `Xms` and `Xmx` properties of the JVM to set the heap memory as per your availability and need.

The takeaway here is that shards are the ones which hold our data so we must do the initial legwork to get the sizing correct. Sizing depends on how much data the index holds (including future requirements) and how much heap memory we can allocate to a node. Every organization must have a strategy for shards in place before onboarding the data. It is imperative to strike a balance between the data requirements and the optimal number of shards.

Shards cannot be modified on a live index

The number of shards cannot be changed once the index is created and is in operation. When we create an index, Elasticsearch associates a single shard and a single replica by default to it (before version 7, the default was 5 shards and 1 replica). While the number of shards is set (much like written in stone), the number of replicas can be changed using the index settings API during the course of the index's life. The documents are housed in a particular shard according to the routing algorithm:

```
shard_number = hash(document_id) % number_of_primary_shards
```

As the algorithm directly depends on the number of shards, changing the shard number during a live run will modify the current document's location and corrupt it. This in turn will compromise the inverted index and retrieval process. There's a way out of this, though: re-indexing. By re-indexing, we can change the shard settings should we want to. We will learn the re-indexing mechanism in detail later on in this book.

We will learn about shard sizing in a dedicated chapter X on Administration.

Shards and replicas make nodes. Nodes make up clusters. Let's learn about both of these in the next section.

3.2.6 Nodes and clusters

When you launch Elasticsearch, it boots up a single instance called a node. Each node hosts a set of shards and replicas (the instances of Apache Lucene). The index, a logical collection to hold our data, is created across these shards and replicas. The figure 3.17 demonstrates a single node forming a cluster.

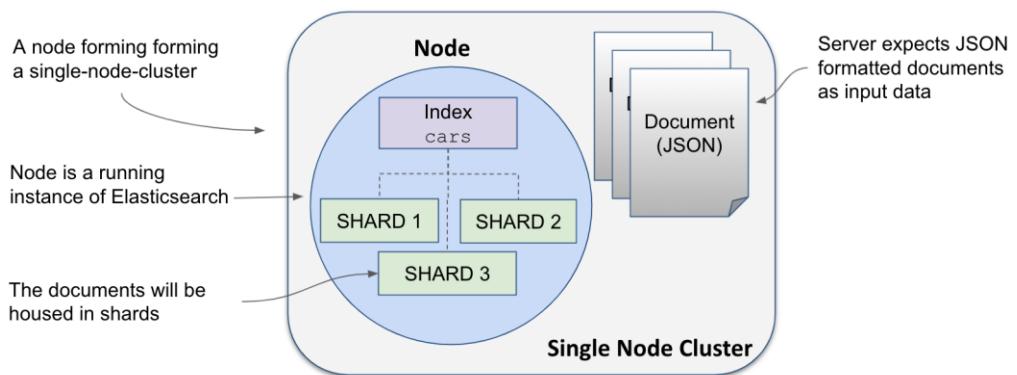


Figure 3.17 A single node Elasticsearch cluster

SINGLE NODE CLUSTER

When you boot up a node for the first time, Elasticsearch forms a new cluster, commonly called a single-node cluster. While we can suffice with a single node cluster for development purposes, it is far from a production-grade setup. In a typical production environment, we can find a farm of data nodes forming clusters (single or multiple clusters) based on the data and application search requirements. If you start another node in the same network, the newly instantiated node will join the existing cluster provided the `cluster.name` property points to the single-node-cluster. We will discuss production grade setup in Chapter X on productionalizing Elasticsearch.

NOTE A property file named `elasticsearch.yml` (the file exists in the `<INSTALL_DIR>/config` directory) mentions a property called `cluster.name`, which dictates the name of our cluster. This is an important property as all the nodes with the same name join together and form a cluster. If we are expecting a 100-node cluster, for instance, all of these 100 nodes should have the same `cluster.name` property. The default cluster name for out of the box server is `elasticsearch`, but it is a best practice to configure a unique name for the cluster.

The cluster can be ramped up by scaling up (horizontal scaling) or scaling out (vertical scaling). When additional nodes are booted up, they can join the same cluster as the existing nodes as long as the `cluster.name` property is the same. Thus, a group of nodes can form a multi-node cluster as the figure 3.18 demonstrates.

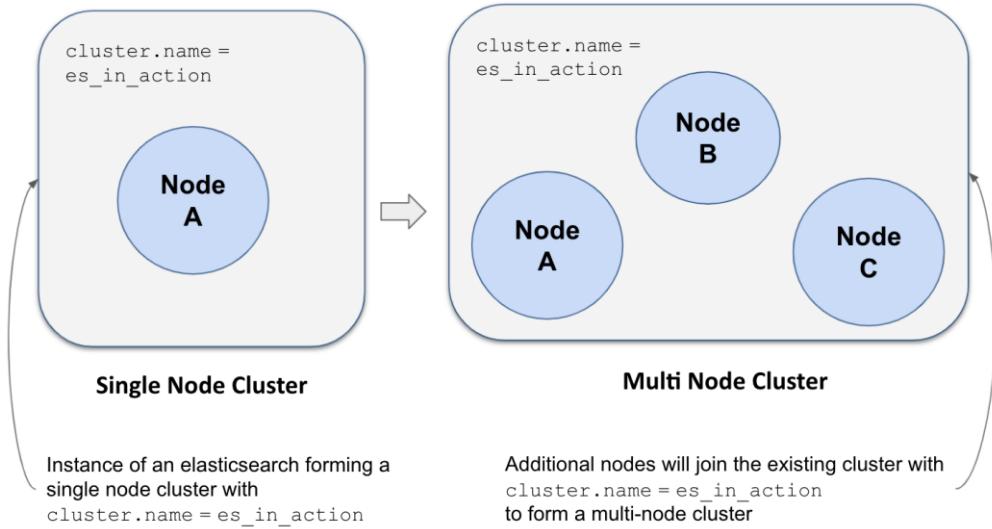


Figure 3.18 Cluster formation from a single node to a multiple node cluster

Adding more nodes to the cluster not only creates redundancy and makes the system fault tolerant, it also brings huge performance benefits. When we add more nodes, we create more room for replicas. Remember, we learned earlier that the number of shards cannot be changed when the index is in operation. So you may ask what's the benefit to shards adding nodes? What usually happens is that the data gets re-indexed from an existing index into a new index. This new index would've been configured with a new number of shards taking the additional new nodes into consideration.

Manage your node disk space judiciously .

To counteract read performance, we can add additional replicas but with that comes higher memory and disk space requirements. While it is not unusual to create clusters with terabytes or even petabytes when working with Elasticsearch, we must give forethought to our data sizing requirements.

For example, if we have a *three-shards-and-15-relicas-per-shard* strategy with each shard sized at 50 GB, we must ensure all the 15 replicas have enough capacity for not only storing the documents on disk but also for heap memory. This means:

Shards memory: $3 \times 50 \text{ GB/shard} = 150 \text{ GB}$

Replicas memory/per shard: $15 \times 50 \text{ GB/replica} = 750 \text{ GB/per shard}$

(Replicas memory for 3 shards = $3 \times 750 \text{ GB} = 2250 \text{ GB}$)

Total memory for both shards and replicas on a given node = $150\text{GB} + 750\text{GB} = 900 \text{ GB}$

(Grand total for 20 nodes = **18 TB**)

That is, a whopping **18 TB** is required for one index with *three-shards-and-15-relicas-per-shard* strategy. In addition to this initial disk space, we also need further disk space for running the server smoothly. So, we must work through the capacity requirements judiciously. We will discuss how to set up clusters with memory and disk space in a couple of chapters later in the book (Chapter X Administration and Chapter X Performance Tuning).

MULTI-NODE MULTI CLUSTERS

We briefly looked at a node as an instance of an Elasticsearch server. When you start the Elasticsearch application, we are essentially initializing a node. By default, this node joins a cluster, a one-node cluster to be precise. We can create a cluster of any number of nodes based on our data requirements. We can also create multi clusters as demonstrated in figure 3.19, but this depends on an organization's use cases.

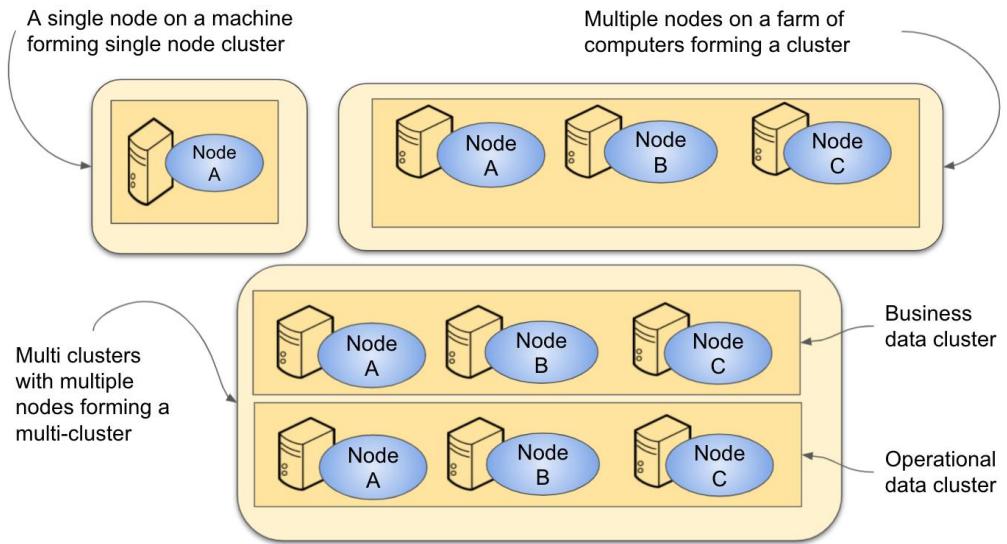


Figure 3.19 Varied cluster configurations

For example, an organization may have different types of data such as business-related data like invoices, orders, customer information, and so on. Or it may have operational information like a web server and database server logs, application logs, application metrics, and so forth.

Bundling all sorts of data into a single cluster is not unusual, but it might not be a best practice. It might be a better strategy to create multiple clusters for varied data shapes with customized configurations for each cluster. For example, a business-critical data cluster might be running on an on-premise cluster with higher memory and disk space options configured, while an application-monitoring data cluster will have a slightly different setup.

NOTE ADDITIONAL NODE ON A PERSONAL MACHINE If you are running a node on your personal laptop or PC, you can instantiate an additional node from the same installation folder. Rerun the shell script by passing two additional parameters: The `path.data` and `path.logs` options pointing to respective directories as shown here:

```
$>cd <INSTALL_DIR>bin
$>./elasticsearch -Epath.data=../data2 -Epath.logs=../log2
```

This command spins up an additional node with the given data and logs folders. If you now issue the `GET _cat/nodes` command, you should have a second node appearing in the list.

Each of these nodes need to work with specific responsibilities at times. Some may need to worry about data-related activities such as indexing and caching while others might be required to coordinate a client's requests and responses. Some tasks might also include node-to-node communications and cluster level management. To enable such responsibilities, Elasticsearch

created a set of roles for nodes so they can undertake a specific set of responsibilities when assigned to that role. Let's discuss the node roles in the next section.

Node Roles

Every node has multiple roles to play, from being a coordinator to managing data to becoming a master and so on. Elasticsearch folks have been fiddling with the nodes over time, so you may see a set of new roles appearing every now and then. Overall, table 3.2 lists the node roles that we can categorize and then, following the table, we'll take a deeper look at each.

Table 3.2 Node roles and their responsibilities

Role	Description
Master node	Its primary responsibility is cluster management.
Data node	Responsible for document persistence and retrieval.
Ingest node	Responsible for the transformation of data via pipeline ingestion before indexing.
Machine learning node	Handles machine learning jobs and requests.
Transform node	Handles transformations requests.
Coordination node	This role is the default role. It takes care of incoming client's requests.

Master Node: A master node is involved in high-level operations such as creating and deleting indexes, node operations, and other admin-related jobs for cluster management. These admin operations are light-weight processes; hence, one master is enough for an entire cluster. If this master node crashes, the cluster will elect one of the other nodes as the master so the baton continues. Master nodes don't participate in the CRUD operations of the documents, but the master node still knows the location of the documents.

Data Node: A data node is where the actual indexing, searching, deleting, and other document-related operations happen. These nodes host the indexed documents. Once an index request is received, they jump into action to save the document to its index by calling a writer on the Lucene segment. As you can imagine, they talk to the disk frequently during CRUD operations and, hence, they are disk I/O and memory-intensive operations.

NOTE DATA NODE VARIANTS There are specific variants of a data node role that will come to use when we deploy multi-tiered deployments. They are `data_hot`, `data_cold`, `data_warm`, and `data_frozen` roles. We will go over them in the chapter on Administration

Ingest node: An ingest node handles the ingest operations such as transformations and enrichment before the indexing kicks in. The documents that get ingested via a pipeline operation

(for example, processing Word or PDF documents) can be put through additional processing before getting indexed.

Machine learning node: The machine learning node, as the name indicates, executes ML algorithms and detects anomalies. It is part of a commercial license, so you must purchase an X-Pack licence to enable the machine learning capabilities.

Transform node: The transform node role is the latest addition to the list. It's used for the aggregated summary of data. This node is required for carrying out transform API invocations, which would create (transform) new indexes that are pivoted based on the existing indexes.

Coordinating node: While these roles are assigned to a node by the user on purpose (or by default), there's one special role that all the nodes take on irrespective of the user's intervention: a coordinating node. As the name suggests, the coordinator looks after the client's requests end to end. When a request is asked of Elasticsearch, one of these nodes picks up the request and dons the coordinator's hat. After accepting the requests, the coordinator asks the other nodes in the cluster for the processing of the request. It awaits the response before collecting and collating the results and sending them back to the client. It essentially acts as a work manager, distributing the in-coming requests to appropriate nodes and responding back to the client.

CONFIGURING ROLES

When we start up the Elasticsearch in development mode, the node is by default set with master, data, and ingest roles (and of course each node is by default a coordinator - there is no special flag to enable or disable a coordinator). We can configure these roles as per our needs, for example, in a cluster of 10 nodes, we can enable one node as a master, 6 as data nodes, 2 as ingest nodes and so on.

All we need to do is tweak a `node.roles` setting in the `elasticsearch.yml` configuration file to configure a role on a node. The setting takes in a list of roles, for example, setting `node.roles: [master]` will enable the node as a master node. Multiple node roles can be set as shown in the following example:

```
// This node dons four roles: master, data, ingest, and machine learning
node.roles: [master, data, ingest, ml]
```

Remember, we mentioned the coordinator role is the default role provided to all nodes. Although we set up four roles in the example (`master, data, ingest, and ml`) , this node still inherits a `coordinator` role.

You can, however, assign a `coordinator` role only to a node specifically by simply omitting the `node.roles` values altogether. In the following snippet, we assign the node with nothing but the `coordinator` role, meaning this node doesn't participate in any other activities other than coordinating requests:

```
// Leaving the roles array empty sets the node as a coordinator
nodes.roles : [ ]
```

There is a benefit for enabling a node (or set of nodes) as dedicated coordinators: they simply perform as load-balancers, working through the requests and collating the result sets. However, should you enable many nodes as just coordinators, the result outweighs the benefits. We will discuss the pros and cons of this setup later in the book.

In our earlier discussions, I mentioned that Elasticsearch stores the analyzed full-text fields in an advanced data structure called an inverted index. If there's one data structure that any search engine (not just Elasticsearch) heavily depends on, that's the inverted index. It is time to understand the internal workings of an inverted index as it would help to solidify the text analysis process, storage, and retrieval. Let's jump on the next section to dig deeper about the inverted index data structure.

3.3 Inverted indexes

If you look at the back of any book, usually you'll find an index which maps keywords to the pages where they are found. This is actually nothing but a physical representation of an inverted index.

Similar to navigating to the pages based on the page numbers associated with the keyword you are searching for in a book, Elasticsearch consults the inverted index for search words and the document association. Once the engine finds the document identifiers for these search words, it returns the full document(s) by querying the server to return them to the client. Elasticsearch uses a data structure called an *inverted index* for each of the full-text fields during the indexing phase as figure 3.20 illustrates.

Inverted Index

Word	Doc Num
hello	1, 2
world	1
mate	2

Figure 3.20 An inverted index data structure

The inverted index at a high level is a data structure much like a dictionary but with the words and the list of documents the words are present in (see figure 3.18). This inverted index is the key to faster retrieval of documents during the full-text search phase. For each document that consists of full-text fields, the server creates the respective inverted indices.

NOTE **BKD TREES** The block k-dimensional (BKD) trees are special data structures used to hold non-text fields like numeric and geo shapes.

3.3.1 Example

We learned a bit of theory about the inverted indices, but now let's look at a simple example of how it works. Say we have two documents with one text field `greeting`:

```
//Document 1
{
  "greeting": "Hello, World"
}
//Document 2
{
  "greeting": "Hello, Mate"
}
```

In Elasticsearch, the analysis process is a complex function carried out by an analyzer module. The analyzer module is further composed of character filters, a tokenizer, and token filters. When the first document is indexed, as in the `greeting` field (a text field), an inverted index is created. Every full-text field is backed up by an inverted index. The value of the greeting “Hello, World” is analyzed so that it gets tokenized and normalized into two words, `hello` and `world`, by the end of the process. But there are few steps in between.

Let’s look at the overall process (figure 3.21): The input line `<h2>Hello WORLD</h2>` gets stripped of unwanted characters such as HTML markup. The cleaned up data gets split into tokens (most likely individual words) based on whitespaces, thus forming `Hello WORLD`. Finally, token filters are applied so that the tokens can be transformed into tokens: `hello, world`. By default, Elasticsearch uses a standard analyzer, which will lowercase the tokens as is the case here. Note that the punctuation (comma) was removed during this process as well.

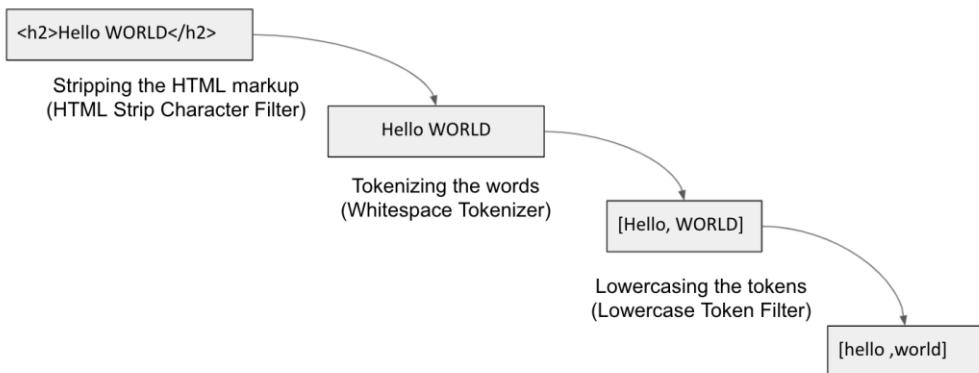


Figure 3.21 Text analysis procedure where Elasticsearch processes some text

After these steps, an inverted index is created for this field. Before digging into the details of how an inverted index is populated by Elasticsearch, let’s understand what an inverted index is.

The inverted index is a popular data structure predominantly used in full-text search functionalities. In essence, it is a hash map with words as keys pointing to documents where these words are present. It consists of a set of unique words, and the frequency of those words occur across all the documents present in the index.

Let's revisit our example. Because our document 1, "Hello, World", was indexed and analyzed, an inverted index was created with the tokens (individual words) and the documents these occur in. This is shown in table 3.3.

Table 3.3 The tokenized words for "Hello, World" and the documents they are found in

Word	Frequency	Document ID
hello	1	1
world	1	1

The words *hello* and *world* are added to the inverted index and the document IDs where these words are found (of course, it's in document ID 1). We also note the frequency of the words across all the documents in this inverted index. When the second document ("Hello, Mate") gets indexed, the data structure gets updated (table 3.4):

Table 3.4 The tokenized words for "Hello, Mate" and the documents they are found in

Word	Frequency	Document ID
hello	2	1,2
world	1	1
mate	1	2

When updating the inverted index for the word "hello", the document ID of the second document was appended and frequency of the word was bumped up. All the tokens from the incoming document are matched up with the keys in the inverted index before appending them to the data structure (like "mate" in this example) as a new record if the token is seen for the first time.

Now that the inverted index was created from these documents, when a search for "hello" comes along, Elasticsearch consults this inverted index first. As the inverted index points out that the word *hello* is present in document ID 1 and 2, the relevant documents are fetched and returned to the client.

We have oversimplified the inverted index here, but that's all right as our aim is to gain a basic understanding about the data structure. If we now search for *hello mate*, for example, both documents 1 and 2 are returned, but most likely, document 2 will have a higher relevancy score over document 1 as the contents of document 2 matches with the query.

While an inverted index is optimized for faster information retrieval, it adds additional complexity of analysis and space requirements. The inverted index grows as the indexing activity increases, thus consuming computing resources and heap space. Fortunately, we will not be working directly with the data structure, but understanding the basics helps.

The inverted index also helps deduce relevant scores; it provides the frequency of the terms, which is one of the ingredients in calculating the relevancy recipe score. We've gone over the term *relevancy* for a while, so it is now time to understand what it actually is and what algorithms a searching engine like Elasticsearch uses to fetch the relevant results to the user. We discuss the concepts relating to relevancy in the next section.

3.4 Relevancy

Modern search engines not only return results based on our query's criteria but also analyze and return the most *relevant* results. We, as developers and DevOps engineers, most likely have used Stack Overflow to search for answers to our technical problems from time to time. Searching on Stack Overflow never disappoints me, at least in the majority of the cases. The results you get for a query are close to what you are looking for. All the results are pretty much sorted in a decent order with highly relevant ones at the top to the least relevant ones at the bottom. It would be highly unlikely that the results wouldn't satisfy your requirements, but perhaps you'd not return to Stack Overflow if the results are all over the place.

3.4.1 Relevancy algorithms

Stack Overflow applies a set of relevancy algorithms to sort the results it returns to the user. Similarly, Elasticsearch returns the results for full-text queries sorted, usually, by a score it calls a relevancy score. *Relevancy* is a positive floating-point number that determines the ranking of the search results. Elasticsearch uses the BM25 (Best Match) relevancy algorithm by default for scoring the return results so the client can expect relevant results.

If you are searching for, say, *Java* in the title of a book, a document containing more than one occurrence of the word *Java* in the title is highly relevant than those in other documents where the title has one or no occurrence. Look at the example results in figure 3.22 retrieved from Elasticsearch for a query searching for keyword *Java* in a title (the full example is provided in the repository):

```

GET books/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": "Java"
    }
  }
}

```

```

"hits" : [
  ...
  "max_score" : 0.33537668,
  ...
  "hits" : [
    {
      "_score" : 0.33537668,
      "_source" : { "title" : "Effective Java" }
    },
    {
      "_score" : 0.30060259,
      "_source" : { "title" : "Head First Java" }
    },
    {
      "_score" : 0.18531466,
      "_source" : { "title" : "Test-Driven: TDD and Acceptance TDD for Java Developers" }
    }
  ]
}

```

Figure 3.22 Relevant results for Java in a title search

The first result shows a higher relevancy score (0.33537668) than the second and third results. All titles have *Java* in them, so to calculate the relevancy score, Elasticsearch utilized the field norm length algorithm: the search word in the first title (*Effective Java*) containing two words is more relevant than the second title (*Head First Java*) consisting of three words. The `max_score` is the highest score of all the available scores, usually the score of the first document.

Relevance scores are generated based on the similarity algorithms employed. In our case, Elasticsearch by default applied the BM25 algorithm to determine the scores. This algorithm depends on term frequency, inverse document frequency, and the field length, but Elasticsearch allows much more flexibility in providing a handful of algorithms in addition to the BM25 algorithms. These algorithms are packaged in a module called `similarity`, which ranks matching documents. Let's look at some of these similarity algorithms in the next section.

3.4.2 Relevancy (similarity) algorithms

Elasticsearch employs a handful of relevance algorithms, the default being the Okapi Best Matching 25 (BM25) algorithm. Elasticsearch provides a module called `similarity` that lets us apply the most appropriate algorithms if the default isn't suited for our requirements.

The similarity algorithms are applied per field by using mapping APIs. Because Elasticsearch is flexible, it allows customized algorithms as per our requirements as well. This is a pretty advanced feature so, unfortunately, it is out of scope for this book. Table 3.5, however, provides a list of available algorithms out of the box:

Table 3.5 Elasticsearch's similarity algorithms

Similarity algorithm	Type	Description
Okapi BM25 (default)	BM25	An enhanced TF/IDF algorithm that considers field length in addition to term and document frequencies
Divergence from Randomness (DFR)	DFR	Uses the DFR framework developed by its authors Amati and Rijsbergen
Divergence from Independence (DFI)	DFI	
LM Dirichlet	LMDirichlet	
LM Jelinek-Mercer	LMJelinekMercer	
Manual scripting		Creates a manual script
Boolean similarity	boolean	Does not consider ranking factors unless the query criteria is satisfied or not

We will briefly go over the BM25 algorithm, which is a next generation enhanced term-frequency/inverse-document-frequency (TF/IDF) algorithm in the next section.

THE OKAPI BM25 ALGORITHM

There are three main factors involved in associating a relevancy score with the results. The term frequency (TF), inverse document frequency (IDF), and field length norm. These three factors dictate the relevancy score of the result set. Let's look at these factors briefly and learn how they affect relevancy.

Term frequency (TF)

Term frequency (TF) represents the number of times the search word appears in the current document's field. If we search for a word (for example, *Java*) in a title field, the number of times the word (*Java* in our case) appears is denoted by the term frequency variable. The higher the frequency, the higher the score. Say, for example, that we are searching for "Java" in a title field across three documents. When indexing, we created the inverted index with the similar information: the word, number of times that word appeared in that field (in the document), and the document IDs. We can create a table with this data as table 3.6 shows.

Table 3.6 Term frequency (TF) for a search keyword

Title	Frequency	Doc ID
Mastering Java: Learning Core Java and Enterprise Java With Examples	3	25
Effective Java	1	13
Head First Java	1	39

As the table indicates, *Java* appears three times in a document with ID 25, while one time in the other two documents. As the table suggests, the search word appears more times in the first document (ID is 25). It is logical to consider that the document ID with 25 is our favorite. Remember, the higher the frequency, the greater the relevance.

While this number seems to be a pretty good indication to deduce the most relevant document in our search result, it is often not enough. There's another factor, inverse document frequency, which when combined with term frequency produces improved score. Let's discuss the inverse document frequency next.

Inverse document frequency (IDF)

The number of times the search word appears across the whole set of documents (i.e., across the whole index) is the document frequency. If the document frequency of a word is higher, we can deduce that the search word is indeed common across the whole index. This means that if the word appears multiple times across all the documents in an index, it is a common term and, accordingly, it's not that relevant.

The words that appear often are not significant. Words like *a*, *an*, *the*, *it*, and so forth are pretty common in a natural language; hence, they can be ignored. The inverse of the document frequency (called inverse document frequency or IDF) provides a higher significance to uncommon words across the whole index. Hence, *the higher the document frequency, the lower the relevancy*. Table 3.7 shows the relationship of the word frequency to relevance.

Table: 3.7 Relationship between word frequency and relevance

Word frequency	Relevancy
Higher term frequency	Higher relevancy
Higher document frequency	Lower relevancy

NOTE STOP WORDS The words such as 'the', 'a', 'it', 'an', 'but', 'if', 'for', 'and' etc are called stop words and are removed by using a stop filter plugin. However, the default standard analyzer doesn't have the `stopwords` parameter enabled (the `stopwords` filter is set to `_none_` by default) so you would see these words analyzed.. We can enable the stop words filter by simply adding the parameter `stopwords` set to `_english_`, as shown in the code snippet below:

```
PUT index_with_stopwords
{
  "settings": {
    "analysis": {
      "analyzer": {
        "standard_with_stopwords_enabled": {
          "type": "standard",
          "stopwords": "_english_"
        }
      }
    }
  }
}
```

We will learn about customizing the analyzers in chapter on Text Analysis, stay put till then.

Up until version 5.0, Elasticsearch used the TF-IDF similarity function to calculate the scores and to rank the results. Unfortunately, the TF-IDF algorithm didn't take the field's length into consideration. For example, which document from the following list is more relevant to the search criteria:

- A field with 100 words having 5 occurrences of a search word
- A field with 10 words with 3 occurrences

If you reason logically, it might be obvious that the second document might be the most relevant document as it has more search words in a short field length. This factor is called the field-length *norm*. Elasticsearch improved their similarity algorithms by enhancing the TF-IDF with an additional parameter: field length. This is what constitutes the BM25 algorithm in the first place. In the next section, we will go over the final factor: field length.

Field length norm

The field-length norm provides a score based on the length of that field: the search word occurring multiple times in a short field is more relevant. For example, the word *Java* appearing once over a long synopsis might not be the book you might think it is. On the other hand, as demonstrated in table 3.8, the same word appearing twice or more in the title field (with fewer words) says that the book is more about the Java programming language.

Table 3.8 Comparing different fields to gather the similarity

Word	Field length	Frequency	Relevant?
Java	Synopsis field with length of 100 words	1	No
Java	Title field with length of 5 words	2	Yes

For most of the cases, the BM25 algorithm would be adequate. However, if we need to swap BM25 with another algorithm, we can surely do so by configuring it using the indexing APIs. Let's go over the mechanics of configuring the algorithm as per our need.

CONFIGURING SIMILARITY ALGORITHMS

NOTE ADVANCED TOPIC: SKIP IT IF YOU WANT TO Working with similarity algorithms is an advanced topic. While I would advise you to go over this section on configuring various similarity algorithms, you can skip this section and revisit when you wish to know more about them. This way your journey in learning Elasticsearch may be less bumpy.

Elasticsearch provides us with the ability to plug in other similarity algorithms if the default BM25 doesn't suit our requirements. There are two similarity algorithms that are provided out of the box without any further customization: BM25 and boolean similarity functions. We can set the similarity algorithm for individual fields when we create the schema definitions using index setting APIs as demonstrated in figure 3.23.

```
PUT index_with_different_similarities
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "similarity": "BM25"
      },
      "author": {
        "type": "text",
        "similarity": "boolean"
      }
    }
  }
}
```

Figure 3.23 Creating an index with two fields by setting both fields with different similarity functions

Tweaking the similarity functions

Each of the similarity functions carry additional parameters so we can alter them to reflect precise search results. For example, although the BM25 function is already set with the optimal parameters, should we want to modify the function, we can do that easily by using the index settings API. There are two parameters that we can change in BM25 if we need to; these are `k1` and `b` and are described in table 3.9.

Table 3.9 Available BM25 similarity function parameters

Property	Default Value	Description
<code>k1</code>	1.2	Nonlinear term frequency saturation variable
<code>b</code>	0.75	TF normalization factor based on the document's length

Let me give you an example. Figure 3.24 shows an index with a custom BM25 similarity function, where the core BM25 function is amended with our own settings of `k1` and `b`.

```
PUT my_bm25_index
{
  "settings": {
    "index": {
      "similarity": {
        "custom_BM25": {
          "type": "BM25",
          "k1": "1.1",
          "b": "0.85"
        }
      }
    }
  }
}
```

The annotations explain the purpose of each part of the JSON code:

- A callout points to the top level of the JSON object with the text: "Configure an index with specific BM25 parameters".
- A callout points to the "custom_BM25" object with the text: "We create a custom similarity function with modified BM25 algorithm".
- A callout points to the "k1" and "b" fields within the "custom_BM25" object with the text: "Setting k1 and b values as per our requirements".

Figure 3.24 Creating a custom BM25 similarity function

In figure 3.24, we create a custom similarity type, albeit a tweaked version of BM25, which can then be reused elsewhere. Once the similarity function is created, we can use it when setting up a field. This is demonstrated in figure 3.25.

```
PUT books/_mapping
{
  "properties": {
    "synopsis": {
      "type": "text",
      "similarity": "custom_BM25"
    }
  }
}
```

Figure 3.25 Creating a field in an index with a custom BM25 similarity function

In figure 3.25, we create a mapping definition, assigning our custom similarity function (`custom_BM25`) to a synopsis field. When ranking the results based on this field, Elasticsearch considers the provided custom similarity function to apply the scores.

Similarity algorithms are a beast

Information retrieval is a vast and complex subject. The algorithms involved to score and rank results are pretty advanced and complicated. While Elasticsearch provides a handful of these similarity algorithms as plug and play, you may need a deeper understanding when working with them. Once you have tweaked the configurational parameters of these scoring functions, make sure that you have tested and tried every possible combination.

You may wonder how Elasticsearch can retrieve the documents in a fraction of a second? How does it know where in these multiple shards the document exists? The key is the routing algorithm, discussed in the next section.

3.5 Routing Algorithm

Every document has a permanent home, i.e., it must belong to a particular primary shard. Elasticsearch uses a *routing algorithm* to distribute the document to the underlying shard when indexing. Routing is a process of allocating a home for a document to a certain shard with each of the documents stored into one and only one primary shard. Retrieving the same document will be easy too as the same routing function will be employed to find out the shard where that document belongs to.

The routing algorithm is a simple formula where Elasticsearch deduces the shard for a document during indexing or searching, as shown here:

```
shard_number = hash(id) % number_of_shards
```

The output of the routing function is a shard number. It is calculated by hashing the document's id and finding out the remainder of the hash when divided (using modulo operator) with the number of shards. The `hash` function expects a unique id, generally a document ID or even custom

ID provided by the user. The documents are evenly distributed so there is no chance of one of the shards getting overloaded.

If you notice, the formula directly depends on the `number_of_shards` variable. That means, once an index is created, we cannot change the number of shards. If we are allowed to change the settings (say from 2 shards we change the number to 4), the routing function will have broken for the existing records and data wouldn't have been found. This is the reason why Elasticsearch wouldn't allow us to change the shards once an index was set up.

What if we have not anticipated the data growth and unfortunately the shards may have been exhausted with the spike in data. Well, all is not lost, there's a way out - *reindex* your data. Reindexing effectively creates a new index with appropriate settings and copies the data from the old index to a new index.

NOTE REPLICAS CAN BE ALTERED ON AN OPERATIONAL INDEX While the shard number cannot be changed when the index is in operation, we can alter the number of replicas should we wish to. Remember the routing function is a function of the number of primary shards not the replicas. However, if we need to change the shard number for whatever the reason, we must close the indices (closed indices will be blocking all the read and write operations), change the shard number and reopen the index.

One of the major goals of Elasticsearch is the scalability of the engine. In the next section, we will glance over the scalability of Elasticsearch at a high level: how to scale, how a vertical and horizontal scaling works, and the reindexing process. We will not be delving into the nitty gritties of the scaling solutions here because we will come back to these discussions later in the book.

3.6 Scaling

When Shay Banon rewrote Elasticsearch from the ground up, one of his goals was to make sure the server scales effortlessly. That is, if the data increases or the query load escalates, adding additional nodes should solve the problems. Of course, there are other solutions such as vertical scaling, performance tuning, and more. Based on the need, predominantly, there are two schools of scaling: scale horizontally or scale vertically, and Elasticsearch supports both of them.

3.6.1 Scaling up (vertical scaling)

In a vertical scaling scenario, we do not go and buy additional virtual machines (VMs), but instead add additional computing resources such as extra memory, CPU, I/O, to the existing machines. For example, we may undertake measures such as increasing the CPU cores, doubling its memory and others.

There is also another way to enhance the power of the cluster. As there is additional room now, we can install additional nodes on the machine, thus creating multi nodes in the single (fat) machine.

Remember that scaling up requires the cluster to be shut down, so there could be potentially a downtime to your application unless you want to lean on a traditional disaster recovery (DR) model. This brings up your secondary or backup system, servicing the clients while your primary system is undergoing maintenance.

There is also a bit of risk attached to this sort of scaling though. Should the whole machine crash in an emergency or due to a hardware failure, there is a chance the data could be lost because all the nodes hosting the data existed on the same machine. Of course, we will have had our backups so restoration, though painful, is possible.

Essentially, our replicas are hosted on the same machines, albeit different nodes, which is asking for trouble.

3.6.2 Scaling out (horizontal scaling)

Alternatively, we can scale out (horizontal scaling) our environments. Instead of attaching additional RAM and memory to the existing machines, we can throw in a number of new machines (most probably VMs with less resource power compared to the fat machines we used for vertical scaling) to form a horizontally-scaled farm.

These new VMs will be booted up as new nodes, thus joining the existing Elasticsearch cluster. As soon as they join the cluster, Elasticsearch, being a distributed architected engine, takes advantage of distributing the data to new nodes instantly. As creating VMs is easy, especially using the modern infrastructure as code (IaC) tools like Terraform, Ansible, Chef, and others, this approach tends to be favored by many.

That's pretty much it for this chapter, so let's wrap it up here. This chapter is your guide to understanding the fundamentals of Elasticsearch, its moving parts, low-level blocks, and search concepts. We will learn more about the concepts and fundamentals along with examples in the next two chapters, so stay tuned.

3.7 Summary

- Elasticsearch expects the data to be brought in to get indexed. The data sources can vary from a simple file to a database to a live stream to a twitter and so on.
- In the indexing process, the data undergoes a rigorous analysis phase during which advanced data structures like inverted indices are created.
- Data is retrieved or searched via the search APIs (along with the document APIs for single document retrievals).
- The in-coming data must be wrapped up in a JSON document. Because the JSON document is the fundamental data-holding entity, it gets persisted to shards and replicas.
- Shards and replicas are the Apache Lucene instances whose responsibility is to persist, retrieve, and distribute documents.
- When we startup the Elasticsearch application, it boots up as a lone-node, single cluster application. Further addition of nodes expands the cluster with these new nodes, making the cluster a multi node cluster.
- For faster information retrieval and data persistence, Elasticsearch develops advanced data structures like inverted indices for structural data such as textual information and BKD trees for non-structural data such as dates, numbers, etc.
- Relevancy is a positive floating-point score attached to retrieved document results. It defines how well the document matches the search criteria.
- Elasticsearch uses the Okapi Best Match (BM) 25 relevancy or similarity algorithm, which is an enhanced version of the Term Frequency/Inverse Document Frequency similarity algorithm.
- You can scale Elasticsearch up or out, based on your requirements and availability of resources. Scaling up beefs up the existing machines (adding additional memory, CPU, RAM, etc.), while scaling out spins up more virtual machines (VMs) and allows them to join the cluster and share the load.

4

Mapping

This chapter covers

- Filed data types
- Implicit and explicit mapping
- Core data types
- Advanced data types
- APIs to create/access the mappings

Data is like a rainbow—it comes multi-coloured. Business data comes in various shapes and forms, usually represented as textual information, dates, numbers, inner objects, booleans, geo-locations, IP addresses, and so on. In Elasticsearch, we model and index data as JSON documents. Each document consists of a number of fields, and every field has a certain type of data. For example, a movie document consists of a title and synopsis represented as textual data, a release date as date, gross earnings as floating-point data, and so on.

In our earlier chapters, when we indexed sample documents, we did not bother about the data type of the fields. That's because Elasticsearch derived these types implicitly by looking at the field and the type of information in it. Elasticsearch created a schema for us without us having to do any up front work unlike in a relational database. It is mandatory to have the table schema defined and developed in a database before proceeding to retrieve or persist the data.

Elasticsearch, on the other hand, does not stop you from priming it with documents without having to define a schema for your data model. This schema-free feature helps developers get up and running with the system from day one. Best practice is to develop a schema upfront rather than letting Elasticsearch define it for us unless your requirements are not needed to have one. Elasticsearch, however, expects us to provide clues as to how it can treat a field when indexing the data. These clues are either provided by us in the form of a schema definition while creating the index or derived by the engine implicitly if we allow it to do so. *This process of creating the schema definition is called mapping.*

Mapping allows Elasticsearch to understand the shape of the data so it can aptly apply a set of predefined rules on the fields before indexing them. Elasticsearch will also consult the manual of mapping rules to apply full-text rules on text fields. Structured fields (otherwise called exact values, like numbers or dates) have a separate set of instructions that will enable them to be part of aggregations and sorting and filtering functions, in addition to being available for general searches.

In this chapter, we set the context for using mapping schemas, explore the mapping process, and work with data types, looking into how to define those using the mapping APIs. Data that gets indexed for Elasticsearch has a definite shape and form. The meticulous shaping of data lets Elasticsearch do a faultless analysis, thus providing the end user with precise results. This chapter is about understanding the treatment of data in Elasticsearch and how the mapping schemas help us avoid hindrances for an accurate search.

NOTE COPY THE FULL CODE TO YOUR KIBANA To make the coding exercises easy, I've created a [ch4_mapping.txt](#) file under the kibana_scripts folder at the root of the repository. Copy the contents of this file to your Kibana as-is. You should be able to work through examples by executing the individual code snippets alongside following the chapter's contents.

Of course, if you wish to want running commentary, follow the wiki page for the chapter [hser](#)

NOTE 100M RUN OR 400M HURDLES? This chapter deals with dozens of hands-on examples around both core and advanced data types. While I would advise you to go over the details in the given order, if you are just starting up with Elasticsearch and wish to focus on the beginner elements then you can skip Section 4.6 on Advanced Data Types for now and revisit once you are a bit more confident and would like more to chew. If all you really want is a 100m run rather than 400m hurdles, then follow the chapter up to core data types (Section 4.4) and feel free to jump to the next chapter.

4.1 Overview of mapping

Mapping is a process of defining and developing a schema definition representing the document's data fields and their associated data types. Mapping tells the engine the shape and form of the data that's being indexed. Being a document-oriented database, Elasticsearch expects a single mapping definition per index. Every field is treated as per the mapping rule. For example, a string field is treated as a text field, a number field is stored as an integer, a date field is indexed as a date to allow for date-related operations, and so on. Accurate and error-free mapping allows Elasticsearch to analyze the data faultlessly, aiding in search-related functionalities, sorting, filtering, and aggregation.

4.1.1 Mapping definition

Every document consists of a set of fields representing business data, and every field has a specific data type (or more) associated with it. The mapping definition is the schema of the fields, and their data types of a document. Based on the data type, each of these fields are stored and indexed in a specific way. This helps Elasticsearch to support a multitude of search queries such as full-text,

fuzzy, term, geo, and so on. Figure 4.1 shows a student document and the data types associated with it.

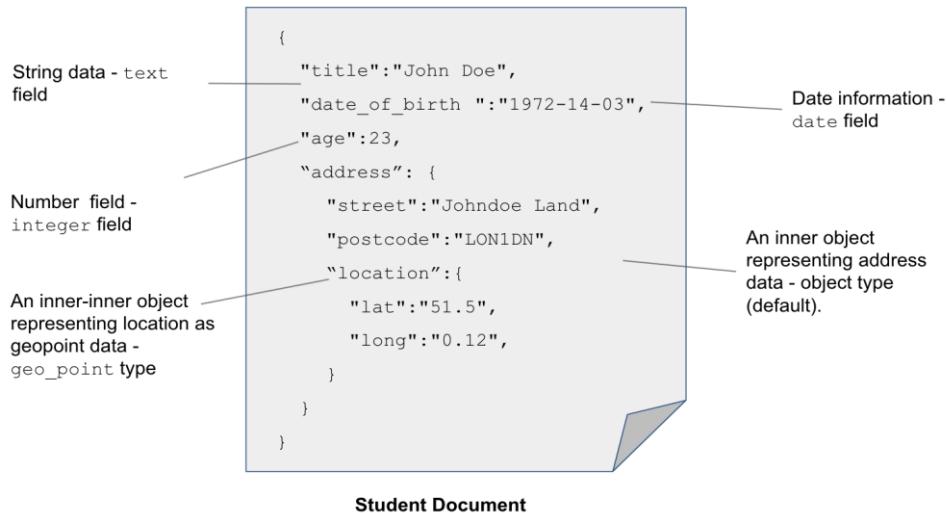


Figure 4.1 An example of a student document with the data represented in JSON

In programming languages, we represent data with specific data types (strings, dates, numbers, objects, and so on). It is a common practice to let the system know about the types of the variables during compilation. In a relational database world, we define a table schema with the appropriate field definitions to persist the records, and it is mandatory for the schema to exist before we start our persistence in the database. The same goes with Elasticsearch too.

Elasticsearch understands the data types of the fields while indexing the documents and, accordingly, stores the fields into appropriate data structures (for example, inverted index for text fields or BKD trees for numericals) for data retrieval. The indexed data with precisely formed data types leads to accurate search results as well as helping to sort and aggregate the data.

4.1.2 Indexing a document for the first time

Let's take an example and find out what happens if we index a document without having to create the schema up front. Say we have a `movie` document that we want to index, as demonstrated in the listing 4.1:

Listing 4.1 Indexing a document with ID 1 into movies index

```

PUT movies/_doc/1
{
  "title": "Godfather", #A The title of the movie
  "rating": 4.9, #B The rating given to the movie
  "release_year": "1972/08/01" #C Movie's release year(note the date format)
}

```

This would be our first document sent to Elasticsearch to get indexed. Remember, we didn't create an index (`movies`) or the schema for this document data prior to the document ingestion. Here's what happens when this document hits the engine:

1. A new index (`movies`) is created automatically with default settings.
2. A new schema is created for the `movies` index with the data types (we learn about data types shortly) deduced from this document's fields. For example, `title` is set to a `text` and `keyword` types, `rating` to a `float`, and `release_year` to a `date` type.
3. The document gets indexed and stored in the Elasticsearch data store.
4. Subsequent documents get indexed without undergoing the previous steps as Elasticsearch consults the newly created schema for further indexing.

Elasticsearch does a bit of ground work behind the scenes when performing these steps to create a schema definition with fields names and additional information. We can fetch the schema definition that Elasticsearch created dynamically using a mapping API. The response from issuing a GET command on the `_mapping` endpoint is shown in the figure 4.2:

```

...
"properties" : {
    "rating" : {"type" : "float"},           ← Due to the rating value of 4.9,
                                             a floating point number, the type
                                             is deduced as "float"
    "release_year" : {
        "type" : "date",
        "format" : "yyyy/MM/dd HH:mm:ss||yyyy/MM/dd|epoch_millis"
    },
    "title" : {
        "type" : "text",                     ← As the release_year value
                                             is in year's format (ISO
                                             format), the type is deduced
                                             as "date"
        "fields" : {
            "keyword" : {
                "type" : "keyword",
                "ignore_above" : 256
            }
        }
    }
}

```

GET movies/_mapping

The figure shows the JSON response from the `GET movies/_mapping` endpoint. It highlights the `properties` object and its contents. Annotations explain the deduced data types for each field:

- `rating`: A float type due to the value `4.9`.
- `release_year`: A date type due to the ISO format value `1997/12/31 14:15:00`.
- `title`: A text type due to the textual value `Toy Story`.
- `fields`: A multi data type for the `title` field, containing a `keyword` type with `ignore_above: 256`.

Figure 4.2 Movie index mapping derived from the document values.

Each field has a specific data type defined, for example, the `rating` field was declared to be a `float`, `release_year` a `date` type, and so on. The individual field can be composed of other fields too, also representing multiple data types. The `title` field is not only defined as `text` but also has a `keyword` type associated with it.

Elasticsearch uses a feature called *dynamic mapping* to deduce the data types of the fields when a document is indexed for the first time by looking at the field values and deriving these

types. During this process, it managed to derive the type of the name field to be `text`, based on the string value it has ("John Doe"). As the field is stamped as a `text` type, all full-text related queries can be performed on this field. We can issue search words like *john*, *doe*, *John Doe*, and so forth to search for the documents consisting of that name in a variety of ways. In this instance, the engine derived the type to be a `text` data type based on the value dynamically.

In addition to creating the name field a `text` type, Elasticsearch also did something extra for us: using the `fields` object, it created an additional type called `keyword` type for the `name` field, thus making the `name` field a *multi-typed* field. Multi typed fields can be associated with multiple data types. In our example, by default, the `name` field was mapped to a `text` as well as `keyword` type. The `keyword` fields are used for exact value searches. These data type fields are untouched and hence they won't go through an analysis phase. That is, they are neither tokenized nor synonymized nor stemmed. Refer to section [4.7 Multiple data types](#) for more information on multi-fields.

Analysis of keyword fields .

Fields declared as `keyword` data types use a special analyzer called `noop` (the no-operation analyzer), thus untouched the `keyword` data during the indexing process. This `keyword` analyzer spits out the entire field as one big token. By default, the `keyword` fields will not be normalized as the `normalizer` property on the `keyword` type is set to `null`. Having said that, we can customize and enable the `normalizer` on the `keyword` data type by setting filters such as `german_normalizer` or uppercase etc. By setting the `normalizer`, we are actually expecting the `keyword` field to undergo normalization prior to indexing

While dynamic mapping is intelligent and convenient, be aware that it can get the schema definitions wrong too. Elasticsearch can only go so far when deriving the schema based on our document's field values. It might make incorrect assumptions, leading to erroneous index schemas that can produce incorrect search results.

We learned Elasticsearch derives and deduces the data types of the fields and creates a schema for us dynamically if the schema doesn't exist when the first document comes through for indexing. In the next section we learn more about dynamic mapping, how Elasticsearch deduces types, and its downsides.

4.2 Dynamic mapping

When we try to index a document for the first time, both the mapping and the index will be created automatically. The engine wouldn't give you trouble for not providing the schema upfront. Elasticsearch is more forgiving on that count. We can index a document without having to let the engine know anything about the field's data types. Consider a movie document consisting of a few fields, shown in figure 4.3.

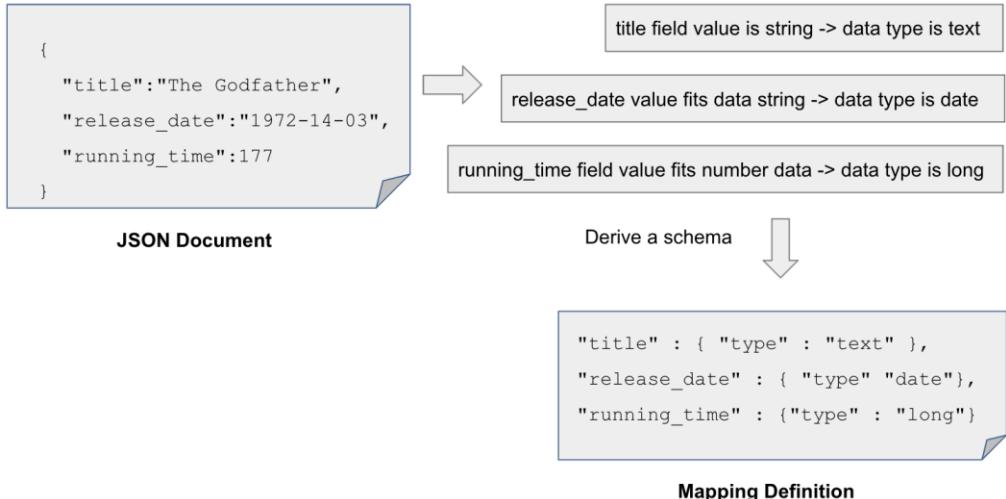


Figure 4.3 Dynamically deriving the indexing schema

Reading the fields and the values in this document, Elasticsearch infers the data types automatically. For example,

- The `title` field value is a string, so it maps the field to a `text` data type;
- The `release_date` field value is a string, but the value matches an ISO 8601 date format so it is mapped to a `date` type.
- Also, the `running_time` field is a number, so Elasticsearch maps it to a `long` data type.

As we are not explicitly creating the mapping but letting Elasticsearch derive the schema on the fly, this type of mapping is termed as *dynamic mapping*. This feature is quite handy during application development.

How did Elasticsearch know the data type of the ratings field is a float type? For that matter, how does Elasticsearch deduce the types? We need to understand the mechanism and rules of how the types are derived by Elasticsearch, a topic discussed in the next section.

4.2.1 The deducing types mechanism

Elasticsearch analyzes the values and guesses the equivalent data type based on the values. When the incoming JSON object is parsed, the `rating` value of 4.9, for example, resembles a floating-point integer in programming language lingo. This “guesswork” is pretty straightforward; hence, Elasticsearch stamps the `rating` field as a `float` field. While deducing the number type might be easy, how about the type of the `release_year` field? The `release_year` field value is compared and matched to one of the two default date formats: `yyyy/MM/dd` or `yyyy-MM-dd`. If it matches, the type is assigned as a `date` data type.

NOTE DATE FORMAT DURING DYNAMIC MAPPING Elasticsearch can deduce a field as a `date` type if the values in the JSON document are provided in either of the format: `yyyy-MM-dd` or `yyyy/MM/dd` although the latter isn't an ISO date format. However, this flexibility is available for dynamic mapping cases only. That is, should you declare the field as a `date` type explicitly (explicit mapping), unless you provide a custom format, by default the `fi` field would be adhering to `strict_date_optional_time` format. The `strict_date_optional_time` conforms to a ISO date format, i.e.,`yyyy-MM-dd` or `yyyy-MM-ddTHH:mm:ss`.

The guesswork is adequate for most cases, but falls short if the data is slightly off-the-default track. Consider the `release_year` field in our movie document. If our document has a value for `release_year` in a different format (say, `ddMMyyyy` or `dd/MM/yyyy`), the dynamic mapping rule breaks down. Elasticsearch will consider that value as a `text` field rather than a `date` field.

Similarly, let's try out indexing another document by setting the `rating` field value to 4 (our intention is that the ratings will be provided as decimal points). Although our intention was to have the `rating` field host floating-point data, Elasticsearch fell short in deriving the appropriate type. It assumed, by looking at the value (4), the data type of the field to be a `long` data type.

In both cases, we have a schema with incorrect data types. Having incorrect types will cause potential problems in an application, making the fields ineligible for sorting, filtering, and aggregations on data. This is the best that Elasticsearch can do when we use its dynamic mapping feature.

Your data - your schema

While Elasticsearch is clever enough to derive the mapping information based on our documents, which allows us to worry not about schemas, there is a chance things could go badly and we would end up with an incorrect schema definition. As we tend to have a good understanding about our domain, and we know the data models pretty much inside and out, my advice is not to let Elasticsearch create our schemas, especially in production environments. Instead, aim to create schemas upfront, most likely creating a mapping strategy across the organization using mapping templates (we will work through some mapping templates later).

So far, we looked at dynamic mappings and learned that although the dynamic mapping feature is attractive, it does carry limitations as discussed in the next section.

4.2.2 Limitations of dynamic mapping

While the dynamic mapping feature is attractive, there are some limitations for letting Elasticsearch derive the document schema. Elasticsearch could misinterpret the document field values and derive an incorrect mapping which voids the fields eligibility for appropriate search, sort and aggregation capabilities. Let's find out how Elasticsearch deduces data types incorrectly and thus aids in formulating inaccurate and erroneous mapping rules.

DEDUCING INCORRECT MAPPING

Say our intention was to provide a field with numeric data but wrapped up as string (for example "`3.14`"). Unfortunately Elasticsearch treats such data incorrectly, as in this example, it would expect the type to be `text`, not `float` or `double` data type.

Let's modify our `student` document and add another field: student's `age` field, which though sounds like a number, we index it as a `text` field by wrapping the value within quotes, as shown in the listing 4.2:

Listing 4.2: Adding `age` field with text values to the student document

```
PUT students_temp/_doc/1 #A This is the first document
{
  "name": "John",
  "age": "12" # B The age variable's value set as a string (in quotes)
}
PUT students_temp/_doc/2 #C This is the second document
{
  "name": "William",
  "age": "14"
}
```

Notice the `age` field's value is enclosed in quotes, which means Elasticsearch will treat it as a `text` field (though it has numbers). Once we get these two documents indexed, we can write a search query to sort the students by ascending or descending based on their age. This is shown in the figure 4.4. The response indicates the operation is not allowed, let's understand the reason.

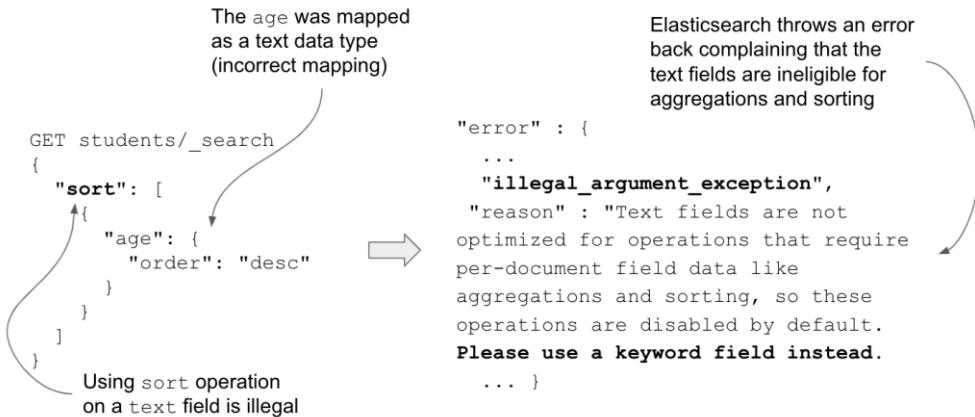


Figure 4.4 A `sort` operation on a `text` field results in an error

As the `age` field of a student document is indexed as a `text` field (by default, any string field is mapped to a `text` field during the dynamic mapping process), Elasticsearch cannot use that field in sorting operations. Sorting functionality is not available on `text` fields.

FIXING THE SORTING USING KEYWORD TYPES

Not all is lost, fortunately Elasticsearch creates any field as a multi-field with `keyword` as the second type. Going with this default feature, the `age` field is indeed tagged as a multi-field, thus

creating `age.keyword` as a second data type with `keyword` being the data type. To sort out students, all we need to do is change the field name from `age` to `age.keyword`, as demonstrated in the listing 4.3:

Listing 4.3: The modified sort query, sorting on age.keyword field

```
GET students_temp/_search #A Using _search API to fetch all docs
{
  "sort": [ #B The sorting function
    {
      "age.keyword": { #C The age.keyword is the name of the field
        "order": "asc" #D Sorting the data in ascending order
      }
    }
  ]
}
```

This query will result in all students in an ascending order of age. The query runs successfully because the `age.keyword` is a `keyword` type field where we can apply a sorting function. We are sorting on the second data type (`age.keyword`) which was created by Elasticsearch by default.

CALLOUTS: The `age.keyword` in the example above is the default name provided by Elasticsearch during dynamic mapping. We have full control of creating the fields, their names and types when defining the mapping schemas implicitly.

Treating a field as a `text` type instead of a `keyword` will unnecessarily affect the engine as the data gets analyzed and broken down into tokens.

DERIVING INCORRECT DATE FORMATS

There's also another potential problem with dynamic mapping: date formats may be derived incorrectly if not provided in Elasticsearch's default date format (`yyyy-MM-dd` or `yyyy/MM/dd`). The date is considered as a `text` type if we post the date in format like UK's `dd-MM-yyyy` or US's `MM-dd-yyyy` formats. We cannot perform date math on the fields associated with non-date data types. Such fields are ineligible for sorting, filtering, and aggregations too.

CALLOUTS: There is no date type in JSON, so it's up to the consuming applications to decode the value and deduce it to be a date (or not). When working with Elasticsearch, we provide the data in a string format for the fields, for example: `"release_date" : "2021-07-28"`

The takeaway when working with dynamic mapping features of Elasticsearch is that there is a scope for erroneous mapping deduced by Elasticsearch at times. Preparing our schema based on the field values may not fit the bill sometimes. Hence, the general advice is to develop the schema as per your data model requirements rather than falling on the mercy of the engine.

To overcome those limitations, we can choose another alternative path: the path of *explicit mapping*, where we define our schema and create it before the indexing process kicks in. We discuss explicit mapping in detail in the next section.

4.3 Explicit mapping

In the last section, we discussed Elasticsearch's schema-less feature using dynamic mapping. Elasticsearch is intelligent to derive the mapping information based on our documents, however, there is a chance things could end up with an incorrect schema definition. Fortunately, Elasticsearch provides the ways and means for us to dictate the mapping definitions the way we want to in the form of indexing and mapping APIs.

The following lists two possible ways of creating (or updating) a schema explicitly. And as an example, look at figure 4.5, which demonstrates using both APIs to create a movies index.

- ***Indexing APIs***—We can create a schema definition at the time of index creation using the `create index` API (not the mapping API) for this purpose. The `create index` API expects a request consisting of the required schema definition in the form of a JSON document. This way, a new index and its mapping definitions are all created in one go.
- ***Mapping APIs***—As our data model matures, at times there will be a need to update the schema definition with new attributes. Elasticsearch provides a `_mapping` endpoint to carry out this action, allowing us to add the additional fields and their data types.

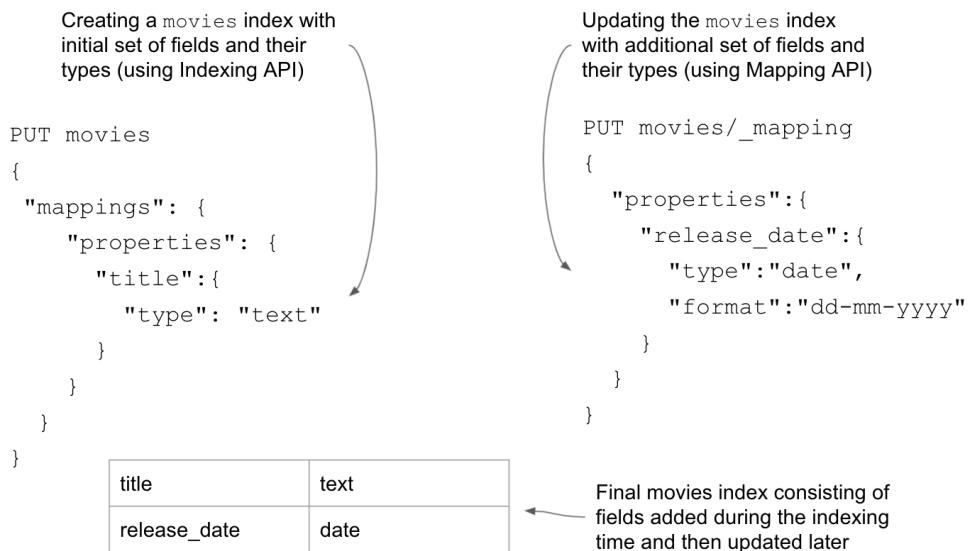


Figure 4.5 Creating and updating schema using indexing and mapping APIs

We will look at these two methods in the next section. We'll follow an example in which we imagine we are working with an employee's data.

4.3.1 Mapping using the indexing API

Creating a mapping definition at the time of index creation is relatively straightforward. We simply issue a PUT command following with the index name and pass the `mappings` object with all the required fields and their details as body of the request. Figure 4.6 explains the constituents visually.

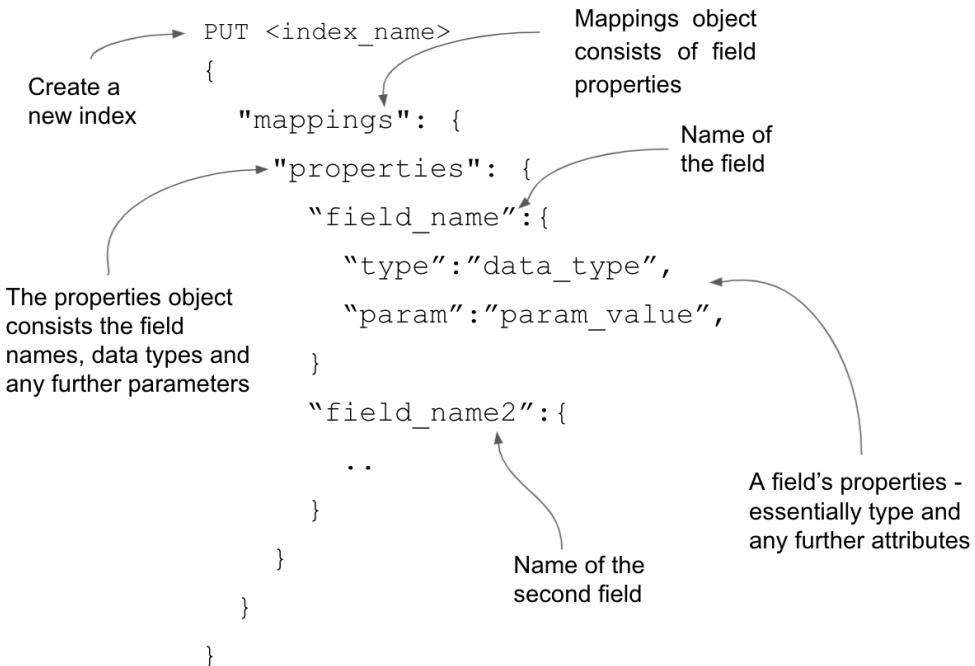


Figure 4.6 Creating a mapping definition during index creation

Now that we know the theory behind using the mapping API for creating a schema, let's put that to work for us by developing a mapping schema for an employee model. The employee information is modelled with a bunch of fields such as name, age, email and an address.

We index the document consisting of these fields into an `employees` index by invoking the mapping API over a HTTP PUT action. The request body is encapsulated with properties for our fields as the listing 4.4 demonstrates.

Listing 4.4 Creating the schema upfront

```
PUT employees
{
  "mappings": {
    "properties": {
      "name": { "type": "text"}, #A The name field is a text data type
      "age": { "type": "integer"}, #B The age field is a number type
      "mmail": { "type": "keyword"}, #C Keyword type (email is structured data)
      "address": { #D Address object in an inner object with further fields
        "properties": { #E Inner object properties
          "street": { "type": "text" },
          "country": { "type": "text" }
        }
      }
    }
  }
}
```

Once the script is ready, execute this command with Kibana's DevTools. You should receive a successful response in the right pane indicating that the index was created. We now have an `employees` index created with the expected schema mapping. In this example, we dictated the types to Elasticsearch; now we are in control of the schema.

Did you notice the `address` field in the listing 4.4? It is an `object` type, consisting of additional fields, `street` and `country`. One important thing to note is that the type of the `address` field has not been mentioned as an `object` type, although we said it was an `object` that encapsulates other data fields. The reason for this is that the `object` data type is deduced by Elasticsearch for any inner objects by default. Also, the sub-field `properties` object wrapped up in the `address` helps define further attributes of the inner object.

Now that we have our `employees` index in production, there's a new requirement bestowed upon us: business management wants us to extend the model to have few more attributes such as department, phone number, and others. To cater to this requirement, we need to add these additional fields using the `mapping API` on a live index. We briefly go over this topic in the next section.

4.3.2 Updating schema using the mapping API

As our project matures, there will be undoubtedly changes to the data model too. Going with our `employee` document, we may want to add a couple of attributes like `joining_date` and `phone_number` shown in the following code snippet.

```
{
  "name": "John Smith",
  "joining_date": "01-05-2021", #A adding joining date
  "phone_number": "01234567899" #B adding phone number
  ...
}
```

The `joining date` is a `date` type as we want to do date-related operations, such as sorting employees by joining date. The phone number is expected to be stored as is, so it fits the `keyword` data type. To amend the existing `employees` schema definition with these additional fields, we

invoke a `_mapping` endpoint on the existing index, declaring the new fields in the request object (listing 4.5):

Listing 4.5: Updating the existing index with additional fields

```
PUT employees/_mapping #A The mapping endpoint to update the existing index
{
  "properties": {
    "joining_date": { #A The joining date field as date type
      "type": "date",
      "format": "dd-mm-yyyy" #C The expected date format
    },
    "phone_number": {
      "type": "keyword" #D Phone numbers are stored as-is
    }
  }
}
```

If you look at the request body closely, the `properties` object is defined at the root level as opposed to the earlier method of creating a schema using the indexing API where the `properties` object was wrapped in the root-level mapping object.

UPDATING AN EMPTY INDEX

We can use the same principle of updating the schema on an empty index too. An *empty index* is an index created without any schema mapping (for example, executing the `PUT books` command creates an empty `books` index).

Similar to how we discussed the mechanism to update the index a few moments ago (listing 4.5) by calling the `_mapping` endpoint with the required schema definition, we can use the same approach for the empty index too. The following code snippet updates the schema on the `departments` index with couple of fields:

Listing 4.6: Updating the mapping schema of an empty index

```
PUT departments/_mapping #A Using the mapping API to update an empty index
{
  "properties": {
    "name": {
      "type": "text" #B The name field declared as text type
    }
  }
}
```

So far we've seen the additive case of updating the schema with additional fields. But what if we want to change the data types of the existing fields? Say, if we want to change a field's type from `text` to a `keyword` type Can we update the mapping just as easily as we added additional fields? The short answer is no. Elasticsearch doesn't allow us to change or modify the data types of existing fields. It requires a bit more work, so let's discuss this in the next section.

4.3.3 Modifying the existing fields is not allowed

Once an index is live (the index was created with some data fields and is operational), any modifications of the existing fields on the live index is prohibited. For example, if a field was defined as a `keyword` data type and indexed, it cannot be changed to a different data type (say, from `keyword` to a `text` data type). There is a good reason for this though.

Data is indexed using the existing schema definitions and, hence, stored in the index. The searching on that field data fails if the data type has been modified, which leads to an erroneous search experience. To avoid the search failures, Elasticsearch does not allow us to modify existing fields.

Well, you may ask, what would be the alternative? Business requirements change as do the technical requirements. How can we fix data types (may have been incorrectly created in the first place) on a live index?

This is where we use a *re-indexing* technique. Re-indexing operations source the data from the original index to a new index (with updated schema definitions perhaps). The idea is that we

1. Create a new index with the updated schema definitions.
2. Copy the data from the old index into the new index using re-indexing APIs. The new index with new schema will be ready to use once the re-indexing is complete. The index is open for both read and write operations.
3. Once the new index is ready, our application switches over to the new index.
4. We shelf the old index once we confirm the new index works as expected.

Re-indexing is a powerful operation, which is discussed in detail in the Chapter 5: Working with Documents, but let me give you a glimpse of how the API works. Say we wish to migrate data from an existing (source) index to a target (dest) index, we issue a reindexing call, as shown in the code listing 4.7 below:

Listing 4.7: Migrating data between indices using reindexing API

```
POST _reindex
{
  "source": {"index": "orders"},
  "dest": {"index": "orders_new"}
}
```

The new index `orders_new` may have been created with the changes to the schema and then the data from the old index (`orders`) is migrated to this newly created index with updated declarations.

NOTE If your application is rigidly tied up with an existing index, moving to the new index may require a code or configuration change. The ideal way to avoid such situations is to use aliases. Aliases are the alternate names given to indices. Aliasing helps us switch between indices seamlessly with zero downtime. We will look at aliases in an upcoming Chapter 6 on indexing operations.

Sometimes the data may have incorrect types when indexing the documents. For example, a rating field of type `float` may have received a value enclosed as a string: `"rating": "4.9"` instead of `"rating": 4.9`. Fortunately, Elasticsearch is forgiving when it encounters mismatched

values for data types. It goes ahead with indexing the document by extracting the value and storing it in the original data type. However, we'll discuss this mechanism in the next section.

4.3.4 Type coercion

At times, the JSON documents might have incorrect values, differing from the ones that are present in the schema definition. For example, an integer-defined field may be indexed with a string value. Elasticsearch tries to convert such inconsistent types, thus avoiding indexing issues. This is a process known as *type coercion*. For example, say that we've defined the `age` of a car field to be an integer type:

```
"age": {"type": "short"}
```

Ideally, we'll index our documents consisting of `age` as an integer value. However, a user may have accidentally set the value of `age` ("23") as a string type and invoked the call, as shown in the listing 4.8:

Listing 4.8: Setting a string value for a numeric field

```
PUT cars/_doc/1
{
  "make": "BMW",
  "model": "X3",
  "age": "23"
}
```

Elasticsearch will get this document indexed without any errors. Although the `age` field is expected to have an integer, it coerces the type for us to avoid errors. The coercion only works if the parsed value of the field matches the expected data type. In the previous case, parsing "2" results in 2, which is a number.

So far we've gone through a lot about mapping and data types without putting much emphasis on data types. Designing your schema with appropriate data types is a crucial step in optimizing the search experience. We need to have a good understanding of the data types, their characteristics, and when to use these. And, unlike any database or programming language, Elasticsearch comes with a long list of data types, catering to almost every shape and form that your data needs. In the next section, we'll look at the data types in detail.

4.4 Data types

Similar to a variable's data type in programming languages, fields in a document have a specific data type associated with them. Elasticsearch provides a rich list of data types, ranging from simple to complex to specialized types. The list of these types keeps growing, so watch for more along the way. In the next section, we classify these data types and learn about them in detail.

4.4.1 Data type classifications

Elasticsearch provides over two dozen different data types, so we have a good selection of appropriate data types based on our specific needs. Data types can be broadly classified under the following categories:

- *Simple types*—The common data types, representing strings (textual information), dates, numbers, and other basic data variants. The examples are `text`, `boolean`, `long`, `date`, `double`, `binary`, etc.
- *Complex types*—The complex types are created by composing additional types, similar to an object construction in a programming language where the objects can hold inner objects. The complex types can be flattened or nested to create even more complex data structures based on the requirements at hand. The examples are `object`, `nested`, `flattened`, `join`, etc.
- *Specialized types*—These types are predominantly used for specialized cases such as geolocation and IP addresses. The common example types are `geo_shape`, `geo_point`, and `ip`, and range types such as `date_range`, `ip_range` and others.

NOTE CONSULT OFFICIAL DOCUMENTATION A full range of types can be accessed from the official documentation here: <https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-types.html>.

Every field in a document can have at least one or more types associated as per the business and data requirements. We have been working with some of the data types without knowing them formally so far. It is time to get a detailed understanding about them. Table 4.2 provides a list of common data types with some examples.

Table 4.2 Common data types with examples

Type	Description	Examples
text	Represents textual information (such as string values); unstructured text	Movie title, blog post, book review, log message
integer/long/short/byte	Represents a number	Number of infectious cases, flights canceled, products sold
float/double	Represents a floating-point number	Student's grade point, moving average of sales, reviewer ratings
boolean	Represents a binary choice, true or false	Is the movie a blockbuster? Are COVID cases on the rise? Has the student passed the exam?
keyword	Represents structured text, text that must not be broken down or analyzed	Error codes, email addresses, phone numbers, Social Security numbers
object	Represents a JSON object	(In JSON format) employee details, tweets, movie objects
nested	Represents an array of objects	An employee's address, email's routing data, a movie's technical crew

As you can imagine, this is not a comprehensive list of data types. As of writing this book for version 8.x, there are about 29 data types defined by Elasticsearch. Elasticsearch defines the types microscopically in some cases for the benefit of optimizing the search queries. For example, the `text` types are further classified into more specific types such as `search_as_you_type`, `match_only_text`, `completion`, `token_count`, and others.

Elasticsearch also tries to group the data types as a family for optimizing space and performance. For example, the keyword family has `keyword`, `wildcard` and `constant_keyword` data types. However, currently this is the only family group that was created, for sure we can expect this family group to grow in the near future.

Single field with multiple data types .

In a programming languages like Java or C#, we can't define a variable with two different types. However, there is no such restriction in Elasticsearch. The engine is pretty cool when it comes to representing a field with multiple data types, allowing us to create multiple types for the same field. For example, we may want the author of a book to be both a text type as well as keyword type. Each of these fields have a specific characteristic, and the keywords will not be analyzed, meaning that the field gets stored as is.

In the next few sections, we will go over these data types, their characteristics and usage. Before we delve into data types and work with them in detail, we need to have an understanding of creating mapping definitions. There is a separate section which digs deeper in working with the mapping features (section 4.3 Explicit Mapping), but for now we briefly look at a mechanism to create a mapping definition as it is a pre-requisite for the next sections.

4.4.2 Developing mapping schemas

When we indexed a movie document (listing 4.1), Elasticsearch created a schema dynamically (figure 4.2) deriving the types by analyzing the field values. We are now at a point where we would like to create our own schema definitions. Elasticsearch uses an indexing API to create these definitions, so let's check it out with an example.

Say, we wish to create a student with `name` and `age` with data types being `text` and `byte` respectively (if you are curious to understand about these data types, jump to the next section as we introduce ourselves to them in detail). We can create the `students` index with these mapping definitions upfront, thus taking control of creating the schema into our hands. The figure 4.7 demonstrates this succinctly, with the mapping definition execution on the left hand side and fetching the definition on the right hand side.

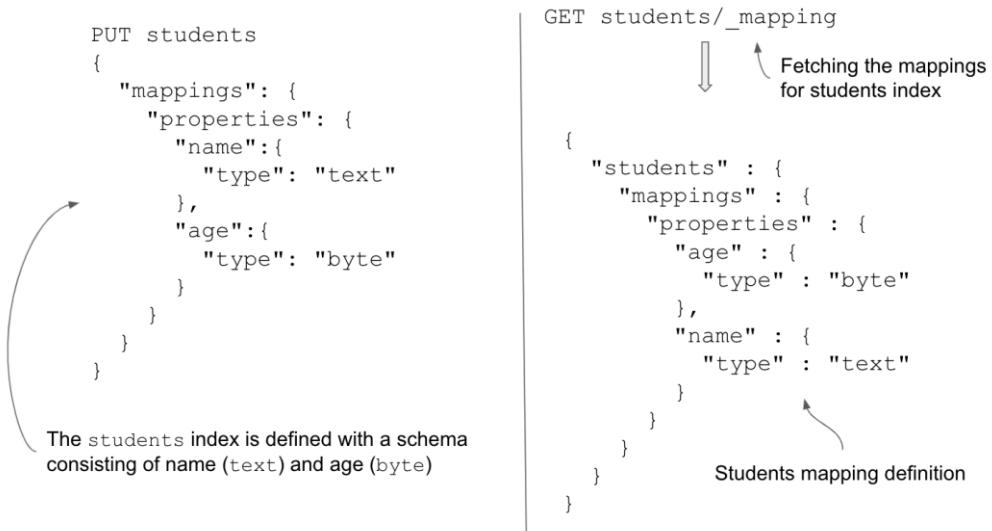


Figure 4.7: Creating the mapping definition (left) and retrieving the schema (right)

We create the index with a request containing a set of field properties wrapped up in a `mappings` object. We invoke the `_mapping` endpoint on the index to check the schema (demonstrated on the right hand side in the figure 4.7) definition.

Now that we are equipped with the knowledge of creating schemas, though at a high level, we can go ahead and learn and understand core data types.

4.5 Core Data Types

Elasticsearch provides a set of core data types such as `text`, `keyword`, `date`, `long`, `boolean` etc for employing them to represent data. We need to have a basic understanding of core data types before we can jump into understanding the advanced types as well as dynamic and explicit mapping in detail. In the next few sections, we will run over the core basic data types with examples where possible.

NOTE **CODE ON GITHUB** For brevity, I have decided to move the code for some of the examples in the coming section from the book to GitHub to avoid bloating the chapter. Please consult the chapter's code on the GitHub for working examples.

4.5.1 The `text` data type

If there's one type of data that search engines must do well is the full-text data type. The human readable text, also termed as full text or unstructured text in a search engine lingo, is the bread and butter of a modern search engine. We consume a lot of textual information in the current digital world, for example as blog posts, research articles, news items, tweets, and many others.

Elasticsearch defines a dedicated data type to handle full text data - the `text` data type - to support such textual information fields.

The idea is that any field that's stamped with the `text` data type gets analyzed before it gets persisted. During the analysis process, the analyzers massage the textual data into various forms and store them in internal data structures for easy access. As we saw in our earlier examples, setting the type is straightforward. Use this syntax in your mapping definition: `"field_name": {"type": "text"}`. We have already worked through an example earlier in section 4.2.2, so we won't repeat that exercise. Instead, let's understand how Elasticsearch treats `text` fields during indexing, which is discussed in the next section.

ANALYZING TEXT FIELDS

Elasticsearch supports two types of text data: structured text and unstructured text. The unstructured text is the full-text data, usually written in a human-readable language such as English, Chinese, Portuguese, and so on. Efficient and effective search on unstructured text is what makes a search engine stand out. The unstructured or as commonly called the full text, undergoes an analysis process whereby the data is split into tokens, characters are filtered out, words are reduced to its root word (stemming), synonyms are added, and other natural language processing rules applied. Though we have a dedicated chapter on text-analysis, let's quickly peek through how Elasticsearch handles full-text with an example. The following shows a user's review comment on a movie:

```
"The movie was sick!!! Hilarious :) :) and WITTY ;) a KILLer 🌟"
```

When this document is indexed, it undergoes an analysis based on the analyzer. Analyzers are text-analysis modules employed by Elasticsearch to analyse the incoming text to tokenize and normalize. By default, Elasticsearch uses a *Standard Analyser* and, analysing the above review comment leads to the following steps (figure 4.8):

1. The tags, punctuation and special characters are stripped away using character filters. This is how it looks after this step:

```
The movie was sick Hilarious and WITTY a KILLer
```

2. The sentence is broken down into tokens using a tokenizer resulting in:

```
[the, movie, was, sick, Hilarious, and, WITTY, a, KILLer, 🌟]
```

3. The tokens are changed to lowercase using token filters, so it looks like this:

```
[the, movie, was, sick, hilarious, and, witty, a, killer, 🌟]
```

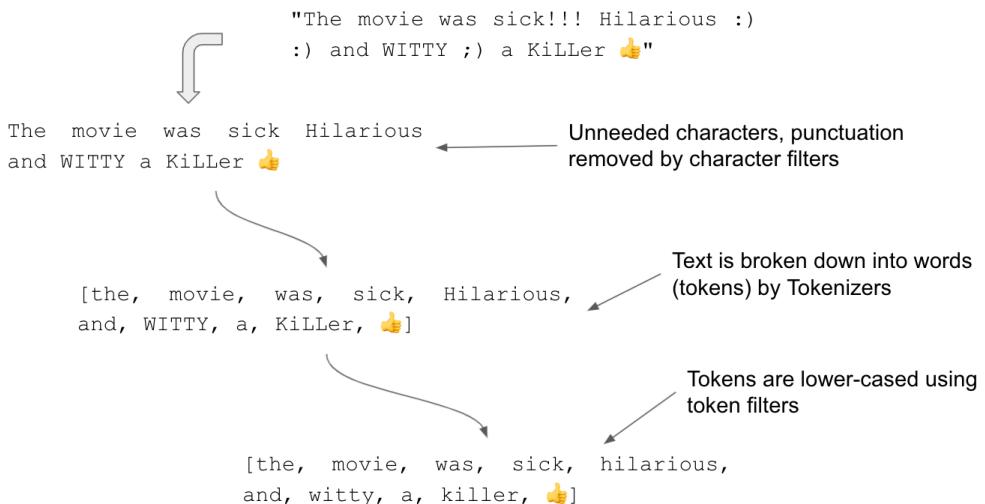


Figure 4.8 Processing a full text field during indexing by standard analyzer module

These steps may vary, depending on the choice of your analyser. For example, if you choose an English analyzer, the tokens are deduced to be root words (stemming):

```
[movi,sick,hilari, witti, killer, 🌟].
```

Did you notice the stemmed words like movi, hilari, witti? They are actually not real words per say but the incorrect spellings don't matter as long as all the derived forms can match the stemmed words.

NOTE **USING _analyze API** We can use the `_analyse` API exposed by Elasticsearch to test out how the text gets analyzed. This API helps us understand the inner workings of the analyzers and supports building complex and custom analyzer modules for various language requirements.

The above code is available on GitHub repository, feel free to experiment with running analyzers on text data.

Stemming

Stemming is a process of reducing the words to their root words. For example, fighter, fight, and fought all may lead to one word, fight. The stemmers are language dependent, for example, one can employ a french stemmer if the chosen language of the documents is French. The stemmers are declared via token filters when composing the analyzer module during the text analysis phase in Elasticsearch.

The same process is retriggered during execution of the search queries on the same field too. We have a dedicated chapter (Chapter X: Text Analysis) on the process of applying analysis on full text whereby we go over the intricacies of analyzers and how Elasticsearch manages full text data.

Remember, I mentioned earlier Elasticsearch defines the types microscopically, for example, further classifying the `text` fields into more specific types such as `search_as_you_type`, `match_only_text`, `completion`, `token_count`, and others? Let's go over these specialized text types briefly in the next few sections as it would be interesting to understand how Elasticsearch puts extra emphasis and effort into the full text world.

TOKEN COUNT (`TOKEN_COUNT`) DATA TYPE

A specialized form of `text` data type, the `token_count` will help define a field that captures the number of tokens in that field. That is, say if we have defined a book's `title` as a `token_count`, we could retrieve all the books based on the number of tokens a book has. Let's create a mapping for this by creating an index with a `title` field. The `title` field is set to type `token_count`, as demonstrated in the code listing 4.9:

Listing 4.9: Creating a `tech_books` index with `title` field as `token_count` its data type

```
PUT tech_books
{
  "mappings": {
    "properties": {
      "title": { #A The field's name
        "type": "token_count", #B The title's data type is token_count
        "analyzer": "standard" #C The analyzer is expected to be provided
      }
    }
  }
}
```

As you can see from the code, the `title` is defined as a `token_count` type (specialized form of `text` type). The analyser is expected to be provided so we are setting the standard analyzer on the `title` field. We can then index three technical books (this and my future books, hopefully) before trying to search for them based on the title's token count feature, as shown here in the listing 4.10:

Listing 4.10 Indexing three new documents into `tech_books` index

```
PUT tech_books/_doc/1
{
  "title": "Elasticsearch in Action"
}

PUT tech_books/_doc/2
{
  "title": "Elasticsearch for Java Developers"
}

PUT tech_books/_doc/3
{
  "title": "Elastic Stack in Action"
}
```

Now that we have the `tech_books` index with few books, let's put the `token_count` type to use. We write a range query (listing 4.11) to fetch the books with title composed of more than 3 words (`gt` is short for greater than) but less than or equal to 5 (`lte` is short form for less than or equal to) words:

Listing 4.11 Range query fetching books with a certain number of words in the title

```
GET tech_books/_search
{
  "query": {
    "range": {
      "title": {
        "gt": 3,
        "lte": 5
      }
    }
  }
}
```

The range query fetches books based on the number of words in a title. This will retrieve the *Elasticsearch for Java Developers* (4 tokens) and *Elastic Stack in Action* (4 tokens) but omits *Elasticsearch in Action* (3 tokens).

We can indeed combine the `title` field as a `text` type as well as a `token_type` too as Elasticsearch allows a single field to be declared with multiple data types (multi-fields - discussed in detail in section 4.6). The listing 4.12 demonstrates this technique:

Listing 4.12: Adding token_count as an additional data type to a text field

```
PUT tech_books
{
  "mappings": {
    "properties": {
      "title": { #A The title field is defined as text data type
        "type": "text",
        "fields": { #B The title field is declared to have multiple data types
          "word_count": { #C The word_count is the additional field
            "type": "token_count", #D The type of word_count
            "analyzer": "standard" #E mandatory to provide the analyzer
          }
        }
      }
    }
  }
}
```

As the `word_count` field is an inner attribute of the `title` field, the term query, for example, is demonstrated in the listing 4.13:

Listing 4.13: Searching the books with title having more than 4 words

```
GET tech_books/_search
{
  "query": {
    "term": {#A We are using a term query
      "title.word_count": {#B The name of the field
        "value": 4
      }
    }
  }
}
```

We will be using `<outer_field>.<inner_field>` as the `word_count` field's name, so `title.word_count` is the accessor of the field.

In addition to the `token_count`, `text` type has other descendants too, like `search_as_you_type` and `completion`. We look at them in the later part of the book in a separate chapter on advanced types (however, sample code is available on the book's GitHub repository if you are curious). We continue our journey of learning about common data types, the `keyword` data type being the next in line.

4.5.2 The keywords data types

The keywords family of data types is composed of `keyword`, `constant_keyword` and `wildcard` field types. Let's look at these types here.

THE KEYWORD TYPE

The structured data, such as pincodes, bank accounts, phone numbers, don't need to be searched as partial matches or produce relevant results. The results tend to provide a binary output: returns results if a match or return none. This type of query doesn't care about how well the document is matched, so you expect no relevance scores associated with the results. Such structured data is represented as `keyword` data type in Elasticsearch.

The `keyword` data type leaves the fields untouched. The field is untokenized and not analyzed. The advantage of `keyword` fields is that these can be used in data aggregations, range queries, and filtering and sorting operations on the data. To set a `keyword` type, use this format:

```
"field_name":{ "type": "keyword" }
```

For example, the following code listing (4.14) creates an email with keyword type:

Listing 4.14: Defining email as keyword type for a faculty document

```
PUT faculty
{
  "mappings": {
    "properties": {
      "email": { #A Define the email property
        "type": "keyword" #B Declaring email as keyword type
      }
    }
  }
}
```

We can also declare numeric values as keywords too, for example, the `credit_card_number` may be declared as a `keyword` for efficient access than as a numeric such as `long`. There's no way we can build range queries on such data. The rule of thumb is if the numerical fields are *not* used in range queries, then declaring them as `keyword` types is advised as it aids faster retrieval.

NOTE [CODE ON GITHUB](#) A sample code for demonstrating keyword data type is available on book's GitHub repository.

THE CONSTANT_KEYWORD TYPE

When the corpus of documents is expected to have the same value, irrespective of any number, `constant_keyword` type comes handy. Let's just say the United Kingdom is carrying out a census in 2031, and for obvious reasons the `country` field of each citizen' individual census document is expected to be "United Kingdom" by default. Ideally, there is no need to send the `country` field for each of the documents when they are indexed into the census index. This is where the `constant_keyword` will be helpful. The mapping schema defines an index (`census`) with a field called `country` and its type being `constant_keyword`.

The code shown here (listing 4.15) demonstrates the mapping definition, with the `country` field's type as `constant_keyword`. Note that we are setting a default value for this field as "United Kingdom" at the time of declaring the mapping definition.

Listing 4.15: Census index with country declared as `constant_keyword` and UK as value

```
PUT census
{
  "mappings": {
    "properties": {
      "country": {
        "type": "constant_keyword",
        "value": "United Kingdom"
      }
    }
  }
}
```

Now, we index a document for John Doe, with just his name (no `country` field):

```
PUT census/_doc/1
{
  "name": "John Doe"
}
```

Now when we search for all residents of the UK (though the document hasn't got that field during indexing), we receive the positive result - returning John's document:

```
GET census/_search
{
  "query": {
    "term": {
      "country": {
        "value": "United Kingdom"
      }
    }
  }
}
```

The `constant_keyword` field will have exactly the same value for every document in that index.

THE WILDCARD DATA TYPE

The `wildcard` data type is another special data type that belongs to the keywords family which supports searching data using wildcards and regular expressions. We define the field as a `wildcard` type by declaring it as `"type": "wildcard"` in the mapping definition. We can query the field by issuing a wildcard query as demonstrated in the listing 4.16 (the document with `"description": "Null Pointer exception as object is null"` was indexed prior to this query)

Listing 4.16: The wildcard query to fetch documents matching to a value set as wildcard

```
GET errors/_search
{
  "query": {
    "wildcard": {#A Use a wildcard query
      "description": {
        "value": "*obj*" #B Search using wildcards
      }
    }
  }
}
```

Keyword fields are efficient and performant, so using them appropriately will improve the indexing and search query performance.

4.5.3 The date data type

Elasticsearch provides a `date` data type for supporting indexing and searching date-based operations. The date fields are considered to be structured data; hence, you can use them in sorting, filtering, and aggregations. Elasticsearch parses the string value and infers it as a date if the value confirms the ISO 8601 date standard. That is, the date value is expected to be in the format of `yyyy-MM-dd` or with a time component as `yyyy-MM-ddTHH:mm:ss`.

JSON doesn't have a date type, so dates in the incoming documents are expressed as strings. These are parsed by Elasticsearch and indexed appropriately. For example, a value such as `"article_date": "2021-05-01"` or `"article_date": "2021-05-01T15:45:50"` is considered a date and is indexed as date type because the value conforms to the ISO standard.

Just as we did earlier with other data types, we can create a field of `date` type during the mapping definition, as the listing 4.17 creates a `departure_date_time` field for a flight document.

Listing 4.17: Creating an index with a date type

```
PUT flights
{
  "mappings": {
    "properties": {
      "departure_date_time": {
        "type": "date"
      }
    }
  }
}
```

When indexing a flight document, setting the `"departure_date_time" : "2021-08-06"` (or as `"2021-08-06T05:30:00"` with time component) will index the document with the date as expected.

NOTE DATE DURING DYNAMIC MAPPING *When no mapping definition for a date field exists in an index, Elasticsearch parses a document successfully when the format of the date is either in yyyy-MM-dd (ISO date format) or in yyyy/MM/dd (non-ISO date format) format. However, once we've created the mapping definition for a date, the date format of the incoming document is expected as per the format defined during the mapping definition.*

We can of course change the format of the date if we need to, that is, instead of setting the date in ISO format (`yyyy-MM-dd`), we can customize the format as per our need by setting the required format on the field during its creation, as shown in the snippet here:

```
"departure_date_time": { #A Customizing the date format
  "type": "date",
  "format": "dd-MM-yyyy||dd-MM-yy" #B Date is set in any of these two values
}
```

The incoming documents can now have the departure field set as

```
"departure_date_time" :"06-08-2021"
```

or

```
"departure_date_time" :"06-08-21"
```

In addition to accepting the date as a string value, we can also provide it in a number format - either seconds or milliseconds since epoch (1st January 1970). The following mapping defining sets three dates with three different formats:

```
{
  ...
  "string_date": { "type": "date", "format": "dd-MM-yyyy" },
  "millis_date": { "type": "date", "format": "epoch_millis" },
  "seconds_date": { "type": "date", "format": "epoch_second" }
}
```

The given dates are converted internally to long values stored in milliseconds since epoch, equivalent to `epoch_millis`. We can use the range queries to fetch the dates, for example the

following snippet of code retrieves flights that were scheduled between 5 and 5.30am on a given date:

```
"range": {##A A range query fetching documents between two dates
  "departure_date_time": {
    "gte": "2021-08-06T05:00:00",##B In between 5 and 5.30am
    "lte": "2021-08-06T05:30:00"
  }
}
```

Finally, we can accept multiple date formats on a single field by declaring the required formats as shown in the following snippet:

```
"departure_date_time":{
  "type": "date",
  "format": "dd-MM-yyyy|dd/MM/yyyy|yyyy-MM-dd|yyyy/MM/dd" #A Setting four different formats on
  the field
}
```

Refer to Elasticsearch's date data type for more information on date data type as covering all options in this chapter is not feasible for this book. Also, refer to GitHub repository for the worked out examples on dates.

4.5.4 Numeric data types

Elasticsearch supplies a handful of numeric data types to handle integer and floating-point data. Table 4.3 provides the list of numeric types:

Table 4.3 Numeric data types

Integer types	byte	Signed 8 bit integer
	short	Signed 16 bit integer
	integer	Singed 32 bit integer
	long	Signed 64 bit integer
	unsigned_long	64 bit unsigned integer
Floating Point types	float	32 bit single precision floating-point number
	double	64 bit double precision floating-point number
	half_float	16 bit half precision floating-point number
	scaled_float	Floating-point number backed by long

We declare the field and its data type as "field_name": { "type": "short"}. The following code snippet demonstrates how we can create a mapping schema with few numeric fields:

```
"age":{  
  "type": "short"  
},  
"grade":{  
  "type": "half_float"  
},  
"roll_number":{  
  "type": "long"  
}
```

4.5.5 The boolean data type

The boolean data type represents the binary value of a field: true or false. For example, we can declare the field's type as boolean, shown in the snippet below:

```
PUT blockbusters
{
  "mappings": {
    "properties": {
      "blockbuster": {
        "type": "boolean"
      }
    }
  }
}
```

We can then index couple of movies: Avatar as a blockbuster and Mulan(2020) as a flop, shown in the code snippet below:

```
PUT blockbusters/_doc/2
{
  "title": "Avatar",
  "blockbuster": true
}

PUT blockbusters/_doc/2
{
  "title": "Mulan",
  "blockbuster": false
}
```

In addition to setting the field as JSON's boolean type (`true` or `false`), the field also accepts "stringified" boolean values such as `"true"` or `"false"` as you can see in the second example (Mulan). You can use a term (booleans are classified as structured-data) query to fetch the results, for example, the following query will fetch the Avatar as the blockbuster:

```
GET blockbusters/_search
{
  "query": {
    "term": {
      "blockbuster": {
        "value": "true"
      }
    }
  }
}
```

You can also provide an empty string for a false value: `"blockbuster": ""`.

4.5.6 The range data type

The `range` data types represent lower and upper bounds for a field. For example, if we want to select a group of volunteers for a vaccine trial, we can segregate the volunteers based on some categories such as age 25–50, 51–70, demographics such as income level, city dwellers, and so on. Elasticsearch supplies a `range` data type for supporting search queries on range data. The range is defined by operators such as `lte` (less than or equal to) and `lt` (less than) for upper bounds and `gte` (greater than or equal to) and `gt` (greater than) for lower bounds.

There are various types of range data types provided in Elasticsearch: date_range, integer_range, float_range, ip_range, and others. In the next section, we understand the date_range type in action.

THE DATE_RANGE TYPE EXAMPLE

The date_range date type helps index a range of dates for a field. We then can use range queries to match some criteria based on the lower and upper bounds of the dates.

Let's code an example to demonstrate the date_range type. Venkat Subramaniam is an award-winning author who delivers training sessions on various subjects from programming to design to testing. Let's consider a list of his training courses and the dates for our example.

We create a trainings index with two fields, name of the course and training dates - with text and date_range types respectively, as given in the listing 4.18:

Listing 4.18: Creating a trainings index with training_dates as date_range type

```
PUT trainings
{
  "mappings": {
    "properties": {
      "name": { #A: The name of the training session
        "type": "text"
      },
      "training_dates": { #B The training_dates is declared as date_range type
        "type": "date_range"
      }
    }
  }
}
```

Now that we have the index ready, let's go ahead and index a few documents with Venkat's training courses and dates.

```

PUT trainings/_doc/1 #A First document
{
  "name": "Functional Programming in Java",
  "training_dates": { #B Set of training dates
    "gte": "2021-08-07",
    "lte": "2021-08-10"
  }
}
PUT trainings/_doc/2 #C First document
{
  "name": "Programming Kotlin",
  "training_dates": { #D Set of training dates
    "gte": "2021-08-09",
    "lte": "2021-08-12"
  }
}
PUT trainings/_doc/3 #E First document
{
  "name": "Reactive Programming",
  "training_dates": { #F Set of training dates
    "gte": "2021-08-17",
    "lte": "2021-08-20"
  }
}

```

The `date_range` type field expects two values: an upper bound and a lower bound. These are usually represented by abbreviations like `gte` (greater than or equal to), `lt` (less than), and so on.

Now we prepped up the data, let's issue a search request (listing 4.19) to find out Venkat's courses between two dates:

Listing 4.19: Searching for Venkat's courses between two dates

```

GET trainings/_search
{
  "query": {
    "range": { #A We use a range query to search
      "training_dates": { # Search for course between these two dates
        "gt": "2021-08-10",
        "lt": "2021-08-12"
      }
    }
  }
}

```

As a response to the query, we see Venkat is delivering *Programming Kotlin* between these two dates (the second document matches for these dates). The `date_range` made it easy to search among a range of data.

In addition to `date_range`, we can create other ranges like `ip_range`, `float_range`, `double_range`, `integer_range`, and so on. Refer to my GitHub repository for more examples.

4.5.7 The IP (`ip`) address data type

Elasticsearch provides a specific data type to support internet protocol (IP) addresses: the `ip` data type. This data type supports both IPv4 and IPv6 IP addresses. To create a field of `ip` type, use `"field": {"type": "ip"}` as the following example shows:

```
PUT networks
{
  "mappings": {
    "properties": {
      "router_ip": { "type": "ip" } #A Type of the field is ip
    }
  }
}
```

Indexing the document is then straight forward:

```
PUT networks/_doc/1
{
  "router_ip": "35.177.57.111" #A Indexing a document with an ip address
}
```

Finally, we can use our search endpoint to search for the IP addresses that match our query. The following query searches for data in the `networks` index to get the matching IP address:

```
GET networks/_search
{
  "query": {
    "term": { #A Term level search for IP addresses
      "router_ip": { "value": "35.177.0.0/16" }#B Search for IPs in this range
    }
  }
}
```

In the previous couple of sections, we went over the core data types and learned about them in detail. Elasticsearch provides a rich set of data types for almost every use case we can think of. Some of the core data types are straightforward and pretty intuitive to work with while others like `object`, `nested`, `join`, `completion`, `search_as_you_type` require a bit of special attention. In the next section, we will learn about some of these advanced data types.

4.6 Advanced data types

In the last section, we learned about some of the core and common data types to represent our data fields. Some of which can be classified as advanced types; this includes some specialized types too. In the next few sections, we touch base on these additional types with definitions and examples.

NOTE **CODE ON GITHUB** Covering all the data types will surely bloat this book and I am also not a big fan of huge long chapters. So, I've taken a judicious call to provide some important and most likely useful advanced data types rather than filling up the chapter with each of them. Do refer to Book's GitHub repository for examples for the types cited here as well as the ones omitted.

4.6.1 The Geopoint (geo_point) data type

With the advent of smartphones and devices, location services and searching for nearest items have become more common. Most of us may have used a smart device to find the location of a nearest restaurant or asked for GPS directions to our mother-in-law's house during Christmas. Elasticsearch developed a specialized data type for capturing the location of a place.

Location data is expressed as a `geo_point` data type, which represents longitude and latitude. We can use this to pinpoint an address for a restaurant, a school, a golf course, and others. Let's see it in action. The code listing (4.20) demonstrates the schema definition of a `restaurants` index - restaurants with name and an address. The only notable point is that the address field is defined as a `geo_point` data type:

Listing 4.20: Mapping schema for restaurants with address declared as geo_point type

```
PUT restaurants
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "address": {
        "type": "geo_point"
      }
    }
  }
}
```

We index a sample restaurant (London based fictitious Sticky Fingers) with an address provided as a longitude and latitude (listing 4.21):

Listing 4.21: Indexing a restaurant with an address represented by longitude and latitude

```
PUT restaurants/_doc/1
{
  "name": "Sticky Fingers",
  "address": { #A The address is provided as a pair of longitude and latitude
    "lon": "0.1278",
    "lat": "51.5074"
  }
}
```

The address of the restaurant is given in the form of longitude (`lon`) and latitude (`lat`) pairs. There are other ways to provide these inputs too, which we look at shortly. But first let's fetch the restaurants within the location perimeter.

We can fetch restaurants using a `geo_bounding_box` query which is used for searching data involving geographical addresses. It takes inputs of `top_left` and `bottom_right` points to create a boxed up area around our point of interest, as demonstrated in the figure 4.9:

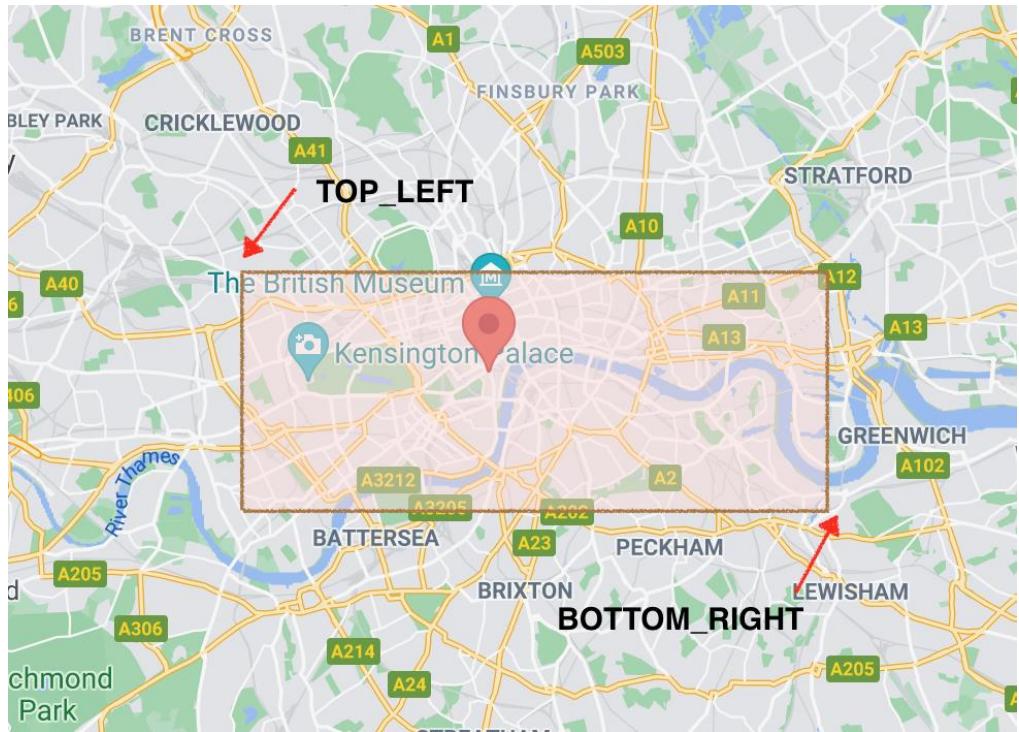


Figure 4.9: Geo bounding box of a location in central London

We provide the upper and lower bounds to this query using the lon (longitude) and lat (latitude) pairs. If you are curious about those coordinates, the address location points to London.

To search for a restaurant in a bounded area, we write the `geo_bounding_box` filter providing the address in the form of a rectangle with `top_left` and `bottom_right` coordinates given as latitude and longitude, as the listing 4.22 shows:

Listing 4.22: Fetching the restaurants around a geographical location

```
GET restaurants/_search
{
  "query": {
    "geo_bounding_box": {
      "address": {#A The top_left part of the box
        "top_left": {
          "lon": "0",
          "lat": "52"
        },
        "bottom_right": {#B The bottom_right part of the box
          "lon": "1",
          "lat": "50"
        }
      }
    }
  }
}
```

The query searches the restaurants that fall in a geo bounding box, represented as two geopoints (`top_left` and `bottom_right` in the query). This query fetches our restaurant because the geo bounding box encompasses our restaurant.

NOTE GEO BOUNDING QUERY COMMON MISTAKE When searching for an address using `geo_bounding_box`, a common mistake is providing the incorrect inputs (`top_left` and `bottom_right`) to the query. Make sure the longitude and latitudes of these two inputs make up the bounding box.

Earlier I mentioned we can provide the location information as not just longitude and latitude during the indexing. We can provide the location information in various formats, not just what you saw previously (providing as an object of `lat` and `lon`). You can provide the same location data in the form of an array or a string too as the following table shows:

Format	Explanation	Example
Array	Geo-point represented as an array. Note the order of geo-point inputs - it takes <code>lon</code> and <code>lat</code> (not the other easy as string format considers).	"address": [0.12, 51.5]
String	Geo-point as string data with <code>lat</code> and <code>lon</code> data.	"address": "51.5,0.12"
Geohash	Geohash is an encoded string formed from hashing the longitude and latitude coordinates. The alphanumeric string points to a place on the earth	u10j4
Point	Known as a well-known-text (WKT), the POINT represents a precise location on a map. The WKT is a standard mechanism to represent the geometrical data.	POINT(51.5, -0.12)

In this section, we have worked with geo-queries without any prior knowledge about them. We will cover them in detail in Chapter X: Advanced Search later in this book.

4.6.2 The object data type

Often we find data in a hierarchical manner - for example - an email object consisting of top level fields like subject as well as inner object to hold attachments, which in turn may have few more properties such as attachment file name, its type and so on.

JSON allows us to create such hierarchical objects - an object wrapped up in other objects. To represent such object hierarchy, Elasticsearch has a special data type to represent a hierarchy of objects - an `object` type.

We already know the data types for the top-level `subject` and `to` fields (`text` and `keyword` respectively). As the attachments is an object itself, its data type will surely be an `object` type. The two properties `filename` and `filetype` in the `attachments` object can be modelled as `text` and `long` field respectively. With this information at hand, we can create a mapping definition as demonstrated in the listing 4.23:

Listing 4.23: Defining the schema definitions for emails model with object data types

```
PUT emails
{
  "mappings": {
    "properties": {#A The top level properties for the emails index
      "to": {
        "type": "text"
      },
      "subject": {
        "type": "text"
      },
      "attachments": {#B The inner object consisting of second level properties
        "properties": {
          "filename": {
            "type": "text"
          },
          "filetype": {
            "type": "text"
          }
        }
      }
    }
  }
}
```

The `attachments` property is something we should draw our attention to. The type of this field is an `object` as it encapsulates the two other fields. The two fields defined inside the `attachments` inner object are no different to the `subject` and `to` fields declared at the top-level, except they are one level down.

Once the command is executed successfully, we can check the schema by invoking `GET emails/_mapping` command (listing 4.24):

Listing 4.24: The GET emails/_mapping response

```
{
  "emails" : {
    "mappings" : {
      "properties" : {#A Attachments is an inner object with other fields
        "attachments" : {#B The type is hidden here but it's object by default
          "properties" : {
            "filename" : {
              "type" : "text",#C Field's type shown as expected
              ...
            }
          }
        }
      }
    }
  }
}
```

The response consists of subject, to and attachments as the top-level fields. The attachments object have further fields, hence encapsulated as properties with the appropriate fields and their definitions. When you fetch the mapping (GET emails/_mapping), while all other fields show their associated data types, the attachments wouldn't. The object type of an inner object is inferred by Elasticsearch as default.

Let's index a document, listing 4.25 shows query:

Listing 4.25 Indexing an email document with all the relevant fields

```
PUT emails/_doc/1
{
  "to": "johndoe@johndoe.com",
  "subject": "Testing Object Type",
  "attachments": {
    "filename": "file1.txt",
    "filetype": "confidential"
  }
}
```

Now that we have primed our emails index with one document, we can issue a simple search query on the inner object fields to fetch the relevant documents (and prove our point), shown in the listing 4.26:

Listing 4.26: Searching for emails with a said attachment file

```
GET emails/_search
{
  "query": {
    "term": {
      "attachments.filename.keyword": "file1.txt"
    }
  }
}
```

This will return the document from our store as the filename matches to that of the document we have in store. Note that we are using a term query on a keyword as we wish to match on an exact field value (file1.txt).

While object types are pretty straightforward, there's one limitation that they carry: the inner objects are flattened out and not stored as individual documents. The downside of this action is

that the relationship is lost between the objects indexed from an array. Let's go over the limitation in the next section using a detailed example.

LIMITATION OF AN OBJECT TYPE

In our earlier emails example, the `attachments` field was declared as an object. While we create the email with just one attachment object, there's nothing stopping us creating multiple of these attachments (usually in an email we attach multiple files as attachments), as shown in the code listing 4.27 below:

Listing 4.27 Indexing a document with multiple attachments

```
PUT emails/_doc/2
{
  "to": "mrs.doe@johndoe.com",
  "subject": "Multi attachments test",
  "attachments": [
    {
      "filename": "file2.txt",
      "filetype": "confidential"
    },
    {
      "filename": "file3.txt",
      "filetype": "private"
    }
}
```

By default the `attachments` field will be of an `object type`, an inner object composed of an array of attachment files. Notice that classification type of the files:`file2.txt` is `confidential` and `file3.txt` is `private`, as indicated in the following table 4.x:

Table 4.x: The attachment's file name and its classified type.

Attachment Name	File Type
file2.txt	confidential
file3.txt	private

We have the document indexed for our email with id 1 - with a couple of attachments. Let's work through a simple search requirement: searching for the matching documents given a filename `file2.txt` and file type `private`. Looking at the data given in table 4.x, this query should return no results - as the `file2.txt`'s classification is `confidential` not `private`. Let's query and check if this is what was returned.

To write such queries, we need to use an advanced query called compound query which combines various leaf queries together to create a complex query. One such compound query is a `bool` search query (we will learn about developing search queries further along, so don't worry if it confuses you for now). Without going into details of how the `bool` query is constructed, let's look at it in action. The `bool` query is developed with two other query clauses:

- The first `must` clause checking for all the documents that match with the attachment's file name using a `term` query; and
- The second `must` clause checks whether the file classification is private

The query is given below in the listing 4.28:

Listing 4.28: Advanced bool query with term queries

```
GET myemails/_search #A Bool query search for a match with a filename and filetype
{
  "query": {
    "bool": {#B The query is defined as a bool query
      "must": [ #C The must clause defining the mandatory clauses
        {"term": { "attachments.filename.keyword": "file2.txt"}},
        {"term": { "attachments.filetype.keyword": "private" }}
      ]
    }
  }
}
```

Once executed, the result is expected to be returning no results because the combination of `file2.txt` with `private` file type doesn't exist. However, unfortunately that is not the case. As you can see the following (incorrect) response, Elasticsearch happily matched the criteria and returned the document, as the snippet below shows:

```
"hits" : [[#A The positive return result from the query
{
  ...
  "_source" : {
    "to:" : "mrs.doe@johndoe.com",
    "subject" : "Multi attachments test",
    "attachments" : [
      {
        "filename" : "file2.txt",
        "filetype" : "confidential"
      },
      {
        "filename" : "file3.txt",
        "filetype" : "private"
      }
    ]
  }
}]
```

This is where the object data type breaks down, i.e., it can't honour the relationships between the inner objects. Ideally, the values `file2.txt` and `private` are in different objects, so the search should't consider them as a single entity. The reason for this is that the inner objects are not stored as individual documents, they are flattened as shown below:

```
{
  ...
  "attachments.filename" :["file1.txt","file2.txt","file3.txt"]
  "attachments.filetype":["private","confidential"]
}
```

As you can see, the `filenames` are collected as an array and stored in the `attachments.filename` field and so are the file types too. Due to this way of storing them, the relationship between them is lost unfortunately. We can't say if the `file1.txt` is `private` or `confidential` as the array doesn't hold the state.

This is the limitation of indexing an array of objects for a field and trying to search them as individual documents. Good news is that we have another data type called the `nested` data to solve this project. Let's discuss this type in the next section.

4.6.3 The nested data type

From our previous example, we know the search query didn't bother to honour the individual document integrity (it fetched the results in spite of searching on data belonging to separate documents). We can fix this issue by introducing a new type called `nested` data type. The `nested` type is a specialized form of an `object` type where the relationship between the arrays of objects in a document is maintained.

Going with the same example of our emails and attachments, this time let's define the `attachments` fields as `nested` data type, rather than letting Elasticsearch derive it as an object type. This calls for creating a schema with declaring the `attachments` field as a `nested` data type. The schema is shown in the listing 4.29 here:

Listing 4.29: Mapping schema definition for `nested` type

```
PUT emails_nested
{
  "mappings": {
    "properties": {
      "attachments": {
        "type": "nested", #A The attachments field is declared as nested type
        "properties": {
          "filename": { #B The field is declared as keyword to avoid tokenizing
            "type": "keyword"
          },
          "filetype": {
            "type": "text" #C We can leave this field as text
          }
        }
      }
    }
  }
}
```

In addition to creating the `attachments` as nested types, we declared the `filename` as a `keyword` type for a reason. The value, for example: `file1.txt`, gets tokenized and gets split up as `file1` and `txt`. As a result, search query may get matched with a `txt` and `confidential` or `txt` and `private` as both records have `txt` as common token. To avoid this, we simply use the `filename` field as a `keyword` field. You can follow this method in the earlier example too (listing 4.28) where we've used `attachments.filename.keyword` in our search query.

Let's come back to the problem at hand - we have a schema definition created, so all we need to do is index a document. The listing 4.30 does this exactly:

Listing 4.30: Indexing a sample document into an index with nested property

```
PUT emails_nested/_doc/1
{
  "attachments" : [ #A Provide a couple of objects as attachments
    {
      "filename" : "file1.txt",
      "filetype" : "confidential"
    },
    {
      "filename" : "file2.txt",
      "filetype" : "private"
    }
  ]
}
```

Once this document is successfully indexed, the final piece of the jigsaw is the search. The listing 4.31 will demonstrate the search query written to fetch documents - criteria being emails with an attachment of `file1.txt` and `private` as the file name and its classification type respectively. This combination doesn't exist and hence the results must be empty, unlike in the case of an object where the data is searched criss-cross across documents returning the false positive results.

Listing 4.31: Fetching the documents that match `file1.txt` with `private` classification

```
GET emails_nested/_search
{
  "query": {
    "nested": { #A Formulating a nested query to fetch data from nested fields
      "path": "attachments", #B Path points to the name of the nested field
      "query": {
        "bool": {
          "must": [ #C Search clauses: must match with file1.txt and private
            { "match": { "attachments.filename": "file1.txt" } },
            { "match": { "attachments.filetype": "private" } }
          ]
        }
      }
    }
  }
}
```

The query in the listing 4.31 is searching for a file named `file1.txt` with a `private` classification, which doesn't exist (refer to our document 4.30). There are no documents returned for this query, which is exactly what we should expect. The `file1.txt`'s classification is `confidential` not `private` hence it didn't match. So, when a nested type represents the array of inner objects, the individual object is stored and indexed as a hidden document.

The nested data types are pretty good at honoring the associations and relationships, so if we ever need to create an array of objects where each of the objects must be treated as an individual object, perhaps the nested data type will become our friend.

No array types

While we are on the subject of arrays, interestingly, there is no array data type in Elasticsearch. We can, however, set any field with more than one value, thus representing the field as an array. For example, a document with one name field can be changed from a single value to an array: "name": "John Doe" to "name": ["John Smith", "John Doe"] by simply adding a list of data values to the field.

How does Elasticsearch deduce the data type when we provide the values as an array during dynamic mapping? The data type is derived from the type of the first element in the array, for example, "John Smith" is a string, hence the name is a text type in spite of its representation as an array.

There's one important point you must consider when creating arrays: you cannot mix up the array with various types. For example, you cannot declare the name field like this: "name": ["John Smith", 13, "Neverland"]. This is illegal as the field consists of multiple types and is not permitted.

4.6.4 Flattened (flattened) data type

So far we've looked at indexing the individual fields parsed from a JSON document. Each of the fields is treated as an individual and independent field when analyzing and storing it. However, sometimes we may not need to index all the subfields as individual fields thus going through the analysis process. Think of a stream of chat messages on a chat system , running commentary during a live football match, a doctor taking notes on his patient's ailments and so on. We can load this kind of data as one big blob rather than declaring each of the fields explicitly (or derived dynamically). Elasticsearch provides a special data type called flattened for this purpose.

A flattened data type holds information in the form of one or more subfields, each subfield's value indexed as a keyword. That is, none of the values are treated as text fields, thus do not undergo the text analysis process.

Let's consider an example of a doctor taking running notes about his/her patient during the consultation. The mapping consists of two fields: the name of the patient and the doctor notes. The notable point in this mapping is the declaration of `doctor_notes` filed as flattened type. The listing 4.32 provides the mapping:

Listing 4.32 Creating a mapping with flattened data type

```
PUT consultations
{
  "mappings": {
    "properties": {
      "patient_name": {
        "type": "text"
      },
      "doctor_notes": { # A This field composes of any number of fields
        "type": "flattened" #B Field is declared as flattened
      }
    }
  }
}
```

Any field (and its subfields) that's declared as flattened will not get analysed. That is, all the values are indexed as keywords; the analyzers are at bay when we index flattened field data. Let's create a patient consultation (listing 4.33) document and index it, as shown here:

Listing 4.33: Indexing the consultation document with doctor's notes

```
PUT consultations/_doc/1
{
  "patient_name": "John Doe",
  "doctor_notes": {# The flattened field can hold any number fields
    "temperature": 103, # All these fields are indexed as keywords
    "symptoms": ["chills", "fever", "headache"],
    "history": "none",
    "medication": ["Antibiotics", "Paracetamol"]
  }
}
```

As you can see, the `doctor_notes` holds a lot of information but remember we did not create these inner fields in our mapping definition. As the `doctor_notes` is a flattened type, all the values are indexed as keywords, that is, the analyzers are at bay when we index flattened field data.

Finally, we search the index using any of the keywords from the doctor notes, as the listing 4.33 demonstrates:

Listing 4.33: Searching through the flattened data type field

```
GET consultations/_search
{
  "query": {
    "match": {
      "doctor_notes": "Paracetamol" # Searching for patient's medication
    }
  }
}
```

Searching for "Paracetamol" will return our John Doe's consultation document. You can experiment by changing the match query to any of the fields, for example: `doctor_notes:chills`" or even write a complex query like the one shown below:

Listing 4.34 An advanced query on a flattened data type:

```
GET consultations/_search
{
  "query": {
    "bool": {
      "must": [{"match": {"doctor_notes": "headache"}}, {"match": {"doctor_notes": "Antibiotics"}}, {"must_not": [{"term": {"doctor_notes": {"value": "diabetics"}}}]}
    }
  }
}
```

In the query, we check for headaches and antibiotics but the patient shouldn't be diabetic - The query returns John Doe as he isn't diabetic but has headaches and is on antibiotics.

The flattened data types come handy especially when we are expecting a lot of fields on an adhoc basis and having to define the mapping definitions for all of them beforehand isn't feasible. Be mindful that the subfiles of a flattened field are always keyword types.

4.6.5 The Join (join) data type

If you are from a relational database world, you would know the relationships between data - the joins that enable the parent-child relationships. Every document that gets indexed is independent and maintains no relationship with any others in that index. Elasticsearch de-normalizes the data to achieve speed and gain performance during indexing and search operations. While maintaining and managing relationships in Elasticsearch is advised under caution, Elasticsearch provides a join data type to consider parent-child relationships should we need them.

Let's learn about join data in action by considering an example of doctor - patients (one-to-many) relationship: one doctor can have multiple patients and each patient is assigned to one doctor. To work with parent-child relationships using the join data type, we need to create a field of join type and add additional information via relations object mentioning the relationship (for example, doctor-patient relationship in the current context).

The query 4.35 here is preparing the doctors index with a schema definition:

Listing 4.36: Mapping of the doctors schema definition

```
PUT doctors
{
  "mappings": {
    "properties": {
      "relationship": {#A declare a property as join type
        "type": "join",
        "relations": {
          "doctor": "patient" #B Names of the relations
        }
      }
    }
  }
}
```

The query has two important points to note:

- Declare a relationship property of type join
- Declare a relations attribute and mention the names of the relations (doctor:patient, teacher:students, parent:children etc)

Once we have the schema ready and indexed, we index two types of documents: one representing the doctor (parent) and the other patients (child). Here's is the doctor's document, with relationship mentioned as doctor (listing 4.37):

Listing 4.37: Indexing a doctor document - note the relationship attribute

```
PUT doctors/_doc/1
{
  "name": "Dr Mary Montgomery",
  "relationship": {
    "name": "doctor" #A The relationship attribute must be one of the relations
  }
}
```

The relationship object declares the type of the document this is: `doctor`. The `name` attribute must be parent value (`doctor`) as declared in the mapping schema. Once we have resident *Dr Mary Montgomery* doctor ready, the next step is to get two patients associated with her. The following query (listing 4.38) does that:

Listing 4.38: Creating two patients for our doctor

```
PUT doctors/_doc/2?routing=mary #A Documents must have the routing flag set
{
  "name": "John Doe",
  "relationship": {#B Define the type of relationship in this object
    "name": "patient", #C The document is a patient
    "parent": 1#D Patient's parent (doctor) is document with ID 1
  }
}

PUT doctors/_doc/3?routing=mary
{
  "name": "Mrs Doe",
  "relationship": {
    "name": "patient",
    "parent": 1
  }
}
```

The relationship object should have the value set as `patient` (remember the parent-child portion of the `relations` attribute in the schema?) and the `parent` should be assigned with a document identifier of the associated doctor (ID 1 in our example).

There's one more thing we need to understand when working with parent-child relationships. The parents and associated children will be indexed into the same shard to avoid the multi-shard search overheads. And as the documents should co-exist, we need to use a mandatory routing parameter in the URL. Routing is a function that would determine the shard where the document will reside. We will look at the routing algorithm in Chapter 4 on Working with Documents.

Finally it's time to search for patients belonging to a doctor with ID 1. The query in listing 4.39 searches for all the patients associated with Dr Montgomery:

Listing 4.39: Fetching the patients of Dr Montgomery

```
GET doctors/_search
{
  "query": {
    "parent_id": {
      "type": "patient",
      "id": 1
    }
  }
}
```

When we wish to fetch the patients belonging to a `doctor`, we use a search query called `parent_id` that would expect the child type (`patient`) and the parent's ID (Dr Montgomery document ID is 1). This query will return Dr Montgomery's patients - Mr and Mrs Doe.

NOTE JUDICIAL USE OF PARENT-CHILD RELATIONSHIPS Implementing parent-child relationships in Elasticsearch will have performance implications. As we touch base in the first chapter, Elasticsearch may not be a right tool if you are considering document relationships, so use this feature judiciously.

4.6.6 Search as you type data type

Most search engines suggest words and phrases as we type in a search bar. This feature has few names - commonly known under few names: search as you type or typeahead or autocomplete or suggestions. Elasticsearch provides a convenient data type - `search_as_you_type` - to support this feature. Behind the scenes, Elasticsearch works very hard to make sure the fields tagged as `search_as_you_type` are indexed to produce n-grams, which we will see in action in this section.

Say, we are asked to support typeahead queries on a books index, i.e., when the user starts typing for a book's title letter by letter in a search bar, we should be able to suggest the book/s with those letters he/she typing.

First we need to create a schema with the field that's in question to be of `search_as_you_type` data type. The listing 4.40 provides this mapping schema:

Listing 4.40: Mapping schema for technical books with the title as search_as_you_type type

```
PUT tech_books
{
  "mappings": {
    "properties": {
      "title": {
        "type": "search_as_you_type" #A The title supports typeahead feature
      }
    }
  }
}
```

The notable point from the listing 4.41 is that the title in the schema definition is declared as `search_as_you_type` data type. We can index couple of documents with various titles (this book, plus my future titles, hopefully), as shown in the code listing 4.x:

Listing 4.41:Indexing a few books

```
PUT tech_books4/_doc/1
{
  "title":"Elasticsearch in Action"
}

PUT tech_books4/_doc/2
{
  "title":"Elasticsearch for Java Developers"
}

PUT tech_books4/_doc/3
{
  "title":"Elastic Stack in Action"
}
```

As the title field's type is of `search_as_you_type` data type, Elasticsearch creates a set of subfields called ngrams, in addition to the root field(title), with various shingle token filters, as shown in the table 4.x

Table 4.x. The subfields created automatically by the engine

Fields		
title._index_prefix	The analyser of title_3gram will be wrapped with an edge n-gram token filter	Generates edge ngrams for the field title._3grams.
title._2gram	The analyser of title will be wrapped up with the shingle token filter of shingle size 2	Generates two tokens for the given text. For example, the 2grams for "action" word ar: ["ac", "ct", "ti", "io", "on"]
title._3gram	The analyser of title will be wrapped up with the shingle token filter of shingle size 3	Generates three tokens for the given text. For example, the 3grams for "action" word ar: ["act", "cti", "tio", "ion"]
title	The default analyser will be applied if no configured analyser is found	If a standard analyzer is used, the title gets tokenized and normalized based on the standard analyzer's rules.

As these fields are additionally created for us, searching on the field is expected to return the typeahead suggestions as the ngrams help produce them effectively.

Let's create the search query as shown in the listing 4.42.

Listing 4.42: Searching in a search_as_you_type field and its subfields

```
GET tech_books4/_search
{
  "query": {
    "multi_match": {
      "query": "in",
      "type": "bool_prefix",
      "fields": ["title","title._2gram","title._3gram"]
    }
  }
}
```

This query should return the *Elasticsearch in Action* and *Elastic Stack in Action* books. We use a multi match query because we are searching for a value across multiple fields - title, title._2gram, title._3gram, title._index_prefix.

Ngrams, Edge ngrams and Shingles .

The ngrams, edge ngrams and shingles are new concepts for us, let me briefly explain about them here but we will learn in detail in the chapter on Text Analysis.

The ngrams are a sequence of words for a given size. You can have 2-ngrams, 3-ngrams etc. For example, if the word is "action", the 3-ngram (ngrams for size 3) are: ["act", "cti", "tio", "ion"] and bi-grams (size 2) are: ["ac", "ct", "ti", "io", "on"] and so on.

Edge ngrams, on the other hand, are ngrams of every word, where the start of the n-gram is anchored to the beginning of the word. Considering the "action" word as our example, the edge n-gram produces: ["a", "ac", "act", "acti", "actio", "action"].

Shingles on the other hand are word n-grams. For example, the sentence "Elasticsearch in Action" will outputting: ["Elasticsearch", "Elasticsearch in", "Elasticsearch in Action", "in", "in Action", "Action"]

We will learn more about these concepts in the Chapter on Text Analysis, stay tuned.

We briefly looked at the multi-fields: fields with more than one data type. Sometimes we may want to have a field that we can declare as more than a single data type. For example, a title of a movie could be both `text` and `completion` data types. Fortunately, Elasticsearch provides us with a facility to declare a single field with multiple data types. Let's find out how we can achieve this multi typed field in the next section.

4.7 Multiple data types

We learned that each field in a document is associated with a data type. However, Elasticsearch is flexible to let us define the fields with multiple data types too. For example, a `subject` field in our emails data can be a `text` or a `keyword` or a `completion` type depending on the requirements. We can create multiple types in our schema definitions using the `fields` object inside the main field definition. It goes with the following syntax:

```
"my_field1":{  
    "type": "text",#A Declare the type of my_field1  
    "fields": {#B Define a fields object to enclose more types  
        "kw":{ "type":"keyword" }#C Declare additional field with a label "kw"  
    }  
}
```

Basically, `my_field1` is indexed as a `text` type as well as `keyword` type. When we expect to use it as `text`, we can simply provide the field as `my_field1` in the queries. We use the label `my_field1.kw` (note the dot notation) as the field name when searching it as a keyword. Let's look at the example schema definition (listing 4.43) that creates our single field `email` with multiple data types (`text`, `keyword`, and `completion`):

Listing 4.43 Schema definition with multi-typed field

```
PUT emails  
{  
    "mappings": {  
        "properties": {  
            "subject":{  
                "type": "text",#A The text type  
                "fields": {  
                    "kw":{ "type":"keyword" },#B The subject is also a keyword  
                    "comp":{ "type":"completion" }#C Subject is completion type too  
                }  
            }  
        }  
    }  
}
```

The `subject` field now has three types associated with it: `text`, `keyword`, and `completion`. If you want to access these, you have to use the format `subject.kw` for the keyword type field or `subject.comp` for the completion type.

We have learned a lot about mapping concepts in this chapter. Let's wrap up here and look forward to the next chapter, which will be all about working with documents.

4.8 Summary

- Every document consists of fields with values and each of the fields has a data type. Elasticsearch provides a rich set of data types to represent these values.
- Elasticsearch consults a set of rules when indexing and searching data. These rules, called mapping rules, let Elasticsearch know how to deal with the varied data shapes.
- Mapping rules are formed either by dynamic or explicit mapping processes.
- Mapping is a mechanism of creating field schema definitions up front. Elasticsearch consults the schema definitions while indexing the documents so the data is analyzed and stored for faster retrieval.
- Elasticsearch also has a default mapping feature where we can let Elasticsearch derive the mapping rather than providing it explicitly ourselves. It deduces the schema based on the first time it sees a field.
- Although dynamic mapping is handy, especially in development, if we know more about the data model, it is best to create the mapping beforehand.
- Elasticsearch provides a wide range of data types such as text, boolean, numericals, dates, etc., and stretching to complex fields like join, completion, geopoints, nested, and others.

5

Working with documents

This chapter covers

- Indexing, retrieving, and reindexing documents
- Manipulating and tweaking responses
- Updating and deleting documents with APIs and query methods
- Working with scripted updates
- Indexing documents in bulk

It is time to work with and understand operations on documents in Elasticsearch. Documents are indexed, fetched, updated, or deleted as per the requirement. We can load the data into Elasticsearch either from various stores such as databases and files or from real time streams. Similarly, we can update or modify the data that exists in Elasticsearch. If needed, we can even delete and purge the documents. For example, we may have a product catalog database that needs to be imported into Elasticsearch to enable search capabilities on the products.

Elasticsearch exposes a set of APIs for working with our documents:

- *The document indexing APIs*—Indexes documents into Elasticsearch
- *The read and search APIs*—Allows clients to fetch documents from Elasticsearch
- *The update APIs*—Allows us to modify the fields of a document
- *The delete APIs*—Removes documents from the store

Elasticsearch classifies the APIs into two categories: single document APIs and multi-document APIs. The *single document APIs*, as the name suggests, are applied to perform operations like indexing, fetching, modifying, and deleting documents one by one, using the appropriate endpoints. These APIs are useful when working with events such as orders generated by an e-commerce application or tweets from a set of Twitter handles, and so forth. We will use the single document APIs to operate on these documents individually.

The *multi-document APIs*, on the other hand, are geared towards working with documents in batches. For example, we may have a requirement to import a product catalog from a database consisting of millions of records into Elasticsearch. Elasticsearch exposes a bulk (`_bulk`) endpoint API for this purpose to help import the data in batches.

Elasticsearch also exposes advanced query-based APIs to work with our documents. We can delete and update a number of documents that match a certain criteria by developing a complex query if needed. We will use sophisticated search queries too to find and update or delete documents. Finally, we can also move data from one index to another by using the re-indexing APIs. Re-indexing helps us migrate our data without any downtime in production, albeit performance implications are to be considered.

In this chapter, we will go over single and multi-document APIs and various operations on documents in detail. The first step in working with Elasticsearch is to get some data into the engine. Let's look at the indexing document APIs and the mechanics of indexing these documents in the next section.

NOTE The code for this chapter is available on GitHub here: [Kibana Script for Ch5. Working with Documents](#).

5.1 Indexing documents

Just like we insert records into a relational database, we add the data (in the form of documents) into Elasticsearch. These documents sit in a logical bucket, called an *index*. The act of persisting the documents to these indexes is known as *indexing*. So, when we hear the term *indexing*, it means nothing but storing or persisting the documents into Elasticsearch.

Text analysis

The documents undergo a process called *text analysis* in Elasticsearch before getting stored. The analysis process prepares the data to make it suitable for varied search and analytical features. It is this analysis that gives a search engine the ability to deliver relevancy and full-text search capabilities.

As we discussed in the opening paragraphs of this chapter, Elasticsearch provides us with APIs to index both single and multiple documents into Elasticsearch. We will go over these document APIs in detail in the next section.

NOTE Refer to Chapter X: Dealing with Data to learn more about documents, their origination, and the popular JSON representation of them in detail. That chapter also deals with the advanced concepts of read/write models and addressing shard failures.

5.1.1 Document APIs

We communicate with Elasticsearch using RESTful APIs over HTTP. We can perform basic CRUD (create, read, update, and delete) operations using single document APIs. There are also APIs designed to work with multiple documents too, instead of targeting a single document. We will learn about both types in this chapter but, for now, let's focus on learning

how to index documents using single document APIs. Before we learn about indexing the documents, however, there's one important concept you need to understand: document identifiers. We discuss this in more detail in the next section.

DOCUMENT IDENTIFIERS

Each document that we index can have an identifier, usually specified by the user. For example, the document for the movie *The Godfather* can be given an identifier (say, `id = 1`), and the movie *Shawshank Redemption* can be given another identifier (`id = 2`). These identifiers, similar to a primary key in a relational database, are associated with that document for its lifetime (unless deliberately changed for whatever reason).

On the other hand, there are times when the client (user) doesn't really need to give an identifier to the document. Imagine an automatic car sending thousands of alerts and heart beats to a server. Each of these messages doesn't need to have a sequence of identifiers. They can exist with a random ID, albeit a unique one. In this case, the system generates a random identifier (UUID) for the document that's getting indexed.

Document APIs allow us to index documents with and without an identifier. There is a subtle difference in using the HTTP verb such as POST or PUT, however.

- If a document has an identifier provided by the client, we use the HTTP PUT method to invoke a document API for indexing the document.
- If the document does not have an identifier provided by the client, we use the HTTP POST method when indexing. In which case, the document inherits the system-generated identifier once indexed.

Both of these HTTP methods do the honors of indexing a new document in Elasticsearch. Let's check out the process of indexing documents using both methods in the following sections.

INDEXING A DOCUMENT WITH AN IDENTIFIER (PUT)

When a document has an identifier, we can use the single document index API with HTTP's PUT action to index the document. The syntax for this method is as follows:

```
PUT <index_name>/_doc/<identifier>
```

The `<index_name>` is the name of the index where the document is going to be housed, and `_doc` is the endpoint that must be present when indexing a document. The `<identifier>` is the document's identity (like a primary key in a database world), which is a mandatory path parameter if using the HTTP PUT method.

Let's index our movie document using the API. Head over to Kibana and write and execute the query to index our first movie document as shown in the listing 5.1:

Listing 5.1: Indexing a new document into movies index

```
PUT movies/_doc/1 #A Document indexing url
{
  #B Body of the request
  "title": "The Godfather",
  "synopsis": "The aging patriarch of an organized crime dynasty transfers control of his
              clandestine empire to his reluctant son"
}
```

The URL `PUT movies/_doc/1` is the RESTful method that invokes the document index API. This request carries a body, which is represented by the enclosing JSON document (with the movie data). The results in the response is shown in figure 5.1:

```

PUT movies/_doc/1
{
  "title": "The Godfather",
  "synopsis": "The aging patriarch
  of an organized crime ..."
}

A request to the server to index a
movie document with an id as 1.
The body of the request is a JSON
document
  
```

```

{
  "_index": "movies",
  "_type": "_doc",
  "_id": "1",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 4,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 0,
  "_primary_term": 1
}
  
```

Response from the server
indicating that the movie
document was indexed
successfully

Figure 5.1 Indexing a document with an identifier.

Let's quickly go over the constituents of the URL as discussed in the table 5.1 here:

Table 5.1: The PUT URL constituents

PUT	The HTTP verb indicates that we are asking the server to create a new resource. The usual convention for a PUT action is to send some data to the resource URL so the server will create the new resource in its store.
movies	The name of the index where we want our movie document to persist.
_doc	The endpoint of the service call. In the earlier Elasticsearch versions (before 5.x), the URL had a type associated with it (something like <code>movies/movie/1</code>). The type of the document was deprecated and, hence, a generic endpoint <code>_doc</code> was amended to the URL (making it <code>movies/_doc/1</code>).
1	The path parameter to indicate the resource's identifier (the movie document's identifier).

Request in cURL format

Kibana shortens the URL to look pretty. Behind the scenes it expands the URL by adding the server details to the request. This is possible because every Kibana instance is connected to the Elasticsearch server implicitly (the `kibana.yml` configuration file defines the server's details). The URL in cURL format is

```
curl -XPUT "http://localhost:9200/movies/_doc/1" -H 'Content-Type: application/json' -d'{ "title":"The Godfather", "synopsis":"The aging patriarch .."}'
```

You can fetch the cURL command from Kibana's DevTools - just click on the spanner icon and copy your request as URL from the "Copy as cURL" as shown in the figure 5.2:

The screenshot shows the Kibana DevTools interface. A JSON document is being edited in the main pane:

```
POST movies_reviews/_doc
{
  "movie": "The Godfather",
  "user": "Peter Piper",
  "rating": 4.5,
  "remarks": "The movie started with a ..."
}
```

A context menu is open over the JSON document, with the following options visible:

- Copy as cURL
- Open documentation
- Auto indent

Figure 5.2: Exporting the query as cURL

While you are here, you can click on Auto indent link to get your code indented in a JSON way.

After executing the preceding script, we expect the response in the pane on the right-hand side of Kibana. Figure 5.3 shows the response.

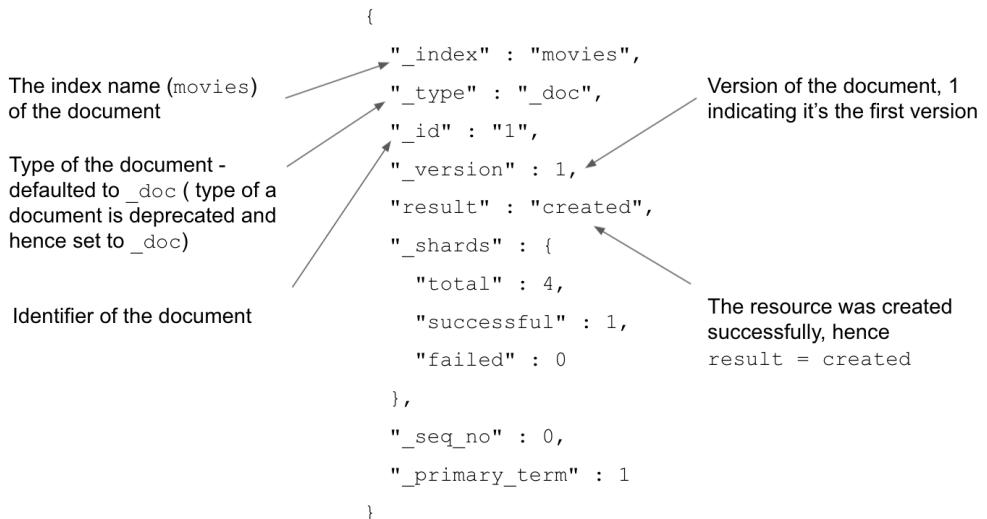


Figure 5.3 Response from the server when a document is indexed successfully

Let's dissect the response briefly. The response is a JSON document with few attributes. The `result` attribute indicates the operation's action: `created` here indicates that we successfully indexed a new document in the Elasticsearch's store. The `_index`, `_type`, and `_id` attributes are derived from our incoming request and assigned to the document. `_version` indicates the current version of this document; the value `1` here means the first version for the document. The number is incremented if we modify the document and reindex it. We will visit the `_shards` information down the line, but all it's saying here is that the document's journey to the store was completed. That's pretty much what you see as a response for creating a document.

If you are curious, re-execute the same command and check the response. You'll notice the `result` attribute changed to `updated` and the version number incremented. Every time we execute the query the version gets updated.

While we are here, let's touch base with the index. When we first tried to insert the document, there was no `movies` index pre-created for us. The document was first seen by the server to only realize the nonexistent index. It took the liberty to create a `movies` index for us, so our journey of indexing documents goes smoothly. The schema of this index was nonexistent too, but the server derived the schema by analyzing the incoming first document. We learned about mapping definitions in the last chapter, so I advise you to refer to chapter 4 (section 4.1.2) if needed.

We just indexed a document that has an identifier. Not every document can or will have a set identity that may have a business need. For example, while it's a good idea for a movie document to have an identifier, the traffic alerts from various traffic lights received by a traffic management system don't need to be associated with an identifier. Indexing a

document without an identifier follows the same process as indexing one with an identifier, but executing the HTTP verb changes from PUT to POST. We see this in action in the next section.

INDEXING A DOCUMENT WITHOUT AN IDENTIFIER (POST)

In the last section, we indexed a document that had an identifier associated with it. We used the HTTP PUT method to create the resource in Elasticsearch. But not every business model promotes data documents with identifiers (for example, price ticks or tweets). In such cases, we will have to let Elasticsearch generate random identifiers. To index these documents, instead of using the PUT method as we did earlier, we use POST action.

The HTTP POST method follows the similar format as the HTTP PUT except the identifier is not provided as part of the URL parameter. The format is provided in the listing 5.2 here:

Listing 5.2 Indexing a document with no identifier using POST

```
POST myindex/_doc #A The URL has no identifier attached to it
{
  "title":"Elasticsearch in Action" #B The request body is a JSON document
}
```

In listing 5.1, we invoked the `_doc` endpoint on the index *without an identifier* but with a body. This POST request tells Elasticsearch that it needs to assign the documents newly generated random identifiers during the indexing process.

NOTE The POST method doesn't expect the user to provide the document's identifier (ID). Instead, it automatically generates a randomly created UUID for the document when it gets persisted.

As an example, let's consider users posting movie reviews . Each movie review is captured as a JSON document, which is sent to Elasticsearch from Kibana. We are not going to provide the document's ID for this request. The listing 5.3 demonstrates this:

Listing 5.3: Indexing a movie review without an ID

```
POST movies_reviews/_doc
{
  "movie":"The Godfather",
  "user":"Peter Piper",
  "rating":4.5,
  "remarks":"The movie started with a ..."
```

Once the server executes the index request, the response is sent back to the Kibana console. Figure 5.4 illustrates this process.

Response from the server

```
{
    "_index" : "movies_reviews",
    "_type" : "_doc",
    "_id" : "53NyfXoBW8A1B2amKR5j",
    "_version" : 1,
    "result" : "created",
    "_shards" : {
        "total" : 4,
        "successful" : 1,
        "failed" : 0
    },
    "_seq_no" : 0,
    "_primary_term" : 1
}
```

The ID is a auto-generated UUID generated by the server and assigned to movie review

The result indicates the document was indexed successfully

Figure 5.4 The server creates and assigns a randomly auto generated ID to the document.

In the response, the `_id` field seems to be randomly generated data while the rest of the information is the same as the earlier PUT request shown in the listing 5.1. You may be wondering how we decide which path to consider when indexing the document, the PUT or POST method. The next section looks at both methods for us.

When to use PUT and/or POST

If you want to control the identifiers or if you already know the IDs of these documents, you should use PUT to index the documents. These could be your domain objects that may have a predefined identity strategy (like primary keys) that you may want to adhere to. Retrieving the documents using an ID might be a reason to consider which way you lean. If we know the document ID, we can use the document APIs to get the document (we'll learn about retrieval in a short while). For example, we provide an ID to a movie document, as the code below shows:

```
POST movies/_doc/1
{
  "movie": "The Godfather",
}
```

On the other hand, using identifiers for documents originating from streaming data or time-series events doesn't make sense (imagine price quotes originating from a pricing server, share price fluctuations, system alerts from a cloud service, tweets, or heartbeats from an automated car, etc.). Having a randomly generated UUID is good enough for these events or messages. But because the IDs are randomly generated, we may need to write a search query to retrieve the documents rather than simply retrieving those using IDs as we do for PUTs. In summary, you should use the HTTP POST action to index documents that don't have a business identity. For example, we don't provide an ID to a movie review document, as the code below shows:

```
POST movies_reviews/_doc
{
  "review": "The Godfather movie is a masterpiece..",
}
```

When indexing a document, the document index API doesn't care if the document exists or not. If we index it for the first time, the document is created and stored as expected. If we index the same document again, it gets saved even if the content is completely different than the earlier document. Does Elasticsearch block the operation of overwriting the document's content? Let's find out in the next section.

USING _CREATE TO AVOID OVERRIDING A DOCUMENT

Let's slightly change course and find out what happens if we execute the following query:

Listing 5.4: Indexing the incorrect document content

```
PUT movies/_doc/1
{
  "tweet": "Elasticsearch in Action 2e is here!" #A Not a movie but a tweet
}
```

In the example, we are indexing a document with a tweet into a `movies` index with a document ID of 1. Hold on a second; didn't we already have a movie document (*The Godfather*) with that ID in our store? Yes, we do. Elasticsearch has no control on such overwriting operations. The responsibility is passed down to the application or to the user.

Rather than depending on the user's discretion that can lead to incorrect data due to accidental overwrites, Elasticsearch provides another endpoint: `_create`. This endpoint

solves the overwrite situation. We can use the `_create` endpoint in place of `_doc` when indexing a document to avoid overriding the existing one. Let's look at this in action.

The `_create` API in action

Let's index the movie document with an identifier 100, but this time let's use `_create` endpoint. Listing 5.5 shows the operation's invocation:

Listing 5.5: Indexing a new movie using the `_create` endpoint.

```
PUT movies/_create/100
{
  "title": "Mission Impossible",
  "director": "James Cameron"
}
```

We've indexed a new movie (*Mission Impossible*), this time using the `_create` endpoint instead of `_doc`. The fundamental difference between these two methods is that the `_create` method will not let you reindex the document with the same ID again, while the `_doc` wouldn't mind.

Next, let's try changing the content on the document completely by sending a tweet message as the movie document. The query is shown in the listing 5.6 here:

Listing 5.6 Adding an additional field to the movie document and updating it

```
PUT movies/_create/100 #A Index a tweet in place of an existing movie
{
  "tweet": "A movie with popcorn is so cool!" #B Not a movie but a tweet
}
```

We are (may be accidentally) overriding the contents of the movie document. However, Elasticsearch will not let us do so and it will throw a version conflict error as shown in the following snippet:

```
{
  "type" : "version_conflict_engine_exception",
  "reason": "[100]:version conflict,document already exists(current version[1])"
}
```

Elasticsearch didn't let the data be overwritten, as you can see in the code snippet, we received a version conflict exception when we attempted to do that. This is the `_create` API's way of indicating the document cannot be updated because the version already exists.

NOTE Although the `_create` endpoint did not allow us to update the document, we can swap the `_create` endpoint with `_doc` to get it updated if that's what your intention is.

The takeaway for this section is that if we need to protect the documents by not allowing them to be overwritten accidentally, we should use the `_create` API instead.

Disabling index autocreation

Elasticsearch, by default, auto-creates a required index if the index doesn't already exist. If we want to restrict this feature, we need to set a flag called `action.auto_create_index` to `false`. This can be done in two ways:

- Set the flag to `false` in the `elasticsearch.yml` config file or
- Explicitly set the flag by invoking `_cluster/settings` endpoint as the listing 5.7 shows here:

Listing 5.7: Disabling index auto creation

```
PUT _cluster/settings
{
  "persistent": {
    "action.auto_create_index": "false"
  }
}
```

This code snippet stops Elasticsearch from creating indexes automatically. The call `PUT my_new_index/_doc/1`, for example, fails if `action.auto_create_index` is set to `false`. You may want to do this if you have already created the index manually (most likely with a predefined settings and mapping schema) and don't need to allow creation of the indices on demand. We will learn more about indexing operations in Chapter 5: *Index Management*.

Now that we understand how the documents get persisted, we need to understand the mechanics of how Elasticsearch stores them. Let's focus on the details about how the indexing process works in the next section.

5.1.2 Mechanics of indexing

We briefly looked at how indexing works in section 3.2.3 in the last chapter (Chapter 3: Architecture). In this section, we will go over the details of the mechanics involved when a document is indexed (figure 5.5). As we already know, shards are Lucene instances that hold the physical data that's logically associated with an index.

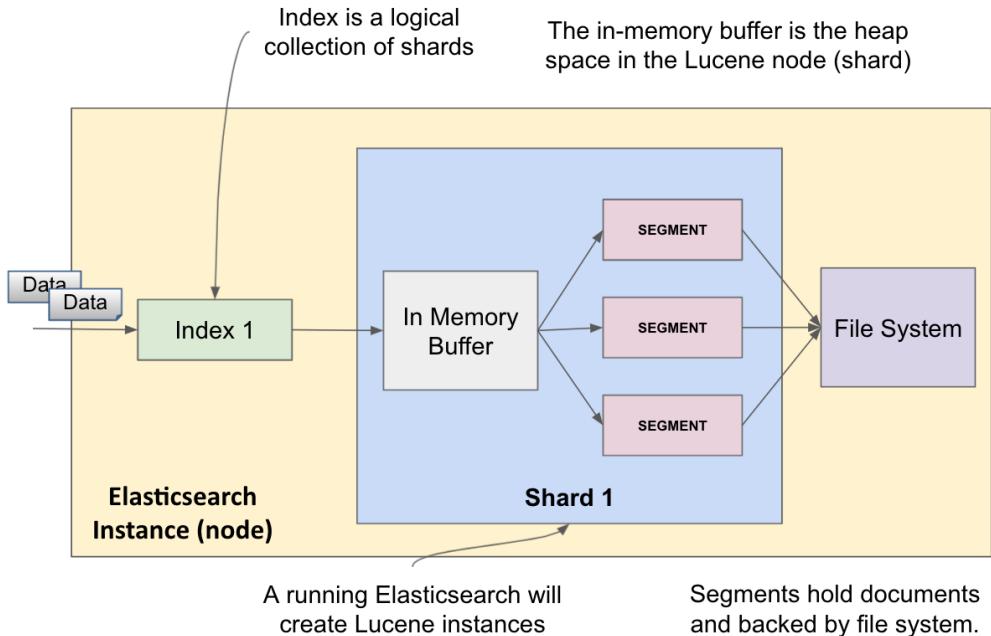


Figure 5.5 Mechanics of indexing documents

When we index a document, the engine decides which shard it will be housed in, based on the routing algorithm (we discussed routing algorithms in section 3.5 in Chapter 3). Each shard comes with heap memory, and when a document is indexed, the document is first pushed into the shard's in-memory buffer. The document is held in this in-memory buffer until a refresh occurs. Lucene's scheduler issues a refresh every 1 second to collect all the documents available in the in-memory buffer, and creates a new segment with these documents. The segment consists of the document data and inverted indexes. The data is first written to the filesystem cache and then committed to the physical disk.

As I/O operations are expensive, Lucene avoids frequent I/O operations when writing data to the disk. Hence, it waits for the refresh interval (default is one second), after which the documents are bundled up to be pushed into segments. Once the documents are moved to the segments, they are then made available for searching.

Apache Lucene is an intelligent library when dealing with data writes and reads. After pushing the documents to a new segment (during the refresh operation), it waits until three segments are formed. It uses a three-segments-merge pattern to merge the segments to create new segments: that is, whenever the three segments are ready, Lucene will instantiate a new one by merging these three segments. And awaits for three more segments to be created so it can create a new one and so on. Every three segments merge to create another segment as shown in figure 5.6.

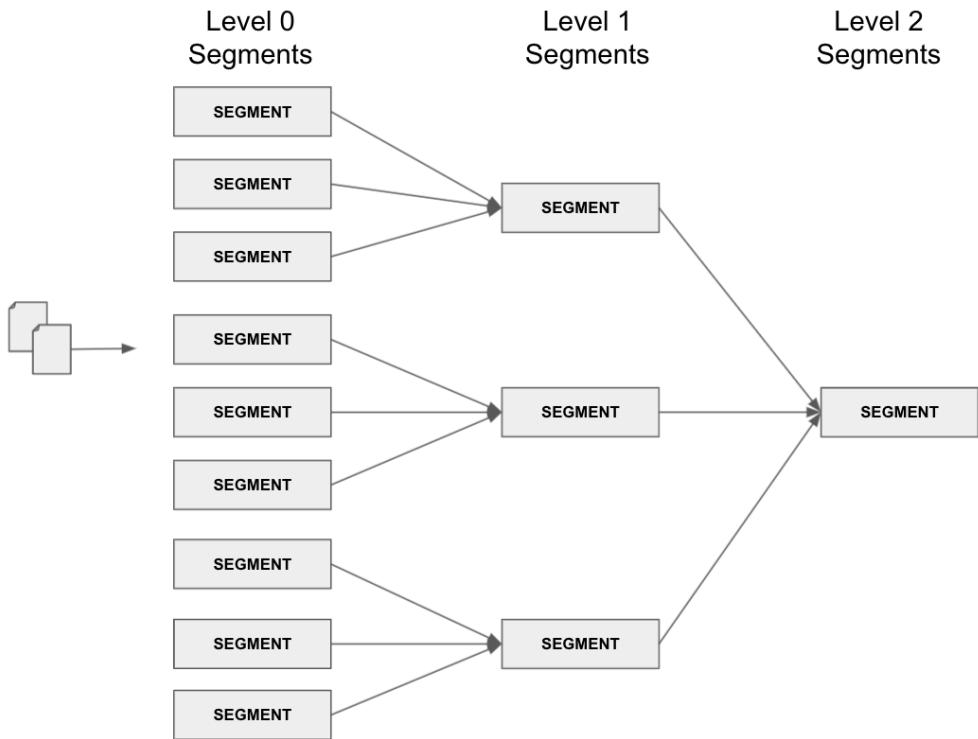


Figure 5.6 Illustration of how segments get merged in Apache Lucene

During the 1-second refresh interval, a new segment is created to hold however many documents are collected in the in-memory buffer. The heap memory of the shard (Lucene instance) dictates the number of documents it can hold in an in-memory buffer before flushing it out to the file store. For all practical purposes, Lucene treats the segments as immutable resources. That is, once a segment is created with available documents from the buffer, any new document will not make its way into this existing segment. Instead, it's moved into a brand new segment. Similarly, the deletes are not performed physically on the documents inside the segments, but they are marked for removal later. Lucene employs this strategy to provide high performance and throughput.

5.1.3 Customizing the refresh

Documents that get indexed live in memory until the refresh cycle kicks in. The documents get moved as segments into the filesystem during the refresh process and, thus, are available to search. The refresh process is expensive, especially if the engine is expected to be taking a larger amount of the indexing requests.

The good news is that we can configure this refresh setup. We can reset the time interval from the default 1 second to, say, 60 seconds by tweaking the settings on an index level using `_settings` endpoint as demonstrated in this code listing 5.8:

Listing 5.8: Setting custom refresh interval

```
PUT movies/_settings
{
  "index": {
    "refresh_interval": "60s"
  }
}
```

This is a dynamic setting, which means that we can change the refresh setting on a live index at any time. To switch off the refresh operation completely, set the value to `-1`. The in-memory buffer will accumulate the incoming documents if the refresh operation is off. The use case for this scenario might be that we are migrating tons of documents from a database into Elasticsearch, and we don't want the data to be searchable until the migration completes successfully. To once again enable refresh manually on the index, we simply need to issue a `POST <index>/_refresh` command.

We can also control the refresh operation from the client side for CRUD operations on documents by setting the `refresh` query parameter. The document APIs (`index`, `delete`, `update`, and `_bulk`) expect `refresh` as a query parameter. For example, the following snippet advises the engine to kick off the refresh once the document has been indexed as opposed to waiting for the refresh interval to expire:

```
PUT movies/_doc/1?refresh
```

The refresh query parameter can expect three values:

- **refresh=false** (default)—By setting the `refresh=false` like for example `PUT movies/_doc/1?refresh=false`, we are letting the engine not to force the refresh operation instead apply the default setting (of 1 second). Engine makes the document available for search only after the predefined interval refresh. We can also not provide the query parameter, which achieves the same effect.

Example: `PUT movies/_doc/1?refresh=false` or `PUT movies/_doc/1?refresh`

- **refresh=true** (or empty string) —Forces the refresh operation and, hence, the document is visible for searching immediately. If our refresh time interval was set to 60 seconds and if we index a thousand documents with `refresh=true` setting, all those thousand documents are expected to be available to search instantly rather than waiting for the 60s refresh interval.

Example: `PUT movies/_doc/1?refresh=true`

- **refresh=wait_for**—A blocking request that compels the client to wait until the refresh operation kicks in and completes before the client's request is returned. If our `refresh_interval` is 60 seconds, for example, the request is blocked for 60 seconds until the refresh is performed. However, It can be manually started, however, by invoking `POST <index>/_refresh` endpoint.

Example: `PUT movies/_doc/1?refresh=wait_for`

Now that we have a couple of documents, we need to understand the mechanics of retrieving them too. Elasticsearch provides a GET API for reading the documents, which is similar to the indexing APIs we saw previously. We will go over the mechanics of reading the documents from the Elasticsearch stash in the next section.

5.2 Retrieving documents

Elasticsearch provides two types of document APIs for retrieving documents:

- The single document API that returns one document given an ID, and
- The multi-document API that returns multiple documents given an array of IDs.

If a document is not available, we will receive a JSON response indicating that the document is not found. Let's find out about retrieving documents using both APIs in this section.

5.2.1 Using the single document API

Elasticsearch exposes a RESTful API to fetch a document given the document ID, similar to the indexing API we learned about in the previous section. The API definition for getting a single document is

```
GET <index_name>/_doc/<id>
```

The GET is the HTTP method that indicates we are fetching a resource. The URL indicates the resource's endpoint - in this case, the `index_name` followed by `_doc` and the ID of the document.

As you may have already noted, the difference between indexing the document and fetching is just that HTTP verb, modifying PUT/POST to GET. There are no changes to the URL in both the indexing and the reading/retrieval API whatsoever. This follows the RESTful services best practices.

Let's retrieve our movie document with an ID as 1, which we indexed in the earlier section. Executing a `GET movies/_doc/1` command on the Kibana console fetches the previously indexed movie document with the ID 1. The response is shown in the figure 5.7 here::.

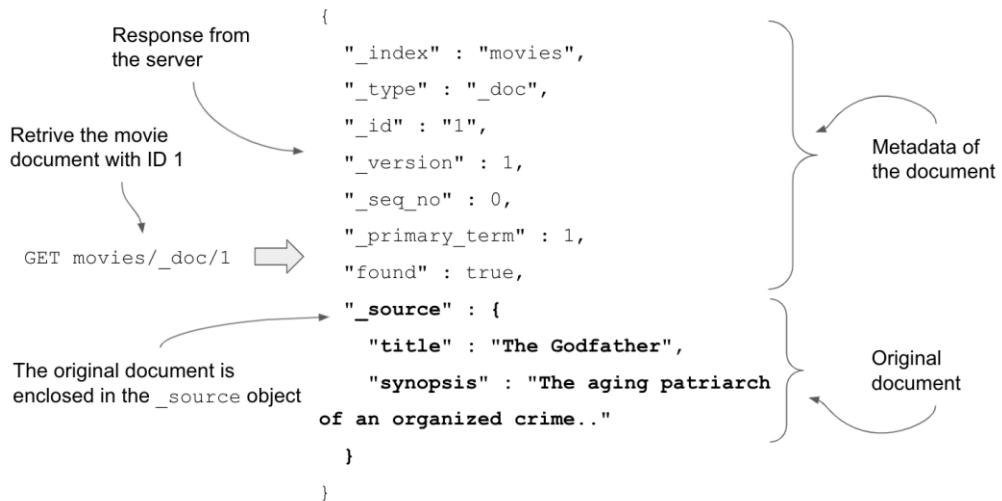


Figure 5.7 Retrieving a document using the GET API call

The JSON response you see in figure 5.8 has two parts: the metadata and the original document. The metadata consists of `_id`, `_type`, `_version`, and so on. The original document is enclosed under the `_source` attribute. That's it! It is as simple as that to fetch a document if we know the identifier of that document.

There may be a chance the document may not exist in the store. If the document is not found, we'll get a response with the attribute `found` set to `false`. For example, finding a document with ID 999 (which doesn't exist in our system) returns the following response:

```
{
  "_index": "movies",
  "_type": "_doc",
  "_id": "999",
  "found": false
}
```

We can of course find out if the document exists in the store beforehand by using the read API but with the HTTP HEAD action on the resource URL. For example, the following query shows in the listing 5.9 checks if the movie with ID 1 exists:

Listing 5.9: Check if the document exists in the engine

```
HEAD movies/_doc/1
```

This query returns 200 – OK if the document exists. If the document is unavailable in the store, a 404 Not Found error is returned to the client as figure 5.8 demonstrates.

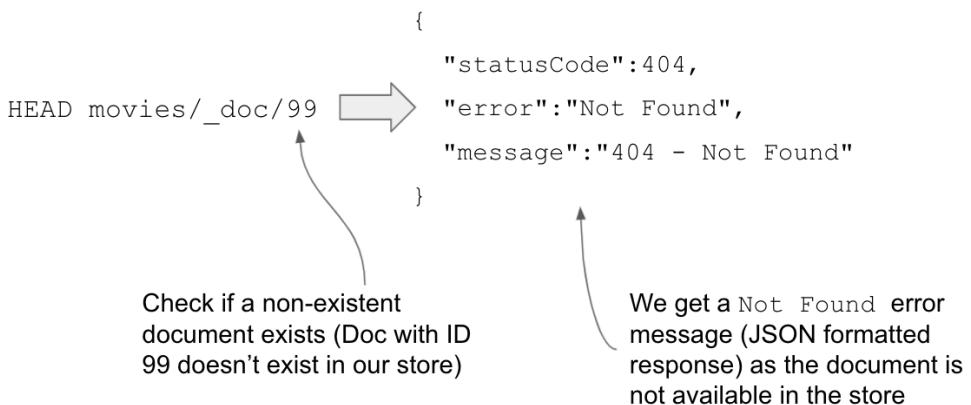


Figure 5.8 Fetching a non-existent document returns a Not Found message.

We can send a HEAD request to the server to find out if the document exists before actually requesting it. If you think this would incur an additional round trip to the server, indeed it does. Based on the HEAD request's response, we may (or may not) have to send another request for fetching the actual object. Instead, we can use a GET request in the first place, which returns the document if it exists or a `Not Found` message if not. Of course, the choice is yours.

So far we were able to fetch only a single document for a single index. How can we satisfy the requirement of fetching multiple documents with identifiers from either the same index or multiple indexes? For example, how can we fetch two documents with the IDs 1 and 2 from the `movies` index? Fortunately, we can use a multi-document API called `_mget`, which is the topic of the next section.

5.2.2 Retrieving multiple documents

In the last section, we used a single document API to fetch one document at a time. However, we might have a couple of requirements like:

- Retrieving a list of documents from an index, given the document identifiers
- Retrieving a list of documents from multiple indexes, given the document identifiers

Elasticsearch exposes a multi-document API (`_mget`) to satisfy these requirements. For example, to fetch a list of documents given identifiers, we can use `_mget` API as shown in the listing 5.10 here:

Login 5.10: Fetching multiple documents in one go

```
GET movies/_mget
{
  "ids" : ["1", "12", "19", "34"]
}
```

In the case of fetching multiple documents from various indices, the figure 5.9 shows the format of the call:

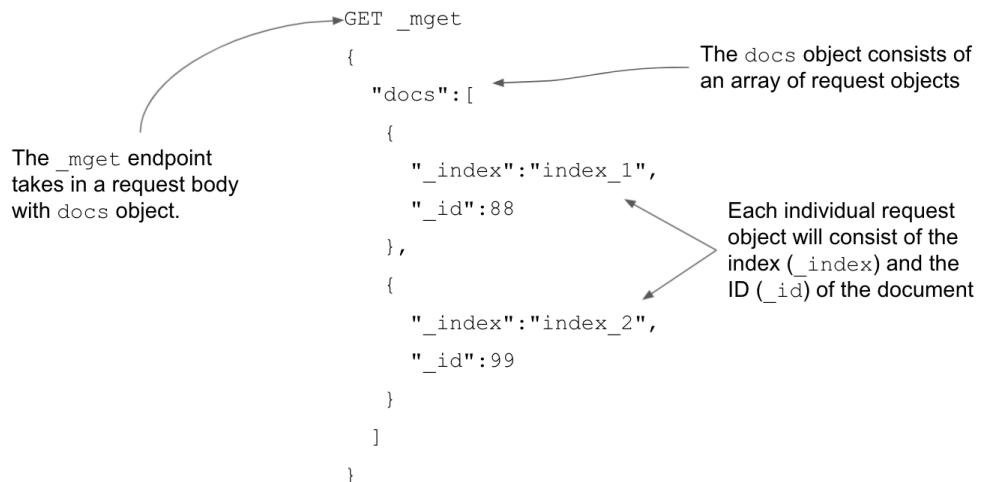


Figure 5.9 Fetching multiple documents using the `_mget` API.

As you can see in figure 5.9, the `_mget` endpoint is provided with a JSON formatted request object. The `docs` key inside the request expects an array of document's `_index` and `_id` pairs that we can use to fetch the documents from multiple indexes. The listing 5.11 fetches the documents from three different indexes.

Listing 5.11 Fetching documents from three different indexes

```
GET _mget #A A _mget call with no index mentioned in the url
{
  "docs": [
    {
      "_index": "classic_movies", #B The index is provided here
      "_id": 11
    },
    {
      "_index": "international_movies", #C index 2
      "_id": 22
    },
    {
      "_index": "top100_movies", #D index 3
      "_id": 33
    }
  ]
}
```

The request is formed with a requirement of fetching three documents from three different indexes: `classic_movies`, `international_movies`, and `top100_movies`. We can provide as many indexes as required, although there's a bit of a downside using this API: we have to create an individual `_index/_id` pair for each of the identifiers. Unfortunately, Elasticsearch does not yet allow the `_id` attribute to accept an array of identifiers. Hopefully, we can expect the Elastic folks to implement this feature in the near future.

The ids query

We looked at using the `_mget` API to fetch multiple documents in the previous section. However, there is also another route to fetch multiple documents: using an `ids` (short for identifiers) query. This simple search query takes in a set of document IDs to return the documents. This `ids` query is available as part of the search APIs. We will go through the details of this API in the search-related chapters, but here's the query in action (listing 5.12) if you are curious:

Listing 5.12: Use `ids` query to fetch multiple documents given their identifiers

```
GET classic_movies/_search
{
  "query": {
    "ids": {
      "values": [1,2,3,4]
    }
  }
}
```

You can fetch from multiple indexes too by adding the indexes to the URL. Here's an example:

```
GET classic_movies,international_movies/_search
{
  # body
}
```

That pretty much wraps up how to retrieve multiple documents from single or multiple indexes. Now, let's move our focus to the responses. Did you notice that our responses (see figure 5.8 in the previous section) get metadata along with the original source document? What if we want to fetch only the source without metadata in our response? Or maybe we want to hide some of the sensitive information in the source document when returning it to the client? Well, we can manipulate the responses as per our requirements, which is discussed in the next section.

5.3 Manipulating responses

The response returned to the client can contain a lot of information, and the client may not be interested in receiving all of it. And, sometimes, there might be some sensitive information that must not be exposed in the source sent back as the response. Also, sending a huge amount of data as a response (if the source has 500 attributes, for example) is a waste of bandwidth! There are ways of manipulating these responses before sending them to the client, but first let's fetch just the source of the document without metadata.

5.3.1 Removing metadata from the response

Usually the response object consists of metadata and the original document. The notable attribute in the response is the `_source` attribute, which encompasses the original input document. We can fetch just the source (original document) without the metadata by simply issuing the query like this:

```
GET <index_name>/_source/<id>
```

Notice that the `_doc` endpoint is replaced with `_source`, everything else in the invocation stays the same. Let's get the movie document by using this `_source` endpoint as the listing 5.13 demonstrates:

Listing 5.13: Fetching the original document with no metadata

```
GET movies/_source/1
```

As the response shown in figure 5.10 indicates, the document we had indexed returned with no additional information.

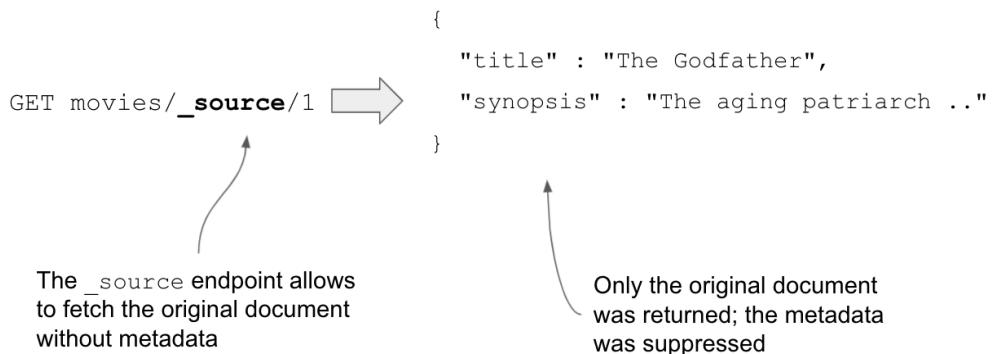


Figure 5.10 Invoking the `_source` endpoint returns the original document with no metadata.

As you can see in the response, there's no metadata fields like `_version`, `_id`, or `_index`; there's just the original source document. But what if we want to get the metadata but not the source document? Sure, we can do that too. Let's learn about how we can suppress the source data in the next section.

5.3.2 Suppressing the source document

There may be instances where the document is loaded with hundreds of fields, for example a full tweet (from Twitter API) consists of more than just a tweet - dozens of attributes such as the tweet, author, timestamps, conversations, attachments and so on. When we are retrieving data from the Elasticsearch, sometimes we wish not to look at the source data at all - that is we want to suppress the source data completely and just return the metadata

associated with the response. In that case, you can avoid returning the original document by setting the `_source` field to `false` as your request parameter in your query as the code in listing 5.14 shows:

Listing 5.14: Suppressing the original source data

```
GET movies/_doc/1?_source=false.
```

The response to this query is shown in the figure 5.11 here:

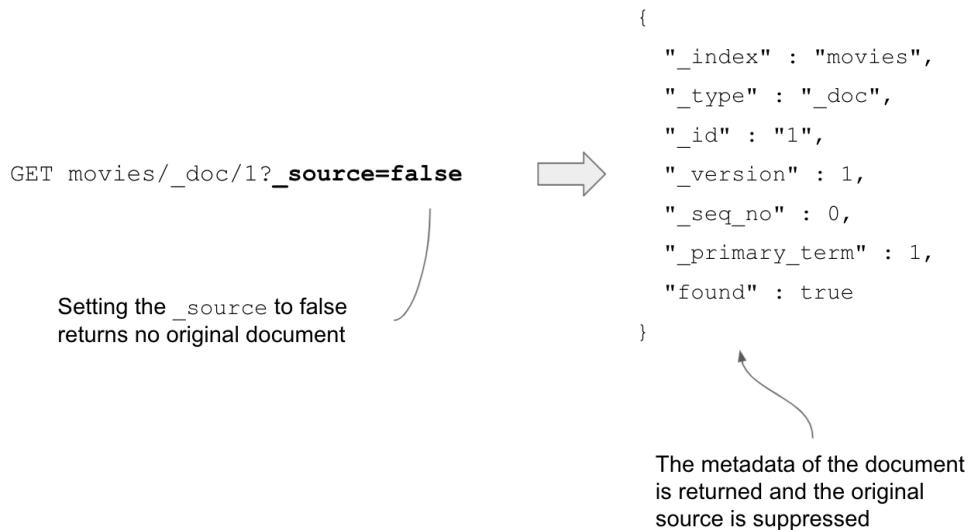


Figure 5.11 Setting the `_source` flag in the request returns metadata only.

The command in figure 5.11 sets the `_source` flag to `false`, indicating to the server not to return the original document. As you can see from the response, only the metadata was returned and not the entire source document. Not fetching the original document frees up the bandwidth too.

We now know how to avoid getting the whole document back, but how about returning a document with some selective (inclusive or exclusive) fields? For example, we might want the `title` and `rating` of the movies to be returned, while the `synopsis` is not. How can we customize what fields are included or excluded in the return list? Let's see exactly how to do that in the next section.

5.3.3 Including and excluding fields

In addition to suppressing the `_source` field as we did in the last section, we can also include and/or exclude fields when retrieving the documents. This is done by using the `_source_includes` and `_source_excludes` parameters, similar to the `_source` parameter we

used previously. We can provide a comma-separated list of fields that we want to return to the `_source_includes` attribute. Similarly, we can use `_source_excludes` to exclude the fields from the response; no surprise there.

We may need to enhance our movie documents for this example as the current document has no more than two fields. Let's add a few extra fields for a third movie (*The Shawshank Redemption*) as the following listing (5.15) shows.

Listing 5.15 Creating a new movie document with additional attributes

```
PUT movies/_doc/3 #A Indexing a new document
{
  "title": "The Shawshank Redemption",
  "synopsis": "Two imprisoned men bond ..",
  "rating": 9.3, #B rating attribute - new field
  "certificate": "15", #C certificate attribute - new field
  "genre": "drama",
  "actors": ["Morgan Freeman", "Tim Robbins"]
}
```

Once this document is indexed, we can experiment with the fields that should or should not return in the response.

Include fields (`_source_includes`)

To include a custom list of fields, append the `_source_includes` parameter with comma-separated fields. Say we want to fetch the `title`, `rating`, and `genre` fields from our movies index in our response and suppress the others. We would execute the following command mentioned in the listing 5.16:

Listing 5.16: Selectively including few fields to be part of the response

```
GET movies/_doc/3?_source_includes=title,rating,genre
```

This returns the document with these three fields. The following snippet demonstrates the response:

```
{
  ...
  "_source" : {
    "rating" : 9.3,
    "genre" : "drama",
    "title" : "The Shawshank Redemption"
  }
}
```

This document has both the original document information (under the `_source` object) and also the associated metadata. You can rerun the query by using the `_source` endpoint instead of `_doc` to lose the metadata and get a document with your custom fields as the following listing shows:

Listing 5.17: Returning selective fields with no metadata

```
GET movies/_source/3?_source_includes=title,rating,genre
```

In the same vein, we can exclude some of the fields too while returning the response. This time we would use the `_source_excludes` parameter.

Exclude fields (`_source_excludes`)

We can exclude fields that we don't want to be returned in the response using the `_source_excludes` parameter. The `_source_excludes` is a URL path parameter that accepts comma-separated fields. The response consists of all the fields of the document minus the fields mentioned in the `_source_excludes` parameter. The following query (listing 5.18) demonstrates this approach:

Listing 5.18: Excluding the fields in response

```
GET movies/_source/3?_source_excludes=actors,synopsis
```

The `actors` and `synopsis` fields are excluded in the response. What if we want to include some fields and explicitly exclude some fields too? Can an Elasticsearch query support this functionality? Sure we can ask Elasticsearch to satisfy these requirements too, discussed in the next section.

Include and exclude fields

We can mix and match what the return attributes we wish to have in our response, because Elasticsearch allows us to fine-tune the response. We need to enhance our movie document to demonstrate this functionality, so let's add various ratings (`amazon`, `metacritic` and `rotten_tomatoes`). The following listing (5.19) provides the updated document.

Listing 5.19 Updated movie model with ratings

```
PUT movies/_doc/3
{
  ...
  "rating":9.3,
  "rating_amazon":4.5,
  "rating_rotten_tomatoes":80,
  "rating_metacritic":90
}
```

Now, how can we return all the ratings except `amazon`? Here's where the power of setting up the `_source_includes` and `_source_excludes` with appropriate attributes shines as listing 5.20 shows:

Listing 5.20: Selectively ignoring certain fields

```
GET movies/_source/3?_source_includes=rating*&_source_excludes=rating_amazon
```

The query and response is visually represented in the figure 5.12 here:

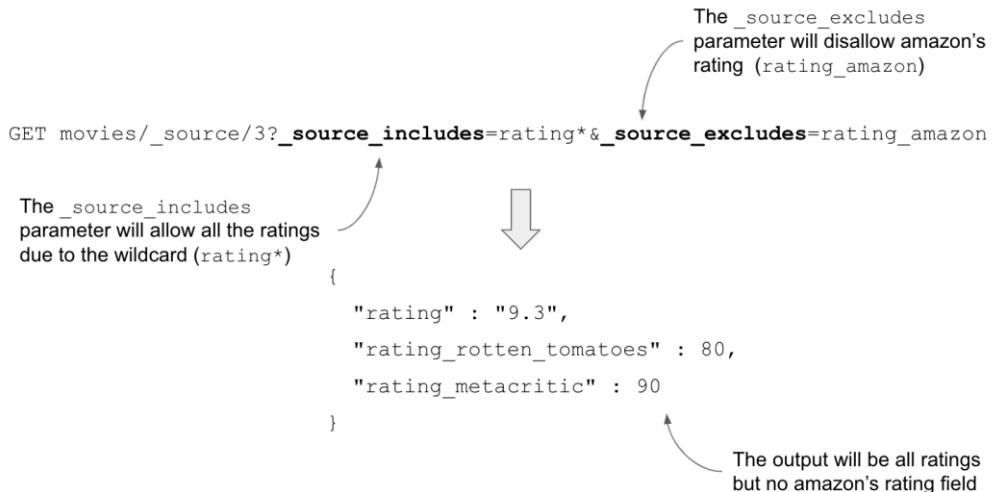


Figure 5.12 Tweaking what attributes can and cannot be as part of the return result.

We've enabled a wildcard field, `_source_includes=rating*`, in this query to fetch all the attributes that are prefixed with the word *rating* (for example, `rating`, `rating_amazon`, `rating_metacritic`, `rating_rotten_tomatoes`). The `_source_excludes`, on the other hand suppresses a field (for example, `_source_excludes=rating_amazon`). The resultant document should consist of all the ratings except the amazon rating.

So far we've learned about how to create and read documents including manipulating the responses. We will now understand the mechanics of updating the documents. There will always be a need to update an existing document, either modifying an existing field's value or adding a new field. Elasticsearch provides a set of update APIs for this purpose, which are discussed in the next section.

5.4 Updating documents

Documents, once indexed, need to be updated with modified values, additional fields, or the entire document may need to be replaced at times. Similar to indexing documents, Elasticsearch provides two types of update queries: one for working against single documents and other for working on multiple documents:

- `_update` API will update a single document
- `_update_by_query` will allows us to modify multiple documents in one go

Before we look at some examples, we need to understand the mechanics involved when updating the documents. Let's do that next.

5.4.1 Document update mechanics

Elasticsearch requires a few steps when we need to update our documents. Figure 5.13 illustrates the procedure.

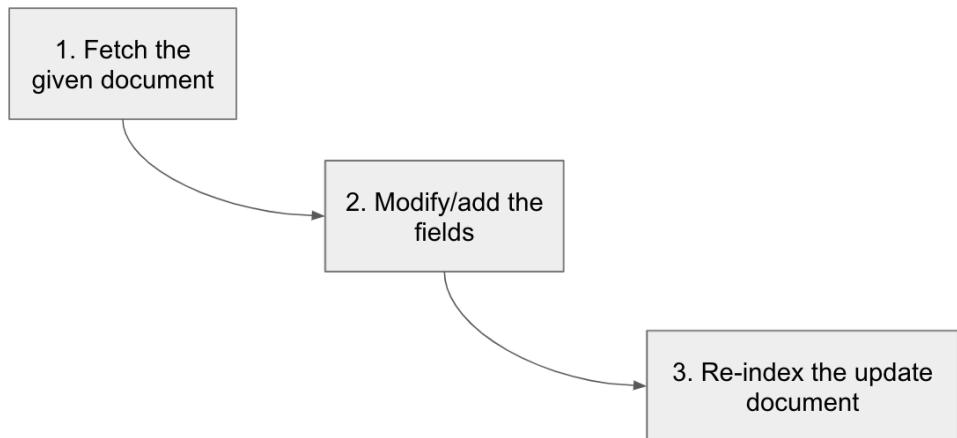


Figure 5.13 Updating or modifying the document is a three-step process.

As figure 5.13 shows, Elasticsearch first fetches the document, modifies it, and then re-indexes it. Essentially, it *replaces* the old document with a new document. Behind the scenes, Elasticsearch creates a new document for updates. During this update, Elasticsearch increments the version of the document once the update operation completes. When the newer version of the document (with the modified values or new fields) is ready, it marks the older version for deletion.

If you think we can make the same updates by ourselves by calling the GET, UPDATE, and POST methods on the document individually, you are absolutely right. In reality, this is what Elasticsearch does. As you can imagine, these are three different calls to the server, each resulting in a round trip to and from the client to the server. Elasticsearch avoids this roundtrip as Elasticsearch cleverly executes the set of these operations on the same shard, thus avoiding network traffic between client and server. By using the `_update` API, Elasticsearch avoids network calls, bandwidth, and coding errors.

5.4.2 The `_update` API

When we plan to update a document, we usually want to focus on one or more of the following scenarios:

- Adding more fields to an existing document
- Modifying existing field values
- Replacing the whole document

All of these operations are performed using the `_update` API. The format is simple, and it goes like this:

```
POST <index_name>/_update/<id>
```

When managing our resources with the `_update` API, we need to use the POST method as per RESTful API conventions. The ID of the document is provided in the URL along with the index name to form the full URL.

NOTE In the previous update URL, we used the `_update` endpoint, but we could have also used `POST <index_name>/_doc/<id>/_update`. Because the types are being deprecated and expected to be removed from version 8, avoid using the `_doc` endpoint for updating the document.

Now that we know the basics of the `_update` API, let's update our movie document (*The Godfather*) with a couple of additional attributes: `actors` and `director`. (Remember we had only two attributes on this original document: `title` and `synopsis` - you can check it out by fetching the document using `GET movies/_doc/1`).

ADDING NEW FIELDS

To amend the document with new fields, we pass a request body consisting of the new fields to the `_update` API. The new fields are wrapped in a `doc` object as the API expects in that manner. The code shown in the listing 5.21 demonstrates how to amend the `movies` document with two additional fields. It should be noted that we are implicitly assuming that `dynamic=true`.

Listing 5.21: Adding additional fields to the existing document using the `_update` API.

```
POST movies/_update/1
{
  "doc": {
    "actors": ["Marlon Brando", "Al Pacino", "James Cann"],
    "director": "Frances Ford Coppola"
  }
}
```

This query adds `actors` and `director` fields to our movie document. If we fetch the document (`GET movies/_doc/1`), the additional fields will be available in the return document.

MODIFYING THE EXISTING FIELDS

Sometimes we may need to change the existing fields too. There's no complexity in this, we just have to provide the new value to the field in a `doc` object as we did with the previous query. Let's just say we wish to rename our `title` field, we write the query like shown in the listing 5.22 here:

Listing 5.22: Updating the title of an existing document

```
POST movies/_update/1
{
  "doc": {
    "title": "The Godfather (Original)"
  }
}
```

When it comes to updating an element in an array (like adding a new actor to the list of `actors` field), we must provide both new and old values together. For example, let's say we want to add another actor (Robert Duvall) to the `actors` field for *The Godfather* document. This is demonstrated in the following code listing 5.23.

Listing 5.23 Updating an existing field with additional information

```
POST movies/_update/1 #A Updating the document ID 1
{
  "doc": { #B the updates must be enclosed in the doc object
    "actors": ["Marlon Brando",
      "Al Pacino",
      "James Cann",
      "Robert Duvall"] #C Old and new values together
  }
}
```

The query in listing 5.23 updates the `actors` field to add Robert Duvall to the list. Note that we are providing the existing actors in the array along with our new actor. Unfortunately, the list gets replaced by Elasticsearch with just Robert Duvall if we only mention Robert Duvall in the `actors` array.

We saw in these last couple of sections how to amend an existing document. There are instances where we must amend the documents based on some conditions. We do this using scripts, which is discussed in the next section.

5.4.3 Scripted updates

In the last couple of sections, we've used update API to modify the documents - to add additional fields or update existing ones. In addition to doing this on a field-by-field basis, we can also execute updates using scripts. Scripted updates allow us to update the document based on some conditions (refer to listing 5.xx for code sample), for example, ranking the movie as a blockbuster if it crosses a certain threshold of box office earnings.

Scripts are provided in a request body using the same `_update` endpoint, with the updates wrapped in a `script` object which in turn consist of the `source` as the key. We provide the updates as a value to this `source` key with the help of a context variable `ctx` - which is used to fetch the original document's attributes by calling `ctx._source.<field>`.

UPDATE ACTORS USING A SCRIPT

Let's update our movie document by adding an additional actor to the existing array. This time, we don't want to use the method demonstrated in listing 5.23, where we attached all

the existing actors to the `actors` field along with the new one. Instead, we simply use a script to update the `actors` field with an additional actor as the listing 5.24 shows.

Listing 5.24 Adding an additional actor to the actors list via a script

```
POST movies/_update/1
{
  "script": {
    "source": "ctx._source.actors.add('Diane Keaton')">#A Additional actor
  }
}
```

The `ctx._source.actors` fetches the `actors` array and invokes the `add` method on that array to insert the new value (`Diane Keaton`) in the list. In the same vein, we can delete a value from the list using the scripted update, although it is a bit involved.

REMOVING AN ACTOR

Removing an element from an array using script requires us to provide the index of the elements. The `remove` method takes an integer that points to the index of the actor we want to remove. To fetch the index of an actor, we can invoke the `indexOf` method on the appropriate array object. Let's check this in action, removing Diane Keaton from the list. The listing 5.25 demonstrates the script:

Listing 5.25 Removing the actor from the list of actors using scripted remove method

```
POST movies/_update/1
{"script": {"source":
  "ctx._source.actors.remove(ctx._source.actors.indexOf('Diane Keaton'))"}}
```

#A the remove expects the integer position of the actor

In the listing, the `ctx._source.actors.indexOf('Diane Keaton')` returns an index of the element in the `actors` array. This is *required* for the `remove` method.

ADDING A NEW FIELD

We can also add a new field to our document using a script as the listing 5.26 demonstrates. Here we add a new field, `imdb_user_rating`, with a value of `9.2`:

Listing 5.26 Adding a new field with a value using the script

```
POST movies/_update/1
{
  "script": {
    "source": "ctx._source.imdb_user_rating = 9.2"
  }
}
```

NOTE **Adding a new value to an array** if we wish to add a new value to an array (as we did it earlier mentioned in the listing 5.24), we invoke the `add` method on the array, as shown here: `ctx._source.<array_object>.add('value')`

REMOVING A FIELD

Removing a field is a straightforward job too. The following listing (5.27) demonstrates how we can remove a field (`metacritic_rating`) from our `movies` document:

Listing 5.27 Removing a field from the source document

```
POST movies/_update/1
{
  "script": {
    "source": "ctx._source.remove('metacritic_rating')"
  }
}
```

Note that if you try to remove a nonexistent field, you will not get an error message letting you know you are trying to remove a field that doesn't exist on the schema. In fact, you'll get a response indicating that the document was updated as well as the field being incremented (kind of false positive, I think).

ADDING MULTIPLE FIELDS

We can write a script that adds multiple fields in one go. The listing 5.28 shows how to do this.

Listing 5.28 Updating the document with multiple new fields using a script

```
POST movies/_update/1
{
  "script": {
    "source": """
      ctx._source.runtime_in_minutes = 175;
      ctx._source.metacritic_rating= 100;
      ctx._source.tomatometer = 97;
      ctx._source.boxoffice_gross_in_millions = 134.8;
    """
  }
}
```

The notable thing in this listing is that the multi-line updates are carried out in a triple-quote block. Each key-value pair is segregated with a semicolon (;).

ADDING A CONDITIONAL UPDATE SCRIPT

We can implement a bit more complicated logic in the script block too. Let's say we want to tag the movie as a blockbuster if the gross earnings are over \$125 million. (I've just made this rule up; in reality, there are other factors involved such as budget, big stars, return of investment, and so on to make it a blockbuster.) To do this, let's create a script with a condition that checks the movie's gross earnings, and if the earnings cross the threshold, we label that movie as a blockbuster. In the listing 5.29, we write a simple if/else statement, whose logic is based on the earnings and sets a `blockbuster` flag accordingly.

Listing 5.29: Conditionally updating the document using an if/else block

```
POST movies/_update/1
{
  "script": {
    "source": """
      if(ctx._source.boxoffice_gross_in_millions > 125)
        {ctx._source.blockbuster = true}
      else
        {ctx._source.blockbuster = false}
      """
    }
}
```

The `if` clause in the listing checks for the value of the field `boxoffice_gross_in_millions`. It then creates a new `blockbuster` field automatically (we don't have that field yet on our schema) and sets the flag to `true` or `false`, based on the outcome of the condition. I agree it's a bit of a mouthful to chew, but it's not difficult to understand what's going on here. Retrieving the document will let us know if we have attached the `blockbuster` tag to the document:

```
_source" : {
  "title" : "The Godfather",
  ...
  "blockbuster" : true
}
```

Indeed, the movie was a blockbuster as the query result reveals this condition. So far we've been working with straightforward examples using scripts. However, scripts allow us to do more - from a simple update to a complex conditional modification on a data set. Understanding the intricacies of scripting is out of scope for this book, but learning a few concepts is advised, so let's briefly discuss the anatomy of the script in the next section.

ANATOMY OF A SCRIPT

Let's take a short pause to understand a brief anatomy of the script. The script has three parts, source, language, and parameters, as figure 5.14 demonstrates.

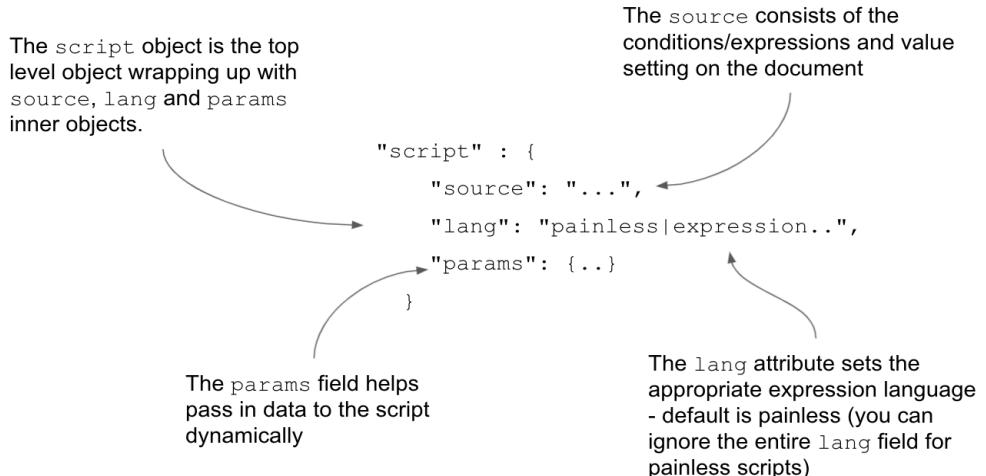


Figure 5.14 The anatomy of a script

The `source` is where we provide the logic, while the `params` are the parameters that the script expects separated by a vertical bar (or pipe) character. We can provide the language our script is written in (for example, one of `painless`, `expression`, `moustache`, or `java`, where the default is `painless`). Let's next check on updating a document by passing the values via the `params` attribute.

PASSING DATA TO THE SCRIPTS

One issue with the code in listing 5.9 is that we hardcoded the threshold earnings in the script (gross earnings as \$125 million). Instead, we can set the threshold value in the script by using `params` property. Let's revisit our blockbuster script, but this time we will pass the value of the gross earnings threshold to the script's logic via the `params` attribute. The listing 5.30 demonstrates this.

Listing 5.30 Dynamically passing a parameter to the script

```
POST movies/_update/1
{
  "script": { #A Business logic goes here
    "source": "" #B checking the value against params
    if(ctx._source.boxoffice_gross_in_millions > params.gross_earnings_threshold)
      {ctx._source.blockbuster = true}
    else
      {ctx._source.blockbuster = false}
    """",
    "params": {#C provide the parameter values
      "gross_earnings_threshold":150
    }
  }
}
```

The script has two notable changes from the previous version mentioned in the code listing 5.29:

- The `if` clause is now compared against a value read from the `params` object (`params.gross_earnings_threshold`), and
- The `gross_earnings_threshold` is set to 150 via the `params` block

When the script gets executed, Elasticsearch consults the `params` object and replaces the attribute with the value from the `params` object. If for whatever reason we want to change the value of gross earnings for setting a blockbuster flag (perhaps the `params.gross_earnings_threshold` needs to be updated to half a billion dollars), we can simply pass the new value in the `params` flag.

Did you notice the hardcoded `params` value in the script? You may be wondering why we are hardcoding the `gross_earnings_threshold`'s value in the script in the `params` object?

Well, there is a lot more to the scripting functionality than what we saw here. Scripts are compiled when they are executed for the first time. The compilation of the scripts carries a performance cost, hence it is considered to be an expensive operation in Elasticsearch. However, scripts that are associated with dynamically changing parameters (using the `params` object) can avoid this compilation cost. This is because the script gets compiled the first time only and then gets updated with the variable's (`param`) value when invoked for the rest of the time. This is a significant benefit, so the general practice is to provide the dynamic variables via the `params` object to the script (refer to listing 5.30).

Scripting languages

The scripts we developed in this chapter were derived from Elasticsearch's special scripting language, called `Painless`, for decoding the logic and executing the scripts. The default language is `Painless` (we didn't specify the language explicitly in our code earlier). There are other scripting languages (`Mustache`, `expression`, or even `Java`) that we can plug into by using a `lang` parameter. Irrespective of the language we use, there's a set pattern that we must follow as this snippet shows:

```
"script": {
  "lang": "painless|mustache|expression|java",
  "source": "...",
  "params": { ... }
}
```

So far, we've updated individual documents, be it using the `_update` API call or scripts. How about updating a bunch of documents that match a criteria? That's exactly what we'll learn in the next section.

5.4.4 Replacing documents

Let's say we need to replace an existing document with a new one. This is super easy; just use the same PUT request we performed earlier in the chapter when indexing a new

document. But what do we do if we need to replace a document? Let's insert a new movie title (*Avatar*) in our `movies` document but associate it with an existing document (ID = 1) as demonstrated in the code listing 5.31:

Listing 5.31: Replacing the content of a document

```
PUT movies/_doc/1
{
  "title": "Avatar"
}
```

The existing movie, *The Godfather*, is replaced with the new data attributes (*Avatar*) after executing this command.

NOTE If your intent is to replace the existing content with something else, that's simply use the `_doc` API on the same ID with a new request body. However, If we do not wish to get the document replaced, we must lean on using the `_create` endpoint (discussed in section 5.1.1)

Sometimes when we try to update an nonexisting document, we might want Elasticsearch to index it as a new document rather than throwing an error. This is what the *upsert* operation is all about, which we'll discuss in the next section.

5.4.5 Upsert

Upserst, short for update and insert, is an operation that either updates a document (if it exists) or indexes a new document with the data provided (if it does not exist). Figure 5.15 demonstrates this operation.

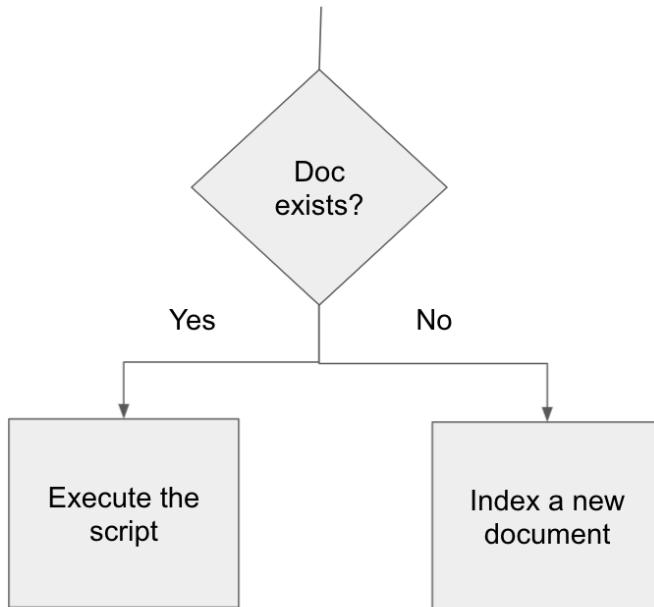


Figure 5.15 Upsert operation's workflow

Let's say we want to update `gross_earnings` for the movie *Top Gun*. Remember, we don't have this movie in our store yet. We'll develop a query to update the `gross_earnings` field and create a new document with this update.

Listing 5.32 illustrates upsert in action. The query has two parts: a `script` and an `upsert` block. The `script` part is where we update the field on an existing document, while the `upsert` block consists of the new movie document information.

Listing 5.32 Upsert block example

```
POST movies/_update/5
{
  "script": {
    "source": "ctx._source.gross_earnings = '357.1m'"
  },
  "upsert": {
    "title": "Top Gun",
    "gross_earnings": "357.5m"
  }
}
```

When we execute this query, the `script` is expected to run and update the `gross_earnings` field on a document with ID 5 (if the document is available). What happens

if that document is not there in our index? Well, that's where the `upsert` block comes into play.

The second part of the JSON request is interesting: the `upsert` block consists of fields that constitute a new document. Because there is no document with that ID in the store, the `upsert` block creates a new document with the specified fields. So, an upsert action provides us with the facility of either updating an existing document or creating a new one if it's the first time we index it.

If you rerun the same query again (for the second time), this time the script gets executed with a field `gross_earnings` changed to \$357.1 m (from \$357.5m in the original document) because the document already exists.. The script part came into action as the document already exists.

UPDATES AS UPSERT

Earlier, in section 5.5.2, we looked at partially updating a document using the `doc` object via the `_update` API. In the following listing 5.xx, we update a movie with an additional field, `runtime_in_minutes`. If document 11 exists, this field gets updated as expected, otherwise an error is thrown (listing 5.33), mentioning that the document does not exist.

Listing 5.33: Updating a non-existent field will throw an error

```
POST movies/_update/11
{
  "doc": {
    "runtime_in_minutes":110
  }
}
```

If we want to avoid errors - maybe we want to create a new document if the said document does not exist - we can use a `doc_as_upsert` flag. Setting this flag to `true` allows the contents of the `doc` object to be stored as a new document if the document (ID = 11) doesn't exist. To do this, simply add the `doc_as_upsert` flag to the script in the listing 5.34 and let it run.

Listing 5.34: Updating a new field in the document if the field doesn't exist

```
POST movies/_update/11
{
  "doc": {
    "runtime_in_minutes":110
  },
  "doc_as_upsert":true
}
```

This time, if we don't have the document with ID 11 in our store, that's alright. The engine does not throw an error, instead it creates a new document with ID 11 and fields extracted from the `doc` object (`runtime_in_minutes`, in this case).

5.4.6 Updating by a query

Sometimes we wish to update a set of documents matching to a search criteria - for example update all movies whose rating is above 4 stars as popular movies. We can run a query to search for such a set of documents and apply the updates on those by using the `_update_by_query` endpoint.

Say, we need to update all the movies with the actor's name matching *Al Pacino* to *Oscar Winner Al Pacino* in all the movies that he acted in. The listing 5.35 does exactly that.

Listing 5.35: Updating the document using a query method

```
POST movies/_update_by_query #A We search for the docs and update the set
{
  "query": { #B Search query - search all movies of Pacino
    "match": {
      "actors": "Al Pacino"
    }
  },
  "script": { #C Apply the following script logic for the matching documents
    "source": """
      ctx._source.actors.add('Oscar Winner Al Pacino');
      ctx._source.actors.remove(ctx._source.actors.indexOf('Al Pacino'))
    """,
    "lang": "painless"
  }
}
```

In the listing, the `match` query executes first (we will learn about match queries in the later chapter on search, but just think of it as a type of the query where we can fetch the documents matching a given criteria) to fetch all the movies where the actor is Al Pacino. When the `match` query returns its results, a script executes to add the name change - changes *Al Pacino* to *Oscar Winner Al Pacino*. Because we need to remove the old name, we invoke a `remove` operation too in the script.

The `_update_by_query` is a handy mechanism to update tons of documents based on a criteria. However, there is a lot going on behind the scenes when we use this method for updating documents. We'll learn the mechanics of this in section 5.7 in detail.

We now know the mechanism to update documents. It is time to learn about delete operations; specifically, how to delete a single document or multiple documents in one go. The next section is dedicated to deleting documents in multiple ways.

5.5 Deleting documents

When you want to delete documents, there are essentially two methods: using an ID or using a query. In the former case, we can delete a single document, while in the latter, we can delete multiple documents in one go. When deleting by query as in the second method, we can set filter criteria (for example, delete documents whose status field is unpublished or documents from last month, etc.). Let's see both of these methods in action.

NOTE There is a lot going on behind the scenes during delete operations. Refer to Chapter X: Dealing with Data for details on the versioning mechanics and optimistic concurrency control during delete operations.

5.5.1 Deleting with an ID

We can delete a single document from Elasticsearch by invoking the HTTP DELETE method on the indexing document API as shown here:

```
DELETE <index_name>/_doc/<id>
```

The URL is the same as the one we previously used for both indexing and retrieving the document. Given the ID of the document to be deleted, we construct the URL by specifying the index, the `_doc` endpoint, and the ID of the document. For example, we can invoke the query in the listing 5.36 to delete the document with ID 1 from the movies index.

Listing 5.36 Deleting a movie document from the index

```
DELETE movies/_doc/1
```

The response received from the server indicates that the document was deleted successfully as this code snippet shows:

```
{
  ...
  "_id" : "1",
  "_version" : 2,
  "result" : "deleted"
}
```

The response will return a `result` attribute set to `deleted` to let the client know the object was deleted successfully. If the document isn't deleted (for instance, it doesn't exist in our store) we'll get a response with the `result` value set to `not_found`. Interestingly, Elasticsearch increments the `_version` flag should the delete operation succeed.

5.5.2 Deleting by query (`_delete_by_query`)

Deleting a single document is easy and straightforward as we saw in the last section. But if you want to delete multiple documents based on a criteria, there's `_delete_by_query` (similar to what we say `_update_by_query` in the last section) that we can lean on. If we want to delete all the movies that are directed by James Cameron, for example, we would write the query in the listing 5.37.

Listing 5.37 Deleting all movies based on a criteria

```
POST movies/_delete_by_query
{
  "query": {
    "match": {
      "director": "James Cameron"
    }
  }
}
```

Here we create the query in the request body using the criteria for all movies directed by James Cameron. Documents that are matched with the criteria are then marked and deleted.

The body of this POST uses a special syntax called Query DSL (domain-specific language), which we can pass in a variety of attributes like `term`, `range`, and `match` (as in this listing), similar to the basic search queries. We will learn more about search queries in the later chapters, but for now, note that `_delete_by_query` is a powerful endpoint with sophisticated delete criteria. Let's see a few examples in the following sections.

5.5.3 Delete by range query

We may wish to delete some records which may fall in a certain range: movies whose reviews were between 3.5 and 4.5 or cancelled flights between two dates and so on. We use a `range` query to set a criteria for a range of values for such requirements. The listing 5.38 demonstrates `_delete_by_query` to delete movies whose earnings between \$350 m to \$400 m.

Listing 5.38 Deleting all movies with gross earnings above \$350 but below \$400 million

```
POST movies/_delete_by_query
{
  "query": {
    "range": {
      "gross_earnings_in_millions": {
        "gt": 350,
        "lt": 400
      }
    }
  }
}
```

As listing 5.38 demonstrates, the `_delete_by_query` accepts a `range` query with a match criteria: find the documents whose gross earnings fall between 350 to 400 million dollars. As you can expect, all matching documents will be deleted.

Of course, you can construct a complex query if you want. For example, listing 5.39 shows a query that constructs a criteria to delete movies directed by Steven Spielberg, where those movies are rated between 9 and 9.5 and earn less than \$100 million. The listing uses a `bool` query as the request.

Listing 5.39 Deleting movies with a complicated query criteria

```
POST movies/_delete_by_query
{
  "query": {
    "bool": {
      "must": [{ #A match movies directed by Spielberg
        "match": {
          "director": "Steven Spielberg"
        }
      }],
      "must_not": [{ #B The ratings shouldn't be less than 9
        "range": {
          "imdb_rating": {
            "gte": 9,
            "lte": 9.5
          }
        }
      }],
      "filter": [{ #B Earning shouldn't fall below 100mil
        "range": {
          "gross_earnings_in_millions": {
            "lt": 100
          }
        }
      }]
    }
  }
}
```

The query uses a complex query logic that's called a `bool` query because it combines all the little queries together to make it work on a grand scale. We have a dedicated chapter on `bool` queries towards the later part of the book, where we will dig deeper into how we can construct complicated and complex queries.

5.5.4 Deleting all documents

WARNING Delete operations are irreversible! Be cautious before hitting your Elasticsearch with a delete query.

You can delete a whole set of documents from an index using the `match_all` query. The listing 5.40 demonstrates this operation.

Listing 5.40 Deleting all documents in one go

```
POST movies/_delete_by_query
{
  "query": {
    "match": {
      "match_all": {}
    }
  }
}
```

The query in listing 5.40 runs a `match` query. In this case, it matches all documents and deletes them all in one go. As this is a destructive operation, do carry out deleting the entire set of documents with due care. You can of course delete the whole index by simply issuing `DELETE movies` command - remember these are irreversible commands.

We have deleted documents so far on a single index. We can also delete documents across multiple indexes by simply providing a comma-separated list of indexes in the API URL. The example format is shown here:

```
POST <index_1>,<index_2>,<index_3>/_delete_by_query
```

The listing 5.41 shows how to delete all documents across multiple movie-related indexes:

Listing 5.41 Deleting documents from three different movie indexes

```
POST old_movies,classics,movie_reviews/_delete_by_query
{
  "query": {
    "match": {
      "match_all": {}
    }
  }
}
```

Of course, be mindful of issuing delete queries because you may lose the whole dataset! Unless your wish is to purge the complete data set, exercise delete operations with caution in production.

We have been indexing the documents individually so far, but in the real world there's always a case to index large sets of documents in one go. We may have a hundred thousand movies to read from a CSV file or a half a million currency fx rates fetched from a third party service to be indexed into our engine. While usually you use an ETL (extract-transform-load) tool like Logstash to extract, enrich and publish data, Elasticsearch provides a bulk (`_bulk`) API to index messages in bulk. We discuss bulk API in the next section.

5.6 Working with documents in bulk

So far, we've worked with indexing documents individually using Kibana. Indexing a single or handful of documents using the API methods is straightforward. While this is well and good for development purposes, we would do this rarely in production though. It is cumbersome

and error-prone if there is a larger set of data (for example, when extracting a larger number of records from a database).

Fortunately, Elasticsearch provides a bulk API to index quantities of data. For this, we can use the `_bulk` API to index large data sets simultaneously, and it is one of the commonly used mechanisms for that. We can use the bulk API not only to index but also to manipulate documents including deleting them.

The `_bulk` API accepts a POST request, which can perform index, create, delete, and update actions all at once. This saves bandwidth as multiple trips to the server can be avoided. There is a special format for the `_bulk` API, which we may find a bit weird, but not difficult to get our head around. Let's understand the format first.

5.6.1 Format of the `_bulk` API

The `_bulk` API consists of a specific syntax with a POST method invoking the API call. Figure 5.16 displays the syntax for this.

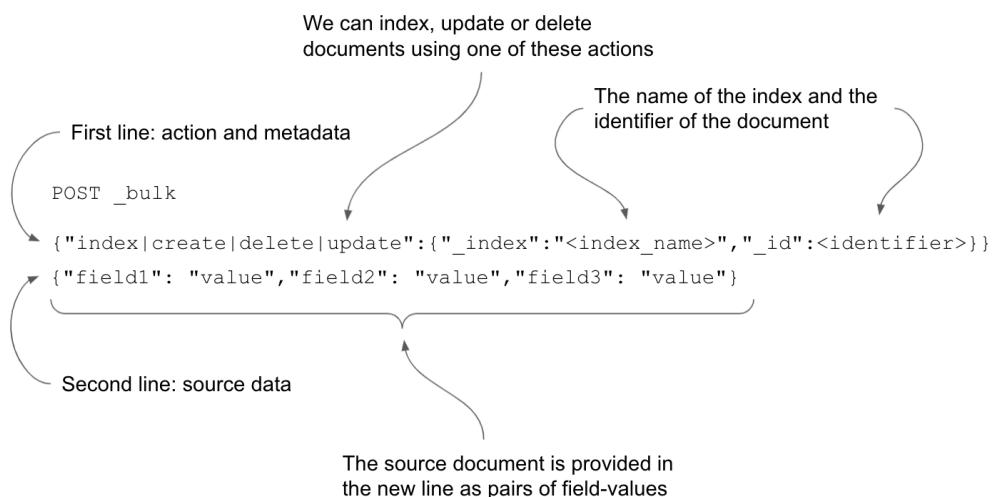


Figure 5.16 `_bulk` API's generic format

The request body consists of two lines for every document that needs to be stored. The first (refer to figure 5.19) indicates one of the actions to be performed on the document: `index`, `create`, `delete`, or `update`. The document is described in the second line, which we will cover shortly.

Once we choose the action, we need to provide a value to this action key with some metadata. The metadata is usually the name of the document's index and the document's ID. For example, the metadata for a document with ID 100 that embeds `index` into the movies index is `"_index": "movies", "_id": "100"`.

The second line that we refer to in figure 5.19 is the document's source itself, which is where we want to store the document. As expected, the document is formatted in JSON and added in the new line to the request. Both metadata and source lines are delimited with new line (\n) separators expressed in JSON (hence, NDJSON, which is short for newline-delimited JSON). NDJSON (<http://ndjson.org>) is a convenient format for storing records to be consumed one by one. If we execute the query in listing 5.19, e'll have a document with ID 100 that is indexed into the `movies` document with the fields that were given in the second line.

NOTE The request that gets attached to the bulk API must adhere to strict NDJSON format; otherwise, the documents won't get bulk indexed. Each of the lines must end with a newline delimiter as the bulk request is newline-sensitive. Make sure your document is formatted as NDJSON.

With this concept in mind, let's create a bulk request to index the movie document *Mission Impossible*.

5.6.2 Bulk indexing documents

We want to index documents using bulk API - so let's check out the example shown in listing 5.42

Listing 5.42: Bulk indexing in action

```
POST _bulk #A the bulk API URL
{"index":{"_index":"movies","_id":"100"}} #B We want to "index" this document
{"title": "Mission Impossible","release_date": "1996-07-05"} #C Document
```

The same request is demonstrated visually in figure 5.17 with annotations:

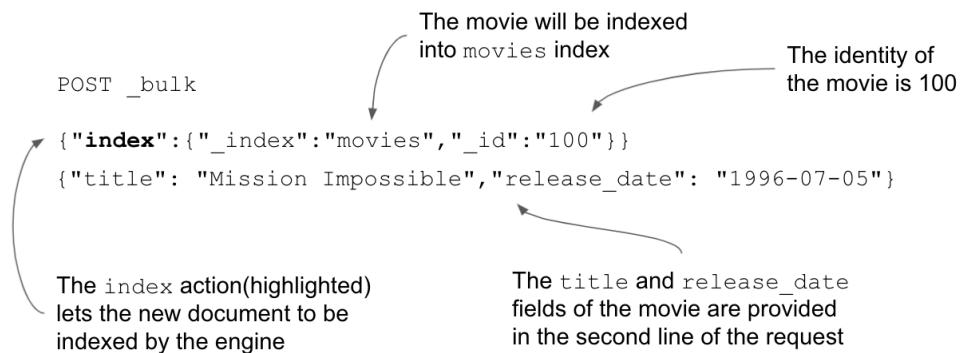


Figure 5.17 Indexing the new movie *Mission Impossible* using the `_bulk` API

If you execute the query in figure 5.17, the *Mission Impossible* movie is created in our store. The two lines make for one request - the idea is that you'll have to code two lines for each of the documents that's expected to be actioned.

We can even shorten the metadata line - for example, we can lose the index and attach it to the URL itself, as shown in the listing 5.43 below:

Listing 5.43: Bulk API with index embedded in the request URL

```
POST movies/_bulk #A The URL has the index
{"index": {"_id": "100"} } #B The _index field was removed
{"title": "Mission Impossible", "release_date": "1996-07-05"}
```

You can get rid of the `_id` field too if you want system-generated random IDs for your movies. The following snippet (listing 5.44) demonstrates this approach:

Listing 5.44: Letting the system generate document identifiers

```
POST movies/_bulk
{"index": {} } #A Both the _index and _id were removed
{"title": "Mission Impossible", "release_date": "1996-07-05"}
```

The ID for this document will be assigned by the system, something of a random UUID.

Of course, you might be wondering why we need to follow this bulk approach when we can index the same document using the document index API (`PUT movies/_doc/100`). That is a fair question, but let me just say that we've not yet unleashed the full power of the bulk API. Suppose we have a requirement to index Tom Cruise's movies into our movies. The request is written in the listing 5.45 here:

Listing 5.45 Bulk indexing Tom Cruise's movies

```
POST movies/_bulk
{"index": {} }
{"title": "Mission Impossible", "release_date": "1996-07-05" }
{"index": {} }
{"title": "Mission Impossible II", "release_date": "2000-05-24" }
{"index": {} }
{"title": "Mission Impossible III", "release_date": "2006-05-03" }
{"index": {} }
{"title": "Mission Impossible - Ghost Protocol", "release_date": "2011-12-26" }
```

As we've attached the index name in the URL (`POST movies/_bulk`) and because we are not concerned with a predefined ID, the query indexes four of Tom Cruise's movies successfully.

5.6.3 Independent entities and multiple actions

We only indexed the *Mission Impossible* movie, but we can get other entities indexed in the same request. The notable thing is that bulk API helps to wrap up multiple entities - that is not just movies, we could potentially have any types - books, flights, logs etc, all bundled up in the request. The listing in 5.46 has a list of multiple, mutually exclusive requests:

Listing 5.46: Bulk request with mixed bag of requests

```
POST _bulk
{"index":{"_index":"books"}} #A Indexing the book
{"title": "Elasticsearch in Action"}
{"create":{"_index":"flights", "_id":"101"}} #B Creating a flight
{"title": "London to Bucharest"}
{"index":{"_index":"pets"}} #C Indexing a pet into a pets index
{"name": "Milly", "age_months": 18}
{"delete":{"_index":"movies", "_id":"101"}} #D Deleting a movie
{ "update" : {"_index":"movies", "_id":"1"} }#E Updating the title of a movie
{ "doc" : {"title" : "The Godfather (Original)"} }
```

The above bulk request has almost all the actions that one could imagine when using the bulk API. Let's drill down into these individual actions.

CREATE ACTION

As you can see in the second action, we swapped the `index` action with `create` so we do not replace the document if it doesn't exist while indexing (see section 5.1.3). The following code (listing 5.47) shows the `create` operation in action individually:

Listing 5.47: Bulk API with create operation

```
POST _bulk
{"create":{"_index":"movies", "_id":"101"}} #A avoids accidental overrides
{"title": "Mission Impossible II", "release_date": "2000-05-24"}
```

UPDATE ACTION

Updating the document follows a similar pattern too, but we must wrap the fields to be updated in a `doc` object as we learned earlier in section 5.5.2. Listing 5.48 updates a movie with ID 200 (*Rush Hour*) with additional fields (`director` and `actors`). Note that the *Rush Hour* movie wasn't in our index, so I've created the movie beforehand (look at GitHub for the source of this indexing query).

Listing 5.48 Bulk updating the *Rush Hour* movie

```
POST _bulk
{"update":{"_index":"movies", "_id":"200"}}
{"doc": {"director": "Brett Ratner", "actors": ["Jackie Chan", "Chris Tucker"]}}
```

DELETE ACTION

Finally, let's use the bulk API to delete a document. The format is slightly different though, as shown in the listing 5.49 below:

Listing 5.49: Bulk call with delete action

```
POST _bulk
{"delete":{"_index":"movie_reviews", "_id":"111"}}
```

As you can see, we don't need a second line for this operation. The query deletes the movie review with ID 111 from the `movie_reviews` index. Remember, you may lose the whole dataset if care is not exercised when issuing delete queries.

5.6.4 Bulk requests using cURL

In the earlier section, we've used Kibana to perform operations on some documents using the `_bulk` API. We can also perform the same actions using cURL. In fact, using cURL may be the preferred method if you have a larger chunk of records to deal with.

To use cURL, we need to create a JSON file with all the data in it and pass that file to cURL with the flag `data-binary`. For your convenience, I copied the same data used in listing 5.50 to a `movie_bulk_data.json` file available in GitHub. You can then pass the file to cURL as the following listing shows.

Listing 5.50 Using cURL to execute a bulk data operation

```
curl -H "Content-Type: application/x-ndjson"
-XPOST localhost:9200/_bulk
--data-binary "@movie_bulk_data"
```

NOTE Make sure to give the `--data-binary` flag the name of your file (without the extension) with a `@` prefix.

We now know how to work with bulk requests, however sometimes we'll want to move documents from one index to another (for example, movies from `blockbuster_movies` to `classic_movies`). Bulk APIs may not be suitable for moving (or migrating) data between indices. Elasticsearch provides us with a popular feature: reindexing API. This is discussed in detail in the next section.

5.7 Reindexing documents

Depending on our application and business needs, we may need to move our documents from one index to another from time to time. This is especially true when we need to migrate an older index to a newer one due to changes in our mapping schema or settings. We can use the `_reindex` API for such requirements as format shows here:

```
POST _reindex
{
  "source": {"index": "<source_index>"},
  "dest": {"index": "<destination_index>"}
}
```

When would we like to put reindexing in action, you may ask. Suppose we want to update our `movies` index with schema modifications that may break the existing index if we implement them directly on the existing index. In this case, the idea is to create a new index with the updated settings (say, a `movies_new` index with the updated schema) and move the data from the old `movies` index to the new one. The query in the following listing 5.51 does exactly that.

Listing 5.51 Migrating data between indexes using the reindexing API

```
POST _reindex
{
  "source": {"index": "movies"},
  "dest": {"index": "movies_new"}
}
```

The query in listing 5.51 grabs a snapshot of the movies index and pushes the records into the newer index. The data gets migrated between these indexes as expected. One of the important use cases of reindexing is zero downtime migration in production if used with aliases. We will go over this in the next chapter to put this concept in action.

This is a long chapter, we've learned quite a lot. Let's wrap it up here and jump right into the next chapter where we discuss indexing operations in detail.

5.8 Summary

- Elasticsearch provides a set of APIs that work on documents. We can use the APIs to execute CRUD actions (create, read, update, and delete) on individual documents.
- Our documents are held in the shard's in-memory buffer and pushed into a segment during the refresh process. Lucene employs a strategy of creating a new segment with the documents during the refresh. It then cumulatively merges three segments together to form a new segment and the process repeats.
- Documents with identifiers (IDs) use the HTTP PUT action when indexing (for example, `PUT <index>/_doc/<ID>`), whereas documents with no IDs invoke the POST method.
- Elasticsearch generates random unique identifiers (UUIDs) and assigns these to documents during the indexing process.
- To avoid overriding a document, we can issue the following commands:
 - `_create`—This API throws an error if the document already exists.
 - `_mget`—This API lets us retrieve multiple documents in one go given their identifiers.
 - `_bulk`—This API performs document operations such as indexing, deleting, and updating multiple documents in one invocation call.
- We can tweak the source and the metadata retrieved as a result of query invocations and then customize the returned document source to include and/or exclude fields by setting the `_source_includes` and `_source_excludes` properties, respectively.
- The `_update` API lets us modify an existing document by updating fields as well as adding additional fields. The expected updates are wrapped in a `doc` object and passed as the request body.
- Multiple documents can be modified by constructing a `_update_by_query` query.
- Scripted updates allow modifying the documents based on conditions. If the conditional clause mentioned in the request body is evaluated to true, the script is put into action.
- Documents can be deleted by a single document invocation using HTTP DELETE or by running a `_delete_by_query` method on multiple documents.
- Migrating data between the indexes is performed by the reindexing API. The `_reindex` API call expects the source and destination indexes to transfer the data.

6

Indexing operations

This chapter covers

- Basic indexing operations
- Index templating
- Status management and monitoring
- Index life-cycle management (ILM)

In the last few chapters, we worked with indices without delving much into the intricate details about them. While that is adequate for getting started with Elasticsearch, it is far from ideal. Configuring an index with appropriate settings not only lets the search engine run efficiently, but it also enables befitting infrastructure, thus increasing resiliency and reducing costs. Having a sound organizational indexing strategy will create a future-proof search engine and, hence, a smoother user experience.

For a healthy and performant Elasticsearch cluster, working with indices at a lower level is necessary. Understanding the inner workings of index management in depth helps when setting up a resilient and coherent search system. This chapter is all about digging deep into indexing management, monitoring, and life cycle.

Indices come with three sets of configurations: settings, mappings, and aliases. Each of these configurations modify the index in one form or another. For example, we use settings for creating the number of shards and replicas as well as other properties for the index. The shards and replicas allow scaling and high availability of the data. The mappings define an effective schema for our data for indexing and querying data efficiently. Aliases, the alternate names given to indices, allow querying across multiple indices easily as well as reindexing data with zero downtime. Learning the indexing operations and tweaking the configurations along the way will help you get a grip on index management.

Though instantiating indices manually is acceptable, it is indeed a tedious, ineffective, and sometimes erroneous process. Instead, organizations should strive for a strategy which

leads to developing indices using index templates. Index templates allow creating indices with a predefined configuration, and understanding the template mechanism lets you develop indices for advanced operations such as rollover.

The indices are expected to grow along with the data over time, which may lead to the system becoming unresponsive if let go unchecked. Elasticsearch provides a mechanism to create life-cycle policy definitions to help manage and monitor indices productively. When an index ages or crosses a certain size, it can then be rolled over to a new index, thus preventing unavoidable exceptions. Similarly, larger indices which were created in the anticipation of more data can be retired automatically after a set period of time. Though an advanced topic, realizing the index life-cycle management is both fun and engaging.

In the initial part of the chapter, we will look at index operations such as creating, reading, deleting, closing, shrinking, splitting, and other major operations. As expected, Elasticsearch provides a set of indexing APIs using REST over HTTP. Let's jump right into this discussion to work with basic index operations.

6.1 Indexing operations

Let's quickly recap what an index is: it is a logical collection of our data backed up by shards (primary and replicas). Documents represented as JSON having similar attributes, (for example, employees, orders, login audit data, news stories by region, and so on) are held in each of their own respective index. Any index consisting of shards is distributed across various nodes in the cluster. A newly created index is associated with a set number of shards and replicas along with other attributes by default.

We are expected to bring the indices into life with custom configurations. There are a multitude of operations that we can perform when developing an index, from creating it to closing, shrinking, cloning, freezing, deleting, and other operations. Understanding these operations allows you to set up the system for efficient data storage and search retrieval. Let's begin by looking at creating the indices and the operations involved in getting them instantiated.

6.2 Creating indices

When we indexed a document for the first time in our earlier chapters, in addition to getting the document index, Elasticsearch created the index *implicitly*. This is one of the ways we can create an index. There is an alternative method to creating the index implicitly and that is creating the indices *explicitly*. We have much more control in customizing the indices when creating them via the latter route. Let's look at both ways here:

- *Implicit (automatic) creation*—When indexing a document for the first time, if the index doesn't exist, Elasticsearch takes the liberty of creating it implicitly with default settings. This method of index creation usually works well but care should be taken when using this approach in production because incorrect or unoptimized indices will bring unexpected consequences to the running system.

Elasticsearch uses dynamic mapping to deduce the fields when creating the mapping schema with this method. Unfortunately, the mapping definitions produced are not foolproof; for example, data in a nonISO date format (`dd-MM-yyyy` or `MM-dd-yyyy`) is deduced as a `text` field rather than as a `date` data type.

- *Explicit (manual) creation*—Choosing this approach lets you control index creation so we can customize it as required. We can configure the index with a mapping schema doctored by resident data architects or allocate shards as per current and projected storage expectations and so on.

Elasticsearch provides a set of index creation APIs that help create indices with personalized configurations. We can take advantage of these APIs when creating the indices upfront, so the indices are optimized for storage and data retrieval. The APIs provide great flexibility; for example, we can create an individual index with features such as appropriately sized shards, applicable mapping definitions, multiple aliases, and many others.

NOTE To control automatic index creation, we can turn off creating indices by setting an `action.auto_create_index` flag to `false` via the cluster settings API or by setting this property in `config/elasticsearch.yml`. By default, this flag is set to true. We will use this feature shortly.

6.2.1 Creating indices implicitly (automatic creation)

When we index a document for the first time, Elasticsearch won't make any complaints about a non existing index; instead, it happily creates one for us. When an index is created this way, Elasticsearch uses default settings such as setting the number of primary and replica shards to one. To demonstrate, let's quickly index a document with `car` information using the document API. The code in the listing 6.1 demonstrates this.

Listing 6.1 First document containing car data

```
PUT cars/_doc/1 #A
{
  "make": "Maserati",
  "model": "GranTurismo Sport",
  "speed_mph": 186
}
```

#A Until this document gets through, the index doesn't exist

Because this is the first document to be stored in the `cars` index, when you send this request to Elasticsearch, the server instantly creates an index called `cars` because that index doesn't exist in the store. The index is configured with the default settings and a document ID of 1. We can fetch the details of the newly created index by invoking the `GET cars` command as figure 6.1 demonstrates.

```

1  GET cars | ▶ 🔍
The GET cars command will fetch the index details
Every index is made of three components: aliases, mappings and settings
1· {
2·   "cars" : {
3·     "aliases" : { },
4·     "mappings" : {
5·       "properties" : {
6·         "make" : { },
7·         "model" : { },
8·         "speed_mph" : { }
9·       }
10·    },
11·    "settings" : {
12·      "index" : {
13·        "routing" : { },
14·        "number_of_shards" : "1",
15·        "provided_name" : "cars",
16·        "creation_date" : "1630798678818",
17·        "number_of_replicas" : "1",
18·        "uuid" : "_AQTk6RgR7Gh5majM02i6w",
19·        "version" : { }
20·      }
21·    }
22·  }
23·
24·
25·
26·
27·
28·
29·
30·
31·
32·
33·
34·
35·
36·
37·
38·
39·
40·
41·
42·
43·
44·
45·
46·
47·
48·
49·

```

number_of_shards and number_of_replicas are set to 1 each, by default

Figure 6.1 Fetching the details of the cars index by issuing a `GET cars` command

There are a few important things that we should note from the response: each index is made of *mappings*, *settings*, and *aliases*, which we will look at shortly. Elasticsearch creates the mappings schema automatically by deducing the field's data type from the field's value. For example, as the `make` and `model` types seem to have textual information, these fields are created as text fields. Also, Elasticsearch took the liberty of allocating one each of primary and replica shards by default.

NOTE Static settings can't be changed on an operational Index Not all the default settings (`number_of_shards`, `number_of_replicas`, etc.) applied by the engine can be changed on an index that's operational. For example, the `number_of_replicas` setting can be modified on a live index but the `number_of_shards` can't. We need to take that shard offline to change the primary shard's number and other static settings

DISABLING AUTO CREATION FOR AN INDEX FEATURE

As discussed briefly in the previous note, Elasticsearch lets us block the automatic creation of indices by setting the `action.auto_create_index` attribute to false (default is true). We can tweak this setting to modify the flag's value by invoking a cluster-wide property change using the `cluster settings` API. The code listing 6.2 disables the feature:

Listing 6.2: Disabling automatic creation of indices

```
PUT _cluster/settings #A
{
  "persistent": { #B
    "action.auto_create_index":false #C
  }
}
```

#A Updates the settings across the whole cluster

#B The changes can be persistent or transient.

#C Shuts down autocreation

The persistent property indicates that the settings will be permanent. On the contrary, using the transient property saves the settings only until the next reboot of the Elasticsearch server.

While disabling this feature sounds cool, in reality this is not advisable. Although we are restricting automatic creation, there may be an instance of an application or Kibana that may need to create an index for administrative purposes; Kibana creates hidden indices often. (A dot placed in front of an index name is treated as a hidden index. For example, .user_profiles, .admin etc.

Having said that, there is a mechanism to tweak this property beyond binary options. We can provide comma-separated regular expressions that allow (or disallow) this change. For example,

```
action.auto_create_index: .admin*, cars*, *books*, -x*, -y*, +z*
```

This setting allows the automatic creation of hidden indices with admin as the prefix, as well as any index prefixed with cars, books, and those following the + sign. However, this setting does not allow any index starting with x or y to be automatically created because the dash (-) indicates that automatic index creation is disallowed. Do keep in mind that any other indices that don't match this pattern will not be instantiated automatically. For example, if we try to index a document into the flights index, index creation fails as the flights index name doesn't match with the regular expressions we just defined (more on this in a bit). Here's the exception issued by the engine:

```
no such index [flights] and [action.auto_create_index] ([.admin*, cars*, *books*, -x*, -y*, +z*]) doesn't match
```

Allowing the server to create the index fosters a rapid development process. However, it is rare that we go into production without tweaking some of these properties. For example, we may decide to have a strategy of 10 primary shards with 2 replicas per shard; in which case, we must change the settings (it would be disastrous to design a search service with just one primary shard). Also, as we learned in chapter 4, Elasticsearch may not rightly deduce the correct data types based on the document's field values. Incorrect data types lead to setbacks during search operations.

Fortunately, Elasticsearch allows us to create indices to meet our requirements by letting us configure and instantiate them explicitly. However, before we jump into the action of developing custom indices, you'll need to know about index configurations, discussed in the next section.

INDEX CONFIGURATIONS

Every index is made of some sort of configuration, consisting of mappings, settings, and aliases, irrespective of whether it was created automatically or explicitly. We covered mappings in chapter 4, so we'll recap that here and present two other configurations:

- *Mappings*—Mapping is the process of creating a schema definition. Data that gets stored usually has multiple data types associated with its fields, such as `text`, `keyword`, `long`, `date`, and so on. Elasticsearch consults the mapping definitions to apply appropriate rules for analyzing the incoming data before storing it for efficient and effective searching. For example, the following snippet sets the mapping for the `car` entity:

```
PUT cars_index_with_sample_mapping
{
  "mappings": {
    "properties": {
      "make": {
        "type": "text"
      }
    }
  }
}
```

Issuing `GET cars_index_with_sample_mapping/_mapping` fetches the schema for our newly created `cars_index_with_sample_mapping` index.

- *Settings*—Every index comes with a set of configuration settings, such as number of shards and replicas, refresh rate, compression codec, and others. There are few settings (called dynamic settings) that can be tweaked on a live index at run time. Other settings (called static settings) are applied to an index that's in a nonoperational mode. We will look at these two types shortly. The code listing 6.3 configures an index with a few settings:

Listing 6.3: Creating an index with custom settings

```
PUT cars_index_with_sample_settings #A
{
  "settings": { #B
    "number_of_replicas": 3,
    "codec": "best_compression"
  }
}
```

#A Creates the index

#B Applies the settings

Invoking `GET cars_index_with_sample_settings/_settings` fetches the settings for this index.

- **Aliases**—Aliases are alternate names given to indices, and an alias can point to a single or multiple indices. For example, an alias named `my_cars_aliases` could point to all car indices. We can also execute queries on aliases as if we are running them on individual aliases. The code listing 6.4 below shows how to create an alias:

Listing 6.4: Creating an index with an alias

```
PUT cars_index_with_sample_alias #A
{
  "aliases": { # B
    "alias_for_cars_index_with_sample_alias": {} #C
  }
}

#A Creates the index
#B Declares the aliases object to configure the alias
#C The alias itself
```

Issue `GET cars_index_with_sample_alias/_alias` to fetch the alias for this index.

When we create an index explicitly, we have an opportunity to set mappings, settings, and aliases up front. This way, the index is instantiated with all the required configurations in place. We can, of course, modify some of these configurations at run time or some others when the index is in a *closed* (non operational) state. In the next section, we will learn how to set these configurations on indices that are created explicitly.

6.2.2 Creating indices explicitly

Indices created implicitly are seldom ready for production configurations. Creating an index implicitly means that we are expected to set custom configurations. We can direct Elasticsearch to configure an index with the required mappings and settings, as well as aliases, rather than depending on the defaults.

We already know that creating an index is an easy task: simply issue `PUT <index_name>` to create an index of your choice. This command creates a new index with the default configuration (similar to the index created when a document is indexed for the first time). For example, `PUT cars` creates a `cars` index, and issuing `GET cars` returns the index. Let's see how we can manage these indices with custom configurations.

6.2.3 Index with custom settings

Every index can be instantiated with some settings, either with defaults or custom ones during index creation. We can also change some of the settings when the index is still in operation. For this purpose, Elasticsearch exposes the `_settings` API to update the settings on a live index. However, as we mentioned, not all properties can be altered on a live index, only dynamic properties. That brings us to a brief discussion about types of index settings. Index settings exist in two forms, static and dynamic settings. We learn about them in the next two sections.

STATIC SETTINGS

Static settings can only be applied during the process of index creation and cannot be changed while the index is in operation. These are properties like the number of shards, compression codec, data checks on startup, and others. If you want to change the static settings of a live index, you will need to close the index to reapply the settings or recreate the index with new settings altogether.

It is always best to instantiate an index with the required static settings because applying them as an afterthought would require the index to be shutdown. You can, however, manage index upgrades (re-indexing being one form of an upgrade) with zero downtime; more on this later in the book.

DYNAMIC SETTINGS

Dynamic settings are those settings that we can modify on a live (the index that's in operation) index. For example, we can change properties like number of replicas, allowing or disallowing writes, refresh intervals, and others on indices that are in operation.

There are a handful of settings that fall in both camps, so having a high-level understanding of each of those properties will help us in the long run. Let's see how we can instantiate an index with some static settings. We want to create an index with these properties: three shards with five replicas per shard, the compression codec, the maximum number of script fields, and the refresh interval. To apply these configuration settings to our index, we'll use a `settings` object. The code in the listing 6.5 demonstrates a query that creates our index with the required properties.

Listing 6.5 Creating an index with custom settings

```
PUT cars_with_custom_settings #A
{
  "settings":{#B
    "number_of_shards":3, #C
    "number_of_replicas":5, #D
    "codec": "best_compression",#E
    "max_script_fields":128,#F
    "refresh_interval": "60s"#G
  }
}
```

#A Creates an index with custom settings
 #B The settings object encloses the required properties.
 #C Sets the number of shards to 3
 #D Sets the number of replicas to 5
 #E Changes the compression from its default value
 #F Increases the maximum number of script fields from its default of 32
 #G Changes the refresh interval from its default of 1 second

As you can see, we've instructed Elasticsearch to create an index with the settings that we think are essential as per our requirements. Issuing the `GET cars_with_custom_settings` command fetches the details of the index, reflecting the custom settings for shards and replicas that we set a moment ago.

If you try to change any of the static properties (for example, the `number_of_shards` property) on a live index, Elasticsearch throws an exception saying that it can't update non dynamic settings.

We learned that some settings are written in stone once the index is live (static settings) and others (dynamic settings) can be changed on a running index. We can update the dynamic settings using the `_settings` endpoint. The listing 6.6 demonstrates how to update the number of replicas on a running index.

Listing 6.6 Updating dynamic property on an index

```
PUT cars_with_custom_settings/_settings
{
  "settings": {
    "number_of_replicas": 2
  }
}
```

The `number_of_replicas` property is a dynamic property, so it doesn't matter if the index is live; the property is applied instantly.

You may need to consult the official documentation for learning more about the static and dynamic settings that Elasticsearch supports:

<https://www.elastic.co/guide/en/elasticsearch/reference/7.15/index-modules.html#index-modules-settings>

NOTE Elasticsearch will not allow changing the shards count once the index is operational. There is a simple, yet valid reason for this: the routing function (`shard_home = hash(doc_ID) % number_of_shards`) is dependent on the number of shards. When a document is indexed, the document's home shard is deduced as a function of the shard count (as you can see from the routing function). Modifying the shard count breaks the routing function, thus existing documents may get misplaced or incorrectly assigned to a shard.

Should we want to reconfigure indices with new settings, we need to carry out a few steps as explained briefly here:

1. Close the current index (that is, index cannot support read/write operations).
2. Create a new index with the new settings.
3. Migrate the data from the old index to the new index (reindexing operation).
4. Repoint the alias to the new index (assuming the index has an existing alias).

We will see this in action in the later part of the book when we discuss reindexing operations.

Fetching configurational settings is a straightforward job. Simply issue a GET request. The next listing 6.7 demonstrates this approach.

Listing 6.7 Fetching the settings of an index

```
GET cars_with_custom_settings/_settings
```

You can also fetch settings of multiple indices using comma-separated indices or even wildcard patterns on the names. The listing 6.8 shows how to do this.

Listing 6.8 Fetching the settings of multiple indices in one go

```
GET cars1,cars2,cars3/_settings #A
GET cars*/_settings #B
```

#A Fetches multiple index settings
#B Fetches the settings for the index identified with a wildcard (*)

We can also get a single attribute too. For example, the following listing shows a request that fetches the number of shards.

Listing 6.9 Fetching a single attribute

```
GET cars_with_custom_settings/_settings/index.number_of_shards
```

Here, the properties are enclosed in an inner object `index`, hence you must prefix the attribute with the top-level object like `index.<attribute_name>`, for example.

6.2.4 Index with mappings

In addition to settings, we can also provide field mappings when creating an index. The listing 6.10 demonstrates the mechanism to create an index with a mapping definition of a `car` type with `make`, `model`, and `registration_year` attributes.

Listing 6.10 Creating an index with field mappings for a car document

```
PUT cars_with_mappings
{
  "mappings": { #A
    "properties": { #B
      "make":{ #C
        "type": "text" #C
      },
      "model":{ #C
        "type": "text" #C
      },
      "registration_year":{ #D
        "type": "date",
        "format": "dd-MM-yyyy" #E
      }
    }
  }
}
```

#A The `mappings` object encloses the properties.
#B The fields with data types of the car are declared here.
#C Declaring the `make` as a text type
#D Declaring the `registration_year` as a date type
#E The field's custom format

We can also combine both settings and mappings. The following listing demonstrates this approach.

Listing 6.11 Creating an index with both settings and mappings

```
PUT cars_with_settings_and_mappings #A
{
  "settings": { #B
    "number_of_replicas": 3
  },
  "mappings": { #C
    "properties": {
      ..
    }
  }
}
```

#A Index with both settings and mappings

#B Settings on the index

#C Mappings schema definition

Now that we know how to set up an index with settings and mappings, the final piece in the jigsaw is to create an alias.

6.2.5 Index with aliases

Aliases are alternate names given to indices for various purposes such as:

- Searching or aggregating data from multiple indices (as a single alias)
- Enabling zero downtime during re-indexing

Once we create an alias, we can use it for indexing, querying, and all other purposes as if it were an index. Aliases are quite a handy and useful tool during development as well as in production. We can group multiple indices and assign an alias to them so one can write queries against a single alias rather than dozen indices.

To create an index that's similar to the one we saw when configuring the index with settings and mapping, we can set the aliases information in an `aliases` object. The listing 6.12 shows this approach.

Listing 6.12 Creating an alias using an aliases object

```
PUT cars_for_aliases #A
{
  "aliases": {
    "my_new_cars_alias": {}#B
  }
}
```

#A Creates an index with an alias

#B Points the alias to the index

However, there's another way to create aliases rather than using the index APIs: using an alias API. Elasticsearch exposes the `alias` API and the syntax goes like this (check the `_alias` endpoint highlighted in bold in the URL):

```
PUT|POST <index_name>/_alias/<alias_name>
```

Let's create an alias called `my_cars_alias` that points to our `cars_for_aliases` index. The following listing demonstrates how to do this.

Listing 6.13 Creating an alias using an `_alias` endpoint

```
PUT cars_for_aliases/_alias/my_cars_alias
```

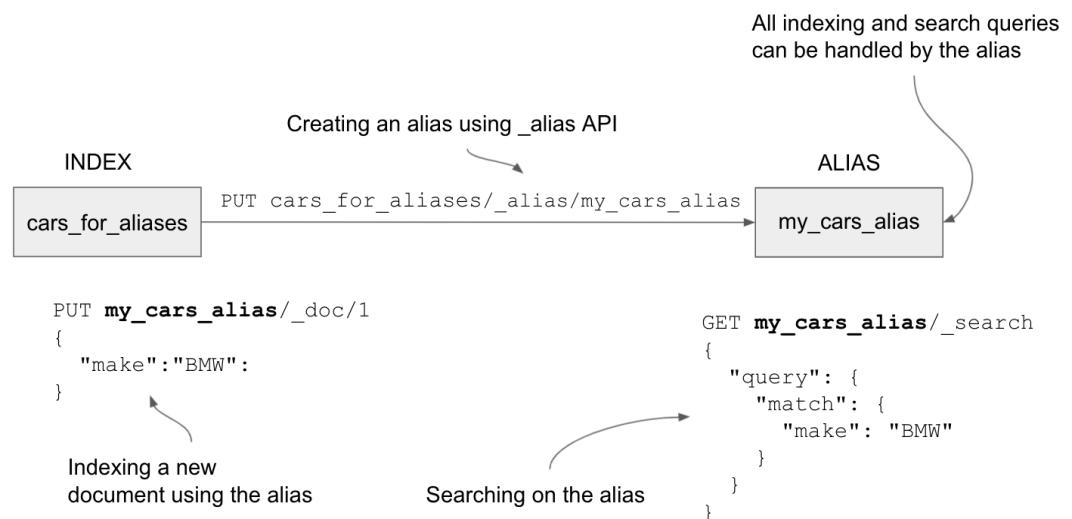


Figure 6.2 Creating an alias for an existing index

We can also create a single alias pointing to multiple indices as figure 6.3 shows, including indices provided with a wildcard.

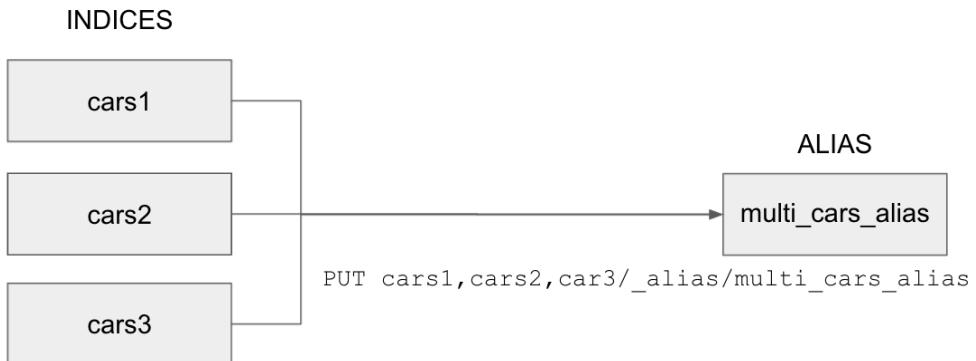


Figure 6.3 Creating an alias pointing to multiple indices

The following listing shows the code that creates an alias (`multi_cars_alias`). This, in turn, points to multiple indices (`cars1, cars2, cars3`).

Listing 6.14 Creating a single alias pointing to multiple indices

```
PUT cars1,cars2,cars3/_alias/multi_cars_alias #A
#A Comma-separated indices list
```

Similarly, we can also create an alias pointing to multiple indices using a wildcard. The listing 6.15 shows this approach:

Listing 6.15 Creating an alias with a wildcard

```
PUT cars*/_alias/wildcard_cars_alias #A
#A All indices prefixed with cars
```

Once the aliases are created, getting the index (`GET <alias_name>`) details will reflect the aliases defined on the index. `GET cars` returns the index with all aliases created on that index (in addition to all the mappings and settings). This is demonstrated in the listing 6.16, which shows that we defined three aliases on the `cars` index.

Listing 6.16 Fetching aliases, settings, and mappings for an index

```
GET cars
```

This call will return all the configurations applicable to the index, as shown in the snippet below:

```
{
  "cars" : {
    "aliases" : { #A
      "all_cars_alias" : { },
      "my_cars_alias" : { },
      "wildcard_cars_alias" : { }
    },
    "mappings" : { ... }, #B
    "settings" : { ... } #C
  }
}
```

#A Shows all aliases created on this index

#B Shows the field mappings

#C Shows the settings

Now that we understand the mechanism of creating aliases, it is time to learn how to fetch the alias details. Similar to settings and mappings, we can send a GET request to the `_alias` endpoint to fetch the details of the aliases, demonstrated in the listing 6.17 here:

Listing 6.17: Getting an alias on a single index

```
GET my_cars_alias/_alias
```

Of course, you can extend the same command to multiple aliases too (listing 6.18):

Listing 6.18: Fetching aliases associated with multiple indices

```
GET all_cars_alias,my_cars_alias/_alias
```

MIGRATING DATA WITH ZERO DOWNTIME USING ALIASES

Configuration settings for indices in production may have to be updated at times with newer properties, perhaps due to a new business requirement or a technical enhancement (or to fix a bug). The new properties may not be compatible with the existing data present in that index; in which case, we can create an index with new settings and migrate the data from the old index into the brand new index.

This may sound well and good, but one potential issue is that the queries that were written against the old index (`GET cars/_search { .. }`, for example) need to be updated because they now need to run against a new index (`cars_new`). If these queries are hardcoded in the application code, there's a chance that you may need a hotfix release to production.

Say you have an index called `vintage_cars` with all the information for ancient and vintage cars, and you now are required to update the index. Here's where we can lean on aliasing; we can devise a strategy with an alias in mind. If you execute the following steps (also illustrated in figure 6.4 below), the migration is achieved most likely with zero downtime.

1. Create an alias called `vintage_cars_alias` to refer to the current index `vintage_cars`.
2. Because the new properties are incompatible with the existing index, create a new index, say `vintage_cars_new` with the new settings.

3. Copy (i.e., reindex) the data from the old index (`vintage_cars`) to the new index (`vintage_cars_new`).
4. Recreate your existing alias (`vintage_cars_alias`), which pointed to the old index, to refer to the new index. Thus, `vintage_cars_alias` will now be pointed to `vintage_cars_new`.
5. Now all the queries that were executed against the `vintage_cars_alias` are carried out on the new index.

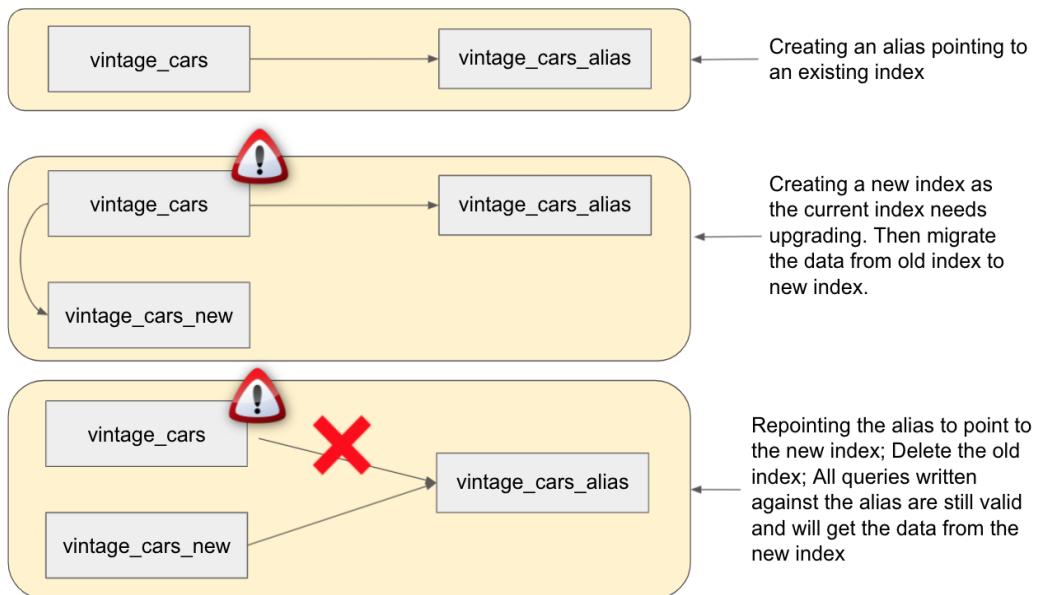


Figure 6.4 Achieving migration with zero downtime

Queries executed on the alias will now fetch the data from the new index, without the application being bounced. Thus, you've achieved zero downtime.

MULTIPLE ALIASING OPERATIONS USING THE `_ALIASES` API

In addition to working with aliases using the `_alias` API, there is another API for working on multiple aliasing actions: the `_aliases` API. It combines a few actions such as adding and removing aliases, as well as deleting the indices, all in one go. Whereas the `_alias` API is used for creating an alias, the `_aliases` API helps create multiple actions on indices related to aliasing.

The code in listing 6.14 performs two distinct aliasing operations on two indices: it removes an alias that was pointing to an old index and repoints (via the add action) the same alias to a new index (see the previous section to understand the need for doing this).

Listing 6.19 Performing multiple aliasing operations

```
POST _aliases #A
{
  "actions": [#B
    {
      "remove": {#C
        "index": "vintage_cars",
        "alias": "vintage_cars_alias"
      }
    },
    {
      "add": { #D
        "index": "vintage_cars_new",
        "alias": "vintage_cars_alias"
      }
    }
  ]
}
```

#A Uses the `_aliases` API to execute multiple actions

#B Lists the individual actions

#C Removes the alias that points to an old index

#D Adds an alias that points to a new index

In listing 6.19, we remove an alias, `vintage_cars_alias`, that was originally created for the `vintage_cars` index. We then reassign it to the `vintage_cars_new` index.

With the `_aliases` API, we can also delete the alias pointing to the existing index and assign it to a new index when the new index is ready after migrating the data. And we can create an alias for multiple indices using the same `_aliases` API by using the `indices` parameter to set up the list of indices. The following listing demonstrates how to do that.

Listing 6.20 Creating an alias pointing to multiple indices

```
POST _aliases
{
  "actions": [
    {
      "add": {
        "indices": ["vintage_cars", "power_cars", "rare_cars", "luxury_cars"],
        "alias": "high_end_cars_alias"
      }
    }
  ]
}
```

The `actions` in listing 6.20 creates an alias called `high_end_cars_alias` that points to four car indices (`vintage_cars`, `power_cars`, `rare_cars`, and `luxury_cars`). Now that we've mastered the art of creating and aliasing indices, let's look at how to read our indices.

6.3 Reading indices

So far the indices we've looked at are *public* indices (those usually created by users or applications for holding data). There's another type of index that exists, however. It's a hidden index, and we'll discuss that in the next section. For now, let's look at reading the public indices.

6.3.1 Reading public indices

We can fetch the details of an index by simply issuing a GET command (like `GET cars`), as we already saw earlier in the chapter. The response provides mappings, settings, and aliases as a JSON object. The response can also return the details of multiple indices. Say we want to return the details of three indices `cars1`, `cars2`, and `cars3`. The following listing shows the way forward:

Listing 6.21 Getting index configurations for multiple indices

```
GET cars1,cars2,cars3 #A
```

#A Retrieves details of three indices

NOTE: Kibana complains if the multiple indices url has a space between the indices. That is `GET cars1, cars2` invocation will fail because there's a white space after the comma. Make sure the multiple indices are separated by just a comma - something to take a note.

This command returns relevant information for all three indices, but providing a long list of a comma-separated indices does not necessarily go well with developers. Instead, we can use wildcards if our indices have a pattern. For example, the code in the following listing fetches all the indices starting with the letters `ca`.

Listing 6.22 Getting multiple index configurations with a wildcard

```
GET ca* #A
```

#A Returns all indices prefixed with `ca`

Using the same principle with comma-separated indices and wildcards, we can get the configurations for specific indices. For example, the following listing (6.23) gets all indices prefixed with `mov` and `stu` (short for movies and students).

Listing 6.23 Fetching configurations for specific indices

```
GET mov*,stu*
```

Although all these GET commands fetch the aliases, mappings, and settings of the specified indices, there is yet another way to return that information. Say we want to fetch an individual configuration for a specific index. We can use the relevant APIs for that as the following listing shows.

Listing 6.24 Getting individual configurations for an index

```
GET cars/_settings #A
GET cars/_mapping #B
GET cars/_alias #C
```

#A Gets the settings from the _settings endpoint
#B Gets the mappings from the _mapping endpoint
#C Gets the aliases from the _alias endpoint

As the listing shows, these `GET` commands return the specified configurations for a public index. Let's look at retrieving this information for a hidden index next.

NOTE *Find If the Index exists* To find out if the index exists in the cluster, we can use the command `HEAD <index_name>`. For example, `HEAD cars` returns the error code `200 - OK` if the index exists, and it returns the error `404 - Not Found` if the index doesn't exist.

6.3.2 Reading hidden indices

As mentioned, there are two types of indices: the normal (public) indices that we've worked with so far and the hidden indices that we'll look at now. Similar to the hidden folders in our computer file systems that are prefixed with a dot (for example, `.etc`, `.git`, `.bash`, etc.), hidden indices are usually reserved for system management. You see them used for Kibana or for application health, and so forth. We can create a hidden index by simply executing the command `PUT .old_cars` (note the dot in front of the index name).

NOTE While we can use hidden indices as public indices because Elasticsearch has no checks and balances in version 8 and earlier, this will change in future versions. All hidden indices will be reserved for system-related work. With this change in mind, my advice is to be mindful of creating hidden indices for business-related data.

The `GET _all` or `GET *` calls fetch all the indices including the hidden indices. For example, figure 6.5 shows the result of issuing a `GET _all` command, which shows the entire index list including the hidden indices (the indices with a dot in front of the name).

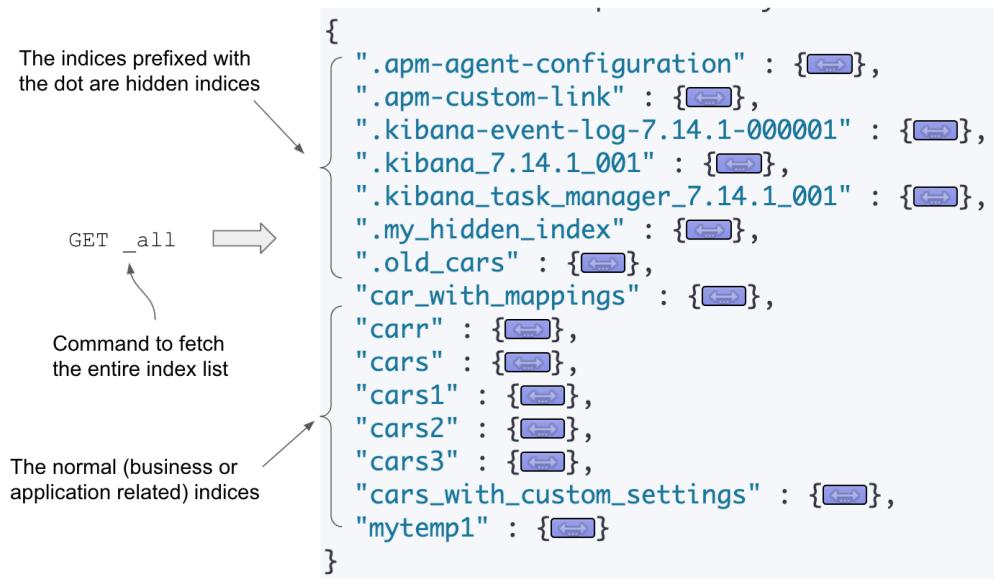


Figure 6.5 Fetching a list of all indices, public as well as hidden.

We looked at reading the public and hidden indices in this section. Now that we know how to create and read our indices, we can, if the need arises, execute delete operations on the indices too. This is discussed in detail in the next section.

6.4 Deleting indices

Deleting an existing index is straightforward: a `DELETE` action on the index (like `DELETE <index_name>`) deletes the index permanently. For example, issuing a `DELETE cars` command deletes the `cars` index when the command is issued, meaning all the documents in that index are gone—forever.

DELETING MULTIPLE INDICES

You can delete multiple indices too. Append a comma-separated list of indices to delete in one go, as shown in the listing 6.25:

Listing 6.25: Deleting documents from multiple indices

```
DELETE cars,movies,order
```

You can delete indices using a wildcard pattern as well: `DELETE *`. However, if you want to delete the hidden indices too, you must use the `_all` endpoint: `DELETE _all`.

WARNING Deleting indices accidentally can result in permanent data loss. When you are working with `DELETE` APIs, extreme caution is advised because accidental invocations can destabilize the system.

DELETING ONLY ALIASES

In addition to deleting the whole index, which internally deletes mappings, settings, aliases, and the data, there's also a mechanism to delete only the aliases. We use the `_alias` API for this purpose as the listing 6.26 demonstrates.

Listing 6.26 Deleting an alias explicitly

```
DELETE cars/_alias/cars_alias #A
```

#A Delete the cars_alias

Deleting an index operation is a destructive task as data is purged permanently, so needless to say, before issuing this operation, know that you really want to delete the index and all its configurations and data. Let's now look at some slightly less destructive operations: closing and opening indices as needed.

6.5 Closing and opening indices

Depending on our use case, we can close or open an index. To begin, let's look at the options we have for closing an index.

6.5.1 Closing indices

Closing the index means exactly that: it is closed for business and any operations on it will cease. There will be no indexing of documents or search and analytic queries.

NOTE Because closed indices are not available for business operations, you must ensure care is taken before closing them. There's a chance it could break the system if indices are closed but referenced in your code. This is one good reason why we should depend on aliases rather than on real indices!

The close index API (`_close`) shuts down the index. The syntax goes like this: `POST <index_name>/_close`. For example, the listing 6.27 closes the `cars` index (until further notice) and, thus, any operations on it results in errors.

Listing 6.27 Closing the cars index indefinitely

```
POST cars/_close?wait_for_active_shards=index-setting
```

You will see an informational message popping up on the Kibana's response pane when you execute the code in listing 6.21. It says that the default value for the property `wait_for_active_shards` has been changed from version 8. To avoid this message, we really should wrap the query with a parameter like that shown here:

```
POST cars/_close?wait_for_active_shards=index-setting
```

The `wait_for_active_shards` property is a positive integer that directs the server to wait until the request processes all the required active shards before returning the response to the client. Through version 7, the default value for `wait_for_active_shards` was 0, but it was changed to `index-setting` with version 8.

CLOSING ALL OR MULTIPLE INDICES

We can close multiple indices by using comma-separated indices (including wildcards). The listing 6.28 shows this procedure.

Listing 6.28 Closing multiple indices

```
POST cars1,*mov*,students*/_close?wait_for_active_shards=index-setting
```

Finally, if we want to halt all the live operations on our indices in a cluster, we can issue a close index API call (`_close`) with either `_all` or `*`, as shown in the listing 6.29:

Listing 6.29: Closing all indices

```
POST */_close?wait_for_active_shards=index-setting #A
```

#A Closes all indices in the cluster

AVOID DESTABILIZING THE SYSTEM

As you can imagine, closing (or opening) *all* indices may destabilize the system. It's one of those super admin capabilities that if executed without forethought, could lead to disastrous results, including taking the system down or making it irreparable. Hence, only administrators should turn this feature off (especially in production). This is done by setting a flag called `action.destructive_requires_name` to `true` in the `elasticsearch.yml` configuration file. We can update the flag using the cluster settings API as listing 6.30 shows.

Listing 6.30 Disabling the action to close all indices

```
PUT _cluster/settings
{
  "persistent": {
    "action.destructive_requires_name":true
  }
}
```

Once we disable this feature, we must provide a specific index name(s) when performing a close action. That means we cannot close all indices using `_all` or `*` operators.

Closing the indices blocks read/write operations and, thus, the overhead of maintaining the shards in the cluster is minimized. The resources will be purged and memory will be reclaimed on the close indices. However, you may be wondering if we can disable closing the indices altogether? Yes, indeed, we can disable the close feature if we never want to close any of the indices. By disabling this feature, we are allowing the indices to be operational forever (unless, of course, you delete them). If this is what you want, we can set a property called `cluster.indices.close.enable` to `false` (it is `true` by default) on the cluster using

the configuration settings. The following listing (listing 6.31) shows how to switch off this feature.

Listing 6.31 Disabling the closing indices feature

```
PUT _cluster/settings
{
  "persistent": {
    "cluster.indices.close.enable":false
  }
}
```

6.5.2 Opening indices

Opening an index kick starts the shards back into business; they are open for indexing and searching once ready. You can open the closed index by simply calling the `_open` API as the following listing shows.

Listing 6.32 Putting the index back into operation

```
POST cars/_open
```

Once the command gets executed successfully, the `cars` index is available and in business instantly. Similar to the `_close` API, the `_open` API can be invoked on multiple indices including specifying indices using wildcards.

We briefly looked at the index settings earlier on and found that Elasticsearch exposes the settings API to work with them. Properties such as number of shards, refresh interval, and so forth can be modified using the settings API. In the next section, we will look at index settings in detail.

So far, we've worked with several index operations, and we've managed to create indices with custom mappings and settings, albeit individually. While this method proves to be effective in development, this is far from ideal in production environments. It can be quite cumbersome to create a set of indices over and over again in these various environments. Also, we may not want developers to create indices with an unexpected number of shards or replicas.

Because tighter control and business standards are expected as part of the overall indexing strategy, Elasticsearch provides a feature to assist in developing indices with an organization strategy in mind. We can use the indexing template feature to apply configurations at scale, which is the topic of the next section.

6.6 Index templates

Copying the same settings across various indices, especially one by one, is tedious and at times erroneous too. If we define an indexing template with a schema upfront, creating a new index will implicitly be molded from this schema if the index name matches the template. Then, any new index will follow the same settings and, hence, be homogenous across the organization. Also, DevOps won't need to advocate the optimal settings to individual teams in an organization over and over.

One use case for an indexing template might be to create a set of patterns based on environments. For example, indices for a development environment would have 3 shards and 2 replicas, while indices for a production environment would have 10 shards with 5 replicas each..

With index templating, we can create a template with predefined patterns and configurations. We can have a set of mappings, settings, and aliases all bundled up in this template, along with an index name. Then, when creating a new index, if the index name matches the pattern name, the template is applied. Additionally, we can create a template based on a *glob* (global command) pattern, such as wildcards, prefixes, and others.

NOTE The *glob* pattern is commonly used in computer software to represent filename extensions (for example, *.txt, *.cmd, *.bat, and so on).

Elasticsearch has upgraded its template functionality beginning with version 7.8. This newer version is abstract and much more reusable. However, if you are still interested in index templates from prior versions, see appendix 3 for details.

6.6.1 Creating index templates

Since Elasticsearch version 7.8, the index templates can be classified into two categories: composable index templates (or simply index templates) and component templates. As the name indicates, composable index templates are composed of zero or more component templates. Having said that, an index template can exist on its own too, without being associated with any component template. This index template can have all the required template features (mappings, settings, and aliases) contained within it. They are used as independent templates when creating an index with a pattern.

On the other hand, although the component template is a template on its own, it's not very useful if it's not associated with an index template. They can, however, be associated with many index templates. Usually, one develops a component template (for example, specifying the codecs for a development environment) and attaches it to various indices via the composable index templates at the same time. This is demonstrated in figure 6.6:

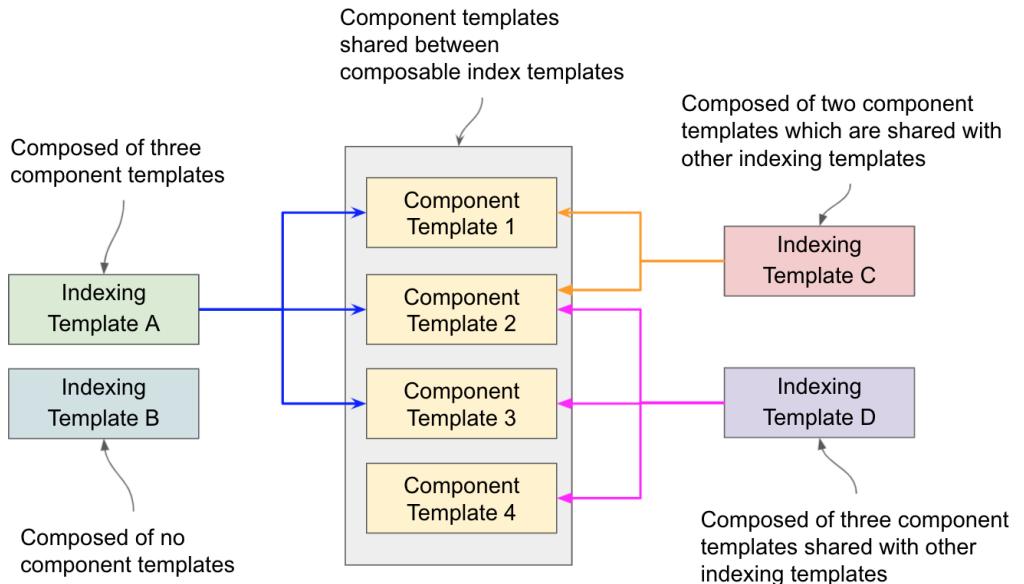


Figure 6.6 Composable (index) templates are composed of component templates.

As you can see in the figure 6.6, the indexing template A and C and D share component templates between themselves (for example, component template 2 is used by all three indexing templates). The indexing template 2 is a standalone template that it composes of no component templates.

We can create an index template with no component templates. This brings us to certain rules when creating templates:

- An index created with configurations explicitly takes precedence over the configurations defined in the index or in component templates. This means if you create an index with explicit settings, don't expect them to be overridden by the templates.
- Legacy templates carry a lower priority than the composable templates.

CREATING COMPOSABLE (INDEX) TEMPLATES

To create an index template, we use an `_index_template` endpoint, providing all the required mappings, settings, and aliases as an index pattern in this template. Let's say our requirement is to create a template for cars, represented with a pattern having wildcards: `*cars*`. This template will have certain properties and settings, such `created_by` and `created_at` properties, as well as shards and replica numbers. Any index that gets matched with this template during its creation inherits the configurations defined in the template. For example, `new_cars`, `sports_cars`, and `vintage_cars` indices would have `created_by` and

`created_at` properties, as well as shards and the replicas 1 and 3, respectively, if these were defined in the template.

Let's create this `cars_template` with an index pattern defined as `*cars*` and with a mapping schema consisting of two properties (`created_by` and `created_at`). The listing 6.33 shows how to create this template.

Listing 6.33 Creating an index template

```
POST _index_template/cars_template
{
  "index_patterns": ["*cars*"],
  "priority": 1,
  "template": {
    "mappings": {
      "properties": {
        "created_at": {
          "type": "date"
        },
        "created_by": {
          "type": "text"
        }
      }
    }
  }
}
```

When you execute this command (you may get a deprecation warning, ignore it for now), the template is created with the index pattern `*cars*`. When we then create a new index, if the name of the new index (say, `vintage_cars`) matches the pattern (`*cars*`), the index gets created with the configuration that we've defined in the template. When the index matches with the given pattern, the templated configurations are applied automatically.

Template Priority

The index template carries a priority, a positive number defined when creating the template: the higher the priority, the higher the precedence. The priority is useful when we have similar or the same settings described in two different templates. If an index matches to more than one index template, the one with higher priority is used. For example, the `cars_template_mar21` overrides the `cars_template_feb21` in the following code snippet (though the names suggest the other way!):

```
POST _index_template/cars_template_mar21
{
  "index_patterns": ["*cars*"],
  "priority": 20, #A
  "template": { ... }

}
POST _index_template/cars_template_feb21
{
  "index_patterns": ["*cars*"],
  "priority": 30, #B
  "template": { ... }
}

#A The lower priority index template
#B Same matching template but with a higher priority
```

When you have multiple templates matching the indices that are being created, Elasticsearch applies all the settings from all matching templates but overrides anything that has higher priority. In the previous example, if the `cars_template_mar21` has a codec that has `best_compression` defined, it's overridden by a codec with a value of `default` (defined in the `cars_template_feb21` template). That's because the latter template has higher priority.

Now that we know more about index templates, the next step is to learn about the reusable component templates. Let me introduce you to component templates in the next section.

CREATING COMPONENT TEMPLATES

If you are from a DevOps background, most likely you'd have a requirement to create indices with a preset configuration for each of the environments. Rather than tediously applying each of these configurations manually, you can create a component template for each of the environments.

A component template is nothing but a reusable block of configurations that we can use to make up more index templates. The component templates are of no value unless they are clubbed with index templates, however. They are exposed via a `_component_template` endpoint. Let's see how this all fits together.

Let's say we need to create a template for a development environment, which is expected to have three primary shards with replicas per each shard. The first step, as the listing 6.34 shows, is to declare and execute a component template with this configuration.

Listing 6.34 Developing a component template

```
POST _component_template/dev_settings_component_template
{
  "template": {
    "settings": {
      "number_of_shards": 3,
      "number_of_replicas": 3
    }
  }
}
```

In this listing, we use the `_component_template` endpoint to create a template. The body of the request holds the template information in a `template` object. Once executed, the `dev_settings_component_template` becomes available for use elsewhere in the index templates. Note that this template does not define any index pattern; it's simply a code block that configures some properties for us. In the same way, let's create another template. This time, let's define a mapping schema, as the listing 6.35 demonstrates.

Listing 6.35 A component template with a mapping schema

```
POST _component_template/dev_mapping_component_template
{
  "template": {
    "mappings": {
      "properties": {
        "created_by": {
          "type": "text"
        },
        "version": {
          "type": "float"
        }
      }
    }
  }
}
```

The `dev_mapping_component_template` consists of a mapping schema predefined with two properties, `created_by` and `version`. Now that we have two component templates, the next step is to put them to use. We can do this by letting an index template for, say `cars`, use it. Check out the code block in the listing 6.36.

Listing 6.36 Creating an index template composed of a component template

```
POST _index_template/composed_cars_template
{
  "index_patterns": ["*cars*"],
  "priority": 200,
  "composed_of": [
    "dev_settings_component_template",
    "dev_mapping_component_template"
  ]
}
```

The highlighted `composed_of` tag is a collection of all component templates (highlighted) that you want to apply. In this case, we are choosing the settings and mappings component templates.

Once this script executes and we create an index with `cars` in the index name (`vintage_cars`, `my_cars_old`, `cars_sold_in_feb`, etc), the index gets created with the configuration derived from both component templates. If you want to create a similar pattern in a production environment, for example, you'll possibly create a composable template with a `prod_*` version of the component templates.

So far we have worked with CRUD operations on indices and, of course, instantiating those using templates. However, we have no visibility on the performance of these indices; we need to let the business know the statistics on the data that's been indexed, deleted, and queried. Fortunately, as the next section discusses, Elasticsearch provides statistics on indices.

6.7 Monitoring and managing indices

Elasticsearch provides detailed statistics on the data that goes into indices as well as what gets pulled out. It provides APIs to knock off reports such as the number of documents that an index holds, deleted documents, merge and flush statistics, and more. In the next couple of sections, we'll go over a few of the APIs to fetch these statistics.

6.7.1 Index statistics

Every index generates statistics such as total number of documents it has, count of documents that were deleted, the shard's memory, get and search request data, and so on. The `_stats` API helps us retrieve statistics of an index, both for primary shards and for replica shards.

The following listing (6.37) demonstrates the mechanism to fetch statistics of a `cars` index by invoking `_stats` endpoint. Figure 6.7 then shows the statistics returned by this call.

Listing 6.37 Fetching the statistics of an index

```
GET cars/_stats
```

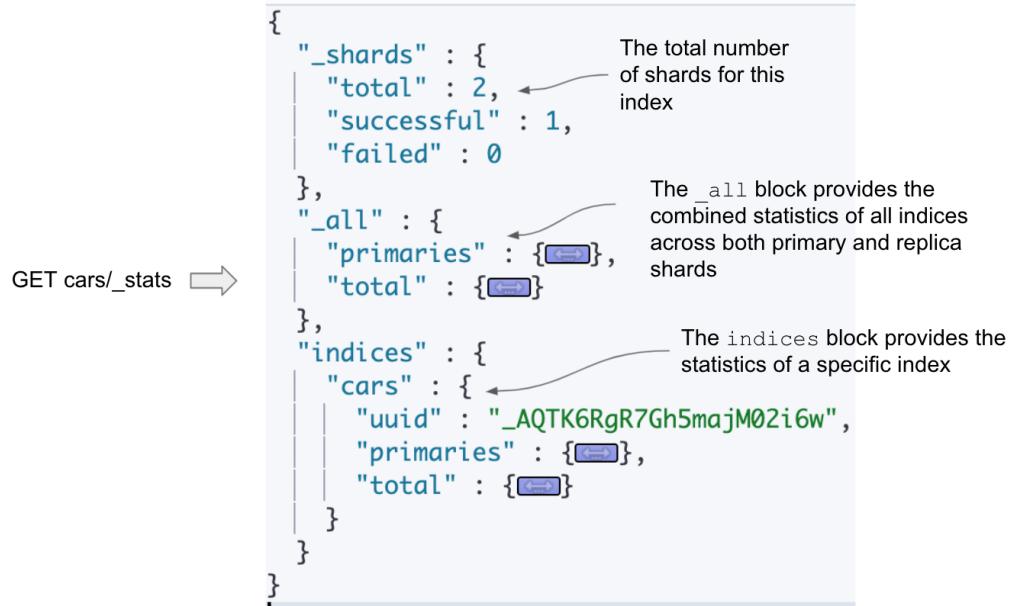


Figure 6.7 Statistics of an cars index

As figure 6.7 shows, the response indicates a `total` attribute, which is the number of total shards (both primary and replica) that are associated with this index. As we have only one primary shard, the `successful` attribute points to this shard number.

UNDERSTANDING THE RESPONSE

The response consists of two blocks primarily:

- The `_all` block, where we see the aggregated statistics of all indices combined
- The `indices` block, where we see the statistics for the individual indices (each of the index in that cluster)

Both these blocks consist of two buckets of statistics: the `primaries` bucket contains the statistics related to just the primary shards, while the `total` bucket indicates the statistics for both primary and replica shards. Elasticsearch returns over a dozen statistics (figure 6.8) that you can find in the response for each of the `primaries` or `total` buckets. Table 6.1 describes a small set of the statistics.

```

"primaries" : {
    "docs" : { },
    "store" : { },
    "indexing" : { },
    "get" : { },
    "search" : { },
    "merges" : { },
    "refresh" : { },
    "flush" : { },
    "warmer" : { },
    "query_cache" : { },
    "fielddata" : { },
    "completion" : { },
    "segments" : { },
    "translog" : { },
    "request_cache" : { },
    "recovery" : { }
}

```

Over a dozen statistics returned by the `_stats` call

Each of these statistics have details on the individual metrics

Figure 6.8 The response from invoking the `GET _stats` endpoint showing multiple statistics. Each block holds individual metrics for the primary (displayed in the figure) and for the replica shards.

Table 6.1 Index statistics fetched by a call to the `_stats` endpoint (the list was shortened for brevity).

Stat	Description
docs	The number of documents that exists in the index as well as the number of documents that were deleted
store	The size of the index (in bytes)
get	The number of GET operations on the index
search	Search operations including query, scroll, and suggest times.
refresh	The number of refresh operations

There are more statistics such as indexing, merge, completion, field data, segments, and others, but due to space constraints, we've omitted them from the table. You can visit <https://www.elastic.co/guide/en/elasticsearch/reference/current/indices.html#monitoring> to find the list of statistics that Elasticsearch exposes via the index statistics APIs.

6.7.2 Multiple indices and statistics

Just like the way we fetched the statistics data on an individual index (as shown in listing 6.37), we can also fetch statistics on multiple indices by providing comma-separated index names. The following listing (6.38) shows the command:

Listing 6.38: Fetching statistics on multiple indices

```
GET cars1,cars2,cars3/_stats
```

We can also use wildcards on indices we chose. The listing 6.37 shows how to do this with wildcards.

Listing 6.39 Fetching the statistics for multiple indices

```
GET cars*/_stats
```

The listing 6.40 shows how to get the stats for all the indices in the cluster (including hidden indices).

Listing 6.40 Fetching the statistics for all indices in the cluster

```
GET */_stats
```

Because we might not need to find all the statistics all the time, we can, for example, find only the statistics about certain segments such as stored fields and terms memory, file size, document count, and others. The following code listing (6.41) shows how to return the statistics in segments:

Listing 6.41: Segment statistics

```
GET cars/_stats/segments
```

This command returns the data on the segments as the following snippet shows:

```
"segments" : {
    "count" : 1,
    "memory_in_bytes" : 1564,
    "terms_memory_in_bytes" : 736,
    "stored_fields_memory_in_bytes" : 488,
    "term_vectors_memory_in_bytes" : 0,
    "norms_memory_in_bytes" : 64,
    "points_memory_in_bytes" : 0,
    "doc_values_memory_in_bytes" : 276,
    "index_writer_memory_in_bytes" : 0,
    "version_map_memory_in_bytes" : 0,
    "fixed_bit_set_memory_in_bytes" : 0,
    "max_unsafe_auto_id_timestamp" : -1,
    "file_sizes" : { }
}
```

While we're on the subject of segments, Elasticsearch provides another API, the index segments API, to allow us to have a peek into the lower-level details of segments. Invoking the code `GET cars/_segments` provides a detailed view of segments managed by Apache

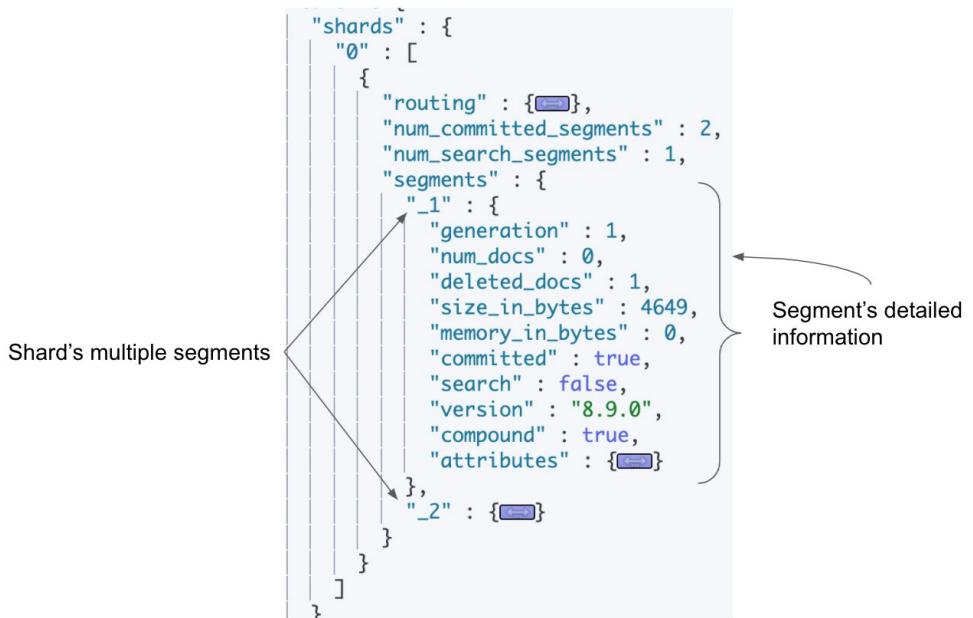


Figure 6.9 The detailed information of every segment in a shard

Of course, we can also get the segment information across the whole index by simply invoking the `GET segments` call.

There are times where we may need to manage an overgrowing index to save space or reduce the infrastructure behind a predominantly dormant index. We can do such operations using some advanced tricks, discussed in detail in the next section.

6.8 Advanced operations

Earlier we looked at some CRUD operations on the indices, such as creating, reading, and deleting them. In addition to such basic operations, we can perform advanced operations like splitting the index to add more shards or, alternatively, downsizing the index by shrinking or rolling over the indices on a periodic basis (such as daily). In this section, we'll go over a few advanced operations that we can perform on our indices.

6.8.1 Splitting an index

Sometimes the indices may need to be overloaded with data so that additional shards may need to be added to the index to manage memory and distribute them evenly. For example,

if an index (`cars`) with 5 primary shards is overloaded, we can split the index into a new index with more primary shards, say 15 shards. This operation of expanding indices from a small size to a larger size is called *splitting*. Splitting is nothing more than creating a new index with more shards and copying the data from the old index into the new index.

Elasticsearch provides a `_split` API for splitting an index. There are some rules as to how many shards a new index can be created with as well as other rules, but first let's see how we can perform splitting an index.

Let's say our `all_cars` index was created with two shards and, as the data is growing exponentially, it is now overloaded. To mitigate the risk of slow queries and degrading performance, we want to create a new index with more space. For this, we can split the index into a brand new index that has more room and additional primary shards.

Before we invoke the split operation on the `all_cars` index, we must make sure the index is disabled for indexing business (the index is changed to a read-only index). To set the index as read-only, the code in the listing 6.42 will help us by invoking the `_settings` API.

Listing 6.42 Making sure the index is read-only

```
PUT all_cars/_settings #A
{
  "settings": {
    "index.blocks.write": "true" #B
  }
}
```

#A Uses the settings API
#B Closes the index for writing operations

Now that the prerequisite of making the index non operational is complete, we can move to the next step of splitting it by invoking the split API. This API expects the source and target indices as described in the format here:

```
POST <source_index>/_split/<target_index>
```

Now, let's split the index into a new index (`all_cars_new`). The listing 6.43 shows how:

Listing 6.43 Splitting the all_cars index

```
POST all_cars/_split/all_cars_new #A
{
  "settings": {
    "index.number_of_shards": 12 #B
  }
}
```

#A `_split` expects a target index
#B Sets the number of shards on the new index

This request kicks off the splitting process. The splitting operation is a synchronous operation, meaning the client's request waits for a response until the process is completed.

Once the split completes, the new index (`all_cars_new`) will have all the data as well as additional space as more shards were added to it.

As mentioned earlier, splitting operation comes with certain rules and conditions. Let's look at some of those in the following list:

- *The target index must not exist before this operation.* This means, other than the configuration that you provide in the request object while splitting (listing 6.41), an exact copy of source index is transferred to the target index.
- *The number of shards in the target index must be a multiple of the number of shards in the source index.* If the source index has 3 primary shards, the target index can be defined with shards as multiples of 3 (that is, 3, 6, 9, . . .).
- *The target index's primary shards can never be less than the source primary shards.* Remember, splitting allows more room for the index.
- *The target index's node must have adequate space.* Make sure the shards are allocated with the appropriate space.

During the splitting operation, all the configurations (settings, mappings, and aliases) are copied from the source index into the newly created target index. Elasticsearch then moves the source segment's hard links to the target index. Finally, all the documents are rehashed as the documents' home has changed.

The target index's primary shard number must be a multiple of the number of primary shards of the source index. If you provide a non-multiple number, reset the `index.number_of_shards` and execute the query. For example, the following snippet resets the number of shards to 14:

```
POST all_cars/_split/all_cars_new

{
  "settings": {
    "index.number_of_shards": 14
  }
}
```

Unfortunately, this query will throw an exception. That's because we violated the second rule in the previous list. Here's the exception:

```
"reason" : "the number of source shards [3] must be a factor of [14]"
```

Splitting indices also helps to resize your cluster by adding additional primaries to the original number. The configurations are copied across the source to the target shards. That means, other than adding more shards, the split API can't change any settings on the target index. If your requirement is to just increase the shards so the data is spread across the newly resurrected one, then splitting is the best way to go. Also, remember the target indices must not exist before invoking a splitting operation.

Now that we've worked on the splitting operations, let's look at another advanced operation. In the next section we'll see the opposite of splitting, which is shrinking the index.

6.8.2 Shrinking an index

While splitting the indices expands the index by adding additional shards for more space, shrinking is the opposite: it reduces the number of shards. Shrinking helps consolidate all the documents spread out in various shards into fewer numbers of shards. Elasticsearch exposes `_shrink` API for this purpose. Let's see it in action.

Let's say we have an index (`all_cars`) distributed among 50 shards and want to resize it to a single digit shard, say, 5 shards. Similar to what we did with a splitting operation, the first step is to make sure our `all_cars` index is read-only, so we'll set the `index.blocks.write` property to `true`. We can then readjust the shards to a single node. The code in the following listing demonstrates these actions as prerequisites before shrinking the index.

Listing 6.44 Carrying out the prerequisites before shrinking the index

```
PUT all_cars/_settings
{
  "settings": {
    "index.blocks.write": true,
    "index.routing.allocation.require._name": "node1"
  }
}
```

Now that the source index is all set for shrinking, we can use the shrink operation. The format goes like this: `PUT <source_index>/_shrink/<target_index>`. Let's issue the shrink command to shrink the `all_cars` index as the following listing shows.

Listing 6.45 Shrinking the index

```
PUT all_cars/_shrink/all_cars_new2 #A
{
  "settings":{
    "index.blocks.write":null,#B
    "index.routing.allocation.require._name":null,#C
    "index.number_of_shards":1,#D
    "index.number_of_replicas":5
  }
}
```

#A Shrinks the source index
#B Removes the read-only instruction
#C Sets the node name to null
#D Reduces the number of shards

We need to understand a few things about the script in listing 6.45. The source index was set with two properties: read-only and the allocation index node name. These settings will be carried over to the new target index if we do not reset them. Hence, in the script, we nullify these properties so the target index wouldn't have these restrictions imposed on when it's created. We also set the number of shards and replicas on the newly instantiated target

index. And, hard links are created for the target index pointing to the source index file segments.

Note Keep in mind that the number of shards must be smaller than (or equal to) the source index's shards (afterall, we are shrinking the index!). And, of course, the target index's shard number must be a factor of the source index's shard number.

While we are here, we can also remove all replicas to the source index so the shrink operation is much more manageable. We just need to set the `index.number_of_replicas` property to zero. Remember the `number_of_replicas` property is a dynamic property, meaning that it can be tweaked on a live index.

There are also a bunch of actions that must be done prior to shrinking indices. The following list provides these actions:

- *The source index must be switched off (made read-only) for indexing.* Although not mandatory, but advised, we can turn off the replicas too before shrinking kicks in.
- *The target index mustn't be created or exists before the shrinking activity.*
- *All target index shards must reside on the same shard.* There is a property called `index.routing.allocation.require.<node_name>` on the index that we must set with the node name to achieve this.
- *The target index's shard number can only be a factor of the source index's shard number.* Our `all_cars` index with 50 shards can only be shrunk to 25, 10, 5, or 1 shard.
- *The target index's node satisfies the memory requirements.*

We can use shrinking operations when the shards are many in number but the data is sparsely distributed. As the name suggests, the idea is to reduce the number of shards. While splitting or shrinking indices is a nice way to manage the indices as our data grows, creating indices on a set pattern with the help of a rollover mechanism is another way to create them. We'll look at the rollover mechanism in the next section.

6.8.3 Rolling over an index alias

Our indices accumulate data over time. Yes, we can split the index as we already saw in an earlier section to handle additional data. However, splitting simply re-adjusts the data into additional shards. Elasticsearch provides another mechanism, called **rollover**, where the current index is rolled over to a new blank index.

Unlike a splitting operation, in a rollover, the documents are not copied to the new index. The old index becomes read-only, and any new documents will be indexed into this rolled over index going forward. For example, if we have an index `app-000001`, rolling over creates a new index `app-000002`. If we rollover once again, another new index, `app-000003`, is instantiated and so on.

The rollover operation is heavily used when dealing with time-series data. The time-series data (data that's usually generated for a specific period like every day, weekly, or monthly) is usually held in an index created for a particular time period. Application logs, for example, are created per date, like `logs-18-sept-21`, `logs-19-sept-21`, and so on.

This will be easy to understand when you see it in action. Let's say we have an index for cars: `cars_2021-000001`. Elasticsearch performs a few steps to rollover the `cars_2021-000001` index. We'll go over these steps in the next couple of sections.

1. Elasticsearch creates an alias pointing to the index (`cars_2021-000001`, for example).

Before Elasticsearch creates this alias, we must make sure the index is writable by setting `is_write_index` to `true`. The idea behind this step is that the alias must have at least one writable backing index.

2. Elasticsearch invokes a rollover command on the alias using the `_rollover` API. This creates a new rollover index (for example, `cars_2021-000002`).

NOTE The trailing suffix (like `000001`) is a positive number, something that Elasticsearch expects the index to be created with. Elasticsearch can increment only from a positive number; it doesn't matter what is the starting number. As long as we have a positive integer, Elasticsearch will increment the number and move forward. If we provide `my-index-04` or `my-index-0004`, for example, the next rollover index will be `my-index-000005`. Elasticsearch automatically pads the suffix with zeros.

CREATING AN ALIAS FOR ROLLOVER OPERATIONS

The first thing we need to do before a rollover operation is to create an alias pointing to the index we want to roll over. We can use the rollover API for index or data stream aliases. For example, the following listing invokes the `_aliases` api to create an alias called `cars_2021_a` for the index `cars_2021-000001` (make sure you have this index created upfront).

Listing 6.46 Creating an alias for an existing index

```
POST _aliases #A
{
  "actions": [ #B
    {
      "add": { #C
        "index": "cars_2021-000001", #D
        "alias": "latest_cars_a", #E
        "is_write_index": true #F
      }
    }
  ]
}
```

#A Uses the `_aliases` API to invoke an add action

#B The set of actions

#C The add action

#D The index

#E Creates the alias

#F Makes the index writable

As you can see, the `_aliases` API request body expects the `add` action with an index and its alias defined. Listing 6.46 creates the alias `latest_cars_a`, pointing to an existing index, `cars_2021-000001`, with the `POST` command.

One important point to note: the alias must point to a writable index. This is why we set `is_write_index` to `true` in the listing. If the alias points to multiple indices, at least one must be a writable index. The next step is to rollover the index, which is discussed next.

ISSUING A ROLLOVER OPERATION

Now that we have an alias created, the next step is to invoke the rollover API endpoint. Elasticsearch has defined a `_rollover` API for this purpose. The endpoint is invoked on the alias as the code in the following listing (6.47) demonstrates.

Listing 6.47 Rolling over the index

```
POST latest_cars_a/_rollover
```

As you can clearly see, the `_rollover` endpoint is invoked on the alias not the index. Once the call is successful, a new index, `cars_2021-000002`, is created (the `*-000001` is incremented by 1). The listing 6.48 shows the response to the call:

Listing 6.48 The response from a `_rollover` call

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "old_index" : "latest_cars-000001", #A Old index name
  "new_index" : "latest_cars-000002", #B New index name
  "rolled_over" : true,
  "dry_run" : false,
  "conditions" : { }
}
```

As the response indicates, a new index (`latest_cars-000002`) was created as a rollover index. The old index was put into a read-only mode to pave the way for indexing documents on the newly created rollover index.

NOTE The rollover API is applied to the alias, albeit the index that was behind this alias is the one that gets rolled over.

Behind the scenes, invoking the `_rollover` call on the alias does a couple of things. In the background, this call:

- Creates a new index (`cars_2021-000002`) with the same configuration as the old one (the name prefix stays the same but the suffix after a dash gets incremented).
- Remaps the alias to point to the new index that was freshly generated (`cars_2021-000002`, in this case).
- Our queries are unaffected because all queries, of course, were written against an alias (not a physical index).
- Deletes the alias on the current index and repoints it to the newly created rollover index.

When we invoke a rollover command, Elasticsearch creates a set of actions (remember, the current index must have an alias pointing the index as a prerequisite):

- Makes the current index read-only (so only queries are executed)
- Creates a new index with the appropriate naming convention
- Repoints the alias to this new index

If you re-invoke the same call (listing 6.46), a new index `cars_2021-00003` is created, and the alias is re-assigned to this new index rather than the old `cars_2021-00002` index. When you are expected to roll over the data to a new index, simply invoking the `_rollover` on the alias will suffice.

Naming conventions.

Let's touch base with the naming conventions we've used when rolling over indices. The `_rollover` API has two formats: one where we can provide an index name and another where the system will deduce it as shown here:

```
POST <index_alias>/_rollover
or
POST <index_alias>/_rollover/<target_index_name>
```

Specifying a target index name as given in the second option lets the rollover API create the index with the given parameter as the target index name. However, the first option, where we don't provide an index name, has a special convention: `<index_name>-0000N`. The number (after the dash) is always made up of 6 digits with padded zeros. If your index follows this format, rolling over creates a new index with the same prefix, but the suffix will be automatically incremented to the next number: `<index_name>-0000N +1`. The increment starts from wherever your original index number is; for example, `my_cars-000034` will be incremented to `my_cars-000035`.

You may be wondering, when would we want to roll over the index? That is actually up to us. When we think the index is clogged or we need to (re)move older data, we can simply invoke the rollover. However, let's first ask ourselves:

- Can we automatically rollover the index when the shard's size has crossed a certain threshold?
- Can we instantiate a new index for everyday logs?

Although we have seen the mechanism of rollover in this section, we can satisfy these questions using the relatively new *index life-cycle management* (ILM) feature, which is discussed at length in the next section.

NOTE Indexing life-cycle management (ILM) is an advanced topic and, most likely, you may not need it when you are just starting out with Elasticsearch. If that's the case, you can skip this discussion and return to it when you need to learn how Elasticsearch deals with time-series data and how we can act upon indices by rolling them over freezing or deleting them based on certain conditions "automagically" (well, based on defined policies), and more.

Indices are expected to grow in size as data pours in over time. Sometimes an index is written too frequently that the underlying shards run out of memory, and other times most of the shards may be sparsely filled. Wouldn't it be ideal to automatically rollover the index in the former case while shrinking the index in the latter?

There is also time-series data that we need to consider. Take an example of logs that are written to a file on a daily basis. These logs are then exported to the indices suffixed with a time period like `my-app-2021-10-24.log`. When a day is rolled off to the next day, you'd expect the respective index to be rolled over too; for example, `my-app-2021-10-24.log` to `my-app-2021-10-25.log` (the date is incremented by day) as the figure 6.10 shows:

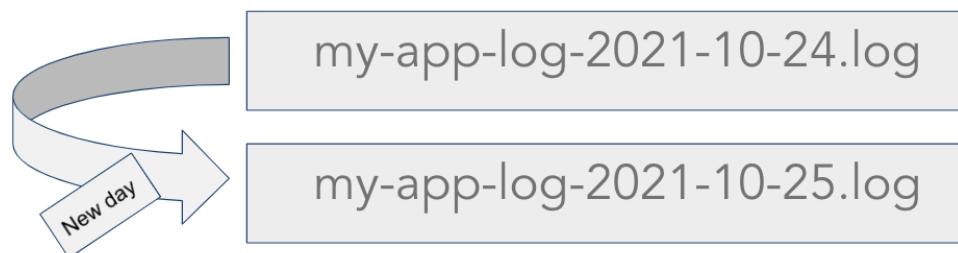


Figure 6.10: Rolling over to a new index when as the new day dawns

We can write a scheduled job that can do this for us, but fortunately, Elastic released a new feature relatively recently called *index life-cycle management* (ILM).

As the name suggests, the ILM is all about managing the indices based on a life-cycle policy. The policy is a definition that declares some rules that are executed by the engine when the conditions of the rules are met. For example, we can define rules based on rolling over the current index to a new index when:

- The index reaches a certain size (say 40 GB, for example)
- The number of documents in the index crossed, say, 10,000
- The day is rolled over

Before we start scripting the policy, let's understand the life cycle of an index: the various phases of an index can progress based on some criteria and conditions.

6.9.1 About the index life cycle

An index has five life-cycle phases - hot, warm, cold, frozen, and delete - as demonstrated in figure 6.11.

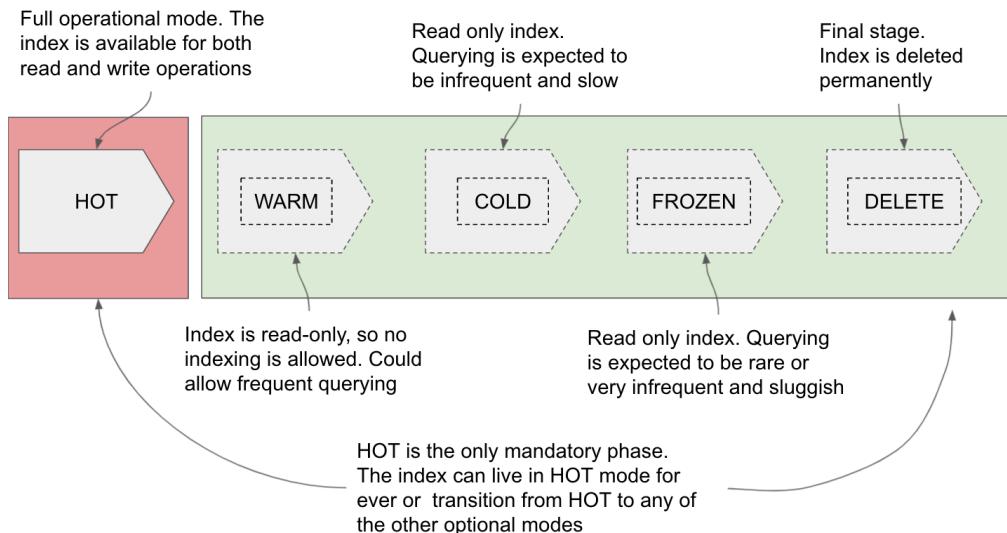


Figure 6.11 Life cycle of an index

Let's look at the brief description of each of these phases:

- **Hot**—The index is in full operation mode. It is available for both read and write, thus enabling the index for both indexing and querying.
- **Warm**—The index is in read-only mode. Indexing is switched off but open for querying so that the search and aggregation queries can still be served by this index.
- **Cold**—The index is in read-only mode. Similar to the warm phase, where indexing is switched off, and it's open for querying, albeit the queries are expected to be infrequent. When the index is in this phase, the search queries might result in slow response times.
- **Frozen**—This is similar to the cold phase, where the index is switched off for indexing but querying is allowed. However, the queries are more infrequent or even rare. When the index is in this phase, users may seem to notice longer response times for their queries.
- **Delete**—This is the index's final stage, where the index gets deleted permanently. As such, the data is erased and resources are freed. Usually, it's expected that we take a snapshot of the index before deleting so that the data from the snapshot can be restored at some point in the future.

Transitioning from the hot phase into every other phase is optional. That is, once created in the hot phase, the index can remain in that phase or transition to any of the other four phases. In the next sections, we will check out a few examples that set an indexing life-cycle policy so that the indices can be managed automatically by the system.

6.9.2 Managing life cycle manually

So far, we've managed to create or delete an index on demand when needed i.e., intervening manually. But we were unable to delete, rollover, or shrink indices based on certain conditions such as size of the index exceeding a certain threshold, after a certain number of days, and so forth. We'll use ILM to help us set up this feature.

Elasticsearch exposes an API for working with the index life-cycle policy, and the format goes like this: `_ilm/policy/<index_policy_name>`. The process is split into two steps - defining a life cycle policy and associating policy with an index for execution - as detailed in the following sections.

STEP 1: DEFINING A LIFE-CYCLE POLICY

The first step is to define a life-cycle policy, where we provide the required phases and set the relevant actions on those phases. The code in the following listing defines such a policy.

Listing 6.49 Creating a policy with hot and delete policies

```
PUT _ilm/policy/hot_delete_policy #A
{
  "policy": { #B
    "phases": {
      "hot": { #C
        "min_age": "1d", #D
        "actions": { #E
          "set_priority": { #F
            "priority": 250
          }
        }
      },
      "delete": { #G
        "actions": {
          "delete" : { }
        }
      }
    }
  }
}
```

```
#A The ILM API
#B Defines the policy and its phases
#C Defines the first phase (the hot phase)
#D Sets the minimum age before other actions are carried out
#E Defines the actions that must be carried out
#F Sets the priority
#G Defines the delete phase
```

The `hot_delete_policy` in listing 6.49 defines a policy with two phases: hot and delete. Here's what the definition states:

- *Hot phase*—The index is expected to live for at least one day before carrying out the actions. The actions block defined in the “actions” object sets a priority (250 in this example). The indices with a higher priority are acted on first during node recovery.
- *Delete phase*—The index is deleted once the hot phase completes all the actions. As there is no `min_age` on the delete phase, the delete action kicks in immediately once the hot phase finishes.

The first step of declaring and defining the policy is complete. Now, how do we get this policy attached to an index of our choice? That’s exactly what we’ll discuss in the next section.

STEP 2: ASSOCIATING THE POLICY WITH AN INDEX

Now that we have the policy defined, the next step is to get an index associated with this policy. To see this in action, let’s create an index and attach the policy from listing 6.48 to the index. The following listing creates the index definition.

Listing 6.50 Creating an index with an associated index life cycle

```
PUT hot_delete_policy_index
{
  "settings": { #A We use settings object to set the property
    "index.lifecycle.name": "hot_delete_policy" # Name of the policy
  }
}
```

This script creates the `hot_delete_policy_index` index with a property setting, `index.lifecycle.name`: the index is now associated with a life-cycle policy as the `index.lifecycle.name` points to our previously created policy (`hot_delete_policy`) in listing 6.50. This means, the index would undergo phase transition as per the policy definition. That is, once the index is created, it first enters into the hot phase and stays in that hot phase for a day (`min_age=1d`) before it applies a few actions (in this case, setting a priority on the index).

As soon as the hot phase completes (one day, to be precise as per the policy definition), the index transitions into the next stage, the delete phase in this case. This is a straight forward phase where the index gets deleted automatically.

NOTE The `hot_delete_policy` policy defined in listing 6.47 deletes the index after one day as per the definition. Be aware that if you are using this policy in production, you may find no index available after the deletion phase (the delete phase purges everything).

To sum up, attaching an index life-cycle policy to an index transitions the index into given phases and executes certain actions defined in each of those respective phases. We can surely define an elaborate policy such as a hot phase for 45 days, then move to a warm phase, which will be alive for one month. We then could move it to the cold phase from the warm phase and keep it on cold for one year, and finally, delete the index after a year.

In this section, we looked at the mechanics of ILM: we learned how we can define a policy with various phases and attach it to an index. We executed rollover scripts so the indices rolled over as needed. However, suppose we want to transition the indices based on

some conditions such as every month or a particular size? Fortunately, Elasticsearch provides just that—automated and conditional index life-cycle rollovers, and that's the topic of the next section.

6.9.3 Life cycle with rollover

In this section, we'll set conditions on a time-series index to magically roll when those conditions are met. Let's say we want the index to be rolled over based on the following conditions:

- On each new day.
- When the maximum number of documents hits 10,000.
- When the maximum index size reaches 10 GB.

We need to define a life-cycle policy that encompasses these conditions. The script in listing 6.49 defines a simple policy declaring a hot phase where the shards are expected to roll over based on these conditions with a few actions.

Listing 6.51 Simple policy definition for hot phase

```
PUT _ilm/policy/hot_simple_policy
{
  "policy": {
    "phases": {
      "hot": {
        "#A
        "min_age": "0ms", #B
        "actions": {
          "rollover": {
            "#C
            "max_age": "1d",
            "max_docs": 10000,
            "max_size": "10gb"
          }
        }
      }
    }
  }
}
```

#A Declares a hot phase

#B Index enters this phase instantly.

#C Index rolls over if any of the conditions are met.

In our policy, we declared one phase, the hot phase, with rollover as the action to be performed when any of the conditions declared in the rollover actions are met. For example, if the maximum number of documents is 10,000 or the size of the index exceeds 10 GB or if the index is one day old, the index rolls over. As we declared the minimum age (`min_age`) to be `0ms`, as soon as the index is created, it gets moved into the hot phase instantly and then rolled over.

The next step is to create an indexing template, attaching the life-cycle policy to it. The script in the listing 6.50 declares an index template with an index pattern `mysql-*`.

Listing 6.52 Attaching a life-cycle policy to a template

```
PUT _index_template/mysql_logs_template
{
  "index_patterns": ["mysql-*"], #A
  "template": {
    "settings": {
      "index.lifecycle.name": "hot_simple_policy", #B
      "index.lifecycle.rollover_alias": "mysql-logs-alias" #C
    }
  }
}
```

#A The index pattern for all mysql indices

#B Attaching the policy

#C Attaching an alias

There's a couple of things we may need to note from the index template script. We must associate our previously defined index policy by setting it as `index.lifecycle.name` with this index template. Also, as the policy's definition has a hot phase with `rollover` defined, we must provide the `index.lifecycle.rollover_alias` name when creating this index template.

The final step is to create an index matching the index pattern defined in the index template, with a number as a suffix so the rollover indices are generated correctly. Another thing to note is that we must define the alias and declare that the current index writeable by setting `is_write_index` to `true` as the listing 6.53 shows.

Listing 6.53 Setting the index as writable for the alias

```
PUT mysql-index-000001 #A
{
  "aliases": {
    "mysql-logs-alias": { #B
      "is_write_index": true #C
    }
  }
}
```

#A Creates an index with the appropriate format

#B Enables the alias

#C The backing index must be writable.

Once we create the index, the index policy kicks in. In our example, the index enters into the hot phase as the `min_age` is set to 0 milliseconds and then moves into the phase's `rollover` action. The index stays in this phase until one of the conditions (age or size of the index or number of docs) is met. As soon as the condition is positive, the rollover phase gets executed and a new index `mysql-index-000002` is created (note the index suffix). The alias is remapped to point to this new index automatically. Then `mysql-index-000002` is rolled over to the `mysql-index-000003` index (again, if one of the conditions is met) and the cycle continues.

Policy scan interval .

By default, policies are scanned every 10 minutes. You need to update the cluster settings using the `_cluster` endpoint if you want to alter this scan period.

Usually, when we are trying out the life-cycle policies in development, a common issue we see is that none of the phases are executed. For example, although you may have set the phases' times (`min_age`, `max_age`) in milliseconds, you may notice none of your phases are executing. Not realizing the scan's interval, we may think the life-cycle policies are not being invoked, but actually this is due to the fact that the policies are waiting to be scanned.

We can reset this scan period by invoking the `_cluster/settings` endpoint with the appropriate time period. For example, the following snippet resets the poll interval to 10 milliseconds:

```
PUT _cluster/settings
{
  "persistent": {
    "indices.lifecycle.poll_interval":"10ms"
  }
}
```

Now that we understand the index rollover using life-cycle policies, let's script another policy with multiple phases. The listing 6.54 shows how to create this.

Listing 6.54 Creating an advanced life-cycle policy

```
PUT _ilm/policy/hot_warm_delete_policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "1d", #A
        "actions": {
          "rollover": { #B
            "max_size": "40gb",
            "max_age": "6d"
          },
          "set_priority": { #C
            "priority": 50
          }
        }
      },
      "warm": {
        "min_age": "7d", #D
        "actions": {
          "shrink": { #E
            "number_of_shards": 1
          }
        }
      },
      "delete": { #F
        "min_age": "30d", #G
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

#A The index waits for one day before becoming hot.

#B Rolls over when one of the conditions is met

#C Sets the priority (an additional action)

#D Wait for 7 days before carrying out the actions

#E Shrinks the index

#F Deletes the index, but ...

#G ... before deleting, stay in this phase for 30 days

This policy consists of hot, warm, and delete phases. Let's look at what happens and what actions are executed in these phases:

- *Hot phase*—The index enters into this phase after one day because the `min_age` attribute was set to `1d`. After one day, the index moves into the roll-over stage and waits for the conditions to be satisfied: the maximum size is 40 GB (`"max_size": "40gb"`) or the age is older than 6 days (`"max_age": "6d"`). Once any of these conditions are met, the index transitions from the hot phase to the warm phase.

- *Warm phase*—When the index enters the warm phase, it stays in the warm phase for about one week ("min_age": "7d") before any of its actions are implemented. After the seventh day, the index gets shrunk to one node ("number_of_shards": 1), then the phase is deleted.
- *Delete phase*—The index stays in this phase for 30 days ("min_age": "30d"). Once this time lapses, the index is deleted permanently. Be wary of this stage as the delete operation is irreversible! My advice is to take a backup of the data before you delete the data permanently.

It's time to wrap up. In this chapter, we learned quite a lot about indexing operations and ILM. In the next chapter, we will go over text analysis in detail so stay tuned.

6.10 Summary

- Elasticsearch exposes index APIs to create, read, delete, and update indices.
- Every index has three sets of configurations: aliases, settings, and mappings.
- Indices can be created implicitly or explicitly:
 - Implicit creation kicks in when an index doesn't exist and a document is indexed for the first time. Default configurations (such as one replica and one shard) are applied on an index that's created implicitly.
 - Explicit creation occurs when one instantiates indices with a custom set of configurations using the index API.
- An index template lets us create indices with predetermined configuration settings that, based on a matching name, are applied during the index creation.
- An index can be resized using a shrinking or splitting mechanism. Shrinking reduces the size of the shards, while splitting adds more primary shards.
- An index can be conditionally rolled over as required.
- Index life-cycle management (ILM) helps transition the indices between these life-cycle phases: hot, warm, cold, and delete. In the hot phase, the index is fully operational and is open for searching and indexing, but in the warm and cold phases, the index is read-only.

7

Text analysis

This chapter covers

- Overview of text analysis
- Anatomy of an analyzer
- Built-in analyzers
- Developing custom analyzers
- Understanding tokenizers
- Learning about character and token filters

Elasticsearch does a lot of ground (and grunt) work behind the scenes on incoming textual data. It preps data to make it efficiently stored and searchable. In a nutshell, Elasticsearch cleans the text fields, breaks the text data into individual tokens, and enriches the tokens before storing them in the inverted indices. When a search query is carried out, the query string is searched against the stored tokens and, accordingly, any matches are retrieved and scored. This process is termed as text analysis and is usually expected to be performed on all text fields.

The aim of the text analysis is to not just return search results quickly and efficiently, but to retrieve relevant results too. The groundwork is carried out in the name of text analysis by employing so-called *analyzers*. The analyzers are software components prebuilt to inspect the input text according to some rules. If the user searches for “K8s”, for example, we should be able to fetch books on Kubernetes, even though the criteria was K8s. Similarly, if the search word is “emojis”, the search engine should be capable of extracting the appropriate results. All this and many more search criteria are honored by the engine due to the way we configure the analyzers.

There are a handful of analyzers out of the box (see section 7.3), each one working on the input text in one form or another. For example, the standard (default) analyzer lets us work easily on English text by tokenizing the words using whitespace and punctuation as well

as lowercasing the final tokens. If we want to stop indexing a set of predefined values (either common words like "a", "an", "the", "and", "if", and so on or, perhaps, swear words), we can customize the analyzer to do so.

Elasticsearch provides a set of prebuilt analyzers that work for most common use cases. The most notable are, in addition to the standard analyzer keyword, simple, stop, whitespace, pattern, language, and a few other analyzers. There are language-specific analyzers too, like English, German, Spanish, French, Hindi, and so on.

While this analysis process is highly customizable, the out-of-the-box analyzers fit the bill for most circumstances. Should we need to configure a new analyzer, we can do so by mixing and matching tokenizers and character and token filters (the components that make an analyzer) from a given vast catalog of them.

It is important and vital to understand how the text is treated and stored by Elasticsearch, how the search is performed effectively, and how the relevancy is scored. At times the search results would be what we were expecting, and the reason could be the way the input was indexed.

Understanding the text analysis helps us to know how the text is analyzed internally and how it is tokenized, synonymized, stemmed, and stored. Having a thorough knowledge about the text analysis process can help us troubleshoot search-related issues. Knowing the mechanics of analyzers and their building blocks helps us learn to work with analyzers, as well as build custom analyzers for varied languages if required.

7.1 Overview

Elasticsearch stores structured and unstructured data. As we've gathered from the previous chapters, working with structured data is straightforward: we match the documents for the given query and return the result. For example, retrieving a customer's information by their email address, finding out the flights that were canceled between a set of dates, getting the sales figures for the last quarter, fetching a list of patients assigned to a surgeon on a given day, and so on. The results are definitive: results are returned if the query matches the documents and not returned if the query does not match.

Querying unstructured data, on the other hand, involves finding out not only if the documents match the query, but also how *relevant* the document is to the query or how well the document matches. For example, searching for "K8s" across book titles should fetch the *Kubernetes in Action* book or "chasing red october" might fetch the *The Hunt for Red October* movie.

NOTE Analyzers during the search Text is analyzed during indexing the data as well at querying time. That is, just as the field gets analyzed during indexing, the search query will go through the same process. It is more often than not to use the same analyzer during searching time too, but there might be a requirement that one might want to use a different analyzer for searching. This feature of using different analyzers for search along with specifying required analyzers for indexing time is discussed in section 7.5 in detail.

7.1.1 Querying unstructured data

Unstructured data is nothing but our day-to-day human language. It consists of free text (like a movie synopsis), a research article, an email body, blog post, and so on. Exact matches wouldn't fetch the results in these cases. Let's say we have a famous quote from Einstein indexed in our search engine:

"Imagination is more important than knowledge"

Users get positive results with queries made from either an individual word or combination of words as the first two rows in the table 7.1 shows.

Table 7.1: Possible search queries and expected results

Search keywords	Results	Notes
imagination, knowledge	Yes	Individual keywords are exact matches, hence positive results are returned.
imagination knowledge, knowledge important	Yes	Combined keywords also match the documents and, hence, returns results.

However, searching for other criteria may yield no results. For example, as demonstrated in table 7.2, if the user searches for "passion", "importance", or "passionate wisdom", "curious cognizance", etc., the engine fails to return matching results on default settings. Words such as "passion", "cognizance", and so forth are synonyms of the word "knowledge". Similarly, abbreviations are missed.

Table 7.2 Possible search queries and expected results

Search keywords	Results	Notes
imagine, passion, curious, importance, cognizance, wisdom, passionate wisdom, extra importance	No	Any synonyms or alternate names do not yield positive results.
important, knowlege, imaginaton	No	Spelling mistakes might result in poor or no matches.
Imp, KNWL, IMGN	No	Abbreviations do not return any positive results.

We should expect users to query our engine with numerous combinations: synonyms, abbreviations, acronyms, emojis, language lingo, and so on. A search engine that helps to fetch the relevant answers for a wide variety of search criteria always wins.

7.1.2 Analyzers to the rescue

To build an intelligent and never disappointing search engine, extra assistance is given to the engine during the data indexing process in the name of *text analysis* and carried out by software modules called *analyzers*. To serve a multitude of queries, one must prepare and help the search engine on the data that it's consuming during the indexing phase.

NOTE Only text fields are analyzed, the rest are not! Elasticsearch analyzes only the text fields before storing them into their respective inverted indices. Any other data types wouldn't undergo text analysis. It also uses the same principle of analyzing the query's text fields when executing the search query.

Analyzing the data during the indexing process and capturing it as per our requirements lets Elasticsearch satisfy a spectrum of search query variations. An analyzer is the component that works on the text to make sure it fits our purpose. An analyzer is a software module that consists of few other components to help analyze the text.

7.2 Analyzer module

The analyzer is a software module essentially tasked with two functions: tokenization and normalization. Elasticsearch employs *tokenization* and *normalization* processes so the text fields are thoroughly analyzed and stored in inverted indexes for advanced query matching. Let's look at these concepts at a high level before drilling down into the anatomy of the analyzer.

7.2.1 Tokenization

Tokenization is a process of splitting sentences into individual words, and it follows certain rules. For example, we can instruct the process to break sentences by a delimiter such as whitespace, a letter, a pattern, or other criteria. This process is carried out by a component called a *tokenizer*, whose sole job is to chop the sentence into individual words called *tokens* by following certain rules when breaking the sentence into words. A whitespace tokenizer is commonly employed during the tokenization process with each word in the sentence separated by whitespace, removing any punctuation and other noncharacters.

The words can also be split based on nonletters, colons, or some other custom separators. For example, a movie reviewer's assessment saying, "The movie was sick!!! Hilarious :) :)" can be split into individual words: "The", "movie", "was", "sick", "Hilarious", and so on (note the words are not yet lowercased). Or "pickled-peppers" can be tokenized to "pickled" and "peppers", "K8s" can be tokenized to "K" and "s", and so on. While this helps to search on words (individual or combined), it can only go so far to answer all the queries such as those with synonyms, plurals, and other searches we mentioned earlier. The normalization process comes to rescue in such cases.

7.2.2 Normalization

Normalization is where the tokens (words) are massaged, transformed, modified, and enriched in the form of stemming, synonyms, stop words, and other features. Stemming is

an operation where the words are reduced (stemmed) to their root words (for example, “author” is a root word for “authors”, “authoring”, and “authored”).

In addition to stemming, normalization also deals with finding appropriate synonyms before adding them to the inverted index. For example, “author” may have additional synonyms such as “wordsmith”, “novelist”, “writer”, and so on. And finally, each document will have a number of words such as “a”, “an”, “and”, “is”, “but”, “the”, and so on that are called *stop words* because they really do not have a place in finding the relevant documents.

Both these functions—tokenization and normalization—are carried out by the analyzer module. An analyzer does this by employing filters and a tokenizer. Let’s dissect the analyzer module and see what it is made of.

7.2.3 Anatomy of an analyzer

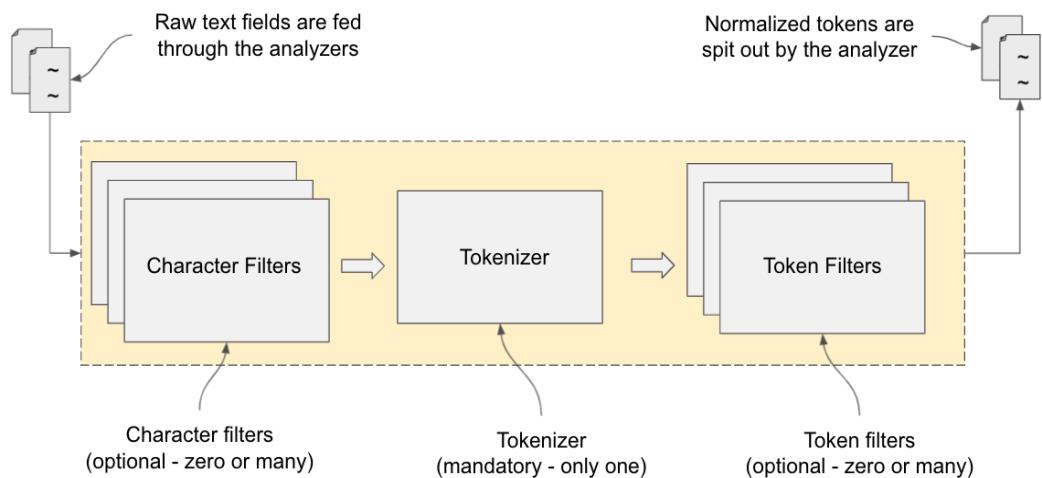


Figure 7.1 Anatomy of an analyzer module

All text fields go through this pipe: the raw text is cleaned by the character filters, and the resulting text is passed on to the tokenizer. The tokenizer then splits the text into tokens (aka individual words). The tokens then pass through the token filters where they get modified, enriched, and enhanced. Finally, the finalized tokens are then stored in the appropriate inverted indices. The search query gets analyzed too, in the same manner as the indexing of the text.

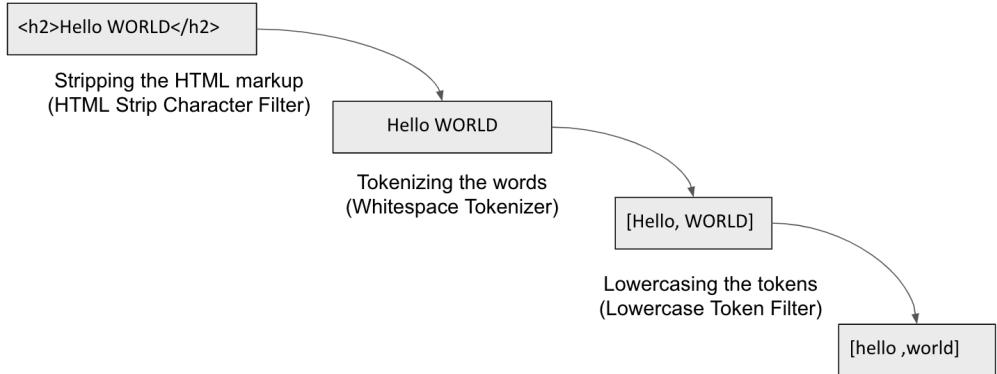


Figure 7.2 An example of text analysis in action

The analyzer is composed of three low-level building blocks. These are

- *Character filters*—Applied on the character level, where every character of the text goes through these filters.

The filter's job is to remove unwanted characters from the text string. This process could, for example, purge HTML tags like `<h1>`, `<href>`, `<src>` from the input text. It also helps replace some text with other text (e.g., Greek letters with the equivalent English words) or match some text in a regular expression (regex) and replace it with its equivalent (e.g., match an email based on a regex and extract the domain of the organization).

Elasticsearch provides three character filters out of the box: `html_strip`, `mapping` and `pattern_replace`. These character filters are optional; analyzers can exist without a character filter.

- *Tokenizers*—Split the sentences into words by using a delimiter such as whitespace, punctuation, or some form of word boundaries.

Every analyzer must have one and only one tokenizer. Elasticsearch provides a handful of these tokenizers to help split the incoming text into individual tokens. The words can then be fed through the token filters for further normalization. A standard analyzer is used by Elasticsearch by default, which breaks the words based on grammar and punctuation.

- *Token filters*—Work on tokens produced by the tokenizers for further processing. For example, the token can change the case, create synonyms, provide the root word (stemming), or produce n-grams and shingles, and so on.

Token filters are optional. They can either be zero or many, associated with an analyzer module. There is a long list of token filters provided by Elasticsearch out of the box.

We will look at these components in detail a bit later in the chapter, but let's first find an API that will help test the analyzers before we put the analyzers into production.

7.2.4 Testing analyzers

You might be a bit curious to find out how Elasticsearch breaks the text, modifies it, and then plays with it. After all, knowing how the text is split and enhanced upfront helps us to choose the appropriate analyzer and customize it if needed. Fortunately, Elasticsearch exposes an endpoint just for testing the text analysis process. It provides an `_analyze` endpoint that helps us understand the process in detail. This is really a handy API that allows us to test how the engine treats a text when indexed. It'll be easy to explain using an example.

Let's just say we want to find out how Elasticsearch deals with this piece of text when indexed: "James bond 007". The following listing shows this in action.

Listing 7.1 Testing the analyzer using the `_analyze` endpoint

```
GET _analyze
{
  "text": "James Bond 007"
}
```

This script produces a set of tokens as shown in figure 7.3.

```
{
  "tokens": [
    {
      "token": "james",
      "start_offset": 0,
      "end_offset": 5,
      "type": "<ALPHANUM>",
      "position": 0
    },
    {
      "token": "bond",
      "start_offset": 6,
      "end_offset": 10,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "007",
      "start_offset": 11,
      "end_offset": 14,
      "type": "<NUM>",
      "position": 2
    }
  ]
}
```

- The analyzer used here is standard analyzer - used by default as it was not specified in the query
- The input text is broken into a set of tokens - three tokens precisely in this case: james, bond, 007
- Elasticsearch deduced types associated with each token: ALPHANUM - for alphanumerics and NUM for numbers
- The start_offset and end_offsets indicate the start and end character offset of the word
- All tokens were lowercased

Figure 7.3 The tokens produced by invoking _analyze endpoint

The output of the query shows us how the analyzer treats the text field. In this case, the field is split into three tokens ("james", "bond", and "007"), all lowercase. Because we didn't specify the analyzer in the code, by default, it is assumed to be the standard analyzer. Each of the tokens has a type ALPHANUM for string, NUM for numeric token, and so on. The position of the token is saved too, as you can see from the result shown in the figure 7.3. That brings to the next point: specifying the analyzer explicitly during the `_analyze` test.

EXPLICIT ANALYZER TESTS

In listing 7.1 in the previous section, we didn't mention the analyzer, although it was applied implicitly by the engine. This is the standard analyzer by default. However, we can explicitly enable an analyzer too. The code in the following listing shows how to enable the `simple` analyzer.

Listing 7.2 Explicitly enabling an analyzer

```
GET _analyze
{
  "text": "James Bond 007",
  "analyzer": "simple"
}
```

The `simple` analyzer (we will learn about various types of analyzers in the next section) truncates text when a nonletter character is encountered, so this code produces only two

tokens, “james” and “bond” (“007” was truncated), as opposed to three tokens from the earlier script that used the standard analyzer.

If you are curious, change the analyzer to `english`. The output tokens would then be “jame”, “bond”, and “007”. The notable point is that “james” has been stemmed to “jame” when the `english` analyzer was applied.

CONFIGURING ANALYZERS ON THE FLY

We can also use the `_analyze` API to club together a few filters and a tokenizer (as though we are creating a custom analyzer on the fly). The idea is that we can create a custom analyzer by mixing and matching the given filters and tokenizers. (We are not really building or developing a new analyzer as such.) This on-demand custom analyzer is demonstrated in the following code listing,

Listing 7.3 Creating a custom analyzer using the `_analyze` endpoint

```
GET _analyze
{
  "tokenizer": "path_hierarchy",
  "filter": ["uppercase"],
  "text": "/Volumes/FILES/Dev"
```

The code in this listing uses a `path_hierarchy` with an uppercase filter, and therefore produces three tokens: `"/VOLUMES"`, `"/VOLUMES/FILES"`, and `"/VOLUMES/FILES/DEV"`. The `path_hierarchy` tokenizer splits the text, based on a path separator; hence, you see three tokens telling us about the three folders of hierarchy.

NOTE *The `_analyze` endpoint* The `_analyze` endpoint helps a lot when understanding the way the text has been treated and indexed by the engine, as well as the reasons why a search query may not have produced the desired output. We can use this as the first point to test our text with the expected analyzers before we proceed to put the code into production.

We’ve been talking about the analyzer module in the last few sections. Elasticsearch provides us with a handful of analyzer modules to work with. In the next section, we look at these built-in analyzers in detail.

7.3 Built-in analyzers

Elasticsearch provides over half a dozen out-of-the-box analyzers that we can use in the text analysis phase. These analyzers most likely suffice for the basic cases, but should there be a need to create a custom one, one can do that by instantiating a new analyzer module with the required components that make up that module. The table 7.3 lists the analyzers that Elasticsearch provides us with:

Table 7.3

Analyzer	Description
Standard analyzer	This is the default analyzer that tokenizes input text based on grammar, punctuation, and whitespace. The output tokens are lowercased.
Simple analyzer	A simple tokenizer splits input text on any nonletters such as whitespaces, dashes, numbers, etc. It lowercases the output tokens too.
Stop analyzer	It is a simple analyzer with English stop words enabled by default.
Whitespace analyzer	The whitespace analyzer's job is to tokenize input text based on whitespace delimiters.
Keyword analyzer	The keyword analyzer doesn't mutate the input text. The field's value is stored as is.
Language analyzer	As the name suggests, the <code>language</code> analyzer helps work with human languages. Elasticsearch provides dozens of language analyzers such as English, Spanish, French, Russian, Hindi and so on, to work with different languages.
Pattern analyzer	The <code>pattern</code> analyzer splits the tokens based on a regular expression (regex). By default, all the non word characters help to split the sentence into tokens.
Fingerprint analyzer	The <code>fingerprint</code> analyzer sorts and removes the duplicate tokens to produce a single concatenated token.

The standard analyzer is the default analyzer and is widely used during text analysis. Let's look at the standard analyzer with an example in the next section, and following that we will look at each of the other analyzers in turn.

NOTE Elasticsearch provides a handful of built-in analyzers as well as letting us create a plethora of analyzers by customizing them with a mix and match of filters and tokenizers. It would be too verbose and impractical to go over each of them, but I will present as many examples as possible in this chapter. However, I advise you to refer to the official documentation for specific components and their integration into your application.

7.3.1 Standard analyzer

The standard analyzer is the default analyzer used in Elasticsearch. The standard analyzer's job is to tokenize sentences based on the whitespaces, punctuation, and grammar. Let's suppose we want to build an index with a weird combination of snacks and drinks. Consider the following text that mentions coffee with popcorn:



"Hot cup of ☕ and a 🍿 is a Weird Combo :(!!"

We can index this text into a `weird_combos` index as shown in the following:

```
POST weird_combos/_doc
{
  "text": "Hot cup of ☕ and a 🍿 is a Weird Combo :(!!"
}
```

The text gets tokenized and the list of tokens are spit as shown here in a condensed form:

```
["hot", "cup", "of", "☕", "and", "", "🍿", "is", "a", "weird", "combo"]
```

The tokens are lowercased as you can tell from the output. The smiley at the end as well as the exclamation marks are removed by the standard tokenizer, but the emojis are saved as if they were textual information. This is the default behavior of the standard analyzer, which tokenizes the words based on a whitespace and strips of non letter characters like punctuation. Figure 7.4 shows the workings of the previous input text when passed through the analyzer.

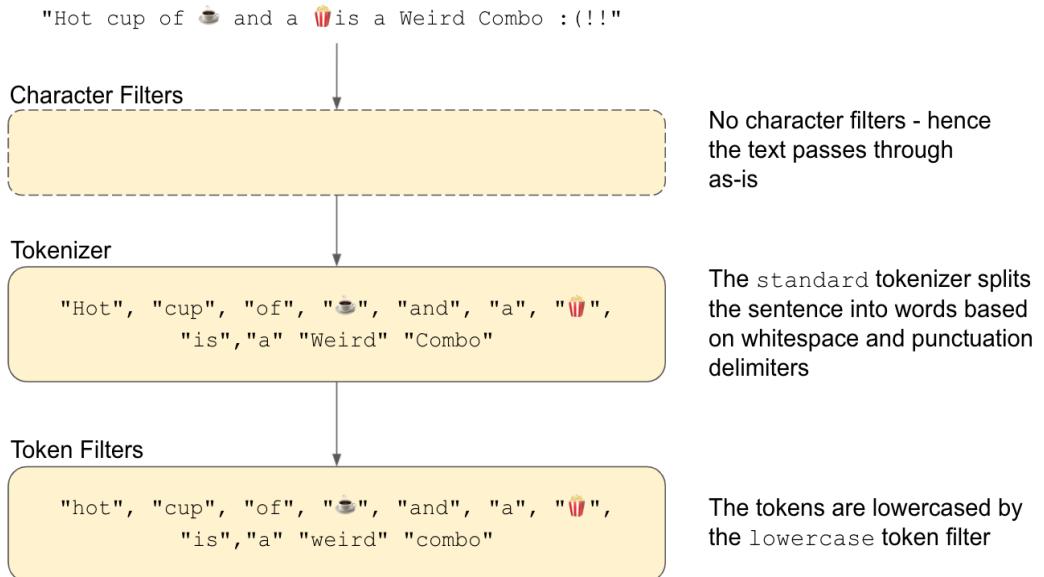


Figure 7.4 The standard (default) analyzer in action

In fact, the following shows how we can use the `_analyze` API to check the output before we index the text. (As discussed earlier, we are not specifying the analyzer because, by default, Elasticsearch uses the standard analyzer if not mentioned explicitly.)

```
GET _analyze
{
  "text": "Hot cup of ☕ and a 🍿 is a Weird Combo :(!!"
}
```

The output of this GET command is shown in the following snippet. (For brevity, other than the first token, the rest of them are condensed.)

```
{
  "tokens" : [
    {
      "token" : "hot", #A
      "start_offset" : 0,
      "end_offset" : 3,
      "type" : "<ALPHANUM>",
      "position" : 0
    },
    { "token" : "cup", ... },
    { "token" : "of", ... }, #B
    { "token" : "☺", ... }, #C
    { "token" : "and", ... },
    { "token" : "a", ... },
    { "token" : "is", ... },
    { "token" : "a", ... },
    { "token" : "weird", ... },
    { "token" : "combo", ... }#D
  ]
}
```

#A The lowercase token filter lowerscases the words.

#B The stop words are not removed as the stop filter is disabled by default.

#C The coffee cup and popcorn are indexed as is.

#D The smiley and exclamation marks are removed by the standard tokenizer.

The output indicates the works of the standard analyzer: the words were split based on whitespace and nonletters (punctuation), which is the mark of the standard tokenizer. The tokens are then passed through the `lowercase` token filter.

NOTE Components of a built-in analyzer As discussed, each of the built-in analyzers comes with a predefined set of components such as character filters, tokenizers and token filters - for example a `fingerprint` analyzer is composed of a standard tokenizer along with a bunch of token filters (`fingerprint`, `lowercase`, `asciifolding` and `stop` token filters) but no character filters. It isn't easy to tell the anatomy of an analyzer unless you have memorized them over time! So, my advice is to check the definition of the analyzer on the documentation page if you need to go over the nitty gritties of an analyzer in detail.

Figure 7.5 shows a condensed output of this command in DevTools. As you can observe, the tokens for “coffee” and “popcorn” are stored as is and the non letter characters such as :(and !! are removed.

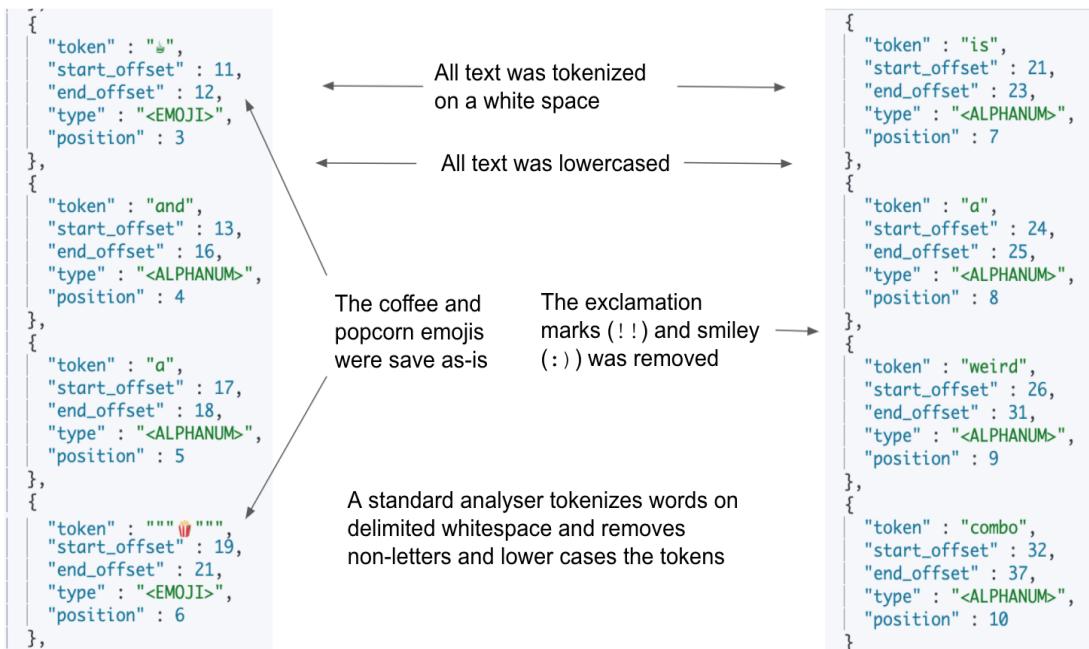


Figure 7.5 The output tokens from a standard analyzer

TESTING THE STANDARD ANALYZER

We can add the specific analyzer during our text analysis testing phase by adding an additional `analyzer` attribute in the code. The following listing demonstrates this.

Listing 7.4 Testing the standard analyzer with an explicit call

```

GET _analyze
{
  "analyzer": "standard",
  "text": "Hot cup of ☕ and a 🍔 is a Weird Combo :(!!"
}

```

#A Specifying the analyzer (though specifying standard is not needed as it is the default analyzer)

You can replace the value of the analyzer to your chosen one if you are testing the text field using a different analyzer, like: `"analyzer": "whitespace"`, for example.

This code produces the same result as that shown in figure 7.5 above. The output indicates that the text was tokenized and lowercased. Figure 7.6 gives us a pictorial representation of the standard analyzer with its internal components and anatomy.

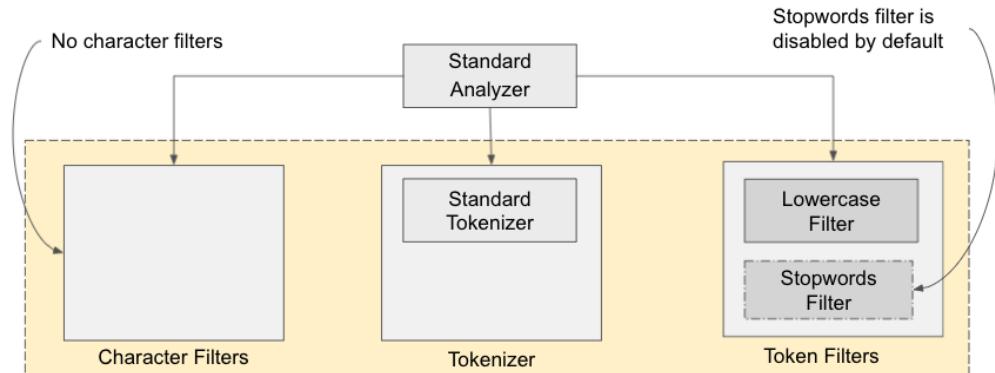


Figure 7.6 Anatomy of a standard analyzer

As the figure 7.6 depicts, the standard analyzer consists of a `standard tokenizer` and two token filters: `lowercase` and `stop` filters. There is no character filter defined on the standard analyzer. To remind ourselves once again, analyzers consist of zero or more character filters, at least one tokenizer, and zero or more token filters.

Although the standard analyzer is clubbed with a stop words token filter, the stop words filter is disabled by default. We can, however, switch it on by configuring its properties, which is the topic of the next section.

CONFIGURING THE STANDARD ANALYZER

Elasticsearch allows us to configure a few parameters such as the stop words filter, stop words path, and maximum token length on the standard analyzer. The way to configure the properties is via the index settings. When we create an index, we can configure the analyzer through the `settings` component:

```
PUT <my_index>
{
  "settings": {
    "analysis": {
      "analyzer": { #B
        "tokenizer": "standard", #A
        "filter": [
          "lowercase",
          "stop"
        ]
      }
    }
  }
}
```

#A The analysis object sets the analyzer.

#B The analyzer that this index is associated with

STOP WORDS CONFIGURATION

Let's take an example of enabling English stop words on the standard analyzer. We could do this by adding a filter during index creation as the following listing shows.

Listing 7.5 Creating an index with a stop words filter-enabled standard analyzer

```
PUT my_index_with_stopwords
{
  "settings": {
    "analysis": {
      "analyzer": {#A
        "standard_with_stopwords":{ #B
          "type":"standard", #C
          "stopwords":"_english_" #D
        }
      }
    }
  }
}
```

#A Sets the analyzer for the index

#B Names the analyzer

#C The standard analyzer type

#D Enables the English stop words filter

As we've noticed earlier, the stop words filter on the standard analyzer was the disabled. Now that we've created the index with a standard analyzer that is configured with the stop words, any text that gets indexed goes through this modified analyzer. To test this, we can invoke the `_analyze` endpoint on the index as demonstrated here in the listing 7.6:

Listing 7.6: Analyzing the text using a standard analyser with stop words

```
POST my_index_with_stopwords/_analyze #A
{
  "text": ["Hot cup of ☕ and a 🍵 is a Weird Combo :(!!"),
  "analyzer": "standard_with_stopwords" #B
}
```

#A Invokes the `_analyze` API on the index

#B The analyzer we had created in listing 7.5

The output of this call shows that the common (English) stop words such as "of", "a", and "is" were removed:

```
["hot", "cup", "☕", "🍵", "weird", "combo"]
```

We can change the stop words for a language of our choice. For example, the code in the following listing shows the index with Hindi stop words and the standard analyzer.

Listing 7.7 A standard analyzer enabling Hindi stop words

```
PUT my_index_with_stopwords_hindi
{
  "settings": {
    "analysis": {
      "analyzer": {
        "standard_with_stopwords_hindi": {
          "type": "standard",
          "stopwords": "_hindi_"
        }
      }
    }
  }
}
```

We can test the text using the aforementioned `standard_with_stopwords_hindi` analyzer:

```
POST my_index_with_stopwords_hindi/_analyze
{
  "text": ["आप कर रहे हो?"],
  "analyzer": "standard_with_stopwords_hindi"
}
```

If you are curious to know what this Hindi sentence represents, its equivalent is “what are you doing?”

The output from the above script is shown here:

```
"tokens" : [{"token" : "रहे", "start_offset" : 3, "end_offset" : 7, "type" : "<ALPHANUM>", "position" : 1}]]
```

The only token that gets output is `रहे` (the second word) because the rest of the words were stop words. (They are common in the Hindi language).

FILE-BASED STOPWORDS

In the previous examples, we provided a clue to the analyzer as to which stop words it should use, English or Hindi, etc., by mentioning the off-the-shelf filter. If our requirement isn't satisfied or catered to by the built-in stop words filters, we can provide the stop words via an explicit file.

Let's say we don't want users to input swear words in our application. We can create a file with all the blacklisted swear words and add the path of the file as the parameter to the standard analyzer. The file must be present relative to the config folder of Elasticsearch's home. The following listing creates the index with an analyzer that accepts a stop word file:

Listing 7.8 Creating an index with custom stop words specified via a file

```
PUT index_with_swear_stopwords
{
  "settings": {
    "analysis": {
      "analyzer": {
        "swearwords_analyzer":{#A
          "type":"standard", #B
          "stopwords_path":"swearwords.txt" #C
        }
      }
    }
  }
}
```

#A Names the analyzer so it can be referenced when indexing/testing
#B Uses the standard analyzer
#C The file must be present in the config folder (relative location)

As discussed previously, the `stopwords_path` attribute looks for a file (`swearwords.txt` in this case) in a directory inside the Elasticsearch's config folder. If it doesn't exist, it creates a text file called `swearwords.txt` and adds the blacklisted words. The following listing demonstrates this approach. In the listing, notice that the blacklisted words are created in a new line.

Listing 7.9 Adding a blacklisted swear words text file

```
file:swearwords.txt
damn
bugger
bloody hell
what the hell
sucks
```

Once the file is created and the index is developed as the following listing shows, we are ready to put the analyzer with the custom-defined swear words to use:

Listing 7.10 Putting the custom swear words to work

```
POST index_with_swear_stopwords/_analyze
{
  "text": ["Damn, that sucks!"],
  "analyzer": "swearwords_analyzer"
}
```

This code should stop the first and last words going through the indexing process because those two words were in our swear words black list. The next attribute that we can configure is the length of tokens: how long we need them as output. This is discussed in the next section.

CONFIGURING TOKEN LENGTH

We can also configure the maximum token length; in which case, the token is split based on the length asked for. For example, the listing 7.10 creates an index with a standard analyzer.

The analyzer is configured to have a maximum token length of 7 characters. If we provide a word that is 13 characters long, the word would be split into 7 and 6 characters (for example, Elasticsearch would become "Elastic", "search").

Listing 7.11 Creating an analyzer with a custom token length

```
PUT my_index_with_max_token_length
{
  "settings": {
    "analysis": {
      "analyzer": {
        "standard_max_token_length": {
          "type": "standard",
          "max_token_length": 7
        }
      }
    }
  }
}
```

So far we've worked with the standard analyzer. The second one in the list of built-in analyzers is the *simple analyzer*. The simple analyzer has the single purpose of splitting the text on nonletters. Let's discuss the details of using a simple analyzer in the next section.

7.3.2 Simple analyzer

While the standard analyzer breaks down the text into tokens when encountered with whitespaces or punctuation, the simple analyzer tokenizes the sentences at the occurrence of a nonletter like a number, space, apostrophe, or hyphen. It does this by using a lowercase tokenizer, which is not associated with any character or token filters. This is represented pictorially in figure 7.7.

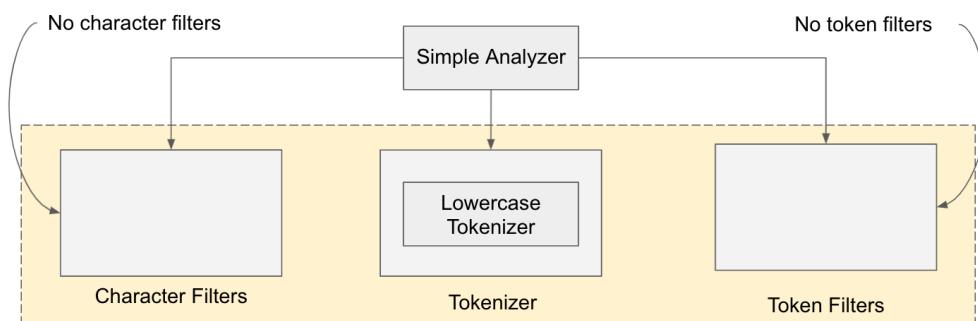


Figure 7.7 Anatomy of a simple analyzer

Let's consider an example of indexing the text "Lukša's K8s in Action" as the script in the following listing shows.

List 7.12: Analyzing text using a simple analyzer

```
POST _analyze
{
  "text": ["Lukša's K8s in Action"],
  "analyzer": "simple"
}
```

This results in

```
["lukša", "s", "k", "s", "in", "action"]
```

The tokens were split when an apostrophe ("Lukša's" becomes "Lukša" and "s") or numbers ("K8s" becomes "k" and "s") were encountered and the resulting tokens were lowercased.

There is not much configuration that a simple analyzer can do, but if we want to add a filter (character or token), the easiest way to do this is to create a custom analyzer with the required filters and the lowercase tokenizer (the simple analyzer has a lone lowercase tokenizer). We see this later in the section on custom analyzers.

7.3.3 Whitespace analyzer

As the name suggests, the whitespace analyzer splits the text into tokens when it encounters whitespaces. There are no character or token filters on this analyzer except a whitespace tokenizer as figure 7.8 shows.

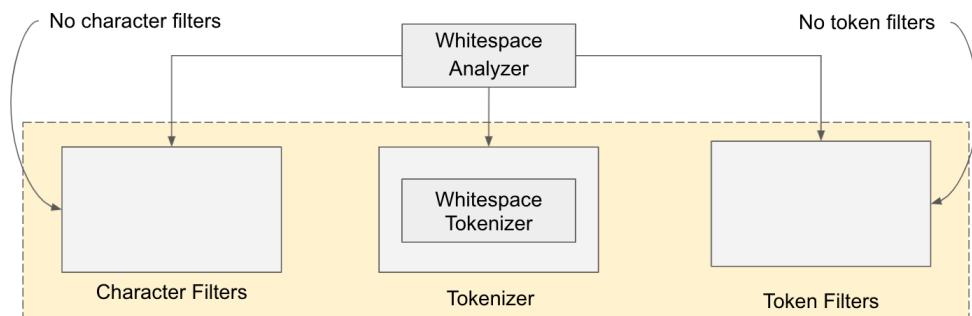


Figure 7.8 Anatomy of the whitespace analyzer

The following listing shows the script for the whitespace analyzer. We can execute the script in the listing to get the desired output as shown:

Listing 7.13 Whitespace tokenizer in action

```
POST _analyze
{
  "text": "Peter_Piper picked a peck of PICKLED-peppers!!",
  "analyzer": "whitespace"
}
```

If we test this script, we'll get this set of tokens:

```
["Peter_Piper", "picked", "a", "peck", "of", "PICKLED-peppers!!"]
```

Two points to note from the result: the text was tokenized only on a whitespace, it was not tokenized on dashes, underscores, and punctuation. The second point is that the case is preserved. The capitalization of the characters and words were kept intact.

As mentioned earlier, similar to the simple analyzer, the whitespace tokenizer is not exposed with configurable parameters. If we need to modify the behavior of the analyzer, we may need to go through the route of creating a modified custom whitespace analyzer. We will run through custom analyzers shortly.

7.3.4 Keyword analyzer

As the name suggests, the keyword analyzer stores the text as is without any modifications and tokenization. That is, the analyzer does not tokenize the text, nor does it undergo any further analysis via filters or tokenizers. Instead, it is stored as a string representing a `keyword` type. As figure 7.9 depicts, the keyword analyzer is composed of just a noop (no-operation) tokenizer and no character or token filters.

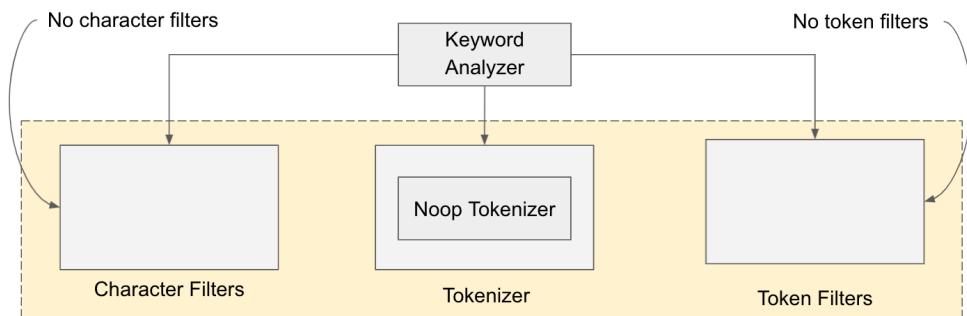


Figure 7.9 Anatomy of the keyword analyzer

The text that gets passed through the analyzer is converted and stored as a keyword. For example, if we pass in "Elasticsearch in Action" through the keyword analyzer, the whole text string is stored as is, unlike earlier instances where the text was split into tokens. The code in the following listing demonstrates this.

Listing 7.14 Keyword analyzer usage

```
POST _analyze
{
  "text": "Elasticsearch in Action",
  "analyzer": "keyword"
}
The output of this script is shown in the following snippet:
"tokens" : [{  
  "token" : "Elasticsearch in Action",  
  "start_offset" : 0,  
  "end_offset" : 23,  
  "type" : "word",  
  "position" : 0
}]
```

As you can see, there's only one token that was produced as a result of processing the text via the keyword analyzer. There's no lowercasing. However, there will be a change in the way we search if we use the keyword analyzer for processing the text. Searching a single word will not match the text string. We must provide an exact match. We must provide the exact group of words as in the original sentence; in this case, "Elasticsearch in Action".

7.3.5 Fingerprint analyzer

The fingerprint analyzer removes duplicate words, extended characters, and sorts the words alphabetically to create a single token. It consists of a standard tokenizer along with four token filters: fingerprint, lowercase, stop words, and ASCII folding filters. Figure 7.10 shows this pictorially.

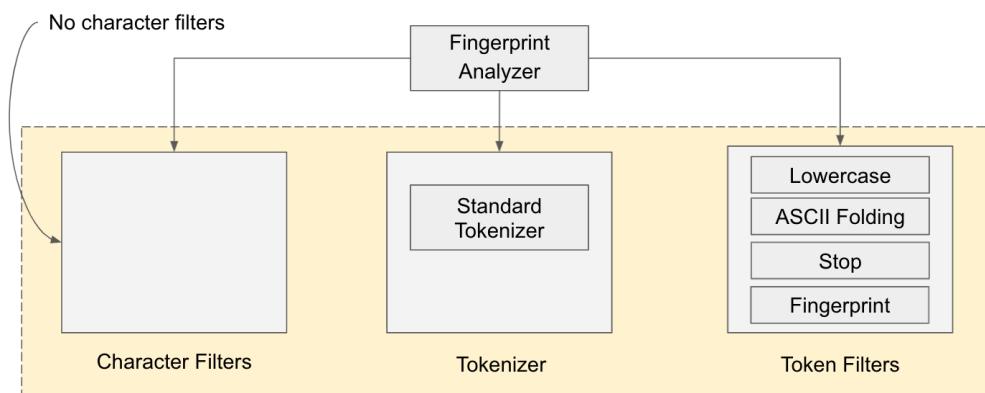


Figure 7.10 Anatomy of the fingerprint analyzer

For example, let's analyze the following text (a definition of a South Indian dish called *dosa*). The following listing includes a description of this fare.

Listing 7.15 Analyzing text using a fingerprint analyzer

```
POST _analyze
{
  "text": "A dosa is a thin pancake or crepe originating from South India. It is made from
          a fermented batter consisting of lentils and rice.",
  "analyzer": "fingerprint"
}
```

The output of the text processed by a fingerprint analyzer is shown here:

```
"tokens" : [{  
  "token" : "a and batter consisting crepe dosa fermented from india is it lentils made of  
          or originating pancake rice south thin",  
  "start_offset" : 0,  
  "end_offset" : 130,  
  "type" : "fingerprint",  
  "position" : 0  
}]
```

When you look closely at the response, you will find that the output is made up of only one token. The words are lowercased and sorted, duplicate words ("a", "of", "from") are removed as well before turning the set of words into a single token.

7.3.6 Pattern analyzer

Sometimes, we may want to tokenize and analyze text based on a certain pattern (for example, removing the first n number of a phone numbers or removing a dash for every four digits from a card number and so on). Elasticsearch provides a pattern analyzer just for that purpose.

The default pattern analyzer works on splitting the sentences into tokens based on nonword characters. This pattern is represented as `\w+` internally. As figure 7.11 demonstrates, the pattern tokenizer, along with lowercase and stop filters, makes the pattern analyzer:

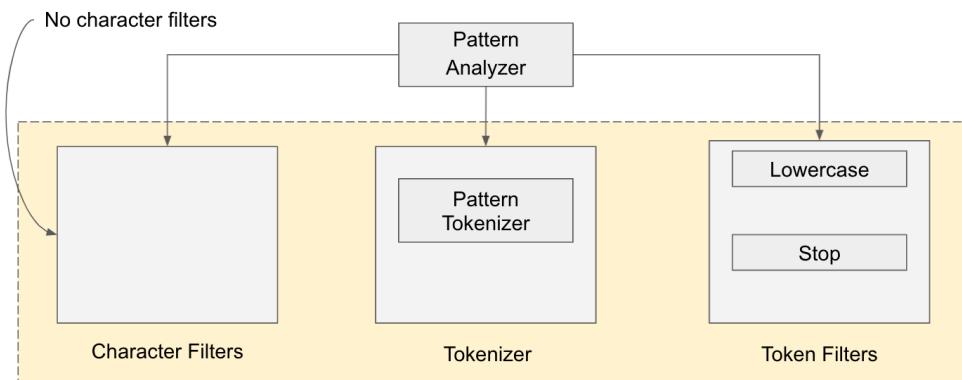


Figure 7.11 Anatomy of the pattern analyzer

As the default (standard) analyzer only works on nonletter delimiters, for any other patterns we need to configure the analyzer by providing the required patterns. Patterns are regular expressions provided as a string when configuring the analyzer. The patterns use Java regular expressions. To learn more about Java regular expressions follow this link:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html>

Let's just say we have an e-commerce payments authorizing application and are actually receiving payment authorization requests from various parties. A 16-digit long card number is provided in the format 1234-5678-9000-0000. We want to tokenize this card data on a dash (-) and extract the four tokens individually. We can do so by creating a pattern that splits the field into tokens based on the dash delimiter.

To configure the pattern analyzer, we must create an index by setting `pattern_analyzer` as the analyzer in the `settings` object. The following listing shows the configuration in action.

Listing 7.16 A pattern that delimits tokens based on a dash

```
PUT index_with_dash_pattern_analyzer #A
{
  "settings": {
    "analysis": {
      "analyzer": {
        "pattern_analyzer": { #B
          "type": "pattern", #C
          "pattern": "[ - ]", #D
          "lowercase": true #E
        }
      }
    }
  }
}
```

```
#A Creates an index with analyzer settings
#B In settings, defines the analyzer in the analysis object
#C Provides the type of analyzer as pattern
#D The regex representing the dash
#E Attaches a lowercase token filter
```

In the code, we create an index with some pattern analyzer settings. The `pattern` attribute indicates the regex, which follows Java's regular expression syntax. In this case, we set the dash as our delimiter, so the text is tokenized when it encounters that character. Now that we have the index created, let's put this analyzer into action as the following listing shows.

Listing 7.17 Testing the pattern analyzer

```
POST index_with_dash_pattern_analyzer/_analyze
{
  "text": "1234-5678-9000-0000",
  "analyzer": "pattern_analyzer"
}
```

The output of this command produces four tokens: `["1234", "5678", "9000", "0000"]`. The text can be tokenized based on a plethora of patterns. I suggest that you experiment with regex patterns to get the full benefit from the pattern analyzer.

7.3.7 Language analyzers

Elasticsearch provides a long list of language analyzers that are suitable when working with most languages. Moreover, you can configure these out-of-the-box language analyzers to add a stop words filter so you don't index unnecessary (or common) words of that language. The list of analyzers are Arabic, Armenian, Basque, Bengali, Bulgarian, Catalan, Czech, Dutch, English, Finnish, French, Galician, German, Hindi, Hungarian, Indonesian, Irish, Italian, Latvian, Lithuanian, Norwegian, Portuguese, Romanian, Russian, Sorani, Spanish, Swedish, and Turkish. The following code listing demonstrates three (English, German, and Hindi) language analyzers in action.

Listing 7.18 English, German, and Hindi language analyzers

```
POST _analyze
{
  "text": "She sells sea shells",
  "analyzer": "english"
}

# German Language Analyzer
POST _analyze
{
  "text": "Guten Morgen",
  "analyzer": "german"
}

# English Language Analyzer
POST _analyze
{
  "text": "The quick brown fox jumps over the lazy dog",
  "analyzer": "hindi"
}
```

We can configure the language analyzers with a few additional parameters to provide our own list of stop words or to ask the analyzers to exclude the stemming operation. For example, there are a handful of words that are categorized as stop words by the stop token filter that is used by the English analyzer. We can override this list as per our convenience. Say we only want to override “a”, “an”, “the”, “and”, and “but”. In this case, we can configure our stop words as the following listing shows.

Listing 7.19 Creating an index with custom stop words on an English analyzer

```
PUT index_with_custom_english_analyzer #A
{
  "settings": {
    "analysis": {
      "analyzer": {
        "index_with_custom_english_analyzer":{ #B
          "type":"english", #C
          "stopwords":["a","an","is","and","for"] #D
        }
      }
    }
  }
}
```

#A Creates an index with analyzer settings

#B Provides a custom name

#C The type of the analyzer here is english.

#D Provides our own set of stop words

As the code indicates, we created an index with a custom English analyzer and a set of user-defined stop words. When we test (see listing 7.20) a piece of text with this analyzer, we can see that the stop words were honored.

Listing 7.20 Testing the custom stop words for the English analyzer

```
POST index_with_custom_english_analyzer/_analyze
{
  "text":"A dog is for a life",
  "analyzer":"index_with_custom_english_analyzer"
}
```

This code outputs just two tokens: “dog”, and “life”. The words “a”, “is”, and “for” are removed as they match the stop words that we specified earlier.

Language analyzers have another feature that they are always eager to implement: stemming. *Stemming* is a mechanism to reduce the words to their root form. For example, any form of the word “author” (“authors”, “authoring”, “authored”, etc.) is reduced to the single word “author”. The following listing shows this behavior.

Listing 7.21 Reducing all forms of “author” to the author keyword

```
POST index_with_custom_english_analyzer/_analyze
{
  "text":"author authors authoring authored",
  "analyzer":"english"
}
```

This code produces four tokens (tokenized, based on whitespace tokenizer) for which all are “author” as the root word for any form of “author” is “author”! But sometimes the stemming might go a bit too far. If you add “authorization” or “authority” to the list of words in the previous listing, unfortunately the words get stemmed and indexed as “author”! Obviously, you will not be able to find pertinent answers when you are searching for “authority” or

“authorization” because those words did not make it to the inverted index in the first place due to stemming.

All is not lost. We can configure our English analyzer, asking it to ignore certain words to stem such as “authorization” and “authority” in this case. These do not need to get through the analyzer. In this case, we can bring the `stem_exclusion` attribute to configure the words that need to be excluded from the stemming. The code in the listing 7.20 does this exactly, by creating an index with custom settings and passing the arguments to the `stem_exclusion` parameter.

Listing 7.22 Creating an index with custom stem-exclusion words

```
PUT index_with_stem_exclusion_english_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "stem_exclusion_english_analyzer": {
          "type": "english",
          "stem_exclusion": ["authority", "authorization"]
        }
      }
    }
  }
}
```

Once you’ve created the index with these settings, the next step is to test the indexing request. The following listing uses the English analyzer to test a piece of text.

Listing 7.23 Stem exclusion in action

```
POST index_with_stem_exclusion_english_analyzer/_analyze
{
  "text": "No one can challenge my authority without my authorization",
  "analyzer": "stem_exclusion_english_analyzer"
}
```

The tokens that were spit as a consequence of the code in listing 7.23 consist of our two words: “authority” and “authorization”. This indicates that they both were untouched!

We can further customize the language analyzers if we want. We will learn about customizing analyzers in a moment or two.

While most analyzers do what we want in most cases, at times, however, we may need to implement text analysis for a few additional requirements. For example, we may want to remove some special characters like HTML tags from the incoming text or to avoid stop words. The job of removing HTML tags is taken care of by the `html_strip` character filter and, unfortunately, not all analyzers have them.

In such cases, we can customize the respective analyzer by configuring the required functionality for our needs. We can add a new character filter like `html_strip` and perhaps enable the stop token filter too. Let’s discuss configuring the standard analyzer to suit advanced requirements.

7.4 Custom analyzers

Elasticsearch provides much flexibility when it comes to analyzers: if off-the-shelf analyzers won't cut it for you, you can create your own custom analyzers. These custom analyzers can be a mix-and-match of existing components from a large stash of Elasticsearch's component library.

The practice is to define a custom analyzer in `settings` when creating an index with the required filters and a tokenizer. We can provide any number of character and token filters but only one tokenizer as figure 7.12 depicts.

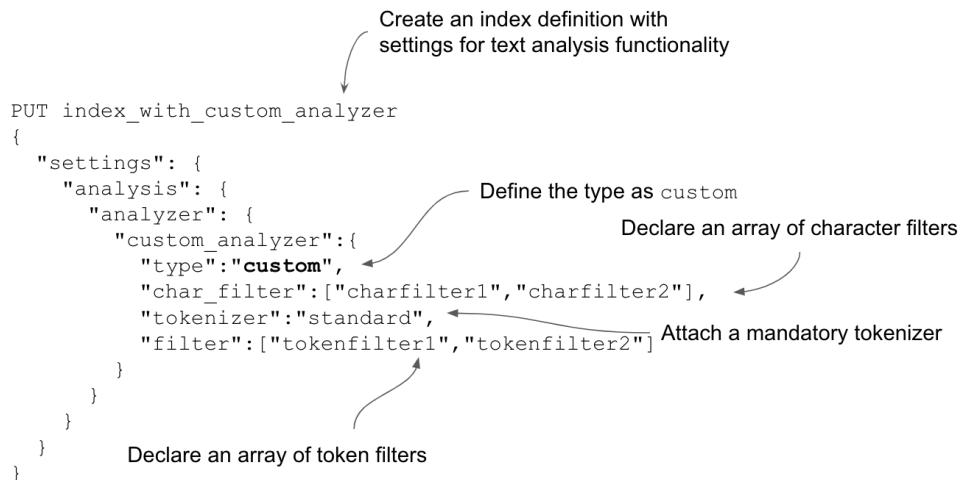


Figure 7.12 Anatomy of a custom analyzer

As the figure shows, we define a custom analyzer on an index by setting the type to `custom`. Our custom analyzer is developed with an array of character filters represented by the `char_filter` object and another array of token filters represented by the `filter` attribute.

NOTE Elasticsearch folks should've named the filter object as `token_filter` instead of `filter` because just the `char_filter` represents the character filter. And one more thing, plural `char_filters` and `token_filters` would've made sense, in my opinion, as they expect an array of stringified filters!

We are expected to provide the tokenizer from our list of off-the-shelf tokenizers with the custom configuration. Let's look at an example of creating a custom analyzer. Listing 7.22 demonstrates the script for developing a custom analyzer. It has

- A character filter (`html_strip`) that strips some HTML characters from the input field.
- A standard tokenizer that tokenizes the field based on whitespace and punctuation.
- A token filter for uppercasing the words.

Listing 7.24 Creating a custom analyzer with filters and a tokenizer

```
PUT index_with_custom_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_analyzer":{#A
          "type":"custom", #B
          "char_filter":["html_strip"],#C
          "tokenizer":"standard",#D
          "filter":["uppercase"]#E
        }
      }
    }
  }
}
```

#A Specifies the custom analyzer
#B The type must be custom to let Elasticsearch know about our custom analyzer
#C An array of character filters
#D Declares a single tokenizer (standard, in this case)
#E A token filter to uppercase the incoming tokens

We can test the analyzer using the following code snippet:

```
POST index_with_custom_analyzer/_analyze
{
  "text": "<H1>HELLO, WoRLD</H1>",
  "analyzer": "custom_analyzer"
}
```

This program produces two tokens: ["HELLO", "WORLD"], indicating that our `html_strip` filter removed the `H1` HTML tags before letting the standard tokenizer split the field into two tokens based on a whitespace delimiter. Finally, the tokens were uppercased as they passed through the `uppercase` token filter.

While the customization helps satisfy a range of requirements, there's even more advanced requirements that can be achieved. Let's discuss that in the next section.

7.4.1 Advanced customization

While default configurations of the analyzer components work most of the time, sometimes we may need to create analyzers with nondefault configurations of the components that make up the analyzer. Say we want to use a mapping character filter that would map characters like & to *and* and < and > to *less than* and *greater than*, respectively, and so on.

Let's suppose our requirement is to develop a custom analyzer that parses text for Greek letters and produces a list of Greek letters as a result. The following listing demonstrates the code to create an index with analysis settings.

Listing 7.25 Parsing text to extract Greek letters with a custom analyzer

```
PUT index_with_parse_greek_letters_custom_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "greek_letter_custom_analyzer":{ #A
          "type":"custom",
          "char_filter":["greek_symbol_mapper"], #B
          "tokenizer":"standard", #C
          "filter":["lowercase", "greek_keep_words"] #D
        }
      },
      "char_filter": { #E
        "greek_symbol_mapper":{
          "type":"mapping",
          "mappings": [ #F
            "α => alpha",
            "β => Beta",
            "γ => Gamma"
          ]
        }
      },
      "filter": {
        "greek_keep_words":{ #G
          "type":"keep",
          "keep_words":["alpha", "beta", "gamma"] #H
        }
      }
    }
  }
}
```

#A Creates a custom Greek-letter parser analyzer

#B The custom analyzer is made of a custom char_filter (see #E).

#C A bog standard tokenizer tokenizes the text.

#D Supplies two token filters. The greek_keep_words is defined in #G.

#E Defines the Greek letters and maps those to English words

#F The actual mappings: a list of symbol to value

#G We don't want to index all the field values, only the words that match the keep words.

#H Keep words; all other words are discarded.

The code in the listing is a bit of a handful, however, understanding it is simple and easy. In the first part, where we define a custom analyzer, we provide a list of filters (both character and token filters if needed) and a tokenizer. You can imagine this as an entry point to the analyzer's definition.

The second part of the code then defines the filters that were declared earlier. For example, the `greek_symbol_mapper`, which is redeclared under a new `char_filter` section, uses the `mapping` type as the filter's type with a set of mappings. The same goes for the filter block, which defines the `keep_words` filter. The `keep_words` filter removes any words that aren't present in the list of `keep_words`.

Once you have the script ready, we can execute the test sample for analysis. In the following listing, we have a sentence that's expected to be passed through the test analysis phase.

Listing 7.26 Parsing the greek letters from a normal text

```
POST index_with_parse_greek_letters_custom_analyzer/_analyze
{
  "text": "\u03b1 and \u03b2 are roots of a quadratic equation. \u03b3 isn't",
  "analyzer": "greek_letter_custom_analyzer"
}
```

The Greek letters (α , β and γ , for example) are processed by the custom analyzer (`greek_letter_custom_analyzer`), and it outputs the following:

```
"alpha", "beta", "gamma"
```

The rest of the words like `roots` and `quadratic equation` were removed.

So far we have gone through the analyzers in detail, including the built-in and custom ones. We can configure the analyzers not just at the field level that we had seen so far but other places such as at an index level too. We can also specify a different analyzer for search queries if the requirement dictates. We discuss all these points in the next section.

7.5 Specifying analyzers

Analyzers can be specified at a few levels: index, field and query level. Declaring the analyzers at index level provides an index-wide default catch-all analyzer for all text fields. However, if further customization is required on a field-level, one could enable a different analyzer at a field level too. In addition to this, we can also provide a different analyzer as opposed to the index time analyzer while searching. Let's look at these options one by one in this section.

7.5.1 Analyzers for indexing

At times we may have a requirement to set different fields with different analyzers - for example, a name field could have been associated with a simple analyzer while the credit card number field with a pattern analyzer. Fortunately, Elasticsearch let's us set different analyzers on individual fields as required; Similarly, we can also set a default analyzer per index so that any fields that were not associated with a specific analyzer explicitly during the mapping process will inherit the index level analyzer. Let's check these two mechanisms in this section.

FIELD LEVEL ANALYZER

We can specify required analyzers at a field level while creating a mapping definition of an index. For example, the code below (listing 7.27) shows how we can leverage this during the index creation:

Listing 7.27: Setting analyzers at field level during index creation

```
PUT authors_with_field_level_analyzers
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text" #A Standard analyzer is being used here
      },
      "about": {
        "type": "text",
        "analyzer": "english" #B Set explicitly with an english analyzer
      },
      "description": {
        "type": "text",
        "fields": {
          "my": {
            "type": "text",
            "analyzer": "fingerprint" #C Fingerprint analyzer on a multi-field
          }
        }
      }
    }
  }
}
```

As the code shows, the `about` and `description` fields were specified with different analyzers except the `name` field which is implicitly inheriting the `standard` analyzer.

INDEX LEVEL ANALYZER

We can also set a default analyzer of our choice at the index level, the following code listing (7.28) demonstrate this:

Listing 7.28: Creating an index with a default analyzer

```
PUT authors_with_default_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "default": { #A Setting this property sets index's default analyzer
          "type": "keyword"
        }
      }
    }
  }
}
```

In the code listing 7.28, we are in effect replacing the `standard` analyzer which comes as default to a `keyword` analyzer. You can test the analyzer by invoking the `_analyse` endpoint on the index as the code listing 7.29 shows:

Listing 7.29: Testing the default analyzer

```
PUT authors_with_default_analyzer/_analyze
{
  "name": "John Doe"
}
```

The first code snippet will output a single token "John Doe" with no lowercasing or tokenizing - which indicates it's been analyzed by our keyword analyzer. You can try the same code using a standard analyzer and you'll notice the difference.

Setting of analyzers at an index level or field level works during the indexing process. We can, however, use a different analyzer during the querying process - let's see why and how in the next section.

7.5.2 Analyzers for searching

Elasticsearch lets us specify a different analyzer during query time than using the same one during indexing. It also allows us to set a default analyzer across the index - this can be set during the index creation. Let's see these two methods in this section as well as some rules that Elasticsearch follows when picking up an analyzer defined at various levels.

ANALYZER IN A QUERY

We didn't run through the search part yet so don't worry if the following code (listing 7.30) baffles you a bit (we discuss search in the next few chapters):

Listing 7.30: Setting the analyzer alongside a search query

```
GET authors_index_for_search_analyzer/_search
{
  "query": {
    "match": { #A
      "author_name": {
        "query": "M Konda",
        "analyzer": "simple" #B
      }
    }
  }
}
```

#A Query to search all the authors with the given criteria

#B The analyzer is specified explicitly, most likely different to the one that field was indexed with

As shown in the code above, we are specifying the analyzer explicitly (most likely the `author_name` field would've been indexed using a different type of analyzer!) while searching for an author.

SETTING THE ANALYZER AT A FIELD LEVEL

The second mechanism to set the search specific analyzer is at the field level. Just as we set an analyzer on a field for indexing purposes, we can add an additional property called the `search_analyzer` on a field to specify the search analyzer. The code below (listing 7.31) demonstrates this method:

Listing 7.31: Setting a search analyzer at a field level

```
PUT authors_index_with_both_analyzers_field_level
{
  "mappings": {
    "properties": {
      "author_name": {
        "type": "text",
        "analyzer": "stop",
        "search_analyzer": "simple"
      }
    }
  }
}
```

As the code above shows, the `author_name` is set with a `stop` analyzer for indexing while a `simple` analyzer for search time.

DEFAULT ANALYZER AT INDEX LEVEL

We can also set a default analyzer for search queries too just as we did for indexing time by setting the required analyzer on the index at index creation time. The following code listing (7.32) demonstrates the setting:

Listing 7.32: Setting a default search analyzer and a default analyzer for indexing

```
PUT authors_index_with_default_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "default_search": { #A
          "type": "simple"
        },
        "default": { #B
          "type": "standard"
        }
      }
    }
  }
}
```

#A Setting the default search analyzer per index by using the `default_search` property
#B The default analyser for the index

In the above code listing, we also have set the default analyzer for indexing too in addition to the search at the same time. You may be wondering if we can set a search analyzer at a field level during the indexing rather than at runtime during the query? The code below (listing 7.33) demonstrates exactly that - setting different analyzers for indexing and searching at a field level during the creation of an index:

Listing 7.33: Specifying index and search analyzers during the index creation

```
PUT authors_index_with_both_analyzers_field_level
{
  "mappings": {
    "properties": {
      "author_name": {
        "type": "text",
        "analyzer": "standard",
        "search_analyzer": "simple"
      }
    }
  }
}
```

As you can see from the above code, the author_name is going to use a standard analyzer for indexing while a simple analyzer during search.

ORDER OF PRECEDENCE

There's a precedence of order in which the analyzer is picked up by the engine when it finds the analyzer at various levels. The following is the order of precedence in which the engine picks up the right analyzer:

- An analyzer defined at a query level has the highest precedence.
- An analyzer defined by setting `search_analyzer` property on a field when defining the index mappings.
- An analyzer defined at the index level.
- If neither of the above were not set, the Elasticsearch engine picks up the indexing analyzer set on a field or an index.

Now that we understand the built-in analyzers and the mechanism to create our own custom analyzers, it is time to look into the individual components of the analyzers in detail. In the next section, we will go over the three components that make up the analyzer: tokenizers and character and token filters. Let's start with character filters.

7.6 Character filters

When a user searches for answers, the expectation is that they won't search with punctuation or special characters. For example, there is a high chance a user may search for "cant find my keys" (without punctuation) rather than "can't find my keys !!!". Similarly, the user is not expected to search the string "<h1>Where is my cheese?</h1>" (with the HTML tags). We don't even expect the user to search using XML tags like <operation>callMe</operation>. The search criteria doesn't need to be polluted with unneeded characters. And, sometimes, we don't expect users to search using symbols: a instead of alpha or β in place of beta, and so on.

Based on these assumptions, we can analyze and clean the incoming text using character filters. Character filters help purge the unwanted characters from the input stream. Though they are optional, if they are used, they form the first component in the analyzer module.

An analyzer can consist of a zero or more character filters. The character filter carries out the following specific functions:

- *Removes the unwanted characters from an input stream.* For example, if the incoming text has HTML markup like "<h1>Where is my cheese?</h1>", the requirement is to get the <h1> tags dropped.
- *Adds to or replaces additional characters in the existing stream.* If the input field has a set of 0's and 1's , then perhaps we may want to replace them with "false" and "true", respectively. If the input stream has the character β , we might map it to the word "beta" and index the field.

Elasticsearch provides three character filters, which we will see in action in the next sections.

7.6.1 Types of character filters

There are three character filters that we use to construct an analyzer: HTML strip, mapping, and pattern filters. We saw these in action in the earlier sections, so in this section we will go over the semantics briefly.

HTML STRIP (HMTL_STRIP) FILTER

As the name suggests, this filter strips the unwanted HTML tags from the input fields. For example, when the input field with a value of <h1>Where is my cheese?</h1> is processed by the HTML strip (html_strip) character filters, the <h1> tags gets purged, leaving "Where is my cheese?". Note that this filter does not touch the punctuation or casing of the words. We can test the html_strip character using the _analyze API as the following listing shows:

Listing 7.34: A html_strip character filter in action

```
POST _analyze
{
  "text": "<h1>Where is my cheese?</h1>",
  "tokenizer": "standard",
  "char_filter": ["html_strip"]
}
```

The character filter simply strips the <h1> tags from the input field to produce "Where", "is", "my", "Cheese" tokens instead. However, there might be a requirement to avoid parsing the input field for certain HTML tags; say, for example, the business requirement could be to strip the <h1> tags from the sentences but to preserve the preformatted (<pre>) tags. For example,

```
<h1>Where is my cheese?</h1>
<pre>We are human beings that lookout for cheese constantly!</pre>
```

Fortunately there is a way out. We can configure the html_strip filter to add an additional escaped_tags array with the list of tags that needs to be unparsed. Let's see it in action. The first step is to create an index with the required custom analyzer as the following listing shows.

Listing 7.35 Custom analyzer with an additional filter configuration

```
PUT index_with_html_strip_filter
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_html_strip_filter_analyzer": {
          "tokenizer": "keyword",
          "char_filter": ["my_html_strip_filter"] #A
        }
      },
      "char_filter": {
        "my_html_strip_filter": {
          "type": "html_strip",
          "escaped_tags": ["h1"] #B
        }
      }
    }
  }
}
```

#A Declares a custom character filter

#B The escaped_tags attribute ignores parsing h1 tags present in the input field.

We just created an index with a custom analyzer made of a html_strip character filter. The notable difference is that the html_strip character is extended in this example to use the escaped_tags option, so the field consisting of <h1> tags will be untouched. To test this, run the code in the following listing, which proves this point.

Listing 7.36 Testing the custom analyzer

```
POST index_with_html_strip_filter/_analyze
{
  "text": "<h1>Hello,</h1> <h2>World!</h2>",
  "analyzer": "custom_html_strip_filter_analyzer"
}
```

This code leaves the word with the <h1> tag as is, stripping the <h2> tag. It results in this output:<h1>Hello,</h1> World!".

MAPPING CHARACTER FILTER

The mapping character filter's sole job is to match a key and replace it with a value. As we saw in our earlier example of conversion of Greek letters to English words, the mapping filter parsed the symbols and replaced them with words: α as alpha, β as beta, and so on.

We can test the mapping character filter. For example, the UK in the following listing will be replaced with the United Kingdom when parsed with the mapping filter.

Listing 7.37: Mapping character filter in action

```
POST _analyze
{
  "text": "I am from UK",
  "char_filter": [
    {
      "type": "mapping",
      "mappings": [
        "UK => United Kingdom"
      ]
    }
  ]
}
```

If we want to create a custom analyzer with a configured mapping character filter, we should follow the same process for creating an index with analyzer settings and the required filters. This code example shows the procedure for customizing a keyword analyzer to attach a character mapping filter:

Listing 7.38: Customizing the keyword analyzer with a custom character filter

```
PUT index_with_mapping_char_filter
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_social_abbreviations_analyzer": { #A
          "tokenizer": "keyword",
          "char_filter": [
            "my_social_abbreviations" #B
          ]
        }
      },
      "char_filter": {
        "my_social_abbreviations": { #C
          "type": "mapping", #D
          "mappings": [ #E
            "LOL => laughing out loud",
            "BRB => be right back",
            "OMG => oh my god"
          ]
        }
      }
    }
  }
}
```

#A A custom analyzer with a mapping character filter

#B Declares a character filter

#C Expands the defined character filter with mappings

#D Specifies the name of the filter; mapping filter in this case

#E Provides a set of mappings in the mappings object as name-value pairs

We've now created an index with custom analyzer settings, providing a bunch of mappings in the character filter. Now that we have the index with custom analyzer, we can follow the same process of testing it using the `_analyze` API, shown in the listing 7.32 below:

Listing 7.39: Testing the custom analyzer

```
POST index_with_mapping_char_filter/_analyze
{
  "text": "LOL",
  "analyzer": "my_social_abbreviations_analyzer"
}
```

The text results in "token" : "laughing out loud", which indicates that "LOL" was replaced with the full form, "laughing out loud".

MAPPINGS VIA A FILE

We can also provide a file with mappings in it, rather than specifying them in the definition. Listing 7.40 demonstrates a character filter with mappings loaded from an external file, secret_organizations.txt. The file must be present in Elasticsearch's config directory (<INSTALL_DIR/elasticsearch/config) or input with an absolute path where it is located.

Listing 7.40 Loading external mappings via a file

```
POST _analyze
{
  "text": "FBI and CIA are USA's security organizations",
  "char_filter": [
    {
      "type": "mapping",
      "mappings_path": "secret_organizations.txt"
    }
  ]
}
```

PATTERN REPLACE CHARACTER FILTER

The pattern_replace character filter, as the name suggests, replaces the characters with a new character when the field matches with a regular expression (regex). Following the same code pattern as the one from the mapping filter, let's create an index with an analyzer associated with a pattern-replace character filter. The code in the following listing (7.41) does exactly that.

Listing 7.41: Pattern replace character filter example

```
PUT index_with_pattern_replace_filter
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_pattern_replace_analyzer":{ #A
          "tokenizer":"keyword",
          "char_filter":["pattern_replace_filter"] #B
        }
      },
      "char_filter": {
        "pattern_replace_filter":{ #C
          "type":"pattern_replace", #D
          "pattern":"_", #E
          "replacement":"-" #F
        }
      }
    }
  }
}
```

#A A custom analyzer with the pattern_replace character filter
#B Declares the pattern_replace filter
#C Expands the defined pattern_replace filter with options
#D Indicates the type of the filter (pattern_replace)
#E Indicates the pattern to search and replace
#F Defines the replacement value

The code in this example demonstrates a mechanism to define and develop a custom analyzer with a pattern_replace character filter. Here, we try to match and replace our input field, replacing the underscore (_) character with a dash (-). If you test the analyzer as shown in the following code listing (7.41), we see that the output, "Apple-Boy-Cat", replaced all the underscores with dashes.

Listing 7.41: Testing a custom pattern replace character filter

```
POST index_with_pattern_replace_filter/_analyze
{
  "text": "Apple_Boy_Cat",
  "analyzer": "my_pattern_replace_analyzer"
}
```

While the sentences are cleaned and cleared of unwanted characters, there remains the job of splitting the sentences into individual tokens based on delimiters, patterns, and other criteria. And that job is undertaken by a tokenizer component, discussed in the next section.

7.7 Tokenizers

The job of a tokenizer is to create tokens based on certain criteria. Tokenizers split the incoming input fields into tokens that are, most likely, the individual words of a sentence.

There are over a dozen tokenizers, each of them tokenizing fields as per the tokenizer's definition.

NOTE As you can imagine, going over all tokenizers in a print book is not only impractical but also boring to read basically similar text with tiny changes. I have picked a few important and popular tokenizers here so you can understand the concept and mechanics behind a tokenizer. Obviously, the code is available for most of them on my GitHub page: <https://github.com/madhusudhankonda/elasticsearch-in-action>

7.7.1 Standard tokenizer

A standard tokenizer splits the words based on word boundaries and punctuation. It tokenizes the text fields based on whitespace delimiters as well as on punctuation like commas, hyphens, colons, semicolons, and so forth. The following code uses the `_analyze` API to execute the tokenizer on a field:

```
POST _analyze
{
  "text": "Hello,cruel world!",
  "tokenizer": "standard"
}
```

This results in three tokens: "Hello", "cruel", and "world". The comma and the whitespace act as delimiters to tokenize the field into individual tokens.

The standard analyzer has only one attribute that can be customized, the `max_token_length`. This attribute helps produce tokens of the size defined by the `max_token_length` property (default size is 255). We can set this property by creating a custom analyzer with a custom tokenizer as the following listing 7.42 shows.

Listing 7.42 Index with a custom tokenizer

```
PUT index_with_custom_standard_tokenizer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "#A": {
          "custom_token_length_analyzer": {
            "tokenizer": "custom_token_length_tokenizer"
          }
        },
        "tokenizer": {
          "#B": {
            "type": "standard",
            "max_token_length": 2
          }
        }
      }
    }
  }
}
```

#A Creates a custom analyzer with a pointer to the custom tokenizer
#B The custom tokenizer with `max_token_length` set to 2 characters

Similar to the way we created an index with a custom component for types of character filters in an earlier section, we can follow the same path to create an index with a custom analyzer that encompasses a standard tokenizer. The tokenizer is then extended by providing the `max_token_length` size (in the previous listing 7.42, the size is set to 2). Once the index is created, we can then use the `_analyze` API to test the field as the following listing 7.42 shows. This code spits out two tokens: "Bo" and "nd", which honors our request for a token size of 2 characters.

Listing 7.43 Testing a token size for the tokenizer

```
POST index_with_custom_standard_tokenizer/_analyze
{
  "text": "Bond",
  "analyzer": "custom_token_length_analyzer"
}
```

7.7.2 N-gram and edge_ngram tokenizers

Before we jump into learning the n-gram tokenizers, let's recap n-grams, edge_ngrams, and shingles.

The n-grams are a sequence of words for a given size prepared from a given word. Take as an example the word "coffee". The two-letter n-grams, usually called bi-grams, are "co", "of", "ff", "fe", and "ee". Similarly, the three-letter tri-grams are "cof", "off", "ffe", and "fee". As you can see from these two examples, the n-grams are prepared by sliding the letter window.

On the other hand, the edge_ngrams produce words with letters anchored at the beginning of the word. Considering "coffee" as our example, the edge_ngram produces "c", "co", "cof", "coff", "coffe", and "coffee". Figure 7.13 depicts the n-grams and edge_ngrams.

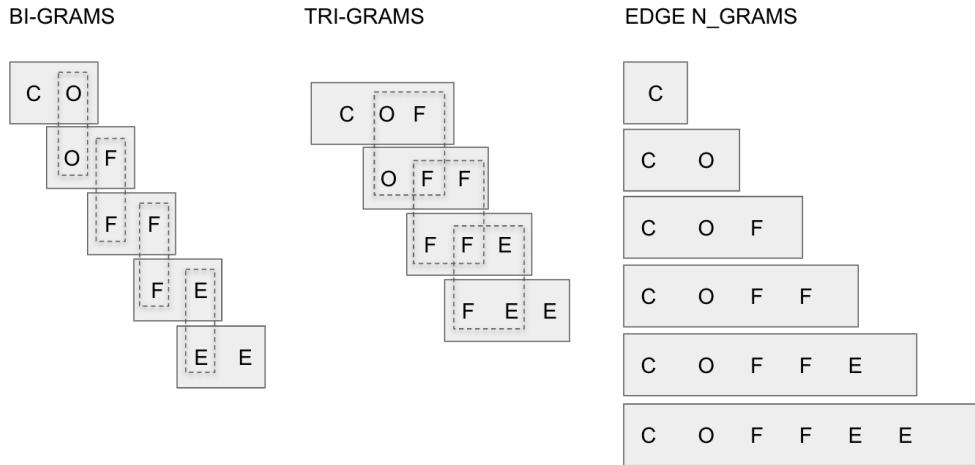


Figure 7.13 Pictorial representation of n-grams and edge_ngrams

The n-gram and edge_ngram tokenizers emit n-grams as the name suggests. Let's look at them in action.

THE N-GRAM TOKENIZER

For correcting spellings and breaking words, we usually use n-grams. The n-gram tokenizer emits n-grams of a minimum size as 1 and a maximum size of 2 by default. For example, this code produces n-grams of the word "Bond".

```
POST _analyze
{
  "text": "Bond",
  "tokenizer": "ngram"
}
```

The output is [B, Bo, o, on, n, nd, d]. You can see that each n-gram is made of one or two letters: this is the default behavior. We can customize the `min_gram` and `max_gram` sizes by specifying the configuration as demonstrated in the following listing 7.44:

Listing 7.44: An n_gram tokenizer

```
PUT index_with_ngram_tokenizer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "ngram_analyzer": {
          "tokenizer": "ngram_tokenizer"
        }
      },
      "tokenizer": {
        "ngram_tokenizer": {
          "type": "ngram",
          "min_gram": 2,
          "max_gram": 3,
          "token_chars": [
            "letter"
          ]
        }
      }
    }
  }
}
```

Using the `min_gram` and `max_gram` attributes of the `ngram` tokenizer (set to 2 and 3, respectively in the example), we can configure the index to produce n-grams. Let's test the feature as shown in the following listing 7.45:

Listing 7.45: Testing the n_gram tokenizer

```
POST index_with_ngram_tokenizer/_analyze
{
  "text": "bond",
  "analyzer": "ngram_analyzer"
}
```

This produces these n-grams: “bo”, “bon”, “on”, “ond”, and “nd”. As you can see, the n-grams were of size 2 and 3 characters.

THE EDGE_NGRAM TOKENIZER

Following the same path, we can use `edge_ngram` tokenizer to spit out edge n-grams. as the code snippet that creates the analyzer with the `edge_ngram` tokenizer demonstrates:

```
..
"tokenizer": {
  "my_edge_ngram_tokenizer": {
    "type": "edge_ngram",
    "min_gram": 2,
    "max_gram": 6,
    "token_chars": ["letter", "digit"]
  }
}
Once we have the edge_ngram tokenizer attached to a custom analyzer, we can test the field
      using the _analyze API. The following shows how:
POST index_with_edge_ngram/_analyze
{
  "text": "bond",
  "analyzer": "edge_ngram_analyzer"
}
```

This invocation spits out these edge grams: “b”, “bo”, “bon”, and “bond.” Note that all the words are anchored on the first letter.

7.7.3 Other tokenizers

As you can imagine, there are a handful of other tokenizers and listing them one by one not only feels repetitious but also impractical. I have, however, created code examples in my GitHub page (<https://github.com/madhusudhankonda/elasticsearch-in-action>), so I suggest you go over the code there to work through the other tokenizer examples. TTable 7.4 provides an explanation of the other tokenizers in brief.

Table 7.4 Out-of-the-box tokenizers

Tokenizer	Description
Pattern	The pattern tokenizer splits the field into tokens on a regex match. The default pattern is to split words when encountered by a nonword letter.
URL and email	The uax_url_email tokenizer parses the fields and preserves URLs and emails. The URLs and emails in text will be spit out as they are without any tokenization.
Whitespace	The whitespace tokenizer splits the text into tokens when whitespace is encountered.
Keyword	The keyword tokenizer doesn't touch the tokens; it spits the text as is.
Lowercase	The lowercase tokenizer splits the text into tokens when a nonletter is encountered and lowercases the tokens.
Path hierarchy	The path_hierarchy tokenizer splits hierarchical text such as filesystem folders into tokens based on path separators.

The final component of an analyzer is a token filter. Its job is to work on the tokens that were spit out by the tokenizers. We will briefly discuss the token filters in the next section.

7.8 Token filters

The tokens produced by the tokenizers may need further enriching or enhancements such as lowercasing (or uppercasing) the tokens, providing synonyms, developing stemming words, removing the apostrophes or punctuation, and so on. Token filters work on the tokens to perform such transformations.

Elasticsearch provides almost 50 token filters and, as you can imagine, discussing all of them here is not feasible. I've managed to grab a few, but feel free to reference the official documentation for the rest of the token filters. We can test a token filter by simply attaching to a tokenizer and using it in the `_analyze` API call as the following listing (7.46) shows:

Listing 7.46 Adding a token filter along with the tokenizer

```
GET _analyze
{
  "tokenizer" : "standard",
  "filter" : ["uppercase","reverse"],
  "text" : "bond"
}
```

The filter accepts an array of token filters; for example, we provided the uppercase and reverse filters in this example). The output would be “DNOB” (“bond” is uppercased and reversed).

You can also attach the filters to a custom analyzer as the following listing 7.47 demonstrates. Then because we know how to attach token filters, we’ll look at a few examples.

Listing 7.47: Custom analyzer associated with additional filters

```
PUT index_with_token_filters
{
  "settings": {
    "analysis": {
      "analyzer": {
        "token_filter_analyzer": {#A
          "tokenizer": "standard",
          "filter": [ "uppercase", "reverse"]#B
        }
      }
    }
  }
}
```

#A Defines a custom analyzer

#B Provides token filters as an array of filters

7.8.1 Stemmer filter

Stemming is a mechanism to reduce the words to their root words (for example, the word “bark” is the root word for “barking”). Elasticsearch provides an out-of-the-box stemmer that reduces the words to their root form. The following listing (7.48) demonstrates an example of stemmer usage.

Listing 7.48 Using a stemmer as the token filter

```
POST _analyze
{
  "tokenizer": "standard",
  "filter": [ "stemmer" ],
  "text": "barking is my life"
}
```

When executed, this code produces a list of tokens: “bark”, “is”, “my”, and “life”. As you can see, the original word, “barking”, is transformed to “bark”.

7.8.2 Shingle filter

Shingles are the word n-grams that are generated at the token level (unlike the n-grams and edge_ngrams that emit n-grams at a letter level). For example, the text, “james bond” emits as “james”, and “james bond”. The following code shows an example usage of shingle filter:

```
POST _analyze
{
  "tokenizer": "standard",
  "filter": ["shingle"],
  "text": "java python go"
}
```

The result of this code execution is [java, java python, python, python go, go]. The default behavior of the filter is to emit unigrams and two-word n-grams. We can change this default behavior by creating a custom analyzer with a custom shingle filter. The following listing (7.49) shows how this is configured.

Listing 7.49: Creating a custom analyzers for shingles

```
PUT index_with_shingle
{
  "settings": {
    "analysis": {
      "analyzer": {
        "shingles_analyzer": {
          "tokenizer": "standard",
          "filter": ["shingles_filter"] #A
        }
      },
      "filter": {
        "shingles_filter": {
          "type": "shingle",
          "min_shingle_size": 2,
          "max_shingle_size": 3,
          "uoutput_unigrams": false #C
        }
      }
    }
  }
}
```

#A Creates a custom analyzer attached with a shingles filter

#B Provides the attributes of the shingle filter (for example, min and max shingle)

#C Turns off the output of single words

Invoking this code on some text (as shown in the listing 7.50 below) produces two and three groups of words.

Listing 7.50 Running the text via the shingles analyzer

```
POST index_with_shingle/_analyze
{
  "text": "java python go",
  "analyzer": "shingles_analyzer"
}
```

The analyzer returns [java python, java python go, python go] because we've configured the filter to produce only 2- and 3-word shingles. The unigram (one word shingle) like "java", "python", and so forth are removed in the output because we disabled our filter to output them.

7.8.3 Synonym filter

We worked with synonyms earlier without really going into detail. *Synonyms* are the alternate meanings given to the words. For example, with football and soccer (the latter being the way football was called in America), both should point to a football game. The synonyms filter helps create a set of words to help produce a richer user experience while searching.

Elasticsearch expects us to provide a set of words and their synonyms by configuring the analyzer with a synonym token filter. We create the synonyms filter on an index's settings as the listing 7.51 demonstrates:

Listing 7.51 Creating an index with a synonym filter

```
PUT index_with_synonyms
{
  "settings": {
    "analysis": {
      "filter": {
        "synonyms_filter": {
          "type": "synonym",
          "synonyms": [ "soccer => football" ]
        }
      }
    }
  }
}
```

In the code example, we created a synonyms list (soccer is treated as an alternate name to football) associated with the `synonym` type. Once we have the index configured with this filter, we can test the text field:

```
POST index_with_synonyms/_analyze
{
  "text": "What's soccer?",
  "tokenizer": "standard",
  "filter": ["synonyms_filter"]
}
```

This produces two tokens: "What's", and "football". As you can see from the output, the word "soccer" is replaced with the word "football".

SYNONYMS FROM A FILE

We can provide the synonyms via a file on a filesystem rather than hard coding them as we did in listing 7.51. To do that, we need to provide the file path in the `synonyms_path` variable as the following listing (7.52) demonstrates.

Listing 7.52 Synonyms loaded from a file

```
PUT index_with_synonyms_from_file_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "synonyms_analyzer": {
          "type": "standard",
          "filter": ["synonyms_from_file_filter"]
        }
      }
    }
  }
}
```

We can call the file from a relative or absolute path. The relative path points to the config directory of Elasticsearch's installation folder.

That's it, it's a wrap! This lesson is fundamental for the discussion on search that we are going to embark on in the next chapter. We completed all the necessary journeys required to work with search functionality in depth. Now let's jump right into the next chapter, where we'll learn the fundamentals of searching.

7.9 Summary

- Elasticsearch analyzes text fields via the text analysis process. The text analysis is carried out using either built-in or custom analyzers. The non text fields are not analyzed.
- The text analysis is composed of two phases: tokenization and normalization. Tokenization is where the input field is split into individual words or tokens, and normalization enhances the word (for example, either the word can be synonymized, stemmed, or removed).
- Elasticsearch employs a software module called analyzer to carry out this text analysis. The analyzer is nothing but a package made of character filters, token filters, and tokenizers
- Elasticsearch uses a standard analyzer by default if no analyzer is mentioned explicitly during indexing and searching. A standard analyzer employs no character filter, a standard tokenizer, and two token filters (lowercase and stop), although the stop filter is off by default.
- Every analyzer must have one (and only one) tokenizer, but it can have zero or more character or token filters.

- Character filters help strip unwanted characters from an input field. Tokenizers act on the fields that were processed by character filters (or even on raw fields as character filters are options). The token filters work on the tokens that are emitted by the tokenizer.
- Elasticsearch provides a handful of out-of-the-box analyzers. We can mix-and-match existing tokenizers with character or token filters to make custom analyzers that suit our requirements.

8

Introducing search

This chapter covers

- **Fundamentals of search**
- **Types of search methods**
- **Introduction to Query DSL**
- **Common search features**

It is now time to enter the world of search. So far, we've looked at priming Elasticsearch with data, and in the last chapter, we went over the mechanics of how the text fields are analyzed whereas non-text fields are not. Although we had a taste of searching through the data using a set of queries, we didn't really explore the mechanics of searching or the nitty-gritty of the variants of search. To remedy that, this chapter and the following few chapters are dedicated to search.

Search is the core functionality of Elasticsearch that answers user queries efficiently and effectively. Once the data is indexed and available for search, users can ask various questions. For example, assuming our fictitious ElasticBooks online bookstore website search is powered by Elasticsearch, we can expect a multitude of queries originating from the clients. This can range from a simple query like finding a book based on a title search word or a complex query like searching books that match multiple criteria: a particular edition, published between a range of dates, hard bound with a review rating higher than 4.5 out of 5, price less than a certain amount, and so on. The UI may support various widgets such as drop downs, sliders, check boxes, and so forth to enable filtering the search even further.

This chapter provides the introduction to search and the fundamental features one should expect to employ while searching. We first will learn about the search mechanics: how a search request is processed and a response is created and sent to the client. We will then learn about the fundamentals of search: the search API and the contexts in which search

queries are performed. We'll dissect a request and response to dig deeper into various components of each of these.

We'll also look at the types of searches, the URI request and the Query DSL, in detail further along in the chapter. We'll look into the reason why we prefer the Query DSL over the URI request method. Finally, we'll learn crosscutting search features such as highlighting, pagination, explanation, manipulating the response fields, and others.

8.1 Overview

Elasticsearch not only supports simple search functionality but also advanced searches that consider multiple criteria including geospatial queries. There are two variants of search in the world of Elasticsearch: *structured search* and *unstructured search*.

Structured search, supported by a term-level search functionality, returns results with no relevance scoring associated. Elasticsearch fetches the documents if they match exactly and doesn't bother with if they are a close match or how well they match. For example, searching for flights between a set of dates, searching for best-selling books during a particular sale promotion, and so on fall in this category. When the search is carried out, Elasticsearch simply checks if the match is successful or not. There are either flights that fall in that data rate or not. There are either a handful of best sellers or not. There's nothing that falls in a *may be* category. This type of structured search is provided by what is called *term-level queries* in Elasticsearch.

On the other hand, in an unstructured search, Elasticsearch retrieves results that are closely related to the query. The results are scored based on how closely they are relevant to the criteria: the highly relevant results get higher scoring and, hence, are positioned on the top of the result hits. Searching on text fields yields relevant results. Elasticsearch provides full text search for the purpose of searching through the unstructured data.

We communicate with the Elasticsearch engine using a RESTful API to execute queries. A search query is written using either a special query syntax, called Query DSL (domain specific language), or an URL standard, called query requests. When a query is issued, any available node in the cluster picks the request and processes it. The response is returned as a JSON object with an array of individual documents as results in the object.

If the query is executed on text fields, the individual result is returned with a relevancy score. If the score is a positive number, the higher the score, the higher the relevancy. The results are sorted in a descending order with the result with highest score on top.

Not every result that a response has is accurate. Just as you can expect an incorrect or irrelevant result at times when searching for something on Google, Elasticsearch may not return 100% relevant results. This is due to the fact Elasticsearch employs two strategies, called precision and recall, that affect the relevancy of the returned results. *Precision* is the percentage of relevant documents retrieved over a set of documents available in the index, whereas *_recall_* is the percentage of relevant documents retrieved over a set of applicable documents. We will go over precision and recall in the next chapter.

In this chapter, we will go over the basics of search and will cover the fundamentals in detail. This will help us lay the foundation for search and its features and APIs so that we can work with the full-text and term-level queries in the following chapters. First, we will learn the fundamentals of search including the search API and search types, as well as

understanding the concept of analyzed versus non-analyzed queries. We will then walkthrough the two types of search invocation methods, URI request and Query DSL, with examples.

There are quite a number of features that are common to search, regardless of the query type (full-text, term-level, geospatial, etc.) we choose. We will discuss these search features in the last section of this chapter, and more than likely, we will find the application of these features in the next few chapters when working with search and aggregations. In the next section, we will explain the mechanics of search: what and how Elasticsearch works on a search query to retrieve matching results.

8.2 How does search work?

A lot happens when a user invokes a search query against Elasticsearch. Although we touched on the mechanics earlier, let's recap what we learned. Figure 8.1 shows the mechanics of how a search is carried out by the engine in the background.

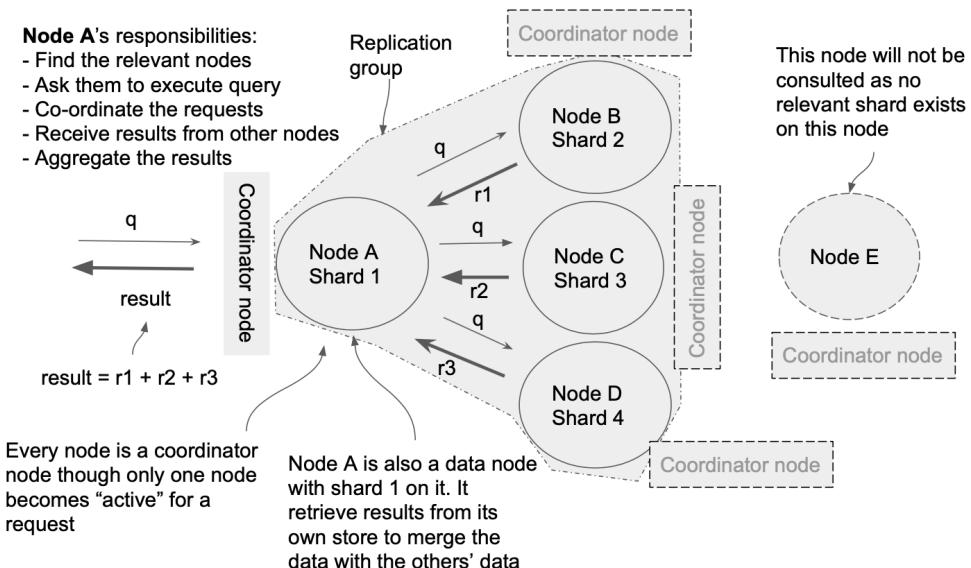


Figure 8.1 A typical search request and the mechanics of how a search works

When a search request is received from a user or a client, the engine forwards that request to one of the available nodes in the cluster. Every node in the cluster is, by default, assigned to a coordinator role; hence, making every node eligible for picking up the client requests on a round-robin basis. Once the request reaches the coordinator node, it then determines the nodes on which the shards of the concerned documents exist.

In figure 8.1, Node A is the *coordinator* node, where it receives the request from the client. It is chosen as a coordinator node for no specific reason other than demonstration

purposes. Once it is chosen as the (coordinator) active role, it creates a *replication group* with a set of shards and replicas on individual nodes in a cluster that consists of the data. Remember, an index is made of shards, and each of these shards can exist independently on other nodes. In our example, the index is made of four shards: shards 1 to 4 exist on Nodes A to D, respectively.

Node A then formulates the query request to send to other nodes, requesting them to carry out the search. Upon receiving the request, the respective node performs the search request on its shard. It then extracts the top set of results and responds back to the active coordinator with the results. The active coordinator then merges the data and sorts it before sending it to the client as a final result.

If the coordinator has a role as a data node, it will also dig into its own store to fetch the results. Not every node that receives the request is necessarily a data node. Similarly, not every node is expected to be part of the replication group for this search query. With this understanding of the search mechanics, let's turn to the fundamentals of search.

8.3 Search fundamentals

Now that we know the inner workings of search, let's look at the search API and learn about ways to invoke the engine to carry out search queries.

Elasticsearch exposes a search endpoint for communicating with it to execute search queries. Let's look at the endpoint details in the next section.

8.3.1 Search endpoint

Elasticsearch provides RESTful APIs to query the data, a `search` endpoint to be specific. We use GET/POST to invoke this endpoint, passing query parameters along with the request or with a request body. The first method of querying is called the URI request method, and the latter is Elasticsearch's special domain query language known as Query DSL. Although both are useful in their own way, Query DSL is powerful and feature-rich.

Query DSL uses a RESTful API for communication with a request body, consisting of the query and other attributes that make up or supplement the query. Query DSL allows a search criteria wrapped in a JSON body to be sent along with the request URL to the server. The result is wrapped in a JSON object as well. We can provide a single query or combine multiple queries, depending on the requirement. Query DSL is also the mechanism to send aggregate queries to the engine. We will learn more about aggregations in the later part of the chapter.

The queries that we construct depend on what type of data we are searching. We'll discuss structured and unstructured search queries in the next section. For now, there are two ways of accessing the search endpoint:

- *URI request*—With this method, we pass the search query along with the endpoint as parameters to the query. For example, `GET movies/_search?q=title:Godfather` fetches all the movies matching the word *Godfather* in the title (*The Godfather* trilogy, indeed).
- *Query DSL*—With this method, Elasticsearch implements a domain-specific language

(DSL) for the search. The criteria is passed as a JSON object in the payload. An example of the same requirement to fetch all the movies with the word *Godfather* in the title field would be

```
GET movies/_search
{
  "query": {
    "match": {
      "title": "Godfather"
    }
  }
}
```

Query DSL is a first-class querying mechanism. It is easier to write complex criteria using Query DSL than with the URI request mechanism. When searching across multiple indices, we can use comma-separated index names like GET <index1>,<index2>,<index3>/_search, including wildcards. We will look at various invocations and mechanisms in the coming sections of this chapter as well as in the next couple of chapters.

Although we will look at these two methods in detail in coming sections, we will work more extensively with Query DSL than with the URI request method. This is for various advantages that you will realize when you start experimenting and working with them.

NOTE Query DSL is the Swiss army knife equivalent when it comes to talking to Elasticsearch and is the preferred option. Elasticsearch developed this domain-specific language extensively to work with its engine. Everything and anything we want to ask Elasticsearch can be retrieved using Query DSL.

Don't fret if you don't understand the search queries and the way they are coded up. We will work through a few examples in this chapter in a moment, but rest assured, they are dealt with in detail in the coming chapters.

8.3.2 Query vs filter context

There's another fundamental concept that we should understand: the execution context. Elasticsearch internally uses an execution context when running searches. This execution context can be either a *filter* context or *query* context. All queries issued to Elasticsearch are carried out in one of these contexts. Although we have no say in asking Elasticsearch to apply a certain type of context, it is our query that lets Elasticsearch decide on applying the appropriate context.

We'll execute a couple of queries to understand the context in which the query is executed. We see this in action in the following sections.

NOTE We did not prime our engine with the movies dataset yet, so the queries in the following sections might look a bit baffling. I suggest going over section 8.4, *Movie sample data*, to index your Elasticsearch with some sample data if you want to execute the examples provided in the query and filter contexts sections that follow.

QUERY CONTEXT

We have used a match query to search for a set of documents that match the keywords with the field's values. The code in the following listing is a simple match query that searches for the word *Godfather* in a title field.

Listing 8.1 Match query the results in hits with scoring - query context by default

```
GET movies/_search
{
  "query": {
    "match": {
      "title": "Godfather"
    }
  }
}
```

This code returns our two Godfather movies as expected. However, if you look at the results closely in the following code snippet, each has an associated relevancy score:

```
"hits" : [{
  ...
  "_score" : 2.879596
  "_source" : {
    "title" : "The Godfather"
    ...
  }
},
{
  ...
  "_score" : 2.261362
  "_source" : {
    "title" : "The Godfather: Part II"
    ...
  }
}]
```

The code output indicates that the query was executed in a query context because the query searched not only if it matched a document, but also how well the document was matched. If you were wondering why the score on the second result (2.261362) is slightly lower than the score of the first one (2.879596), it's due to the fact that the engine's relevancy algorithm found the word *Godfather* in a title of two words ("the", "godfather"), which ranks a match higher than in a title of four words ("the", "godfather", "part", "III").

NOTE The queries on full-text search fields run in a query context because they are expected to have a scoring associated with each of the matched documents.

Although fetching the result with a relevancy score is fine for most cases, there may be some use cases where we do not need to find out how well the document matched. Instead, all we may want to know is if there's a match or not. This is where the filter context comes into play.

FILTER CONTEXT

Let's rewrite the query from listing 8.1, but this time, wrapping our `match` query in a `bool` query with a `filter` clause. The following listing shows the filter query in action.

Listing 8.2 A `bool` query where no score returns

```
GET movies/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "match": {
            "title": "Godfather"
          }
        }
      ]
    }
  }
}
```

In this listing, the results do not have a score (score is set to 0.0) because Elasticsearch got a clue from our query that it must be run in a filter context. This means that if we are not interested in scoring the document, we can ask Elasticsearch to run the query in a filter context by wrapping the query in a `filter` clause.

The main benefit of running a query in this context is that, because there is no need to calculate scores for the returned search results, Elasticsearch can save on some computing cycles. Because these filter queries are more idempotent, Elasticsearch tries to cache these queries for better performance.

In addition to using the `filter` clause in a `bool` query, we can also use the `must_not` clause. Remember, though, the `must_not` clause is completely opposite to the `must` query's intent. In addition to using the filter context in a `bool` query, we can also use it in a `constant_score` (the name is a giveaway!) query.

COMPOUND QUERIES .

The `bool` query is a compound query with a few clauses (`must`, `must_not`, `should`, and `filter`) to wrap leaf queries. In addition to the filter clause, the `must_not` clause gets executed in a filter context.

The `constant_score` query is another compound query where we can attach a query in a filter clause. The following query shows this in action:

```
GET movies/_search
{
  "query": {
    "constant_score": {
      "filter": [
        {
          "match": {
            "title": "Godfather"
          }
        }
      ]
    }
  }
}
```

}

We will look at compound queries in a dedicated chapter later in the book.

Knowing the execution context gets you one step closer to understanding the inner workings of the Elasticsearch engine. It helps create performant queries because the additional effort of running the relevancy algorithm is not required.

We will look at some examples to demonstrate these contexts in the upcoming chapters, but in the meantime, we need some sample data to work with the search examples throughout this chapter. Let's take a moment to load some movie data into our Elasticsearch engine.

8.4 Movie sample data

We will create some movie test data along with movie mappings for this chapter. Because we don't want Elasticsearch to deduce the field types, we will provide the relevant data types for each of the fields as mappings when we create the index (especially the `release_date` and `duration` fields because they can't be text fields). The following listing shows the `movies` index mapping.

Listing 8.3: Mapping schema for the `movies` domain

```
PUT movies #A Movies index
{
  "mappings": { #B Mappings schema
    "properties": { #C Fields and their types
      "title": {
        "type": "text",
        "fields": { #D Multi-field construct
          "original": {
            "type": "keyword"
          }
        }
      },
      "synopsis": {
        "type": "text"
      },
      "actors": {
        "type": "text"
      },
      "director": {
        "type": "text"
      },
      "rating": {
        "type": "half_float"
      },
      "release_date": {
        "type": "date",
        "format": "dd-MM-yyyy"
      },
      "certificate": {
        "type": "keyword"
      },
      "genre": {
        "type": "text"
      }
    }
  }
}
```

```

    }
}
}
```

Table 8.1 shows a few notable things from the mapping in the previous listing. The rest of the attributes are self-explanatory.

Table 8.1 Some fields and their respective data types

Field	Declared data types
title	text and keyword
release_date	date with the format as dd-MM-YYYY
certificate	keyword

Now that the mapping is in place for the `movies` domain, the next task is to index the sample data using the `_bulk` API. The following listing shows this API in action with a sample of data.

Listing 8.4: Indexing sample movie data using the `_bulk` API

```

PUT _bulk #A
{"index":{"_index":"movies","_id": "1"} } #B
{"title": "The Shawshank Redemption", "genre": "Drama", ...} #C
{"index":{"_index":"movies","_id": "2"} } #D
 {"title": "The Godfather", "genre": "Crime, Drama", ...}
 {"index":{"_index":"movies","_id": "3"} }
```

#A The `_bulk` API in action

#B The second document with an ID of 1 for indexing

#C The document itself

#D The second document with an ID of 2 for indexing

For brevity, we show only part of the script. You can visit my GitHub repository to fetch the full script at (<https://github.com/madhusudhankonda/elasticsearch-in-action/blob/main/datasets/top-movies-kibana.json>).

NOTE To avoid too many domain models and faffing around sample data, we will use the same movie data for this and the next chapter on term-level queries.

Now that we have an overview of search in general (as well as some search data at hand), let's jump into the action so that we can understand what constitutes a search request and the components that make up one. We'll also dissect the results (search response) too, all this in the next section.

8.5 Anatomy of a request and response

We had a quick glance at the search request and response in the last few chapters without having to worry about the details of the attributes and their explanations. It is equally important to understand the request and response objects as well. This will help us in formulating the query object without any errors and in understanding the meaning of the attributes featured in the response. In this section, we will delve into dissecting the request and response objects.

8.5.1 Search request

Search queries can be executed by using either a URI request method or a Query DSL. As we discussed earlier, our focus will be on the Query DSL in this book as it is more powerful and expressive. Figure 8.2 demonstrates the anatomy of a search request.

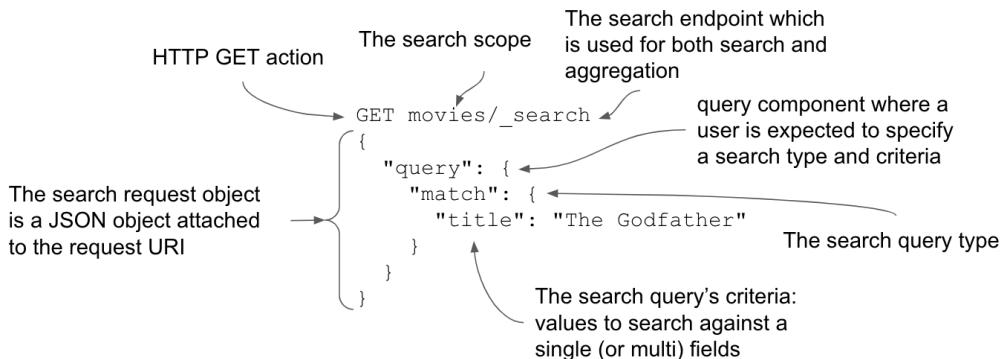


Figure 8.2 Components of a search request

The GET method is the HTTP action that specifies the intention: getting the data from the server with the request details carried over in a body. There's a school of thought that the GET method in a RESTful architecture shouldn't send any parameters using the body of the request. Instead, one should use a POST method if we are querying the server. Elasticsearch implements the GET method request that accepts a body, which helps formulate the query parameters. You can replace GET with POST as both GET or POST act on the resource exactly the same way.

NOTE The Internet is full of hot debates and discussions about using GET with and without a payload. Elasticsearch has taken the route of using GET with a payload. (If you want to understand the schools of thought, search “HTTP get with body” on the internet). In this book, we use GET with a payload for search and aggregation queries, although you can replace GET with POST if you are not comfortable using the GET method with a body.

The search scope in the GET (or POST) action defines the index or an alias that the engine uses to carry out the search. We can also include multiple indices, multiple aliases, or even no index in this URL. To specify multiple indices (or aliases), enter comma-separated index (or alias) names. Providing no index or alias on the search request tells the search to run against all the indices across the cluster. For example, if we have a cluster with 10 indices, running a search query like `GET _search {...}` without specifying the index or alias names searches for all the matching documents against all 10 indices.

The search request object or payload is a JSON object with the request details enclosed. The request details consist of the query component and can include other components such as pagination-related size and form attributes, a list that indicates which source fields to return in the response, sort criteria, highlighting, and so forth. We will go over individual features in the later part of this chapter.

The main constituent of the request is the query. It's the query's job to compose the question that needs to be answered. It does this by creating a `query` object that defines a query type and its required input. We can select from a multitude of query types, serving various search criteria. We will learn about these query types in detail in the next few chapters.

We create a specific query for a specific use case, ranging from match and term-level queries to special queries like geo shapes. Query types can help build a simple leaf query that targets a single search requirement or construct a complex requirement using compound queries that can deal with advanced searching with logical clauses.

Now that we learned about search request anatomy, it's time to dissect the response.

8.5.2 Search response

We briefly looked at the search responses in our earlier chapters but never looked at its details. Here's the opportunity to learn more about responses. Figure 8.3 demonstrates a typical response.

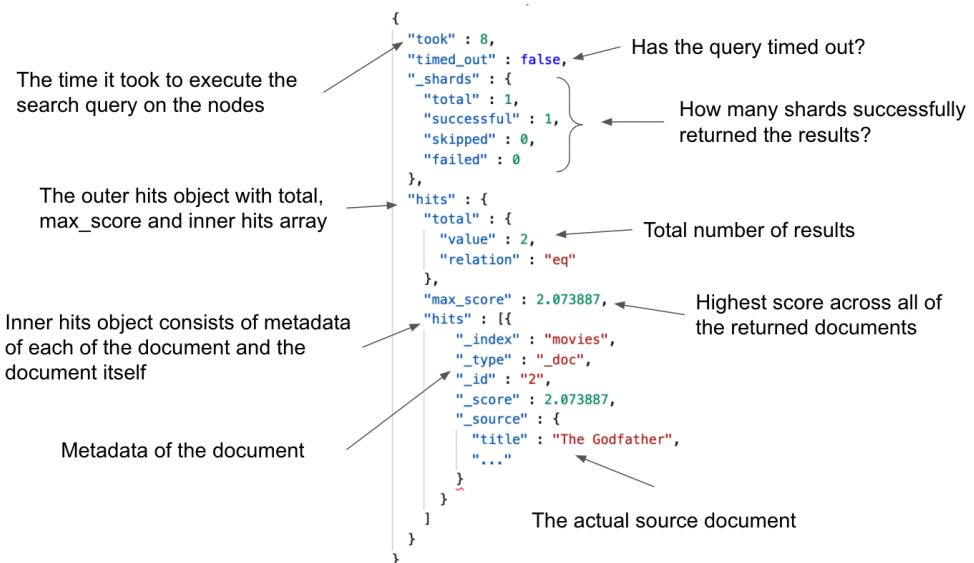


Figure 8.3 Different constituents of a search response

Let's go over these attributes briefly to understand what they represent.

The `took` attribute, measured in milliseconds, indicates the time it takes for the search request to complete. This is the time measured from when a coordinator node receives the request to the time it manages to aggregate the response before sending it back to the client. It doesn't include any of the client-to-server marshaling/unmarshalling times.

The `timed_out` attribute is a Boolean flag that indicates if the response has partial results, meaning if any of the shards failed to respond in time. For example, if we have three shards and one of them fails to return the results, the response will consist of the results from the two shards but indicates the failed shard's state in the next object under the `_shards` attribute.

The `shards` attribute provides the successful number of shards that executed the query and returned the results, as well as the failed ones. The `total` field is the number of shards expected to be searched, and the `successful` field denotes the shards that have returned the data. A `failed` flag, on the other hand, mentions the shards that were failed during the execution of search query - the flag is denoted by the `failed` attribute.

The `hits` attribute (we call it *outer hits*) consists of information about the results. It has another `hits` field inside, which we call the *inner hits*. The `outer hits` object consists of the returned results as well as the maximum score and total results. The maximum score, represented by the `max_score` property, is the returned document with the highest score. The `inner hits` object consists of the results (the actual documents). This is an array of all

the individual documents sorted by relevancy in ascending order. Each of the resultant documents receive this `_score` attribute if the query is executed in a query context.

We briefly mentioned the two types of requests that we can create for a query that asks a question: the URI `request search` and the Query DSL. In the next two sections, we will go over these in detail.

NOTE Although the URI request search method has its own benefits, due to the versatility and features that a Query DSL provides over the URI request, we use Query DSL for our search mechanism in this book. However, for completeness, we discuss the URI request search in the next section briefly to give you an idea of that mechanism. That way, you can experiment by creating equivalent URI request methods if you want to create cross-function search queries.

8.6 URI request search

The URI request method is an easy way to search simple queries. We invoke the search endpoint by passing required parameters. For example, the syntax in the following code snippet shows how to implement this search:

```
GET|POST <your_index_name>/_search?q=<name:value> AND|OR <name:value>
```

The `_search` endpoint is invoked on your index (or multiple indices) with a query in the form of `q=<name:value>`. Note that the query is appended after the `_search` endpoint with a question mark (?) delimiter. With this method, we are expected to pass in the query parameters attached to the URL as `name:value` pairs. Let's issue some search queries using this method.

8.6.1 Search movies by title

Let's say we want to find movies by searching a word in the title field (*Godfather*, for example). We use the `_search` endpoint on the `movies` index with the query parameter `Godfather` as the `title` attribute. The following listing demonstrates this.

Listing 8.5: A search query to fetch all movies matching Godfather

```
GET movies/_search?q=title:Godfather
```

The URL is composed of the `_search` endpoint followed by the query represented by the letter `q`. This query returns all movies with a title matching the word *Godfather* (we should get two movies in the response: *The Godfather Part I* and *Part II*).

If we want to search for movies matching multiple titles, we can do so by adding the additional title as search keywords with a space between them as the query in the following listing. It searches all the movies matching the words *Godfather*, *Knight*, and *Shawshank*.

Listing 8.6: Searching for movies by multiple titles

```
GET movies/_search?q=title:Godfather Knight Shawshank
```

This query returns four movie titles: *The Shawshank Redemption*, *The Dark Knight*, *The Godfather Part I*, and *The Godfather Part II*. Note that by default, Elasticsearch uses the `OR` operator between the query inputs, so we don't need to specify it (it is assumed to be present implicitly).

If you want to check the scores, *The Shawshank Redemption* and *The Dark Knight* received the same score (3.085904), and *The Godfather Part I* scored a bit higher than *The Godfather Part II*. We can ask Elasticsearch to explain how it derived this scoring by passing an `explain` flag:

```
GET movies/_search?q=title:Godfather Knight Shawshank&explain=true
```

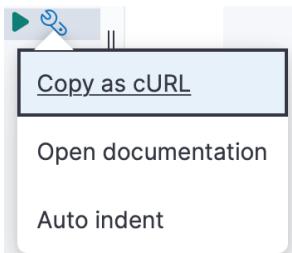
We discuss the `explain` flag later in section 8.8.3.

CURL FORMAT

If you are curious, the equivalent cURL for the previous code snippet is

```
curl -XGET "http://localhost:9200/movies/_search?q=title:Godfather Knight Shawshank"
```

You can generate a cURL command automatically by clicking the Spanner icon next to the green Play button in the DevTools UI as demonstrated in the following figure.



8.6.2 Search a specific movie

To fetch a specific movie, we can combine the criteria with the `AND` operator. The following query (listing 8.7) returns a single movie, *The Dark Knight*, matching the `title` and `actors` fields.

Listing 8.7: Fetch a specific movie by a title AND an actor

```
GET movies/_search?q=title:Knight AND actors:Bale
```

The `AND` operator helps narrow down the results. If you omit the `AND` operator and run the query as the following listing shows, you'd get two additional movies (the *Godfather* movies in addition to *The Dark Knight*).

Listing 8.8: Fetch a specific title OR an actor combination

```
GET movies/_search?q=title:Godfather actors:Bale
```

This is because we are searching for movies that match *Godfather* **OR** movies that match the actor Christian Bale. (Remember that Elasticsearch uses the `OR` operator by default.) Alternatively, to specify the `AND` operator in a multifields search, you can simply append the query with the `default_operator` param as this listing shows.

Listing 8.9: Setting a default_operator as AND

```
GET movies/_search?q=title:Godfather actors:Bale&default_operator=AND
```

This query does not return any results because there is no Godfather movie with Christian Bale (at least not so far).

Make sure the `default_operator` parameter is prefixed with an ampersand (&) as in listing 8.9. Also, there shouldn't be a space after the & because Kibana doesn't like a space between the & and the `default_operator` attribute, unfortunately.

8.6.3 Additional parameters

We can also pass a few more parameters with the URL in addition to adding multiple criteria to the query. For example, we can set two attributes, `from` and `size`, to fetch the paginated results (we'll look at pagination later in this chapter). We can also sort the results using, for example, `rating` to sort by the movie's rating. Again, we can ask Elasticsearch for an explanation of how it calculates the scoring using the `explain` parameter. Let's write a query with all these requirements and add a few more like the following listing shows.

Listing 8.10: Furthering our query with additional parameters

```
GET movies/_search?q=title:Godfather actors:(Brando OR Pacino) rating:(>=9.0 AND <=9.5)&from=0&size=10&explain=true&sort=rating&default_operator=AND
```

This query is a mashup of a lot of parameters, but here is the gist. We are searching for Godfather movies (`title`) starring Brando **OR** Pacino (`actors`) with a `rating` between 9 **AND** 9.5. We also add pagination (`from`, `size`) and sort by `rating`, as well as ask Elasticsearch to `explain`. We should have retrieved two movies (both Godfathers), although the actor Brando is not in *The Godfather Part II*. (If you're curious, the reason that two movies were returned in this case is because we used Brando **OR** Pacino in the `actors` field.)

As you may have guessed, the request URL method of writing queries is crude and error prone. Ideally, you should write queries using the Query DSL. Fortunately, you can wrap a URL request inside a Query DSL to get the goodness of both worlds.

8.6.4 Query DSL supports URI requests

Query DSL has a `query_string` method that allows us to wrap a URI request call (we will touch base on Query DSL in the next section). We can send the URI query parameters for searching movies with multiple title keywords in the request body as a `query_string`. This is demonstrated in the following listing.

Listing 8.11: Using a URI method wrapped up in a Query DSL

```
GET movies/_search
{
  "query": {
    "query_string": {
      "default_field": "title",
      "query": "Knight Redemption Lord Pulp",
      "default_operator": "OR"
    }
  }
}
```

The `query_string` is equivalent to the `q` parameter we used in the URI search method. While it is much better than the URI request method, the `query_string` method is strict with syntax and has some unforgiving characteristics. Unless there's a strong reason not to, we can use Query DSL queries rather than using `query_string`. We can use `query_string` for quick testing but relying on it for complicated and in-depth queries might be asking for trouble.

NOTE We write and execute search queries using QueryDSL throughout this book. The only time you'll find the search requests written using the URI request method is in the previous sections. If you want to use the URI request method, my advice is to use the `query_string` with Query DSL as the preferred query mechanism.

It's time to venture into the mighty kingdom of Query DSL. Because Query DSL is the Swiss army knife for searching, it certainly warrants its own dedicated section.

8.7 Query DSL

Elasticsearch developed a search-specific, all-purpose language and syntax we call Query DSL (domain-specific language). Query DSL is a sophisticated, powerful, and expressive language that creates a multitude of queries ranging from basic to complex, in addition to nested and more complicated ones. It can also be extended for analytical queries too. It is a JSON-based query language that can be constructed with queries both for search and analytics. The syntax and format goes this:

```
GET books/_search #A
{
  "query": { #B
    "match": { #C
      ...
    }
  }
}

#A Invokes the _search endpoint on the books index
#B All your queries are wrapped in this object.
#C The type of a query
#D The query criteria is enclosed here.
```

We invoke the `_search` endpoint with a `query` object, passed in as the body of the request. The `query` object consists of the logic for creating the required criteria.

8.7.1 Sample query

So far, we have worked with a few queries written using the Query DSL format. For completeness, let's write a `multi_match` query that searches a keyword, *Lord*, across two fields, `synopsis` and `title`. The query in the following listing demonstrates a search query written in Query DSL format.

Listing 8.12 Query DSL sample query

```
GET movies/_search
{
  "query": {
    "multi_match": {
      "query": "Lord",
      "fields": ["synopsis", "title"]
    }
  }
}
```

`GET movies/_search` is the shorthand search request from a client to the Elasticsearch server. The full request for this is something like `GET http://localhost:9200/movies/_search` (my Elasticsearch server is running locally, of course). This request expects a JSON-formatted body that consists of the query.

8.7.2 Query DSL for cURL

The same query can be invoked via cURL. The following listing demonstrates this invocation.

Listing 8.13 Query DSL via cURL

```
curl -XGET "http://localhost:9200/movies/_search" -H 'Content-Type: application/json' -d'
{
  "query": {
    "multi_match": {
      "query": "Lord",
      "fields": ["synopsis", "title"]
    }
  }
}'
```

The query is provided as an argument to the `-d` parameter as you can see in the code in listing 8.13. Note that the entire query (beginning with `Content-Type`) is enclosed in a single quote when sending the request via cURL.

8.7.3 Query DSL for aggregations

Although we haven't been introduced to the analytics part of Elasticsearch yet, here's a quick primer. With Query DSL, we use a similar format for aggregations (analytics) with an `aggs`

(short for aggregations) object instead of a `query` object. The following listing shows this format.

Listing 8.14 Average aggregation query written in Query DSL format

```
GET movies/_search
{
  "size": 0,
  "aggs": {
    "average_movie_rating": {
      "avg": {
        "field": "rating"
      }
    }
  }
}
```

This query fetches the average rating of all movies by utilizing a metric aggregation called `avg` (short for average). We will write most of our queries using Query DSL in depth in the next couple of chapters.

Now that we understand the overall form for Query DSL, let's look a bit more at leaf queries and compound queries, which we touched upon earlier.

8.7.4 Leaf and compound queries

Query DSL supports leaf as well as compound queries. The body of the search query can cater to simple or complex query criteria in the form of leaf or compound queries.

We call the queries that are straightforward with no clauses a *leaf query*. These are the queries that fetch results based on a certain criteria (for example, getting the top-rated movies, movies that are released during a particular year, the gross earnings of a movie, and so on).

With leaf queries, we can find out results for criteria against certain fields. Listing 8.15 is an example of a leaf query. (Don't worry about the contents of the query, we will go over these queries in the next few chapters.)

Listing 8.15 Leaf query that matches a phrase

```
GET movies/_search
{
  "query": {
    "match_phrase": {
      "synopsis": "A meek hobbit from the shire and eight companions"
    }
  }
}
```

Leaf queries cannot fetch multiple query clauses. For example, they are not designed to search for movies that match a title but must NOT match a particular actor AND released during a specific year AND rating must not fall below a certain number, and so on. The advanced requirement of logically combining certain clauses to serve a complex query is not possible with a leaf query, which leads to the introduction of compound queries.

Compound queries allow us to create complex queries by combining leaf queries and even other compound queries using logical operators. A Boolean query, for example, is a popular compound query that supports writing queries with clauses like `must`, `must_not`, `should`, and `filter`. We can write significantly complex queries using compound queries. The example query in the following listing demonstrates this.

Listing 8.16 Compound query

```
GET movies/_search
{
  "query": {
    "bool": {
      "must": [{"match": {"title": "Godfather"}}, {"must_not": [{"range": {"rating": {"lt": 9.0}}}]}, {"should": [{"match": {"actors": "Pacino"}}, {"filter": [{"term": {"actors": "Brando"}]}]}]
    }
  }
}
```

The compound query in this listing combines a handful of leaf queries joined up by logical operators. It fetches all movies that `must` match the title `Godfather` AND `must not` have a rating less than 9. The query `should` also consider movies with the actor `Pacino`. Finally, it filters out everything except for the movies with the actor `Brando`. Well, that's a mouthful. If you feel like it is too much to fathom, it is indeed. However, fear not, we will work with advanced querying using compound queries in the later chapters.

Leaf queries (as well as advanced queries) are wrapped in the `query` object of the search request. Other than implementing the advanced query's logic (which might at times be too complex), you should see no significant difference when writing compound queries.

There are a handful of cross-cutting features, such as sorting, pagination, highlighting, etc., that any type of a search query can use. They are not specific to `term` or `match`-level queries, compound queries, or leaf queries. We discuss these features in detail in the following section. We will also use these features every now and then during search and aggregation.

8.8 Search features

Elasticsearch provides capabilities to add additional features to the queries and its results. We can manipulate source documents in the return response by asking Elasticsearch to return a full document or only specific fields. We can sort the documents based on one or more fields in addition to sorting on the document's relevancy score. Elasticsearch let's us paginate results, say every page consists of a hundred documents instead of the default ten documents Elasticsearch returns. There's also a function to highlight the results with the search match words in the results. We can even ask the engine to grab the results only from a set of specific shards using the shard routing function.

Additionally, there are a handful of cross-cutting functions that most search queries support, irrespective of the type of query, whether it is a term-level, full-text, or geospatial query. Some are irrelevant to a specific type of query (for example, sorting isn't advisable on

text fields, hence it is limited to term-level queries. In the next few sections, we will venture into these features to understand their application in detail.

8.8.1 Pagination

More times than not, our query yields a lot of results, possibly hundreds or even thousands. Sending all the results for a query in one go is asking for trouble because both the serverside and clientside needs more memory and processing capacity to deal with the additional data load.

Elasticsearch, by default, sends the top-ten results, but we can change this number by setting the `size` parameter on the query, with a maximum set to 10,000. You can change this limit, and we will discuss that shortly. The query in the next listing sets `size` as 20, returning the top 20 results in one go.

Listing 8.17 Query to fetch a specific number of results

```
GET movies/_search
{
  "size": 20,
  "query": {
    "match_all": {}
  }
}
```

Setting `size` to 20 returns the top 20 results. If you have an index with one million documents, setting `size` to 10,000 retrieves that many documents (ignore the performance considerations for a moment!).

RESETTING THE 10 K SIZE LIMIT

The maximum number of results we can fetch by setting the `size` attribute is 10 K results. If you set the `size` to say, 10001, and execute the following query

```
GET movies/_search
{
  "size": 10001,
  "query": {
    "match_all": {}
  }
}
```

you would get an exception like that shown here:

Result window is too large, from + size must be less than or equal to: [10000] but was [10001]...

While 10 K is a pretty good number for most searches, if your requirement is to get more than that number, you need to reset the `max_result_window` setting on the index. The `max_result_window` is a dynamic setting on the index, so one can change it by executing the following query on a live index with a required changes like so:

```
PUT movies/_settings
{
  # Setting the size to 20000
```

```

    "max_result_window":20000
}

```

Having said that, it is *not* advisable to use this form of search when fetching large data sets. Instead, you should use the `search_after` feature, which we will discuss later in this section.

Elasticsearch provides a `scroll` API to fetch large data sets, but we would advise you to use the `search_after` feature over the `scroll` API. We describe the `search_after` feature in the next chapter.

Of course, fetching 10 K results in one go isn't a practical solution as clients usually expect results in a set of pages with each page containing a certain predefined number of results. For example, the 10 K results can be sliced as 100 pages with each page showing 100 results. For this, Elasticsearch has a feature for building paginated results. To do so, we use another attribute called `from`.

The `from` attribute is an offset, a specific page in a set of pages. If the `from` is set to 2, for example, the first two pages are ignored, and the third page is returned. The index starts from zero, so the page at index 0 is the first page, the page at index 1 is the second page, and the third page is the page at index 2, which is the `from` value in this case. The following listing shows the mechanism of how we can paginate the results by setting the `size` and `from` attributes.

Listing 8.18 Paginating results using size and from

```

GET movies/_search
{
  "size": 100, #A
  "from": 3, #B
  "query": {
    "match_all": {}
  }
}

#A Fetches every page with 100 results
#B Fetches from the third page, ignoring the first two pages

```

In the listing, by setting the `size` to 100, we are fetching a set of 100 documents in every page, and we fetch those pages starting with the third page, ignoring the first two pages with 200 results.

If the result set is too large (more than 10 K), rather than working with the pagination using the `size` and `from` attributes, we need to work with the `search_after` attribute. We'll look at an example of this (deep pagination) in the next chapter. For now, let's look at another common search feature, highlighting.

8.8.2 Highlighting

When we search for a keyword(s) on a website in our internet browser using Ctrl-F, we can see the results highlighted so they stand out. For example, see how the word *dummy* is

highlighted in figure 8.4. Highlighting keywords in the results for your clients are engaging and visually appealing too.

What is Lorem Ipsum?

Lorem Ipsum is simply **dummy** text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard **dummy** text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has

Figure 8.4 An example of highlighted text.

In a Query DSL, we can add a `highlight` object at the same level as the top-level `query` object as demonstrated in this code snippet:

```
GET books/_search
{
  "query": { ... },
  "highlight": { ... }
}
```

The `highlight` object expects a `fields` block, which can have multiple fields that you want to emphasize in the results. We provide the individual fields to highlight in the query inside the `fields` object. For example,

```
GET books/_search
{
  "query": { ... },
  "highlight": {
    "fields": {
      "field1": {},
      "field2": {}
    }
  }
}
```

When results are returned from the server, we can ask Elasticsearch to highlight the matches with its default settings by enclosing the matched text in emphasis tags (`match`). The code in the next listing creates a `highlight` object, indicating the text to highlight in the `title` field of the results.

Listing 8.19 Highlighting the results when matched

```
GET movies/_search
{
  "_source": false, #A
  "query": {
    "term": {
      "title": {
        "value": "godfather"
      }
    }
  }, "highlight": { #B
    "fields": {
      "title": {} #C
    }
  }
}
```

#A: Suppresses the source to be returned

#B Includes a highlight object along with the fields on which highlights are expected

#C The field on which we require the highlight

The following code snippet shows how to highlight the word *Godfather* with the `` (short for emphasis) tags. The source is suppressed in the results because we've set `_source` to `false` in the query.

```
{
  ...
  "highlight" : { "title" : ["The <em>Godfather</em>"] }
},
{
  ...
  "highlight" : { "title" : ["The <em>Godfather</em> II"] }
}
```

We use the `` tags to show emphasis on the font in HTML-based browsers. We can also use custom tags. For example, this code snippet demonstrates the mechanism to create a pair of curly braces (`{}{}`) as a tag

```
...
"highlight": {
  "pre_tags": "{{",
  "post_tags": "}}",
  "fields": {
    "title": {}
  }
}
```

which results in "The {{Godfather}}" (with curly braces as the highlights). Now that we know how to highlight our search results, let's turn our attention to relevancy scores in the data.

8.8.3 Explanation

Elasticsearch provides a mechanism to understand the makeup of relevancy scores. This mechanism tells us exactly how the engine calculates the score. This is achieved by using an `explain` flag on a search endpoint or an `explain` API. The `explain` API is also used to find out the reason why a document has or has not matched a query. In this section, we look at both these methods to understand their commonalities and subtle differences.

EXPLAIN FLAG

You may have noticed a positive number (a relevancy scoring value) in the results for some queries earlier. Although we know that value was computed and set by the engine, we didn't explain how it was computed. If we are curious about the calculation, Elasticsearch provides a flag, called `explain`, that we can set in the body of the query. If we set the `explain` attribute to `true`, Elasticsearch returns the results with the details of how it arrived at that score. In other words, it provides us with an explanation about the logic and the calculations that were carried out by the engine behind the scenes.

The query in listing 8.20 shows a `match` query. Because we want to get the details of how the scores were calculated, we've set `explain` to `true`.

Listing 8.20 Asking the engine to explain the score

```
GET movies/_search
{
  "explain": true,
  "_source": false,
  "query": {
    "match": {
      "title": "Lord"
    }
  }
}
```

The `explain` attribute is set at the same level as the `query` object. The result of this query is interesting as figure 8.5 demonstrates.

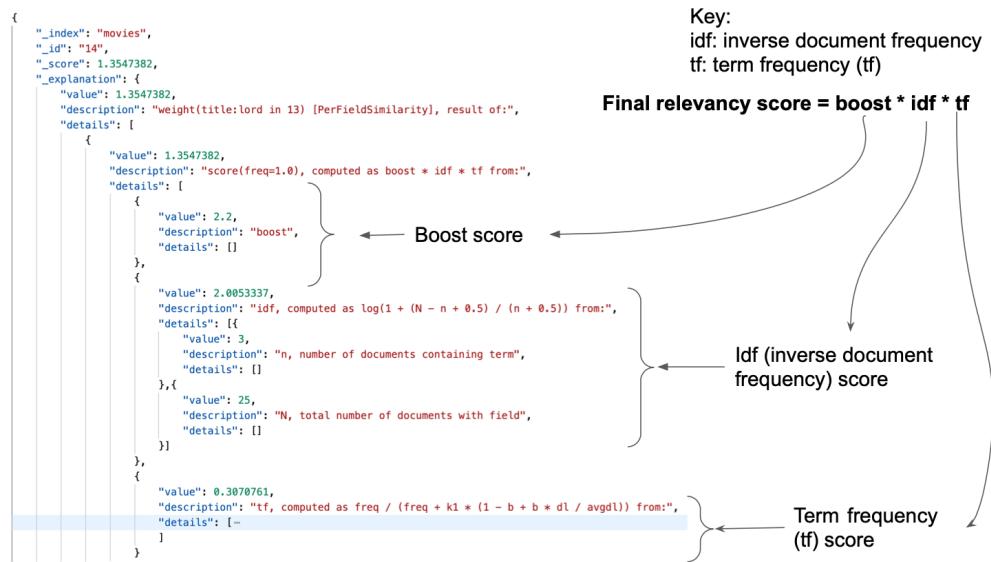


Figure 8.5 The explanation for how Elasticsearch calculates the relevancy scoring

As figure 8.5 shows, the relevancy score is calculated by multiplying three components: the inverse document frequency (*idf*), the term frequency (*tf*), and the boost factor. Elasticsearch goes into detail about how it evaluates and measures each of these components. For example, the *idf* is computed as

```
log(1 + (N - n + 0.5) / (n + 0.5))
```

where

- *n* is the total number of documents containing the term (in figure 8.5, there are 3 documents containing the word *lord*).
- *N* is the total number of documents (figure 8.5 shows 25 documents in our index).

You can find out what is *idf* made of by looking at the description field in the return response (see the above figure).

Similarly, the term frequency (*tf*) is calculated using the formula

```
freq / (freq + k1 * (1 - b + b * dl / avgdl))
```

An explanation on how each of these variables were calculated is provided in the `details` section of the results. I recommend that you take a look at this section to check the application of the formulae by the engine to produce the score.

You may be wondering what would happen if there's no match in the first place? That is, if you perform the search with *Lords* instead of *Lord*, you'd quickly find out that the results would be empty. Try this in the code for yourself to view the results.

EXPLAIN API

Although we use the `explain` attribute to understand the mechanics of relevancy scoring, there's also an `explain` API that provides insight into why a document matched (or not), in addition to providing the scoring calculations. The query in the following listing uses an `_explain` endpoint with a document ID as the parameter to demonstrate this approach.

Listing 8.21 Query that explains the scores using `_explain` endpoint

```
GET movies/_explain/14
{
  "query": {
    "match": {
      "title": "Lord"
    }
  }
}
```

This query is the same as the query in listing 8.20, but this time, we invoke the `_explain` endpoint instead of setting the `explain` flag on the `_search` endpoint. The result in listing 8.21 provides an explanation about the scores.

Finally, let's misspell the word "*Lord*" as *Lords* in the `match` attribute in listing 8.21 and rerun the query. As you might expect, we don't get the same results, instead we get a clue as the following code snippet shows:

```
{
  "_index": "movies",
  "_type": "_doc",
  "_id": "14",
  "matched": false,
  "explanation": {
    "value": 0.0,
    "description": "no matching term",
    "details": []
  }
}
```

As the `explanation` object's `description` says, *Lords* does not match the indexed data. Understanding the reasons for a match (or not a match) helps us troubleshoot the status of the queries (for example, in the previous example, we know that the matching term doesn't exist in our index). If you retry the same query in listing 8.21 using the `explain` flag instead, you may not get any information back from the engine other than an empty array of hits.

A search query built using the `explain` flag on the `_search` API can produce a lot of results. Asking for an explanation of the scores for all documents at a query level is simply a waste of computing resources in my opinion. Instead, pick one of the documents and ask for an explanation using the `_explain` API.

8.8.4 Sorting

The results returned by the engine are sorted by default on the relevancy score (`_score`): the higher the score, the higher on the list that the engine returns. However, Elasticsearch not only lets us manage the sort order of the relevancy score (from ascending to descending), we can also sort on other fields including multiple fields.

SORTING THE RESULTS

To sort the results, we must provide a `sort` object at the same level as the `query` (see the following snippet). The `sort` object consists of an array of fields, where each field contains a few “tweakable” parameters.

```
GET movies/_search
{
  "query": {
    "match": {
      "genre": "crime"
    }
  },
  "sort": [
    { "rating": { "order": "desc" } }
  ]
}
```

Here, the results of the `match` query that searches for all the movies in the crime genre are sorted by the movie’s rating. The `sort` object defines the field (`rating`) and the order in which the results are expected to be sorted, descending order in this case.

SORTING ON THE RELEVANCY SCORE

The documents carrying the relevancy score are sorted on `_score` in descending order by default if no sorting is specified in the query. For example, the query in the following listing sorts the results in a descending order because a sort order is not mentioned.

Listing 8.22 Default sorting for the score in descending order

```
GET movies/_search
{
  "size": 10,
  "query": {
    "match": {
      "title": "Godfather"
    }
  }
}
```

This is equivalent to issuing the `sort` block in a query. The following listing provides the code for this.

Listing 8.23 Sorting on _score

```
GET movies/_search
{
  "size": 10,
  "query": {
    "match": {
      "title": "Godfather"
    }
  },
  "sort": [ #A
    "_score" #B
  ]
}
```

#A Enables sort by setting the sort block at the same level as the query block

#B Without specifying the order, the results are descending by default

If you want to reverse the order with an ascending sort, meaning the lower scored documents are at the top of the list, we simply need to add the `_score` field to specify the order. The following listing shows how to do this.

Listing 8.24 Sorting the results in ascending order by score

```
GET movies/_search
{
  "size": 10,
  "query": {
    "match": {
      "title": "Godfather"
    }
  },
  "sort": [
    {"_score":{"order":"asc"}}
  ]
}
```

You probably have guessed the query to sort on a non scoring document field. The next listing demonstrates exactly this by sorting the data based on `rating` from highest to lowest in descending order.

Listing 8.25 Sorting the results by a field

```
GET movies/_search
{
  "size": 10,
  "query": {
    "match": {
      "genre": "crime"
    }
  },
  "sort": [
    {"rating":{"order":"desc"}}
  ]
}
```

After running this query, you'll find the results sorted with the highest rating movie on top of the list. (I am omitting the results here for brevity.) If you carefully observe the results, you'll see that the score is set to `null`.

When we use any field for sorting, Elasticsearch does not compute the score. However, there is a way you can ask Elasticsearch to compute the score even if you move away from sorting on `_score`. To do that, you would use the `track_scores` Boolean field. The following listing shows how to set the `track_score` for the engine to calculate the scores in this instance.

Listing 8.26 Enabling scoring when sorting on other fields

```
GET movies/_search
{
  "track_scores":true,
  "size": 10,
  "query": {
    "match": {
      "genre": "crime"
    }
  },
  "sort": [
    {"rating":{"order":"asc"}}
  ]
}
```

The highlighted `track_scores` attribute in the listing provides a cue to the engine to calculate the relevancy scores of the document. They will not be sorted on the `_score` attribute, though, because a custom field is used for sorting.

We can also enable sorting on multiple fields. The query in the following listing shows us how we can enable sorting on the `rating` and `release_date` fields.

Listing 8.27 Sorting by multiple fields in ascending order

```
GET movies/_search
{
  "size": 10,
  "query": {
    "match": {
      "genre": "crime"
    }
  },
  "sort": [
    {"rating":{"order":"asc"}},
    {"release_date":{"order":"asc"}}
  ]
}
```

When we sort on multiple fields, the sort order is important! The query's results in listing 8.27 are sorted in ascending order on the `rating` field first. If any of the movies are of the same rating, then the second field (`release_date`) is used to break the tie, so the results with the same rating will be sorted by `release_date` in ascending order.

NOTE We will look at the *geo sorting* in a later chapter. I think it warrants a dedicated chapter to understand geo queries and sorting on geopoints.

8.8.5 Manipulating the results

You may have observed that search queries return the results from the original documents specified with the `_source` field. Occasionally, we may want to fetch only a subset of fields. For example, we may need just the title and the rating of a movie when a user searches for a certain type of rating, or we might not need the document sent out in the response by the engine. Elasticsearch lets us manipulate the response, whether fetching selected fields or suppressing the whole document.

SUPPRESS THE FULL DOCUMENT

To suppress the document returned in the search response, we simply need to set the flag `_source` to `false` in the query. The following listing returns the response with just the metadata.

Listing 8.28 Suppressing the source document

```
GET movies/_search
{
  "_source": false, #A
  "query": {
    "match": {
      "certificate": "R"
    }
  }
}
```

#A Setting the `_source` flag to `false` removes the source document from the result.

The response, as shown in this code snippet, shows no mention of the original document at all:

```
"hits" : [
  {
    "_index" : "movies",
    "_type" : "_doc",
    "_id" : "1",
    "_score" : 0.58394784
  },
  {
    "_index" : "movies",
    "_type" : "_doc",
    "_id" : "2",
    "_score" : 0.58394784
  },
  ...
]
```

FETCHING SELECTED FIELDS

We can of course fetch a nominal set of fields if the intention is to not fetch the whole document but just a few selected fields. Let's see how we can do this in this section.

Elasticsearch provides a `fields` object to indicate which fields are expected to be returned. We define the fields explicitly in this object. For example, the query in the following code snippet fetches only the `title` and `rating` fields in the response.

```
GET movies/_search
{
  "_source": false,
  "query": {
    "match": {
      "certificate": "R"
    }
  },
  "fields": [
    "title",
    "rating"
  ]
}
```

The following snippet displays the response. It shows the resorted document with only the `title` and `rating` fields as expected.

```
{
  "_index" : "movies",
  "_type" : "_doc",
  "_id" : "1",
  "_score" : 0.58394784,
  "fields" : {
    "rating" : [
      9.296875
    ],
    "title" : [
      "The Shawshank Redemption"
    ]
  }
}
```

Note that each of the fields is returned as an array instead of as a single field. Because Elasticsearch doesn't have an array type, it expects multiple values; hence, each of the fields is wrapped in an array.

You can also use wildcards in the field's mapping. For example, setting `title*` retrieves `title`, `title.original`, `title_long_descripion`, `title_code`, and all other fields that have the `title` prefix. (We do not have all these fields in our mapping, other than `title` and `title.original`, so you can add them to the mapping to experiment with the wildcard setting.)

SCRIPTED FIELDS

We may at times need to compute a field on the fly and add it to the response. Say, for example, we want to set a movie as top rated if it falls within the highest ratings returned (say the rating is greater than say 9). For that, we can use scripting features when adding such ad hoc fields on demand.

To use the scripting feature, append the query with the `script_fields` object at the same level with the required name of the new dynamic file and the logic to populate it. The

following listing demonstrates this usage by creating a new field, `top_rated_movie`, by setting a flag based on the ratings the movie receives.

Listing 8.29 Source filtering with a script field

```
GET movies/_search
{
  "_source": ["title*", "synopsis", "rating"],
  "query": {
    "match": {
      "certificate": "R"
    }
  },
  "script_fields": {
    "top_rated_movie": {
      "script": {
        "lang": "painless",
        "source": "if (doc['rating'].value > 9.0) 'true'; else 'false'"
      }
    }
  }
}
```

The script consists of the `source` element where the logic of populating the new field (`top_rated_movie`) is defined: we stamp the movie as top rated if the rating of the movie is greater than 9. For completeness, look at the output (edited for brevity) with the new `top_rated_movie` field given here:

```
"hits" : [{
  ...
  "_source" : {
    "rating" : "9.3",
    "synopsis" : "Two imprisoned men bond ...",
    "title" : "The Shawshank Redemption"
  },
  "fields" : {
    "top_rated_movie" : ["true"]
  }
}
...]
```

Because Elasticsearch provides extensive support to create our own scripts, we have a dedicated chapter on the mechanics of scripting later in the book. In that chapter, we discuss a lot of use cases and implementation details of scripting.

SOURCE FILTERING

Earlier we've set the `_source` flag to `false` to suppress the documents returned in the response. Although we used it for all-or-nothing scenarios, there are a couple of use cases where we can implement the `_source` option to tweak the response further. For example, listing 8.30 sets `_source` to `[title*, synopsis, rating]` so that the results will return the `synopsis` and `rating` fields along with all fields with `title` as the prefix.

Listing 8.30 Using the _source tag to fetch custom fields

```
GET movies/_search
{
  "_source": ["title*","synopsis", "rating"],
  "query": {
    "match": {
      "certificate": "R"
    }
  }
}
```

In fact, you can take the `_source` option even further by setting a list of `includes` and `excludes` to further control the return fields. The following listing demonstrates this in action.

Listing 8.31 Source filtering using includes and excludes

```
GET movies/_search
{
  "_source": {
    "includes": ["title*","synopsis", "genre"],
    "excludes": ["title.original"]
  },
  "query": {
    "match": {
      "certificate": "R"
    }
  }
}
```

The `_source` object expects two arrays:

- `includes`, which consists of all the fields that are expected to be returned in the result
- `excludes`, which defines the list of fields that must be excluded from the fields returned from the `includes` list

In listing 8.31, we expect that all `title` fields (`title` and `title.original`), as well as the `synopsis` and `genre` fields are returned from the query. However, we can suppress `title.original` by including it in the `excludes` array. We can play with the `includes` and `excludes` arrays to gain a finer control of what fields we can return and which to suppress. For example, if you add an `excludes` array to the `_source` object, something like `"excludes": ["synopsis", "actors"]`, all fields except `synopsis` and `actors` will be returned.

8.8.6 Searching across indices and data streams

Our data is more likely than not spread across indices and data streams. Fortunately, Elasticsearch lets us search the data across these multiple indices and data streams by simply appending the required indices in the search request. For example, omitting the index name(s) on the search request is a clue to the engine to search across all indices. The following code snippet shows this technique:

```
GET _search
{
  "query": {
    "match": {
      "actors": "Pacino"
    }
  }
}
```

In fact, you can use `GET */_search` or `GET _all/_search` too, which is equivalent to the previous query. All these forms search across all indices in the cluster.

BOOSTING INDICES

When we search across multiple indices, we may want to have a document found in one index take precedence over the same document found in another index. That is, we may want to boost certain indices over others when performing a search across multiple indices. For that, we can attach an `indices_boost` object at the same level as the `query` object. We can input multiple indices with the appropriate boost rating set on this `indices_boost` object.

To demonstrate this concept, I've created two new indices (`index_top` and `index_new`) and indexed *The Shawshank Redemption* movie in these two new indices(see the code on my GitHub page). Now that we have the same movie across three indices, let's create the query with a requirement of enhancing the score of the document obtained from `movies_top` so that it's the topmost result.

Listing 8.32 Boosting the scores of a document fetched from a given index

```
GET movies*/_search
{
  "indices_boost": [
    { "movies": 0.1},#A
    { "movies_new": 0}, #B
    { "movies_top": 2.0} #C
  ],
  "query": {
    "match": {
      "title": "Redemption"
    }
  }
}
```

#A Lowers indices_boost to 0.1

#B Lowers indices_boost to 0

#C Raises indices_boost to 2.0

As the query indicates, we bumped up the score by double if the document is found in the Query DSL's `movies_top`, and we decreased it to 0.1 for the documents fetched from the `movies` index. Finally, we set `indices_boost` to 0 for the `movies_new` documents. If the document's original score is 0.2876821 in `movies_top`, for example, the new score will be 0.5753642 (2 * 0.2876821), whereas the other documents' scores will be calculated as per the setting in the `indices_boost` object.

And that's a wrap! There are a few advanced concepts like searching across multiple clusters (cross-cluster search), searching with templates (search templates), routing the search to a specific shard, and others. We will discuss these features in a dedicated chapter on advanced search in later chapters. In the meantime, the next chapter discusses term-level queries, now that we have a better understanding of the Query DSL and URI search features.

8.9 Summary

- Searching can be categorized into structured and unstructured search types.
- Structured data works with non text fields like numeric and date fields, or fields that are not analyzed during indexing time and produce binary results (either they exist or don't).
- Unstructured data deals with text fields that are expected to carry a relevancy scoring. The engine scores the results based on how well the resultant documents match the criteria.
- We use a term-level search for structured queries and a full text search for unstructured data.
- Every search request is processed by one of the coordinator nodes. It is the responsibility of the coordinator nodes to ask other nodes to execute the query, return the partial data, aggregate it, and respond to the client with a final result.
- Elasticsearch exposes a `_search` endpoint for queries and aggregations. We can invoke the `_search` endpoint by using either a URI request with parameters or building a full request using a special syntax called Query DSL.
- Query DSL is the preferred choice for creating search queries. We can construct a plethora of queries including advanced queries using Query DSL.
- Query DSL allows us to create leaf and compound queries. Leaf queries are simple search queries with a single criteria. Compound queries are used for advanced queries clubbing with conditional clauses.
- There are cross-cutting features available for most types of queries (for example, pagination, highlighting, explanation of the scoring, manipulation of the results, and so on).

9

Term-level search

This chapter covers

- Overview of term-level queries
- Term-level queries in action

Term-level search is designed to work with structured data such as numbers, dates, IP address, enumerations, keyword types, and others. We use term-level queries to find an exact match. This short chapter solely focuses on understanding the term-level search in detail and works through the various query types with examples.

The source code for this chapter is available on my GitHub page here:

https://github.com/madhusudhankonda/elasticsearch-in-action/blob/main/kibana_scripts/ch9_term_level_queries.txt

Let's begin with an overview of term-level searches and then we will look at specific queries.

9.1 Overview of term-level search

The term-level search is a structured search where the queries return results in exact matches. They search for structured data such as dates, numbers, and ranges. With this type of search, we don't care about how well the results match (like how well the documents correspond to the query) but that it returns the data (or not) if the query is matched. Hence, we do not expect a relevancy score associated with the results from a term-level search.

The term-level search produces a Yes or No binary option similar to the database's WHERE clause. The basic idea for this kind of search is that the results are binary: the query results are fetched if the condition is met; otherwise, it returns none if the condition fails.

Although the documents have a score associated with them, the scores really don't matter. The documents are returned if they match the query but not with relevancy. In fact, we can run the term-level queries with a constant score. They can be cached by the server,

thus, gaining a performance benefit should the same query be rerun. The traditional database search is like this sort.

9.1.1 Term-level queries are not analyzed

One important characteristic of term-level queries is that the queries are not analyzed (unlike full-text queries). The terms are matched against the words stored in the inverted index without having to apply the analyzers to match the indexing pattern. This means that the search words must match with the fields indexed in the inverted index.

For example, if you search for *Java* in a title field using a term-level query, chances are the documents won't match. The reason for this is that during the indexing process, assuming we have a standard analyzer in action, the word *Java* gets converted to a lowercase *java* and gets inserted into the inverted index. Because term-level queries are not analyzed, the engine tries to match the search word *Java* with the word in the inverted index, *java*, hence, the match fails. We can return the same query (with the capitalized *Java*) if we use a keyword type instead (we will go over the usage and explanation for this shortly, so hang tight).

The term-level queries are, hence, suitable for keyword searches, not text-field searches because we know any field identified as a keyword is not analyzed during the indexing process. The field is added to the inverted index without carrying out the analysis on it. Like the keywords, the numerics, Booleans, ranges, etc., are not analyzed and directly added to the respective inverted indices.

Let's take a simple example of the movie, "The Godfather." Figure 9.1 pictorially demonstrates the indexing and term-level search. As you'll see, the standard analyzer doesn't find a hit because "The Godfather" doesn't exist as a single token stored in the inverted index (it was split into two tokens by the analyzer). Similarly, using just *Godfather* as a search word in the term-level query doesn't return any results either because, again, the word *Godfather* does not match the lowercase *godfather*.

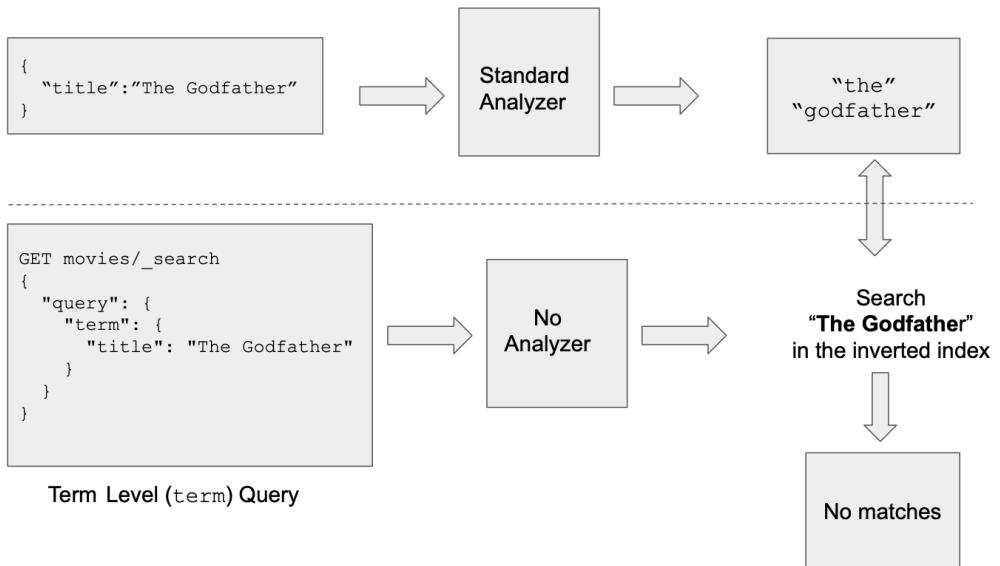


Figure 9.1 Indexing and term-level searching for the movie, “The Godfather”.

As the figure shows, there are two processes: indexing the document and searching for the document. If the field is a text field, assuming the standard analyzer is applied, the title is broken into two tokens and lowercased [“*the*” “*godfather*”] during the indexing process.

On the other hand, during the term-level search, the search terms are passed as is, without any text analysis. If the term-level query searches for “*The Godfather*”, the engine attempts to search for the exact string “*The Godfather*” in the inverted index.

We can still run term-level queries on text fields, although it’s not advisable on fields with lengthy text. If the text has enumerations like days of a week, movie certificates, or gender, etc., we can also use term-level queries. If we are indexing gender such as Male and Female, the term-level queries must use “*male*” and “*female*” in order to successfully return any results because of the standard analyzer’s activity during the indexing process. The takeaway is that term-level queries search for *exact* words.

Elasticsearch exposes a handful of term-level queries, which include term, terms, IDs, fuzzy, exists, range, and others. In the next section, we will go over a few important ones. In the rest of the chapters, we will go over these with hands-on examples.

NOTE *Movie sample data from the last chapter*

Because we already indexed movie sample data in the last chapter, we will build all our queries based on that movie data. For completeness, the source code for this chapter provides the steps to index the movie mapping and sample data.

9.2 Term queries

The term query fetches the documents that exactly match a given field. The field is not analyzed, instead it is matched against the value that's stored as is during the indexing in the inverted index. For example, using our movie dataset, if we were to search an R-rated movie, we can develop a term query as shown in the following listing.

Listing 9.1 Fetching movies with a rating

```
GET movies/_search
{
  "query": {
    "term": { #A
      "certificate": "R"
    }
  }
}
```

#A: The term query declaration

The name of the query (`term` in this case) identifies that we are about to perform a term-level search. The object expects the field (`certificate`, in this case) and the search value. Keep in mind, the certificate is a `keyword` data type hence, during the indexing process, the value "R" was not processed by any analyzer (actually it's a `keyword` analyzer which doesn't alter the case) hence it's stored as is.

If you run this query, you'd get all R-rated movies (14 in our sample data set are R-rated). These are wrapped up in the return JSON response. In the next section we will observe the effect of running term-level search on `text` fields (instead of `keyword` types).

9.2.1 Term queries on text fields

Let's see what happens if we change the query with the rating value to `r` from `R` by lowercasing our search criteria (such as `"certificate": "r"`). To our surprise, we notice this query didn't get any results. Can you guess the reason?

If you recall from the *chapter 7 on Text Analysis*, Elasticsearch analyzes text fields during indexing as well as when searching. As the `certificate` field is a `keyword` type, so the field never goes through the analysis process. This means that it will always be matched with the contents of the inverted index. When indexing the document, the `certificate` value "R" is never tokenized or passed through by filters; hence, it's inserted into the inverted index as is.

The other side of the coin is searching: Term queries do not get analyzed as well. Unlike a standard tokenizer that tokenizes the query's field into multiple tokens and lowercases them, the query field remains as is. If you use `R`, it is considered as `R` because no lowercase conversion (via the standard tokenizer) is applied behind the scenes. Therefore, when we search for a certificate as lowercased (`r`, for example), unfortunately, there are no matches (`R` was indexed not `r`), so there is no result.

This brings up an important point to consider when working with term queries: term queries are not suitable when working with text fields. Although nothing stops you from

using them, they are intended to be used on non text fields like keywords, numericals, and dates.

For whatever the reason, if you want to use a term query on a `text` field, make sure the `text` field is indexed like an enumeration or a constant. For example, an order status field with CREATED, CANCELLED, FULFILLED states can be a good candidate to use a `term` query though the field was a `text` field.

However, should the `text` fields have been populated with unstructured text like non-enumeration styled values, we will not get the expected results when `term` queries are run on them. Let's checkout an example of what happens when we run a term query on a text field in the next section.

9.2.2 Example: Applying a term query on a movie's title

Let's see what happens if we search a text field called `title` using a term query. In listing 9.2, we search for "The Godfather" in the title of the movie using the term query. (If you remember our movie mappings, the `title` field has an explicit text field mapping set.)

Listing 9.2 Using a term-level query on a text field

```
GET movies/_search
{
  "query": {
    "term": {
      "title": "The Godfather"
    }
  }
}
```

Running the code in the above listing, we receive no results (refer to figure 9.1 for pictorial illustration). The reason for this is that the `title` field is a text field, meaning that the field has undergone an analysis process and is stored in the index prior to the search. "The Godfather" is broken down into tokens and stored as lowercase tokens (because we are using the standard analyzer by default) with `["the", "godfather"]` in the inverted index. The search queries are not analyzed for term queries; they instead take the word as is and compare it against the inverted index. In this case, the "The Godfather" query criteria does not match with the tokens (`the, godfather`) for the `title` field.

Also, rerunning the query using "the godfather" does not return any results (try running the query, lowercasing the title like this). The term query tries to match the exact value, "the godfather", which is not in the inverted index (remember, it's tokenized and stored as two words: `the` and `godfather`). However, searching on the word "godfather" returns in the results because the word `godfather` was analyzed and inserted into the inverted index during the data indexing and hence a match is found.

The takeaway is that we need to run the `term` query over a non text field. Should you want to use the term query to search text fields, make sure the text field has data in the form of enumerations or constants.

9.2.3 Shortened term-level queries

The `term` query you see in the listing 9.1 and 9.2 are shortened versions of term queries. While it is convenient to write the shortened version, we should take a moment to see the original full version, shown in the listing (9.3) below:

Listing 9.3: Full version syntax of a query

```
GET movies/_search
{
  "query": {
    "term": {
      "certificate": {#A
        "value": "R",#B
        "boost": 2 #C
      }
    }
  }
}
```

#A: The certificate field has values enclosed in an object

#B: The certificate's search criteria

#C: In addition to the value, we can also provide other parameters like boost

As the code shows, the certificate expects an object with value and other parameters. The value, which was mentioned at the same level as the field in the shortened version, is provided one level down further than the field. The enclosed object can also host other attributes such as boost, as the query shows in the above example.

While the full version has further features to add to a query, the shortened version is pretty straightforward and is used extensively if the query is simple and carries no further tweaking. We will be using shortened versions pretty much throughout the book unless we are interested in working the other parameters.

So far, we looked at searching words against a single field using a `term` query. The `term` query will look at exact matches for a single word like `"certificate": "R"`. However, what if we want to search multiple values in a single field? How can we search both R-rated and PG-13 movies, for example, on the `certificate` field? That's where the `terms` query comes into the picture, which is discussed next.

9.3 Terms queries

As the name suggests, the `terms` (note down the plural) query searches multiple criteria against a single field. That is, we can throw in all the possible values of the field that we would like the search to be performed. Say, we want to search for all movies with multiple content ratings, like PG-13, 15, 18, R-rated. We use `terms` query for this purpose as the following listing 9.4 below demonstrates.

Listing 9.4 Searching for multiple search criteria in a single field

```
GET movies/_search
{
  "query": {
    "terms": { #A
      "certificate": ["PG-13","R"] #B
    }
  }
}
```

#A: The terms query expects an array of search criteria.

#B: Multiple search criteria against a single field

The terms query expects a list of search words to be queried against a field, passed in as an array to the `terms` object. The array values will be searched against the existing documents one by one to fetch matches. Each of the words are matched exactly. In the above listing 9.4, we are searching for all the movies with PG-13 and R ratings in the `certificate` field. The resultant documents would be a combination of all PG-13 and R movies together.

There's a limit of how many terms we can set in that array - a whopping of 65,536 terms. If you need to modify this limit (to increase or decrease it), you can use the index's dynamic property setting to alter the limit: `index.max_terms_count`. The following query in the code listing 9.5 below sets the `max_terms_count` to 10:

Listing 9.5 Resetting the maximum terms count

```
PUT movies/_settings
{
  "index": {
    "max_terms_count": 10
  }
}
```

This setting will restrict a user not to set more than 10 values in the terms array. Remember this is a dynamic setting on the index, so you can change as and when you want it on a live index.

There's a slightly different variant to terms query - a terms lookup query. The big idea is to create the terms array from an existing document's values rather than setting it specifically by us. The next section discusses it in detail with an example.

9.3.1 Terms lookup

So far we were providing the list of values in an array as the search criteria to the terms query. However, terms lookup query is a slight variant of terms query in that it lets the terms set by reading an existing document's field values. It is best understood with an example. We may have to deviate slightly from our movies dataset and create a new index with appropriate schema and index few documents to explain this feature. So, let's do that now.

We create a `classic_movies` index with two properties: `title` and `director`, as the listing 9.6 shows below:

Listing 9.6: Creating a new index

```
PUT classic_movies
{
  "mappings": {
    "properties": {
      "title": { #A
        "type": "text"
      },
      "director": { #B
        "type": "keyword"
      }
    }
  }
}
```

#A: The title field is a text field

#B: Declaring director field as keyword type

As the code illustrates, there is nothing special about this index - except that the notable point is that we are defining the `director` field as a `keyword` type - for no better reason than avoiding complexity.

Now that we have the index prepared, let's drop in a few movies, as the listing 9.7 shows:

Listing 9.7: Indexing movies - with two of Steven Spielberg's movies

```
PUT classic_movies/_doc/1
{
  "title": "Jaws",
  "director": "Steven Spielberg"
}

PUT classic_movies/_doc/2
{
  "title": "Jaws II",
  "director": "Jeannot Szwarc"
}

PUT classic_movies/_doc/3
{
  "title": "Ready Player One",
  "director": "Steven Spielberg"
}
```

The documents are self explanatory. Now that we have indexed these three documents, let's go back to terms lookup query discussion. Say we wish to fetch all movies directed by the director like Speilberg. However, we wouldn't want to construct a terms query with the terms upfront, instead we will let the query know to pick up the values of the terms from a document. That is, we let the terms query lookup the criteria from the field values of a document rather than providing them directly. The listing 9.8 below does exactly this:

Listing 9.8 Terms lookup search

```
GET classic_movies/_search
{
  "query": {
    "terms": { #A
      "director": { #B
        "index": "classic_movies", #C
        "id": "3", #D
        "path": "director" #E
      }
    }
  }
}
```

#A: The terms query (with a twist!)

#B: The field that we are interested in searching against

#C: The index denotes name of the index where the document resides

#D: Field name which makes up the terms for the query

#E: The search Field in the current document

The code listing requires a bit of explanation: we are creating a `terms` query with the `director` being the field against which multiple search terms are arranged. In a usual terms query, we would've provided an array with all the list of names. However, here we are asking the query to look up the values of the director from *another* document instead: the document with id as 3.

The document with this ID 3 is expected to be picked up from the `classic_movies` index as the `index` field mentions in the query. And of course the field to fetch the values is called `director` and is noted as `path` in the above code listing. Running this query will fetch two documents that were directed by Spielberg.

The terms lookup query helps build a query based on values obtained from another document than a set of values passed in the query. It has a greater flexibility when constructing the query's terms: we could easily swap the index with any other index to pick a document from.

For example, say we have an index called "`movie_search_terms_index`" with a handful of documents of search terms (say document 1 with director terms, document 2 with actors terms and so on). We can then reference this document with director terms from `movie_search_terms_index` in our main query and fetch the results. This way, the main query can be a constant query while the lookup documents can be changed as and when required.

Now that we have a good understanding of the `terms` query, let's jump on to a query type where we fetch documents given a set of IDs, discussed in the following section.

9.1 IDs queries

At times, we may have a set of IDs that we would like from Elasticsearch, for example. The IDs query, as the name suggests, fetches the matching documents given a set of document IDs. It's a much simpler way to fetch the documents in one go. The following listing shows how to retrieve some documents using a list of document IDs.

Listing 9.9 Fetching multiple documents using an `ids` query

```
GET movies/_search
{
  "query": {
    "ids": { #A
      "values": [10,4,6,8] #B
    }
  }
}
```

#A The name of the query

#B Provides the document IDs as an array

This query returns four documents with the corresponding four IDs. Each document that gets indexed has a mandatory `_id` field.

NOTE *The metadata fields (like `_id`) can't be part of the mapping schema*

The metadata fields are not allowed to be part of the schema definition. The `_id` field, along with other metadata fields like `_source`, `_size`, `_routing` etc are part of the metadata fields pack and hence are not allowed to be part of the index mapping exercise.

We can also use a terms query to fetch documents if we have a set of document IDs instead of an IDs query as we saw in the listing 9.4. The following listing (listing 9.10) shows how we can do this.

Listing 9.10 A terms query using a set of IDs to fetch documents

```
GET movies/_search
{
  "query": {
    "terms": {
      "_id": [10,4,6,8]
    }
  }
}
```

Here, we use a terms query, setting the array of document identifiers on the `_id` field as our search criteria. Now, let's look at another term-level query type: the `exists` query.

9.5 Exists queries

Sometimes, I see documents having hundreds of fields in some projects. Fetching all the fields in a response is a waste of bandwidth, and knowing if the field exists before attempting to fetch it is a better precheck. To do that, the `exists` query fetches the documents for a given field if the field exists.

For example, if we run the query in the following listing (9.11), we get a response with the document because the document with the `title` field exists.

Listing 9.11 Running an exists query to check if a field exists

```
GET movies/_search
{
  "query": {
    "exists": {#A
      "field": "title" #B
    }
  }
}
```

#A Defines the query type as exists

#B Provides the field that we want to check in the document

If the field doesn't exist, the results return an empty `hits array` (`hits[]`). If you are curious, try out the same query with a nonexistent field like `title2`, for example, and you'll see an empty hit.

9.5.1 Non existent field check

There's another subtle use case of an `exists` query: when we want to retrieve all documents that don't have a particular field (a nonexistent field). For example, in listing 9.12, we check all the documents that aren't classified as confidential (assuming classified documents have an additional field called `confidential` set to true).

Listing 9.12 Finding documents that are nonconfidential

```
PUT top_secret_files/_doc/1 #A
{
  "code": "Flying Bird",
  "confidential": true
}

PUT top_secret_files/_doc/2 #A
{
  "code": "Cold Rock"
}
GET top_secret_files/_search #B
{
  "query": {
    "bool": {
      "must_not": [
        {
          "exists": {
            "field": "confidential"
          }
        }
      ]
    }
  }
}
```

#A Adds two documents, one with a flag that says confidential

#B Writes a compound query, fetching documents without a confidential field

As the listing shows, we add two documents to the `top_secret_files` index: one of the documents has an additional field called `confidential`. We then write an `exists` query in a

`must_not` clause of a `bool` query to fetch all the documents that are not categorized as confidential. We will go over compound queries in Chapter 11: Advanced search.

There are times we wish to work with data that falls in a certain predefined range: fetching movies from last month or sales in a quarter or highest grossers etc. These queries are grouped under the range queries which we will go over in the next section.

9.6 Range queries

Oftentimes, we need a set of data that falls within a range: flights that were delayed between certain dates, profit sales on a particular day, pupils with average height in a class, and so on. Elasticsearch provides a range query for these types of inquiries.

The range query returns the documents for a range in a field. The query accepts lower and upper bounds on the field. If we need to fetch all the movies for a rating between 9.0 and 9.5, for example, we can execute the range query in the following listing 9.13:

Listing 9.13 Fetching movies with a specified range of ratings

```
GET movies/_search
{
  "query": {
    "range": {
      "rating": {
        "gte": 9.0,
        "lte": 9.5
      }
    }
  }
}
```

In the listing, the range query fetches the movies that fall within the specified `rating` brackets. The `rating` field is an object that accepts bounds, which are defined as operators. The following table shows the operators that we can use to specify the range.

Table 9.1 Operators in a range query

Operator	Meaning
gt	Greater than
gte	Greater than or equal to
lt	Less than
lte	Less than or equal to

We use range queries for searching across a range of dates or numbers. If you want to fetch all the movies *after* 1970, for example, you simply stitch the query as the next listing (9.14) shows.

Listing 9.14 Fetching movies after 1970 using a range query

```
GET movies/_search
{
  "query": {
    "range": {
      "release_date": {
        "gte": "01-01-1970"
      }
    }
  },
  "sort": [
    {
      "release_date": {
        "order": "asc"
      }
    }
  ]
}
```

The `release_date` declares a `gte` operator with the search requirement; in this case, the year 1970. Notice that we also *sort* the movies in ascending order on the release date using the `sort` attribute in the query. The results of the movies returned are therefore oldest to newest.

Because we are discussing range queries, let's use this opportunity to go over date math in a range query. We discuss this in the following section.

9.6.1 Range queries with data math

Elasticsearch supports sophisticated data math in queries. For example, we can ask the engine questions like:

- Fetch the book sales a couple of days back (current day minus two days).
- Find the access denied errors in the last 10 minutes (current hour minus 10 minutes).
- Get the tweets for a particular search criteria from last year.

Elasticsearch expects a specific data expression that deals with data math. An anchor date followed by `||` is the first part of the expression, appended with the time we want to add or subtract from the anchor date: anchor date plus or minus a set number of hours, days, or years. For example, to fetch movies two days after a specific day (say, 1st of January 2022), we form the expression as

```
01-01-2022 ||+2d
```

The first part of the expression (01-01-2022) is called an *anchor date*, whereas the `||` indicates that the anchor date is being manipulated either by adding or by subtracting a certain number of time units (minutes, seconds, years, days, etc.). For example, we want to

fetch the movies released a couple of days back, so we set the anchor date as today (as of writing, today's date is 01-03-2022) and subtract two days as this listing 9.15 shows:

Listing 9.15 Fetching movies released two days ago

```
GET movies/_search
{
  "query": {
    "range": {
      "release_date": {
        "lte": "01-03-2022||-2d" #A
      }
    }
  }
}
```

#A The anchor date followed by || minus two days

As you can see in the listing, the range query's `lte` operator takes a date value expressed as date math. In this case, `01-03-2022` is the anchor date, and we subtract two days from it.

Instead of mentioning the current date specifically, Elasticsearch lets us use a specific keyword: `now`. The `now` represents the current date. For example, using `now-1y` sets the date to one year back as the next listing shows.

Listing 9.16 Fetching movies from last year

```
GET movies/_search
{
  "query": {
    "range": {
      "release_date": {
        "lte": "now-1y" #A
      }
    }
  }
}
```

#A Fetches all the movies from last year: the current date (represented as now) minus one year

We built the release date expression using `now` and subtracting one year from it. Elasticsearch has a dictionary of these letters: `y` for years, `M` for months, `w` for weeks, `d` for days, `h` for hours, `m` for minutes, `s` for seconds, and so on. There are a lot of options to manipulate dates in Elasticsearch, so I advise you to consult the documentation.

We looked at range queries in this section - the type of queries which helps fetch a set of documents given a range parameters. In the next section, we look at wildcard queries - the queries will not let you down if you provide a partial search criteria as it uses wildcards to build up the expression. We discuss Elasticsearch details with wildcards in the next section.

9.7 Wildcard queries

Wildcard queries, as the name implies, let you search on words with missing characters, suffixes, and prefixes. At times, we want to use wildcards to do a search, for example, all possible combinations of movies with titles ending with `father` or `god`, or even missing a

single character like *god?ather*, for instance. This is where we use a wildcard query. The wildcard query accepts an asterisk (*) or a question mark (?) in the search word. The following table describes these characters.

Character	Description
*	Lets you search for zero or more characters.
?	Lets you search for a single character.

Let's search for documents where the movie title starts with *god*. The listing in 9.17 shows the query in action:

Listing 9.17 Wildcard search in action

```
GET movies/_search
{
  "query": {
    "wildcard": { #A
      "title": {
        "value": "god*" #B
      }
    }
  }
}
```

#A The wildcard query type

#B Searches the value with a wildcard

We should see three movies (*Godfather*, *Godfather II*, and *City of God*) returned for this wildcard query. Of course, any movie (*Godzilla*, *God's Waiting List*, etc.) can also be fetched because we expect all titles with a prefix of *god*.

NOTE *Wildcard queries on text fields*

We run the query in listing 9.15 on the `title` field, which is a `text` data type. Because term-level queries are not analyzed, we use a lowercase `god`. Also, the `title` field was indexed with the standard analyzer that has, by default, lowercased the word. Should you want to use the `keyword` field instead of a `text` field, change the `title` to `title.original` (the `title.original` is defined as `keyword` in our `movies` schema) and run with the value “The God*”.

However, if you omit “The” from “The God*” and run the query, you can't find any results. The reason is that `title.original` is a keyword typed field and the value is persisted during index without any text analysis.

We can tweak our queries by placing wildcards anywhere in the word. For example, the query “g*d” fetches two movies from our stash: *The Good*, *the Bad and the Ugly* and *City of God*. If you want to find out the match of a given query criteria in a return document, you

can use highlighting (we discussed highlighting in the last chapter). The following listing shows this approach.

Listing 9.18 Searching with a wildcard in a word

```
GET movies/_search
{
  "_source": false,
  "query": {
    "wildcard": {
      "title": {
        "value": "g*d" #A
      }
    },
    "highlight": { #B
      "fields": {
        "title": {}
      }
    }
  }
}
```

#A The wildcard in between the letters of a word

#B The highlight block to let us visualize the results

The output of this program shows us that two movies matched. The following code snippet indicates this.

```
"title": [ "The <em>Good</em>, the Bad and the Ugly" ]
"title": [ "City of <em>God</em>" ]
```

We use the ? wildcard only if we want to match one character; for example, "value": "go?ather" searches for all the words that match the third character at the wildcard's position. Of course, you can club multiple ? characters if you want; for example, "g???ather".

9.7.1 Expensive queries

There are a few queries that can be expensive to run by the engine due to the nature of how we implement them. The `wildcard` query is one of them. The others are the range, prefix, fuzzy, regex, and join queries as well as others. Furthermore, using one of these queries occasionally might not impact server performance, but overusing these expensive queries will perhaps destabilize the cluster, leading to bad user experiences.

Elasticsearch allows us to execute these expensive queries on the cluster, thus leaving the discretion up to us. However, if we want to put a stop to the execution of such expensive queries on the cluster, there's a setting that we can turn off: setting the `allow_expensive_queries` attribute to `false`. We should set this on the cluster settings as the following listing 9.19 shows:

Listing 9.19: Turning off the expensive query setting

```
PUT _cluster/settings
{
  "transient": {
    "search.allow_expensive_queries": "false"
  }
}
```

By switching off the `allow_expensive_queries` setting, we are protecting the cluster from overload. Do note, however, that if we set `allow_expensive_queries` to false, the wildcard queries noted as one of the expensive queries will not be executed. You'll get the following error if you try execute any of these:

```
"reason" : "[wildcard] queries cannot be executed when 'search.allow_expensive_queries' is
set to false."
```

Wildcards will fetch the results with missing characters in a word or a sentence, however, there might be times when we wish to query words with a prefix - that's when the prefix queries come into picture, which are discussed in the following section.

9.8 Prefix queries

At times we might want to query for words using a prefix, like *Leo* for Leonardo or *Mar* for Marlon Brando, Mark Hamill, or Martin Balsam. We can use the prefix query for fetching records that match the beginning part of a word (a prefix). The prefix query in the following listing (9.20) does exactly that:

Listing 9.20 Using a prefix query

```
GET movies/_search
{
  "query": {
    "prefix": { #A
      "actors.original": {
        "value": "Mar" #B
      }
    }
  }
}
```

#A Specifies the prefix type query

#B Queries for the words that start with Mar

The above query fetches three movies with the actors Marlon, Mark, and Martin when we search for the prefix *Mar*. Do note that we are running the `prefix` query on `actors.original` field which `keyword` datatype.

NOTE *Prefix query is an expensive query*

The prefix query is also an expensive query that can destabilize the cluster at times. See section 9.7.1 to take measures on how to prevent an overloaded cluster and section 9.8.2 to speed up the prefix queries

By the way, as we discussed at the beginning of this chapter, we don't need to add an object consisting of value at the field block level. Instead, you can create a shortened version as the next listing demonstrates for brevity:

Listing 9.21 Shortening the prefix query usage

```
GET movies/_search
{
  "query": {
    "prefix": {
      "actors.original": "Leo"
    }
  }
}
```

As we wish to find out the matching fields in the results, we will highlight the results by adding highlights to the query. We add a highlight block to the prefix query. This accentuates one or more fields that match as the following listing in 9.22 shows.

Listing 9.22 Prefix search with highlighting

```
GET movies/_search
{
  "_source": false,
  "query": {
    "prefix": {
      "actors.original": {
        "value": "Mar"
      }
    },
    "highlight": { #A
      "fields": {
        "actors.original": {}
      }
    }
  }
}
```

#A Highlights the matched actors in the results

Because we don't want the source to be returned in the response ("_source": false), the results in the following snippet highlight where the prefix matched with the word:

```

"hits" : [{  
    ..  
    "highlight" : {  
        "actors.original" : ["<em>Marlon Brando</em>"]  
    },  
    {  
        ..  
        "highlight" : {  
            "actors.original" : ["<em>Martin Balsam</em>"]  
        },  
        {  
            ..  
            "highlight" : {  
                "actors.original" : ["<em>Mark Hamill</em>"]  
            }  
        }]  
}

```

We discussed earlier that the prefix queries exert additional computation strain when running the queries. Fortunately, there is a way to speed up such painstakingly ill-performant prefix queries - discussed in the next section.

9.8.1 Speeding up prefix queries

This is because the engine has to derive the results based on a prefix (any lettered word). The prefix queries, hence, are slow to run, but there's a mechanism to speed them up: using the `index_prefixes` parameter on the field.

We can set the `index_prefixes` parameter on the field when developing the mapping schema. For example, the mapping definition in listing 9.23 sets the `title` field (remember, the `title` field is a `text` data type) with an additional parameter, `index_prefixes`, on a new index, `boxoffice_hit_movies`, that we are creating for this exercise.

Listing 9.23 A new index with the `index_prefixes` parameter

```

PUT boxoffice_hit_movies #A
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "index_prefixes": {} #B
      }
    }
  }
}

```

#A: Creates a new index with just one property, `title`
#B: Sets the `index_prefixes` parameter on the `title` field

As you can see from the code in the listing, the sole `title` property includes an additional property, `index_prefixes`. This indicates to the engine that, during the indexing process, it should create the field with prebuilt prefixes and store those values. For example, when you index a new document as shown in the following snippet:

```
PUT boxoffice_hit_movies/_doc/1
{
  "title": "Gladiator"
}
```

Because we set `index_prefixes` on the `title` field in listing 9.23, Elasticsearch indexes the prefixes with a minimum character size of 2 and a maximum character size of 5 by default. This way, when we run the prefix query, it doesn't need to calculate the prefixes. Instead, it picks them up from storage.

Of course, we can change the default `min` and `max` sizes of the prefixes that Elasticsearch tries to create during indexing for us. This is done by tweaking the sizes of the `index_prefixes` object as the following listing 9.24 demonstrates.

Listing 9.24 Custom character length settings for `index_prefixes`

```
PUT boxoffice_hit_movies_custom_prefix_sizes
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "index_prefixes": {
          "min_chars": 4, #A
          "max_chars": 10 #B
        }
      }
    }
  }
}
```

#A: Sets the minimum number of characters for the prefix

#B: Sets the maximum number of characters for the prefix

In the listing, we ask the engine to precreate prefixes with a minimum and maximum character length of 4 and 10 letters, respectively. Note that `min_chars` must be greater than 0, and `max_chars` should be less than 20 characters. This way, we can customize the prefixes that Elasticsearch should create beforehand during the indexing process.

9.9 Fuzzy queries

Spelling mistakes during a search are common. We may at times search for a word with an incorrect letter or letters; for example, searching for `rama` movies instead of `drama` movies. The search can correct this query and return "`drama`" movies instead of failing. The principle behind this type of query is called *fuzziness*, and Elasticsearch employs fuzzy queries to forgive spelling mistakes.

Fuzziness is a process of searching for similar terms based on the Levenshtein distance algorithm (also referred to as the edit distance). The Levenshtein distance is the number of characters that need to be swapped to fetch similar words. For example, searching for "`cake`" can fetch "`take`", "`bake`", "`lake`", "`make`", and others if a fuzzy query is employed with `fuzziness` (edit distance) set to 1. The following query (listing 9.25) should return all the drama genres because applying a fuzziness of 1 to "`rama`" results in "`drama`".

Listing 9.25 A fuzzy query in action

```
GET movies/_search
{
  "query": {
    "fuzzy": {
      "genre": {
        "value": "rama",
        "fuzziness": 1
      }
    }
  },
  "highlight": {
    "fields": {
      "genre": {}
    }
  }
}
```

In this example, we use the edit distance of 1 (one character) to fetch similar words. You can also try removing a character from the middle of the word too like *dama* or *dram* and so forth. These all result in positive returns when fuzziness is set to 1.

If you drop one more letter (for example, "value": "ama" with fuzziness set to 1), the fuzzy query in listing 9.25 does not return any results. Because we are missing two letters, we need to set the edit distance to 2 to solve this issue. The listing in 9.26 below shows this approach:

Listing 9.26 Fuzzy query with two letters missing in a word

```
GET movies/_search
{
  "query": {
    "fuzzy": {
      "genre": {
        "value": "ama", #A
        "fuzziness": 2 #B
      }
    }
  }
}
```

#A: A word with missing letters

#B: Setting fuzziness to 2 is to forgive two letter substitutions/modifications

This might be a clumsy way of handling things because sometimes you wouldn't know if the user has mistyped one letter or a few letters. This is the reason Elasticsearch provides a default setting for fuzziness: the AUTO setting. If the fuzziness attribute is not supplied, the default setting of AUTO is assumed. The AUTO setting deduces the edit distance based on the length of the word as shown in the following table.

Table 9.2 Using the AUTO fuzziness setting

Word length in characters	Fuzziness (edit distance)	Explanation
0 to 2	0	If the word is shorter than two characters, fuzziness is not applied. This means the misspelled words cannot be corrected.
3 to 5	1	If the length of the word is between 3 to 5 characters in length, an edit distance of 1 is applied.
More than 5	2	If the length of the word is longer than 5 characters, an edit distance of 2 is applied.

Sticking to the default setting of `AUTO` for the `fuzziness` attribute is preferred unless you know exactly the use case at hand. That's pretty much it for the term-level queries. In the next chapter, we will look at the full-text search queries on text fields that produce relevant results.

NOTE Fuzzy vs wildcard query

Unlike a wildcard query where a wildcard operator such as `*` or `?`, fuzzy query doesn't use operators and instead goes with the task of fetching similar words using Levenshtein edit distance algorithm.

Let's wrap up here: we learned a lot about term-level searching in this chapter. While term-level search helps finding out answers amongst the structured data, the real power of a search engine lies in its ability of searching unstructured data. In Elasticsearch, the unstructured data is associated with full-text searching - searching text fields and yielding results with relevance scores. We will go over these full-text queries in detail in the next chapter. Stay tuned.

9.10 Summary

- The term-level searching is carried on stricter data such as numbers, keywords, Booleans, and dates
- A term-level search produces a binary result: the search leads to a result or none. There is no *likely match* scenario.
- Term-level queries are not analyzed, meaning that when applied to text fields undergoing text analysis during indexing, they can yield incorrect or no results.
- There are a multitude of term-level search queries: `term`, `terms`, `prefix`, `range`, `fuzzy`, and a few others.
- The `term` query searches for a single term against a field while the `terms` (plural) queries search multiple values against a single field.

- Range queries help search through data within a set of bounds, for example, searching for crimes happened in London in the last month.
- Wildcard query uses * or ? operators to fetch results
- Fuzziness is the mechanism of employing Levenshtein's edit distance to fetch similar looking words (or ability to forgive spelling mistakes). Elasticsearch employs fuzzy query to support user's spelling inconsistencies.
- Prefix queries help retrieve results given a prefix (no need to specify a wildcard operator). As prefix operations are expensive to be executed on a live index, we can ask Elasticsearch to pre-create them during the time of index to avoid finding them during a live query phase.

10

Full-text search

This chapter covers

- Overview of full text queries
- Working through match queries
- Looking at match phrase, multi-match, and other queries
- Looking at query strings and simple query string queries

In the last chapter, we looked at term-level searching, which is the mechanism that we use to search structured data. Although a structured data search is important, the power of modern search engines vests in an efficient and effective way to run when we search unstructured data. Elasticsearch is one such modern search engine that stands as a front runner in searching unstructured data with relevance.

Elasticsearch provides the capability to search unstructured data in the form of full text search queries. A full text search is all about relevancy: fetching the documents that are relevant to the user's search. For example, when searching for the word *Java* in an online bookstore, one shouldn't expect to receive details about the Indonesian island of Java or the wet pressed coffee grown on this island.

In this chapter, we will go over the mechanics of searching unstructured data by employing full text search APIs. Elasticsearch provides a handful of full text queries in the form of match, query string, and others. As all match queries, especially the match query, are the most common queries one would use when working with full text, we will dedicate a good chunk of the chapter for various match queries. We will also work through query string searches, which is equivalent to using an URI request search method but with the request body similar to a Query DSL.

10.1 Overview

When we search on a retailer website such as Amazon or eBay, we see results close to what we are looking for. Should these results not be what we expect, we express our frustration and vow never to return to that site or application, don't we? A user's search experience is a crucial part of retaining a happy customer.

Relevance is all about how appropriate the search results are and how closely related to what the user is searching for. Elasticsearch thrives to produce relevant results with speed and accuracy by employing sophisticated relevance algorithms.

When we talk about relevance, there's two measures that usually spring up: precision and recall. Relevance is measured using these two factors and understanding them at a high level is important, though not crucial.

10.1.1 Precision

Precision is measured as the percentage of relevant documents in the overall number of documents returned. When a query returns results, not all of the results are directly related to the query. There may be a few non relevant documents in the results.

For example, if we are searching for a 4K television from a particular manufacturer (say, for example, LG), let's say we obtain 10 results. However, not all of these results are relevant: a few of them might be 4K cameras (e.g., two in figure 10.1) and a couple of them could be projectors because LG produces these as well. Let's look at this pictorially in figure 10.1.

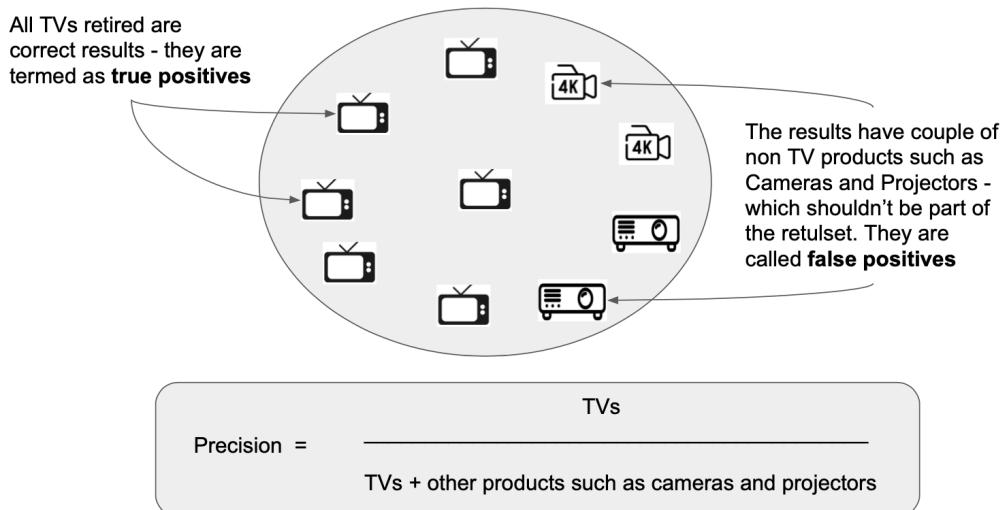


Figure: 10.1 A precision example in action, returning search results for a 4K television

As figure 10.1 demonstrates, the result consists of true positives (relevant documents) as well as false positives (irrelevant documents). Precision is about how many of the retrieved documents are relevant. In this case, we know 6 documents are TVs out of 10 resulting documents, so 6 documents are relevant, but the remaining 4 are irrelevant. Hence, the precision is calculated as

$$\text{precision} = 6 / 10 \times 100\% = 60\%$$

What this tells us is that only 60% of the documents returned are relevant. Some of the documents that are not directly related to the query (irrelevant documents) also appear in the results bag for various reasons.

10.1.2 Recall

Recall is the other side of the coin. It measures how many documents that were returned are relevant. For example, there may have been a few more relevant results (more TVs) that weren't returned as part of the results set. The percentage of relevant documents that were retrieved is called *recall*. Figure 10.2 defines recall using a pictorial representation.

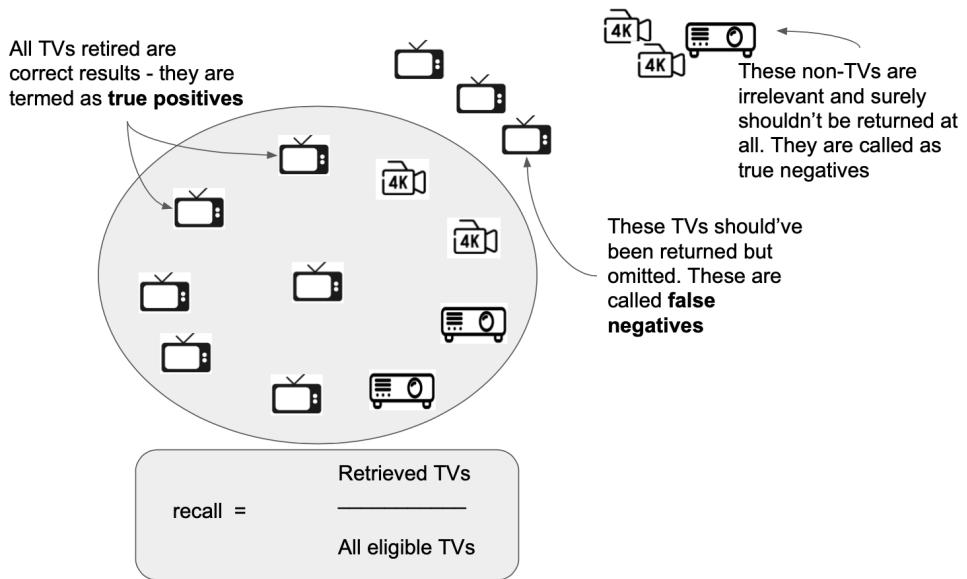


Figure: 10.2 The recall measure in action for our 4K television example search

As the figure 10.2 shows, we have three more TVs that fell into the search bucket but were never returned. These are called *false negatives*. On the other hand, there are a few products like cameras and projectors that weren't returned, which are genuinely irrelevant and, hence, not expected to be part of the result. These are termed as *true negatives*. In the current scenario, recall is calculated as

$$\text{Recall} = 6 / (6+3) \times 100\% = 66.6\%$$

Ideally, we want precision and recall to be a perfect match with no omissions (no relevant documents omitted). However, this is pretty much impossible because these two measures always work against each other. They are indeed inversely proportional to each other: the higher the precision (number of best match documents), the lower the recall (the number of documents returned). Figure 10.3 shows us the inverse relationship between these two factors.

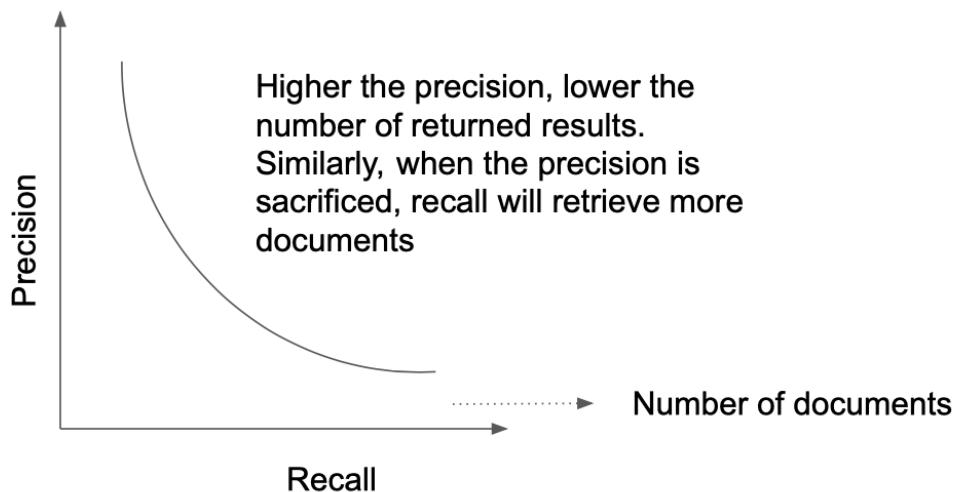


Figure 10.3 Precision and recall are always at odds with each other.

We need to make sure that the returned results strike a balance between the precision and recall strategies. Figure 10.4 summarizes precision and recall measures.

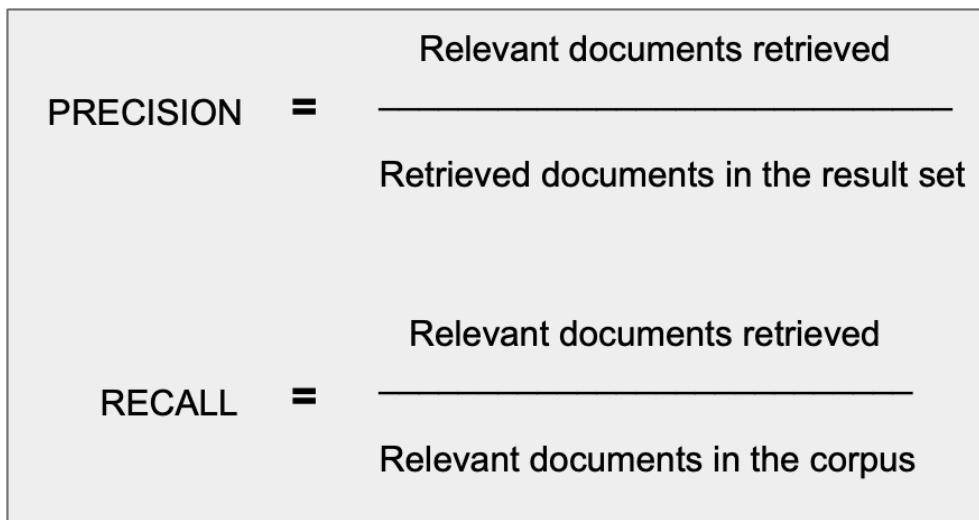


Figure 10.4 The formulas for precision and recall

We can tune the results for precision and recall by employing various strategies when designing and executing queries. We can use match queries, filters, and boost to tweak precision and recall to fine tune the balance. In the remainder of this chapter, we will not directly work with tuning these parameters, but we will work on the queries to see how the results are affected. If you are interested in learning a bit more about precision and recall in the information retrieval context, I suggest you go over beautifully written wikipedia article:

https://en.wikipedia.org/wiki/Precision_and_recall

Now that we understand the concept of relevance and the controlling factors (precision and recall), the rest of the chapter is dedicated to working with the full text queries. First, though, let's get the sample data sorted.

10.2 Sample data

We will work with a fictional book store in this chapter, where we will index a set of 50 technical books into an index named `books` by invoking the `_bulk` API. We are not tweaking the mapping for this part of the sample data, so you can go ahead and index the books as is. The data for the books is available on my GitHub page here:

<https://github.com/madhusudhankonda/elasticsearch-in-action/blob/f334df2dd5f10acc15bfc745f580ec9be0b129d2/datasets/books.txt>

Once you index the books, we can go over the examples. For this, use the script available on my GitHub page here:

https://github.com/madhusudhankonda/elasticsearch-in-action/blob/f334df2dd5f10acc15bfc745f580ec9be0b129d2/kibana_scripts/ch10_full_text_queries.txt

Elasticsearch provides a handful of full text queries such as match and its sister types, intervals, common terms, query string, simple query string, and others. Let's look at these, starting with the match queries in the next section. Because each of the query types have a lot of implementation details, I used separate sections to describe each for ease of flow. The first query we will look at is the `match_all` query. This query returns everything as discussed in the next section.

10.3 Match all (`match_all`) queries

As the name suggests, the match all (`match_all`) query fetches all the documents available in the index. Because this query is expected to return all the available documents, it is the perfect partner for honoring 100% recall.

10.3.1 Building the `match_all` query

We form the query with a `match_all` object, passing it no parameters. The sample code in the following listing demonstrates how to build the `match_all` query.

Listing 10.1 Fetching all documents using `match_all`

```
GET books/_search
{
  "query": {
    "match_all": { } #A
  }
}
```

#A The `match_all` query with no parameters

This query returns all documents available in the `books` index. The notable point is that the response indicates each of the books with a score of 1.0 by default as the following code snippet shows:

```
{
  "max_score" : 1.0,
  ...
  "hits" : [
    {
      "_index" : "books",
      "_type" : "_doc",
      "_id" : "2",
      "_score" : 1.0,
      "_source" : {
        "title" : "Effective Java",
        ...
      }
    },
    {
      ...
      "_score" : 1.0,
      "_source" : {
        "title" : "Java: A Beginner's Guide",
        ...
      },
      ...
    }
  ]
}
```

}]

That said, we can boost the score if needed by simply amending the query. The next listing shows the code for this.

Listing 10.2 Boosting the query with a predefined score

```
GET books/_search
{
  "query": {
    "match_all": {
      "boost": 2 #A
    }
  }
}
```

#A: The match_all query sets a score of 2 for all returned documents.

As you can see, we add a `boost` parameter to the query. This parameter returns all the documents with a boosted score.

10.3.2 Short form of a match_all query

We wrote a `match_all` query with a query body (see listing 10.1), however, providing the body is redundant. That is, the same query can be rewritten in a shorter format like this:

```
GET books/_search
```

Behind the scenes, Elasticsearch executes a `match_all` query with a default `boost` of 1 when the query body is not provided. Unless you want to change the boost value, you can invoke the search endpoint without a body.

10.4 Match none (match_none) queries

While the `match_all` query returns all the results from an index or multiple indices, the opposite query, called `match_none`, returns no results. The following listing shows the syntax for the `match_none` query.

Listing 10.3 The match_none query

```
GET books/_search
{
  "query": {
    "match_none": {} #A
  }
}
```

#A The query that matches nothing

I am providing this query for completeness though I don't see the real benefit of using `match_none` (at least I haven't found its usefulness yet).

In the next section, we'll learn about the versatile query: the `match (match)` query. Most of the queries that we often use when working with Elasticsearch tend to use a `match` query in one form or another.

10.5 Match queries

The `match` query is the most common and powerful query for multiple use cases. It is a full text search query returning the documents that match the specified criteria. The `match` query can be improvised for querying a multitude of options.

10.5.1 Format of a match query

Let's first look at the format of the `match` query as this snippet shows:

```
GET books/_search
{
  "query": {
    "match": { #A
      "FIELD": "SEARCH TEXT" #B
    }
  }
}
```

#A: The type of the query is a match query.

#B: The query expects a criteria to be specified in the form of a name-value pair.

As you can see in the snippet, the `match` query expects the search criteria to be defined in the form of a field value. The field can be any of the text fields present in a document, whose values are to be matched. The value can be a word or multiple words, given either as uppercase, lowercase, or camel case.

There are a handful of additional parameters in the query's full form that we can pass to the `match` query too. The one we have discussed so far is a shortened form of the `match` query. The following code snippet provides an example of the full form:

```
GET books/_search
{
  "query": {
    "match": {
      "FIELD": { #A
        "query": "<SEARCH TEXT>", #B
        "<parameter>": "<MY_PARAM>", #C
      }
    }
  }
}
```

#A: Declares FIELD as an object with additional parameters

#B: The query attribute holds the search text.

#C: The parameter (such as analyzer, operator, prefix_length, fuzziness, etc.) expects a value to be set.

We can search across multiple indices by providing comma-separated indices in the search URL as the following demonstrates:

```
GET new_books, classics, top_sellers, crime* /_search
{
  ...
}
```

As you can see, any number of indices can be provided when invoking the `_search` endpoint, including wildcards.

NOTE If we omit the `index` (or `indices`) in the search request, we effectively search the entire index. For example, `GET _search { ... }` searches across all the indices in the cluster.

10.5.2 Searching using a match query

Now that we know the format for a `match` query, let's look at an example where we want to search for Java books with `Java` in the `title` field. The following listing demonstrates this, setting the `title` field to the word `Java` as the text to search.

Listing 10.4 Searching for Java books

```
GET books/_search
{
  "query": {
    "match": { #A
      "title": "Java" #B
    }
  }
}
```

#A The match query in action
#B Sets the search criteria, searching for the word Java in the title field

As the listing shows, we are creating a `match` query with a search criteria of searching a word in a `title` field. Elasticsearch fetches all the documents that match the word `Java` in the `title` field as expected.

10.5.3 Match query analysis

In the last chapter on term-level queries, we saw that these queries are not analyzed. The match queries that work on text fields, on the other hand, are analyzed. The same analyzers used during the indexing process (unless search queries were explicitly defined with different analyzers) process the search words in match queries. If a standard analyzer (default analyzer) is used during the indexing of our document, the search words are analyzed using the same standard analyzer before the search is executed.

Additionally, the standard analyzer applies the same lowercase token filter (remember, the lowercase token filter is applied during the indexing) to the search words. Thus, if you provide the search keywords as uppercased, they are converted to lowercase letters and searched against the inverted index. For example, if we change the `title` value to use uppercase criteria such as `"title": "JAVA"`, for example, and rerun the query, the results are the same as the search query in listing 10.4. If you change the `title` value to lowercase or in any other way (e.g., `java`, `java`, etc.), the query still returns the same results.

10.5.4 Searching multiple words

In our earlier code example (listing 10.4), we used a single word (`Java`) as the search criteria against a `title` field. We can expand this criteria to accommodate searching for multiple words or a sentence in a single field. For example, we could search for the words `Java Complete Guide` in the `title` field or for `Concurrency and Multithreading` in the `synopsis` field, and so on. Indeed, searching a string of words (like a broken sentence) is more common than searching for a single word. The query in the following listing does exactly that.

Listing 10.5 Match query for a set of words

```
GET books/_search
{
  "query": {
    "match": {
      "title": {
        "query": "Java Complete Guide"
      }
    },
    "highlight": {
      "fields": {
        "title": {}
      }
    }
  }
}
```

Here, our intention is to search for a specific title (`Java Complete Guide`). That is, we want to fetch a book titled `Java Complete Guide`, if available, and return nothing if not. However, if we execute the query with these words, you may be surprised to see more documents than just the one that matches exactly with the search query.

The reason for this behavior is that Elasticsearch employs an `OR` Boolean operator by default for this query, so it fetches all the documents that match with any of the words. The words are matched individually rather than as a phrase: in our example, Elasticsearch searches for `Java` and returns the relevant documents, followed by another search for `Complete` and adds the results to the list, and so on. The query returns either `Java`, `Complete`, or `Guide`, including combinations of the words as its results.

By default, Elasticsearch uses an `OR` operator when searching for a set of words (as you may have already guessed). The same search in listing 10.5 can be rewritten (though the `OR` operator is redundant) as the next listing shows.

Listing 10.6 The query from listing 10.5, specifying the OR operator explicitly

```
GET books/_search
{
  "query": {
    "match": {
      "title": {
        "query": "Java Complete Guide",
        "operator": "OR" #A
      }
    }
  }
}
```

#A Specifies the OR operator explicitly (although, it is set by default)

If you want to change this behavior to find the documents that have all three words in the title, then you need to enable the AND operator. The following listing shows this approach.

Listing 10.7 Specifying the AND operator explicitly

```
GET books/_search
{
  "query": {
    "match": {
      "title": {
        "query": "Java Complete Guide",
        "operator": "AND" #A
      }
    }
  }
}
```

#A Explicitly specifies an AND clause

This query tries to find a book or all books that match all three words (the title must have *Java AND Complete AND Guide*). However, in our data set, we do not have a book called *Java Complete Guide*, so no results are returned.

10.5.5 Matching at least a few words

The OR and AND operators are opposing conditions. The OR condition fetches either of the search words, and the AND condition gets matching documents exactly for all of the words. What if we want to find documents that match at least a few words from the given set of words? In the previous example, suppose we want at least two words out of three to match (say, *Java and Guide*, for example). This is where the `minimum_should_match` attribute comes in handy.

The `minimum_should_match` attribute indicates the minimum number of words that should be used to match the documents. The next listing demonstrates this in action.

Listing 10.8 Matching at least two words

```
GET books/_search
{
  "query": {
    "match": {
      "title": {
        "query": "Java Complete Guide",
        "operator": "OR",
        "minimum_should_match": 2 #A
      }
    }
  }
}
```

#A Sets the minimum number of words that should match

This query will match at least two words (the `minimum_should_match` attribute is set to 2) and will fetch the documents with a combination of two words out of the given three words. The `OR` operator is redundant here because it is applied by default.

NOTE Setting the value to 3 in the previous listing is as good as changing the operator to `AND`: all the words must be matched.

10.5.6 Fixing typos using the keyword fuzziness

When searching for things, we sometimes might incorrectly type the search criteria (we all have been there); for example, instead of searching for Java books, we might post a query with `Kava` as the search criteria. While we know, the intention is to search Java books, so too does Elasticsearch. It employs a concept called *fuzziness*. Simply put, fuzziness is a mechanism to correct a user's spelling mistakes in query criteria.

Fuzziness makes character changes to string input so that it is the same as the string that may exist in the index. It employs the Levenshtein distance algorithm to fix incorrect spellings. We will look at fuzziness further along in this chapter when we discuss fuzzy queries, but we can still use it with match queries as the next paragraph explains.

A match query also allows us to add a `fuzziness` parameter to fix spelling mistakes. We can set it as a numeric value, where the expected values are 0, 1, or 2, meaning none, one, or two character changes (insertions, deletions, modifications), respectively. In addition to setting these values, we also use an `AUTO` setting; we let the engine deal with the changes by setting `AUTO` as its fuzziness parameter. The following listing shows how we use the fuzziness (with a value of 1) to sort our `Kava` spelling typo.

Listing 10.9 Sorting out a spelling mistake with the fuzziness parameter

```
GET books/_search
{
  "query": {
    "match": {
      "title": {
        "query": "Kava",
        "fuzziness": 1 #A
      }
    }
  }
}
```

#A :The fuzziness set to 1 replaces one letter with all other combinations.

Again, we will cover fuzziness and fuzzy queries in the later part of this chapter, so do not fret if you are overwhelmed by this instance. In the example, when searching the text string, *Java Complete Guide*, we used a set of words to search for a book (or books), and most likely, the words were expected to be treated individually (like a set of search words). However, at times we may want to search for a phrase or a sentence. That's when the `match_phrase` query comes into the picture.

10.6 Match phrase (`match_phrase`) queries

The match phrase (`match_phrase`) query finds the documents that match exactly a given phrase. The idea behind the match phrase is to search for the phrase (group of words) in a given field in the same order. For example, if you are looking for the phrase "book for every Java programmer" in the synopsis of a book, documents are searched with those words in that order.

From our previous section on match queries, we learned that words can be split individually and searched with an AND/OR operator when using a `match` query. The `match_phrase` query is the opposite. It returns the results matching the search phrase exactly. The following listing illustrates the `match_phrase` query in action.

Listing 10.10 The `match_phrase` query in action

```
GET books/_search
{
  "query": {
    "match_phrase": { #A
      "synopsis": "book for every Java programmer" #B
    }
  }
}
```

#A Specifies the match phrase query

#B Specifies the phrase (group of words) to be matched

The `match_phrase` query expects a phrase as you can see in the code in listing 10.10. It returns exactly one document because we only have one in our books index with that phrase in the synopsis field.

10.6.1 Match phrase with the keyword slop

What if we drop a word or two in between the said phrase? Say, for example, we remove the `for` or `every` (or both) from the phrase "book for every Java programmer" and rerun the same query. Unfortunately, the query wouldn't return any results! The reason for this is that `match_phrase` expects the words in a phrase to match the exact phrase, word by word. Searching "book Java programmer" returns no results. Fortunately, there is a fix to this problem: using a parameter called `slop`.

The `slop` parameter allows us to ignore the number of words in between the words in that phrase. We can drop the in-between words in the phrase. However, we need to let the engine know how many words to drop. This is done by setting a value for the `slop` parameter. The attribute `slop` is an integer value indicating the number of words that can be ignored in a phrase when searching `match_phrase`. For example, `slop` with 1 ignores one word, `slop` with 2 forgives two words missing in a phrase, and so on. The default value of `slop` is 0, meaning we will not be forgiven for providing a phrase with missing words.

Coming back to our example, let's drop a word from the given phrase, so instead of a "book for every Java programmer," we'll search for the phrase "every Java programmer," dropping the word `for`. Because we drop a single word, we need to set the `slop` parameter to 1 (the missing word is just one word). The query in listing 10.11 demonstrates this. Obviously, we need to expand the query by providing two further parameters in the `query` and `slop` objects for the `synopsis` field.

Listing 10.11 The `match_phrase` query with `slop` that drops one word

```
GET books/_search
{
  "query": {
    "match_phrase": {
      "synopsis": { #A
        "query": "book every Java programmer", #B
        "slop": 1 #C
      }
    }
  }
}
```

#A Expands this field to include an object with additional parameters

#B The phrase with one missing word, for

#C Sets `slop` to 1, meaning the query looks for phrases with a single missing word

If you want to use the `slop` parameter, both `query` and `slop` must be provided along with the field's object as demonstrated in listing 10.11 (the long form of the query). Because `slop` is set to 1, the query matches if one word is missing in an entire phrase in the `synopsis` field. Without a doubt, this query returns the book matching our entire phrase. The takeaway from this example is that a match phrase query looks for an exact phrase, but if you are not sure of the exact phrase, you can use the `slop` parameter to indicate how forgiving your query should be.

There's a slight variation to the match phrase query - the match phrase prefix (`match_phrase_prefix`) query. In addition to matching an exact phrase, we can expect the last word to be matched as a prefix. Let's discuss this in the next section with an example.

10.7 Match phrase prefix (`match_phrase_prefix`) queries

The `match_phrase_prefix` (`match_phrase_prefix`) query is a slight variation of the `match_phrase` query in that, in addition to matching the exact phrase, the query matches all the words, using the last word as a prefix. This is easiest to understand via an example. The following listing illustrates searching for the prefix `Co` in the title, which could mean Collections or Concurrency, etc.

Listing 10.12 Using a `match_phrase_prefix` query

```
GET books/_search
{
  "query": {
    "match_phrase_prefix": { #A
      "tags": {
        "query": "Co" #B
      }
    },
    "highlight": {
      "fields": {
        "tags": {}
      }
    }
  }
}
```

#A Specifies the `match_phrase_query` query

#B Specifies the prefix to search

This query fetches all the books with tags matching `Co`. This includes prefixes such as `Component`, `Core`, `Code`, and so on.

10.7.1 Match phrase prefix using `slop`

Similar to the `match_phrase` query, the order of the words is important in the `match_phrase_prefix` query too. Of course, `slop` is here to the rescue. For example, when we want to retrieve books with the phrase `concepts and foundations` across the `tags` field, we can omit `and` by adding the keyword `slop` as the following listing demonstrates.

Listing 10.13 Using the `match_phrase_prefix` query with `slop`

```
# Match phrase prefix
GET books/_search
{
  "query": {
    "match_phrase_prefix": {
      "tags": {
        "query": "concepts found", #A
        "Slop":1 #B
      }
    }
  }
}
```

#A The phrase has one word (`and`) omitted as well as has a prefix (“`found`” instead of “`foundations`”).

#B Sets `slop` to 1 because one word is dropped from the phrase

Setting the keyword `slop` as 1 queries the books with the `tags` `concepts` and `found`*, but it ignores the word `and`. The query should return the book *Kotlin Programming* as the result because the query matches the phrase *Kotlin concepts and foundational APIs* in the `tags` field.

So far, we’ve queried search criteria across a single field. However, let’s say that we want to find the words *Software Development* across the `title`, `synopsis`, and `tags` fields. That’s exactly what happens when we use a `multi_match` query, discussed in the next section.

10.8 Multi-match (`multi_match`) queries

The multi-match (`multi_match`) query, as the name suggests, searches the query across multiple fields. For example, if we want to search for the word `Java` across the three fields `title`, `synopsis`, and `tags`, then the `multi_match` query is the answer. The following listing shows a query that searches for `Java` across these three fields.

Listing 10.14 Searching for a keyword across multiple fields using multi_match

```
GET books/_search
{
  "_source": false, #A
  "query": {
    "multi_match": { #B
      "query": "Java", #C
      "fields": [ #D
        "title",
        "synopsis",
        "tags"
      ]
    }
  },
  "highlight": { #E
    "fields": {
      "title": {},
      "tags": {}
    }
  }
}
```

#A Suppresses the source document showing up in the results

#B Specifies the multi_match query

#C Defines the search criteria as the word Java

#D Searches across multiple fields provided in an array

#E Highlights the matches returned in the results

The `multi_match` query expects an array of fields along with the search criteria. We get the aggregated results from combining all the results for individual fields.

10.8.1 Best fields

You may be wondering what would be the relevancy of the document when we search multiple fields? Fields where more words are matched are scored higher. That is, if we search for *Java Collections* across multiple fields, a field (say `synopsis`) where two words match is more relevant than a field with one (or no) matches. The document with this `synopsis` field in this case is set with a higher relevancy score.

Fields that match all the search criteria are called *the best fields*. In the previous example, assuming `synopsis` holds both words, *Java* and *Collections*, we can simply say that `synopsis` is the best field. Multi-match uses a `best_fields` type under the hood when running queries. This type is the default for `multi_match` queries. There are, of course, other types of fields, which we will see shortly.

Let's rewrite the query we wrote in listing 10.14, but this time instead of letting Elasticsearch use the default setting of the `best_fields` type, we specifically mention it by overriding the `type` field. The following listing shows the resulting query.

Listing 10.15 Specifying the best_fields type explicitly

```
GET books/_search
{
  "_source": false,
  "query": {
    "multi_match": {
      "query": "Design Patterns",
      "type": "best_fields", #A
      "fields": ["title","synopsis"]
    }
  },
  "highlight": { #B
    "fields": {
      "tags": {},
      "title": {}
    }
  }
}
```

#A Sets the type of multi_match query to best_fields

#B Suppresses the source but shows the highlights

In the listing, we query for *Design Patterns* across the `title` and `synopsis` fields. This time, we explicitly assign the query to use the `best_fields` type for the `multi_match` query.

NOTE The default type for a `multi_match` query is set as `best_fields`. The `best_fields` algorithm ranks the field that has most words higher than those with the least amount of words.

If you look at the response and the scores (see the following code snippet), you'll find the *Head First Design Patterns* book has a score of 6.9938974 compared to the *Head First Object-Oriented Analysis Design* book, which has a score of 2.9220228:

```
"hits" : [
  {
    "_index": "books",
    "_id": "10",
    "_score": 6.9938974,
    "highlight": {
      "title": [
        "Head First <em>Design</em> <em>Patterns</em>"
      ]
    }
  },
  {
    "_index": "books",
    "_id": "8",
    "_score": 2.9220228,
    "highlight": {
      "title": [
        "Head First Object-Oriented Analysis <em>Design</em>"
      ]
    }
  }
...]
```

There are other types of multi-match queries too, such as `cross_fields`, `most_fields`, `phrase`, `phrase_prefix`, and others. We can use the `type` parameter to set the type of query to search for the best match among multiple fields. We won't delve into all these types, however, so consult Elasticsearch's documentation for more information.

If you are wondering how Elasticsearch carries out the `multi_match` query, behind the scenes, it is rewritten using a *disjunction max* (`dis_max`) query. We discuss this query type in detail in the following section.

10.8.2 Disjunction max (`dis_max`) queries

In the previous section, we looked at the `multi_match` query, where the criteria was searched against multiple fields. How does this query type get executed behind the scenes? Elasticsearch rewrites the `multi_match` query using a *disjunction max query* (`dis_max`). The `dis_max` query splits each field into a separate match query as the following listing shows.

Listing 10.16 Disjunction max (`dis_max`) query in use

```
GET books/_search
{
  "_source": false,
  "query": {
    "dis_max": {
      "queries": [ #B
        {"match": {"title": "Design Patterns"}}, #C
        {"match": {"synopsis": "Design Patterns"}}
      ]
    }
  }
}
```

#A: Specifies the `dis_max` query type

#B: Defines the set of queries to include in a `dis_max` query block

#C: Specifies the match query

As you can see from this listing, multiple fields are split into two `match` queries under the `dis_max` query. The query returns the documents with a high relevancy `_score` for the individual field.

NOTE The `dis_max` query is classified as a compound query: a query that wraps up other queries. We discuss compound queries in the next chapter.

In some situations, there is a chance that the relevancy scores of the multi-fields during the `multi_match` query could be the same. In that case, the scores end up in a tie. To break the tie, we use a tie breaker, discussed in the next section.

10.8.3 Tie breakers

The relevancy score is based on the single field's score, but if the scores are tied, we can specify `tie_breaker` to relieve the tie. If we use `tie_breaker`, Elasticsearch calculates the

overall score slightly differently which we will see in action shortly, but first, let's checkout an example.

Listing 10.17 queries a couple of words against a couple fields: `title` and `tags`. However, the listing also shows an additional parameter, `tie_breaker`.

Listing 10.17 A multi_match query with a tie breaker

```
GET books/_search
{
  "query": {
    "multi_match": { #A
      "query": "Design Patterns", #B
      "type": "best_fields", #C
      "fields": ["title", "tags"], #D
      "tie_breaker": 0.9 #E
    }
  }
}
```

#A Specifies the multi_match query
#B Queries Design Patterns
#C Sets the query's default type to best_fields
#D Defines the set of fields to search
#E Sets the tie breaker

When we search for *Design Patterns* using the `best_fields` type and specify multiple fields (the fields `title` and `synopsis` in listing 10.17), we can provide a `tie_breaker` to overcome any tie on equal scores. When we provide the tie breaker, the overall scoring is calculated as:

```
Overall score = _score of the best match field + _score of the other matching fields * tie_breaker
```

A few moments ago, we worked with a `dis_max` query. In fact, Elasticsearch converts all `multi_match` queries to the `dis_max` query. For example, the `multi_match` query from listing 10.17 can be rewritten as a `dis_max` query as the next listing demonstrates.

Listing 10.18 The dis_max query with a tie breaker

```
GET books/_search
{
  "_source": false,
  "query": {
    "dis_max": {
      "queries": [
        {"match": {"title": "Design Patterns"}},
        {"match": {"synopsis": "Design Patterns"}],
        "tie_breaker": 0.5 #A
      }
    },
    "highlight": {
      "fields": {
        "title": {},
        "synopsis": {},
        "tags": {}
      }
    }
  }
}
```

#A The tie breaker

As you can see, the same query that was written as a `multi_match` query is now rewritten as `dis_max` query. Indeed, that's exactly what Elasticsearch does behind the scenes.

Because there are multiple fields that we search, at times we may want to give additional weight to a particular field (for example, finding our search words in a title is more relevant than the same search words appearing in a lengthy `synopsis` or `tags` fields). How do we let Elasticsearch know that our intention is to give extra weight to the `title` field? That's exactly what we do when boosting individual queries, discussed in the following section.

10.8.4 Individual field boosting

There's usually a search bar provided to the users on a website or application so that they can search for something such as a product, book, or review and so forth. When the user enters a few words, they don't say that they are interested in searching only those words in a particular field. For example, when we search for *C# book* on Amazon, we don't ask Amazon to search only in a particular category such as title or synopsis. We simply input the string in the text box and let Amazon do the job of figuring out the result. That's exactly what we can do using individual field boosts!

In a `multi_match` query, we can bump up (boost) the criteria for a specific field. Say, when searching for *C# Guide*, we decide finding the word in the title is more important than in the tags. In this case, we simply boost the importance of the title field by using a caret and a number: `title^2`, for example. The following listing shows the full query for this scenario.

Listing 10.19 Boosting the score for individual field in a `multi_match` query

```
GET books/_search
{
  "query": {
    "multi_match": {
      "query": "C# Guide",
      "fields": ["title^2", "tags"] #A
    }
  }
}
```

#A Doubles up the title field

In this listing, we double up the `title` field's importance, so if the text *C# Guide* is found in the `title` field, that document will have a higher score than the document found in the `tags` field.

As we wrap up match queries, let's move on to learn about the `query_string` query. This is a type of query that helps us build a request URL search, mimicking the Kibana Query Language (KQL) in a Query DSL format. Let's go over the need for a query string query and work with it in the following section.

10.9 Query strings

Earlier on in chapter 8, we looked at the URI search method (one of the search query methods in addition to Query DSL). There, we created the request by passing the search query and its parameters attached to the URL, unlike in a request body. We also learned that, though simple, the URI method becomes error-prone as the complexity of the query criteria grows.

If you are familiar with Kibana's Discover tab (see figure 10.5), we usually use the KQL to create a search criteria using operators. For example, if we want to search for "2nd edition Java book written by Bert, released after 2010," the equivalent query written in the Discover's KQL tab is as follows:

```
title:Java and author :Bert and edition:2 and release_date>=2000-01-01
```

The screenshot shows the Kibana Discover tab interface. At the top, there are tabs for 'Discover' (which is selected), 'New', 'Open', 'Share', 'Inspect', and 'Save'. Below the tabs, a search bar contains the KQL query: 'title:Java and author :Bert and edition:2 and release_date>=2000-01-01'. To the right of the search bar are 'Options', 'New', 'Open', 'Share', 'Inspect', and 'Save' buttons. A 'Refresh' button is also present. Below the search bar, there is a '+ Add filter' button. On the left side, there is a sidebar with a dropdown menu set to 'book*' and a search input field labeled 'Search field names'. Under 'Available fields', there is a dropdown menu set to '0' and a list of fields: '_id', '_index', '_score', '_type', 'author', 'amazon_rating', 'book', 'edition', 'isbn', 'language', 'release_date', 'synopsis', 'tags', and 'title'. The main area displays a table with one hit. The table has columns for 'Document' and '_source'. The '_source' column shows a document with the following fields and values: 'author: Kathy Sierra and Bert Bates', 'title: Head First Java', 'amazon_rating: 4.3', 'edition: 2', 'release_date: Feb 18, 2008', 'isbn: 089721551X', 'language: English', 'synopsis: The most important selling points of Head First Java is its simplicity and super-effective real-life analogies that pertain to the Java programming concepts.', 'tags: IT Certification Exams, Object-Oriented Software Design, Design Pattern Programming', '_id: 5', '_index: books', '_score: 6.152', and '_type: -'.

Figure 10.5 Kibana's Discover tab showing a KQL query

As the query in figure 10.5 shows, in KQL, we usually use OR, AND, and the other logical operators mixed within the query criteria. The principle behind KQL simply uses the URI request API. There surely is goodness in using the URI type query without having to worry about constructing a query in Query DSL mode; instead, throwing in operators and constructing an easily flowing query is a snap. We wish there was something that gives us the best of both worlds. Fortunately, there is. It's the URI request params using Query DSL mode—isn't it?

The good news is that we can achieve the same URI functionality using a special query called `query_string`. This query type lets us define a query using logical operators in a request body. There are two types of query string queries that we'll discuss in the next couple of sections.

10.10 Query string (`query_string`) queries

The `query_string` type lets us construct a query using operators such as `AND` or `OR`, as well as the logical operators such as `>` (greater than), `<=` (less than or equal to), `*` (contains in), and so one. This is easily understood when explained with an example, so let's jump right in to getting our code ready.

In an earlier example (figure 10.5), we fetched Bert's books using a long query on Kiban's Discover tab: `title:Java` and `author :Bert` and `edition:2` and `release_date>=2000-01-01`. We achieved this by writing a `query_string` query as the code in the following listing demonstrates.

Listing 10.20 Creating a query string with operators

```
GET books/_search
{
  "query": {
    "query_string": { #A
      "query": "author:Bert AND edition:2 AND release_date>=2000-01-01" #B
    }
  }
}
```

#A Specifies the `query_string` query
#B Provides the search criteria in the query

The `query_string` query expects a query parameter where we provide our criteria. The query is constructed as name-value pairs: for example, in listing 10.20, `author` is the field, and `Bert` is the value. Looking at the code in that listing, we notice a few things:

- The search query is built using the Query DSL syntax (the GET request has a body).
- The search criteria is written to concatenate the fields using operators.

The query is as simple as writing a question in plain English. We can create a complex criteria using these operators (in plain English) and provide it to the engine to get the results for us.

Sometimes, we do not know which fields users are expected to search; they may want the query to focus on the `title` field, or on the `synopsis` or `tags` fields, or all of them. In the next section, we will go over the ways of specifying fields.

10.10.1 Fields in a query string query

In a typical search box, the user does not need to mention the field when searching for something. For example, take a look at the query in the following listing.

Listing 10.21 Query string with no fields specified

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "Patterns" #A
    }
  },
  "highlight": { #B
    "fields": {
      "title": {},
      "synopsis": {},
      "tags": {}
    }
  }
}
```

#A The query doesn't mention the fields.

#B Highlights the responses

We want to search for the keyword *Patterns* in the previous query (listing 10.21). One quick thing to remember is that the query is not asking us to search any fields. It is a generic query that's actually expected to be executed across all fields. The response (shown in the following snippet) shows that some results were highlighted on a different field for individual documents:

```
"highlight" : {
  "synopsis" : ["Head First Design <em>Patterns</em> is one of ..."],
  "title" : ["Head First Design <em>Patterns</em>"]
},
...
"highlight" : {
  "synopsis" : [ "create .. using modern application <em>patterns</em>"]
},
...
```

Instead of letting the engine search query against all the available fields, we can assist Elasticsearch by providing the fields to run the search on. The next listing shows how to do this.

Listing 10.22 Specifying the fields explicitly in a query string query

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "Patterns", #A
      "fields": ["title", "synopsis", "tags"] #B
    }
  }
}
```

#A The query criteria with no fields mentioned.

#B Explicitly declares the fields as an array of strings

Here, we specify the fields explicitly in an array in the `fields` parameter and mention the fields that this criteria is expected to be performed against. If we are not sure of the fields when constructing this query, we can use another parameter, `default_field` as the query in the following listing demonstrates.

Listing 10.23 Query string with a default field

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "Patterns",
      "default_field": "title" #A
    }
  }
}
```

#A A default field declaration

If a field is not mentioned in the query, the search is carried out against the `title` field. That's because the `title` field is declared as the `default_field` in listing 10.23.

10.10.2 Default operator

In our earlier code example in listing 10.23, we searched for a single word, *Patterns*. If we extend that search to include an additional word such as *Design*, we may get multiple books (two books with the current dataset) instead of the correct one: *Head First Design Patterns*. The reason is that Elasticsearch uses the `OR` operator by default when searching. Hence, it finds books with both words, *Design OR Patterns* in the `title` field.

If this is not our intention (say, for example, we want to fetch a book with the exact phrase *Design Patterns* in the title), we should use the `AND` operator. The `query_string` query in the following listing has an additional parameter, `default_operator`, where we can set the operator to `AND`.

Listing 10.24 Query string with the AND operator

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "Design Patterns",
      "default_field": "title",
      "default_operator": "AND" #A
    }
  }
}
```

#A Changes the operator from OR to AND

This `query_string` query is declared with the `AND` operator. Hence, you'd expect *Design Patterns* to be treated as a single word.

10.11 Query string with a phrase

If you are wondering if there's support for a phrase search using `query_string`, indeed there is. We can rewrite the code in listing 10.24 using a phrase rather than changing the operator. The only thing we need to take note of is that phrases must be enclosed in quotes. That means, the quotes that correspond to the phrase must be escaped; for example, `"query": "\\"Design Patterns\\\""`. The query in the next listing searches for a phrase.

Listing 10.25 Query string with a phrase

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "\"making the code better\"", #A
      "default_field": "synopsis"
    }
  }
}
```

#A: Quotes around the sentence make it a phrase query.

As you can expect, this code searches for the phrase "making the code better" in the `synopsis` field and fetches the *Effective Java* book. Going with the flow, we can use the `slop` parameter (read about `slop` earlier in this chapter and in chapter 8) if we are missing one or two words in the phrase. For example, the code in the following listing demonstrates how the `phrase_slop` parameter allows for a missing word in the phrase (`the` is dropped from the phrase) and still gets a successful result.

Listing 10.26 Query string with a phrase and a slop

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "\"making code better\"", #A
      "default_field": "synopsis",
      "phrase_slop": 1 #B
    }
  }
}
```

#A The phrase with one word removed

#B Sets the phrase_slop to 1 so the phrase with one missing word is honored

The query misses a word, but the `phrase_slop` setting forgives the omission and, hence, we get the desired result.

When building a search service, one must take into account supporting spelling mistakes. The application should handle these mistakes gracefully and, instead of returning incorrect or no results, it should identify the spelling issue to enhance and accommodate the results. This makes the user search experience wonderful. Elasticsearch provides support for handling incorrect spelling issues in the form of *fuzzy queries*. We discuss fuzzy queries in the following section.

10.12 Fuzzy queries

We can also ask Elasticsearch to forgive spelling mistakes by using fuzzy queries with `query_string` queries. All we need to do is suffix the query criteria with a tilde (~) operator. This is best understood by an example as the following listing demonstrates.

Listing 10.27 A fuzzy query string query

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "Pattenrs~", #A
      "default_field": "title"
    }
  }
}
```

#A Incorrectly spelt “Pattenrs” as the search word

By setting the suffix with the ~ operator, we are cueing the engine to consider the query as a fuzzy query. By default, the edit distance of 2 is used when working with fuzzy queries. The edit distance is the number of mutations required to transform a string to another string. For example, CAT requires an edit distance of 1 to transform it into CAP.

Earlier in chapter 8, we learned about fuzzy queries, where the queries utilized the Levenshtein distance algorithm. However, there’s another type of edit distance algorithm called the Damerau–Levenshtein distance algorithm. In fact, the Damerau–Levenshtein

distance is used to support the fuzzy queries. It supports insertions, deletions, or substitution of a maximum of two characters as well as transposition of adjacent characters.

NOTE The Levenshtein distance algorithm defines the minimal number of mutations that are required on a string to be transformed into another string. These mutations include insertions, deletions, and substitutions. The Damerau–Levenshtein distance algorithm goes one step further. In addition to having all the mutations as defined by Levenshtein, the Damerau–Levenshtein algorithm considers the transposition of adjacent characters (for example, TB-> BT -> BAT).

By default, the edit distance in a `query_string` query is 2, but we can reduce it if needed by setting the 1 after the tilde like so: "Pattenrs~1". In the next two sections, we'll look at some simpler queries.

10.13 Simple string queries

The `query_string` query is strict on syntax, and errors in the input are not forgiven. For example, the following query in the next listing throws an error because the input has some parsing issues (purposely, we added a quote to the input criteria).

Listing 10.28 Query string query with an illegal quote character

```
GET books/_search
{
  "query": {
    "query_string": {
      "query": "title:Java\" #A
    }
  }
}
```

#A Query with a syntax error (no corresponding quote)

This query will not be parsed. Elasticsearch throws an exception, mentioning that the syntax was violated:

```
"error" : {
  "root_cause" : [{
    "type" : "json_parse_exception",
    "reason" : "Unexpected character ('\\" (code 34)): was expecting comma to separate object entries\n at ...]"
  }],
  ...
}
```

The query threw a JSON parse exception to the user, thus validating the stricter syntax for `query_string` queries. However, if you want Elasticsearch to ignore syntactical errors and go with the job, here's an alternative: use a `simple_query_string` query, discussed in the following section.

10.14 Simple_query_string queries

As the name suggests, the `simple_query_string` query is a variant of the `query_string` query with a simple and limited syntax. We can use operators such as `+`, `-`, `|`, `*`, `~` and so forth for constructing the query. For example, searching for "Java + Cay" produces a Java book written by Cay as the next listing shows.

Listing 10.29 A simple query string

```
GET books/_search
{
  "query": {
    "simple_query_string": { #A
      "query": "Java + Cay" #B
    }
  }
}
```

#A Specifies the type of the query as `simple_query_string`
#B Searches the query with the AND operator

The `+` operator in the query allows the query to search for Java AND Cay across all fields. We can specify the fields if we want to check a set of fields instead of all the fields by setting the `fields` array. Table 10.1 describes the set of operators that we can use in `simple_query_string`.

Table 10.1 Operators for a simple query string

Operator	Description
<code> </code>	OR
<code>+</code>	AND
<code>-</code>	NOT
<code>~</code>	Fuzzy query
<code>*</code>	Prefix query
<code>"</code>	Phrase query

Unlike the `query_string` query, the `simple_query_string` query doesn't respond with errors if there's any syntax error in the input criteria. It takes a quieter side of not returning anything should there be a syntactic error in the query as the code in the following listing demonstrates.

Listing 10.30 The simple query string in use: no issue with the incorrect syntax

```
GET books/_search
{
  "query": {
    "simple_query_string": { #A
      "query": "title:Java\"" #B
    }
  }
}
```

#A The simple_query_string query
#B The query with incorrect syntax

Though the same query with incorrect syntax (extra quote at the end) was issued, there is no error returned to the user, except no return documents. The `simple_query_string` query is helpful in such situations.

And, that's a wrap! This chapter is all about full text queries, queries on unstructured data. In the following chapter, we will look at compound queries in detail. Compound queries are an advanced form of search queries, which wrap leaf queries such as term-level and full text queries.

10.15 Summary

- Elasticsearch is big on searching unstructured data using full text queries. Full text queries yield relevancy, meaning the documents that were matched and returned have a positive relevancy score.
- Elasticsearch provides a `_search` API for querying purposes.
- A handful of match queries work for various use cases when searching full text with relevance. The most common query is the `match` query.
- The `match` query searches on text fields for a search criteria and scores the documents using the relevance algorithm.
- Match all (`match_all`) queries search across all indices, and it requires no body.
- To search a phrase, we use the `match_phrase` query or its variant, `match_phrase_prefix` query. Both types of queries let us search for a specific set of words in a defined order. Additionally, we can use the `phrase_slop` parameter should the phrase contain missing words.
- Searching the user criteria across multiple fields is enabled by using a `multi_match` query.
- A query string (`query_string`) query helps write a query using logical operators such as AND, OR, NOT, and so on. The `query_string` query is strict on syntax, however, so you'll receive an exception if the input syntax is incorrect.
- If we need Elasticsearch to be a bit less strict on the query string syntax, then instead of using a `query_string` query we should prefer a `simple_query_string` query. With that query type, all syntactical errors are suppressed by the engine.

11

Compound queries

This chapter covers

- **Introduction to compound queries**
- **Boolean search queries**
- **Constant score queries**
- **Boosting queries**
- **Disjunction maximum queries**
- **Function score queries**

In the last two chapters, we looked at the term-level and full-text queries. We learned about searching structured and unstructured data using a multitude of queries, some producing relevance scores and others working in a filter context where scores were irrelevant. Most of those queries allowed setting a simple search criteria, working on a limited set of fields, such as finding books written by an author or searching for best selling books and so on.

In addition to providing queries for complex criteria, we may sometimes need to boost the scores based on a certain criteria, while at the same time negating the score for negative matches (for example, searching through all books that were launched during a training program may get a positive boost while simultaneously suppressing (negating) books that were pricey). Or maybe we want to set the scores based on custom requirements rather than using Elasticsearch's inbuilt relevance algorithms.

The individual leaf queries we've worked with so far are limited in that, other than searching through one or a handful of criteria, searching through more complex and complicated requirements wouldn't be catered to (for example, searching for books written by a specific author, published between certain dates, listed as a best seller or have a star rating of 4.5 (out of 5) with a specific number of pages, etc.). Such advanced queries require

advanced searching query capabilities. That's exactly what we are going to look at in this chapter in the name of compound queries.

Compound queries are advanced constructs of searching capabilities to query complex search criteria in Elasticsearch. They are made of individual leaf queries wrapped up in conditional clauses and other constructs to provide capabilities such as letting the user set custom scores using a set of predefined functions, boosting positive search matches while suppressing negative clauses, developing scores using scripts, and so on. They allow us to use individual leaf queries to develop fully fledged advanced queries of various types.

In this chapter, we will understand the requirements that compound queries satisfy and their semantics and usage. We'll look at the Boolean query, where multiple leaf queries are cast in a few conditional clauses to design an advanced search query, and we'll use clauses like `must`, `must_not`, `should`, and `filter` to arrange the leaf queries in to a compound query. We will look at a boosting query for enhancing the query's score when a positive match occurs, while simultaneously lowering the score for negations. We'll then learn about a predefined static scoring query called constant score, which helps set a static score on the retired results.

We will also run through function score queries to help set user-defined custom scoring algorithms using a set of functions. We will go over the mechanics of setting scores using scripts and weights, based on other field values present in the document, as well as random numbers. But first, let's set the scene: in the next section, we'll go over the need for compound queries and why and how they help build advanced queries.

11.1 Compound queries

We worked with leaf queries in the last two chapters (term-level and full-text queries): the queries that work on essentially single (individual) fields. If our requirement is to find the top selling books for a period, we can use a leaf query to fetch those results. Leaf queries help find answers for simple questions, with no support for conditional clauses. However, the real world is seldom made of simple query requirements.

Most requirements demand developing complex queries with multiple clauses and conditions. A complex query, for example, consists of finding the best selling books, written by a particular author, and published between a specific period or a particular edition, or returning all books categorized by various geographical zones except a specific country, ordered by the highest grosser, and so on.

Here's where the compound queries come to life: they help develop complex search queries by combining one or more leaf queries, as well as integrating compound queries themselves. Fortunately, all we need to do is use the same Query DSL (we learned about Query DSLs in chapter 8) for writing compound queries: we use the same `_search` endpoint with a request body that consists of a query object. Figure 11.1 shows the syntax for a compound query at a glance.

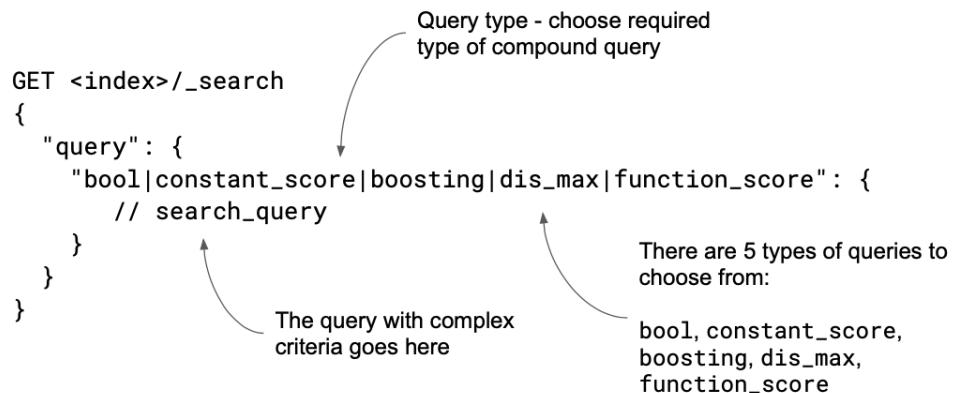


Figure 11.1 Compound query syntax

As you can gather from the figure 11.1, the basic syntax for compound queries is no different from other queries. However, the body of the query object is made of different components, depending on the type of compound query that we want to use.

Elasticsearch provides five such queries for varied search requirements: Boolean (`bool`), constant score, function score, boosting, and disjunction max. Table 11.1 briefly describes these five compound queries. For example, if our requirement is to develop an advanced query using conditional clauses, we can use a Boolean query (called a `bool` query) that encompasses multiple leaf queries using AND, OR, and other conditions. Similarly, if the requirement is to set a static score across all the results, perhaps a constant score query is what we need. We'll apply these five queries for different use cases in the coming sections of this chapter.

Table 11.1 Compound queries briefly described

Compound query	Description
Boolean (bool) query	A combination of conditional clauses that wraps individual leaf (term and full text) queries. This works similar to the AND, OR, and the NOT operators. Example: products= TV AND color = silver NOT rating < 4.5 AND brand = Samsung OR LG
Constant score (constant_score) query	Wraps up a filter query to set constant scores on the results. It also helps to boost the score. Example: Search for all TVs with user ratings higher than 5 but set a constant score of 5 for each of the results, irrespective of the search engine's calculated score.
Function score (function_score) query	A set of user-defined functions to assign custom scores to the resultant documents. Example: Search for products - and if the product is from LG and is a TV, then boost the score by three (by writing a script or weight function).
Boosting (boosting) query	Boosts the score for positive matches while negating the score for nonmatches. Example: Fetch all the TVs but lower the score on those that are pricey.
Disjunction max (dis_max) query	Wraps a number of queries to search multiple words in multiple fields (similar to a <code>multi_match</code> query). Example: Search for smart TV in two fields (say, overview and description) and return the best match.

Among all these types, the `bool` query is the most commonly used compound query due to its greater flexibility and support for multiple conditional clauses. In the next section, we'll concentrate on dissecting and understanding the `bool` query along with others. Because we have a lot to cover when learning about the `bool` query, it deserves to have its own dedicated section. However, before we delve deeper into the rest of the chapter, we need to set our playground to experiment with and understand compound queries.

11.2 Sample products data

In this section, we will use product data as our sample data, and we will go over the definitions and indexing processes. In our playground, we'll work with a dataset of electrical and electronic products such as televisions (TVs), laptops, mobile phones, refrigerators (Fridges), and so on.

11.2.1 Products schema

As part of developing the `products` index, the first step is to create a data schema defining the fields and their data types. The following listing shows the schema at a glance (the full schema is available in my GitHub repository).

Listing 11.1 Defining the `products` schema

```
PUT products
{
  "mappings": {
    "properties": {
      "brand": {
        "type": "text"
      },
      "colour": {
        "type": "text"
      },
      "energy_rating": {
        "type": "text"
      },
      ...
      "user_ratings": {
        "type": "double"
      },
      "price": {
        "type": "double"
      }
    }
  }
}
```

As you can see from the listing, there is not much of a surprise in the `products` definition except that a couple attributes (`price` and `user_ratings`) are defined as `double`, but the rest are declared as `text` fields. Most fields are declared as multiple data types (for eg: `text` and `keyword`) to accommodate working with term-level queries on some data (e.g., energy rating or color).

The mapping in listing 11.1 creates an empty `products` index, with the relevant schema for the e-commerce electrical products that we are about to index. In the next section, we will index a set of sample products, ranging from TVs to laptops to others.

11.2.2 Indexing products

Now that we have the schema ready, let's index our products data set for Elasticsearch. The data set is available in my GitHub, so make sure you copy the contents of products.txt and paste them in Kibana. We'll use the `_bulk` API to index this data. This code snippet shows a sample of the data:

```
PUT _bulk
{"index": {"_index": "products", "_id": "1"}}
{"product": "TV", "brand": "Samsung", "model": "UE75TU7020", "size": "75", "resolution": "4k", "type": "smart tv", "price": 799, "colour": "silver", "energy_rating": "A+", "overview": "Settle in for an epic..", "user_ratings": 4.5, "images": ""}
{"index": {"_index": "products", "_id": "2"}}
{"product": "TV", "brand": "Samsung", "model": "QE65Q700TA", "size": "65", "resolution": "8k", "type": "QLED", "price": 1799, "colour": "black", "energy_rating": "A+", "overview": "This outstanding 65-inch ..", "user_ratings": 5, "images": ""}
{"index": {"_index": "products", "_id": "3"}}
...

```

Now that we've indexed our products data set, let's go over the popular and most useful compound query. In the following section, we'll look at the Boolean query.

11.3 The Boolean (bool) query

The Boolean (bool) query is the most popular and flexible compound query we can use to create a set of complex criteria for searching data. As the name indicates, it is a combination of Boolean clauses, with each clause having a leaf query made of term-level or full-text queries. Each of these clauses has a typed occurrence of `must`, `must_not`, `should`, or `filter` clauses, briefly explained in the table 11.2.

Table 11.2 Brief description of Boolean clauses

Clause	Description	Example
<code>must</code>	An AND query where all the documents must match the query criteria.	Fetching TVs (product is a TV) that fall within a specific price range.
<code>must_not</code>	A NOT query where none of the documents match the query criteria.	Fetching TVs (product is a TV) that fall within a specific price range BUT with an exception such as not black in color.
<code>should</code>	An OR query where one of the documents must match the query criteria.	Searching for Fridges that are frost-free OR are energy rated above C grade.
<code>filter</code>	A filter query where the documents must match the query criteria (similar to the <code>must</code>	Fetching TVs (product is a TV) that fall within a specific price range (but expect

	clause) except that the <code>filter</code> clause does not boost the matches.	the score of the documents returned to be zero).
--	--	--

A compound query consisting of these clauses can contain multiple leaf queries or even additional compound queries. As you can envisage, the scope of creating an advanced and complicated search query is quite achievable by combining the leaf and compound queries to create an advanced compound query.

From the available four clauses in table 11.2, the `must` and `should` clauses contribute to the relevance scoring, whereas `must_not` and `filter` will not. We will look at the details of all these clauses in a moment, but first, we should understand the `bool` query syntax and structure. Let's discuss that in the following section.

11.3.1 Bool query structure

As mentioned earlier briefly, the `bool` query is a combination of Boolean clauses producing a unified output. Figure 11.2 illustrates the basic structure of a `bool` query with empty clauses.

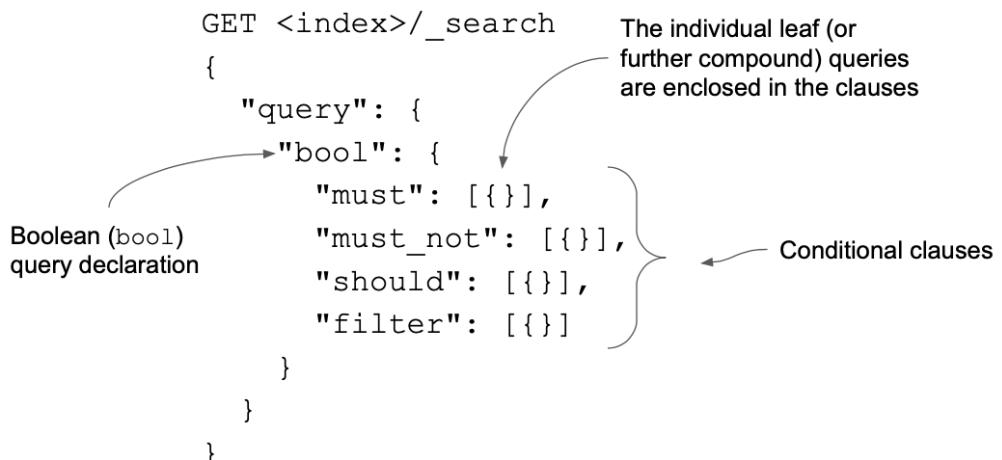


Figure 11.2 Syntax of a sample Boolean (bool) query with a set of four conditional clauses

As you can see in figure 11.2, a `bool` query is configured with a set of conditions captured in the clauses. A `bool` query can accept at least one of the queries embedded in a clause. Each of these clauses can then host one or more leaf or compound queries as an array of the queries. As the following code snippet demonstrates, you can provide multiple term-level and full-text queries (shown in bold) inside any of these clauses (highlighted in italics):

```

GET books/_search
{ 
```

```

"query": {
  "bool": {
    "must": [
      { "match": {"FIELD": "TEXT"}},
      { "term": {"FIELD": {"value": "VALUE"}}}
    ],
    "must_not": [
      {"bool": { "must": [{}]}}
    ],
    "should": [
      { "range": { "FIELD": { "gte": 10,"lte": 20}}},
      { "terms": { "FIELD": [ "VALUE1", "VALUE2"]}}
    ]
  }
}

```

This snippet indicates that we have three clauses, where each clause houses additional leaf queries such as match, term, range, and so on. We also have a compound clause housed in the `must_not` clause, where we can further expand our criteria using the same set of clauses. As you can see, these individual queries joined together in a set of clauses leads to writing a search query that satisfies the advanced query requirements.

While you may have an understanding of the book query from a theoretical perspective, unless you write and execute queries in action, you won't really enjoy the full potential that a book query has on offer. In the next section, we delve into understanding the bool query from ground up. We build the query with one clause at a time and evolve the query as we go. Let's start our sprint with the `must` clause, discussed in the next section.

11.3.2 The must clause

The criteria we declare in a bool query's `must` clause yields positive results when the queries defined in the (clause) block satisfy the criteria. That is, the output contains all the documents that match with the conditions laid out in the `must` clause.

Because we are getting introduced to a boolean query, let's explore a simple query to begin with. Let's say our requirement is to find all TVs in our `products` index. For that, we'll write a bool query with a `must` clause. Because we are looking only for TVs, we can house the search criteria in a match query, matching products that are TVs. The following listing provides the code.

Listing 11.2 Bool query's must clause with a match query

```

GET products/_search
{
  "query": {
    "bool": { #A
      "must": [ #B
        {
          "match": { #C
            "product": "TV" #D
          }
        }
      ]
    }
  }
}

```

```

    }
}
}

#A A bool query
#B The must clause in a bool query
#C A match query that queries by author
#D The search criteria

```

Let's dissect this query a bit. The `bool` declaration inside the `query` object indicates that this is a `bool` query. The `bool` object is then wrapped with a `must` clause with the criteria to match the given search word, `TV`. When we execute this query, we get a few TVs returned as expected (not showing the output due to brevity).

A MATCH QUERY IS A BOOL QUERY

The match query we've worked with so far is indeed a type of Boolean query. For example, the same `bool` query we saw in listing 11.2 can be rewritten as a `match` (full-text) query to fetch the TVs. The following code snippet shows that this leaf query returns exactly those TVs that were returned by the previous `bool` query.

```

GET books/_search
{
  "query": {
    "match": {
      "author": "Joshua"
    }
  }
}

```

You may be inclined to use a `match` query for a simple (single) criteria, but, alas, there's more complexity in the real world; hence, you may have to lean on the `bool` query. Let's then explore how we can enhance the `bool` query.

11.3.3 Enhancing the must clause

Fetching TVs isn't that exciting, is it? Let's make it a bit more interesting. In addition to fetching TVs, let's add a condition: fetch only TVs whose value is in a certain price range. This means we have two queries that need to be joined together to achieve what we want: TVs within a price range.

This requirement seems to ask us to use a `must` clause with two `match` queries: one to match on a `product` field and another to match on the `price`. Remember, the `must` clause accepts an array of leaf level queries? While a `match` query is sufficient to search on the type of the product, we can add a `range` query to a `bool` query to fetch all TVs fitting in a certain price range. The following listing shows this query in action.

Listing 11.3 Finding TVs within a price range

```

GET products/_search
{
  "query": {
    "bool": {

```

```

"must": [ #A
  {
    "match": { #B
      "product": "TV"
    }
  },
  {
    "range": { #C
      "price": {
        "gte": 700,
        "lte": 800
      }
    }
  }
]
}

```

#A The must query with two individual leaf queries

#B A match query to find TVs

#C A range query with a price range

Here, we create two leaf queries with a conditional AND between the match and range queries. The queries go like this: search for TVs AND fetch those in a specified price range. The result would be just one TV because our data set has only one TV with a £799 price.

Of course, we can add further criteria to the query. For example, the query in the next listing searches for all TVs with 4k resolution, whose color is either silver or black.

Listing 11.4 Three leaf queries wrapped in a must clause

```

GET products/_search
{
  "query": {
    "bool": {
      "must": [ #A
        {
          "match": { #B
            "product": "TV"
          }
        },
        {
          "term": { #C
            "resolution": "4k"
          }
        },
        {
          "terms": { #D
            "colour": [
              "silver",
              "black"
            ]
          }
        }
      ]
    }
  }
}

```

```
}
```

#A The must query with three leaf queries
#B The match leaf query to search the TVs
#C The term query to find TVs with a 4K resolution
#D The terms query to fetch TVs in silver or black

As you can imagine, we can combine multiple full-text and term-level queries (or other leaf queries) to develop a search solution for complex and complicated criteria using a bool query. Remember, this is just scratching the surface of a bool query with one of the four clauses listed in table 11.2. We can build even more advanced queries using the other clauses; so let's discuss the `must_not` clause in the next section.

11.3.4 The `must_not` clause

The opposite to `must`, as you can guess, is the `must_not` query clause. For example, when on a shopping site, we might want to search for products with specific details, asking the retailer to ignore products that are not available. Figure 11.3 shows this as an example (taken from a UK retailer, John Lewis).

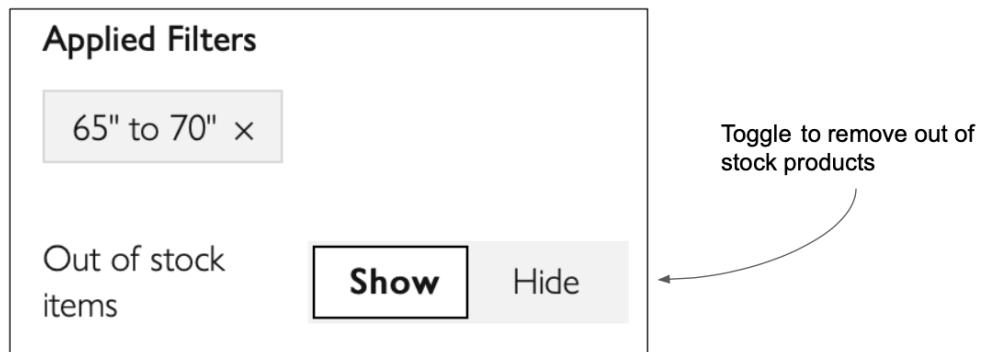


Figure 11.3: Hiding the out-of-stock items during search

The search engine hides all the out-of-stock-items before showing the results of the search illustrated in figure 11.3. This sort of functionality is what the `must_not` clause can satisfy.

Similar to the `must` clause, the `must_not` clause accepts an array of leaf queries to build advanced search criteria. The query's sole aim is to filter out matches that do *not* meet the criteria specified in the query. The best way to understand this is by an example. The query in the following listing searches for all TVs that are not of a specific brand; for example, the criteria might be fetch me any brand TV but not Samsung or Philips.

Listing 11.5 Fetching TVs using must_not to exclude certain brands

```
GET products/_search
{
  "query": {
    "bool": {
      "must_not": [ #A
        {
          "terms": { #B
            "brand.keyword": [
              "Samsung",
              "Philips"
            ]
          }
        }
      ]
    }
  }
}
```

#A A must_not clause to host a terms query

#B The terms query that searches for specific brands

This listing demonstrates the bool query provided with a must_clause, which holds a terms query. At first, the terms query, as expected, fetches products that are Samsung and Philips. However, because the terms query is wrapped in a must_not clause, the result is just the opposite: it fetches all products that are NOT manufactured by Samsung and Philips.

The problem with the query is that it fetches ALL products (TVs, Fridges, monitors, and so on). But, remember, our requirement is to get *TVs only* that are *not* manufactured by Samsung or Philips? We should modify the query to fetch TVs, omitting those particular brands.

If you are thinking of adding a must clause to the must_not clause in listing 11.5, you are correct. We can create a term query wrapped in a must clause to fetch all TVs and to remove (filter out) the specific brands from the results using must_not. Let's update the query to reflect this as the following listing demonstrates.

Listing 11.6 Fetching TVs using must_not to exclude certain brands

```
GET products/_search
{
  "query": {
    "bool": {
      "must_not": [ #A
        {
          "terms": {
            "brand.keyword": [
              "Philips",
              "Samsung"
            ]
          }
        }
      ],
      "must": [ #B
        {
          "match": {
            "brand.keyword": "TV"
          }
        }
      ]
    }
  }
}
```

```

        "product": "TV"
    }
}
}
}
```

#A The must_not clause that ignores a set of brands

#B A must clause that looks for (matches) TVs

As the listing demonstrates, we have two query clauses: a `must` and a `must_not` clause, both wrapped in a single `bool` compound query. Now we are effectively searching for TVs that are not brands of the two specific companies. Can we add more to the `must_not` clause? The `must_not` clause can be henced with multiple leaf queries, the details of which are discussed in the following section.

11.3.5 Enhancing the must_not clause

Similar to how we enhanced the query for `must` clauses (section 11.3.3), it's a no-brainer to provide multiple query criteria inside the `must_not` clause too. For example, in addition to fetching products that are not manufactured by Philips and Samsung, we can query only TVs that have 4-star ratings or above (the must not query uses a range query that filters TVs with user ratings below 4.0). This is demonstrated in the following listing.

Listing 11.7 Enhancing the must not query

```

GET products/_search
{
  "query": {
    "bool": {
      "must_not": [ #A
        {
          "terms": { #B
            "brand.keyword": [
              "Philips",
              "Samsung"
            ]
          }
        },
        {
          "range": { #C
            "user_ratings": {
              "lte": 4.0
            }
          }
        }
      ],
      "must": [ #D
        {
          "match": {
            "product": "TV"
          }
        },
        {

```

```

    "term": {
      "resolution": {
        "value": "4k"
      }
    },
    {
      "range": {
        "price": {
          "gte": 500,
          "lte": 700
        }
      }
    }
  ]
}

```

#A The must_not clause with two individual queries

#B The terms query that matches the given values in the brand field

#C The range query that fetches the TVs rated less than or equal to 4 stars

#D The must clause with a few leaf queries

Although the query in listing 11.7 is verbose, dissecting it helps us to understand its crux. This bool query is composed of two clauses: `must` and `must_not`. The `must` clause searches for all 4K resolution TVs with a price tag between £500 to £700. This list is then applied with the second clause, the `must_not` clause, which has two leaf queries working on the list of TVs produced by the `must` clause. The first query is to filter all TVs that are Philips or Samsung (that is, keep TVs that are of all brands other than Philips and Samsung). This list is further filtered by the next leaf query, the `range` query, - which drops all the TVs whose ratings are less than 4.0.

NOTE The `must_not` clause does not influence the relevance score of the returned results. This is because `must_not` queries are executed in a filter context. Queries executed in a filter context *do not* produce a score; they answer with binary results (yes or no). Hence, the score generated by other clauses such as `must` and `should` are not modified by the queries declared in the `must_not` clause.

So far, we've looked at `must` and `must_not` clauses to construct our compound queries. The queries from these clauses are located on either side of the scale: the `must` clause is particular about the criteria matching exactly (what it says on the tin, so to say). The `must_not` clause is exactly the opposite; it brings forward the results that do not match any of the criteria.

In our earlier examples, we fetched all TVs that matched a certain condition while dropping certain brands. Suppose we want to fetch results for a partial match; for example, fetching TVs that are sized greater than 85 inches OR manufactured by a specific company? This type of query is supported by the third clause of the bool query, the `should` clause, and it's the topic of the next section.

11.3.6 The should clause

Simply put, the `should` clause is an OR clause that evaluates the search based on an OR condition (whereas the `must` clause is based on the AND operator). For example, look at the query in the following listing.

Listing 11.8 Using a should query with a few criteria to fetch TVs

```
GET products/_search
{
  "_source": ["product", "brand", "overview", "price"],
  "query": {
    "bool": {
      "should": [ #A
        {
          "range": { #B
            "price": {
              "gte": 500,
              "lte": 1000
            }
          }
        },
        {
          "match_phrase_prefix": { #C
            "overview": "4K Ultra HD"
          }
        }
      ]
    }
  }
}
```

#A The `should` clause with two individual queries

#B The query to match products within a specific price range

#C The query to search products matching a phrase in the `overview` field

The `should` clause is made of two queries, searching for products that lie in the price range of £500 to £1,000 OR products with a phrase that says “4K Ultra HD” in the `overview` field. The results would be a lot number than you might have expected (remember the OR condition) as this response demonstrates:

```
{
  ...
  "_score" : 12.059638,
  "_source" : {
    "overview" : "... 4K Ultra HD display ...",
    "product" : "TV",
    "price" : 799,
    "brand" : "Samsung"
  }
},
{
  ...
  "_score" : 11.199882,
  "_source" : {
    "overview" : "... 4K Ultra HD ...",
    "product" : "TV",
    "price" : 799,
    "brand" : "Samsung"
  }
}
```

```

    "product" : "TV",
    "price" : 639,
    "brand" : "Panasonic"
}
},
{
...
"_score" : 10.471219,
"_source" : {
    "overview" : "... 4K Ultra HD screen.. ",
    "product" : "TV",
    "price" : 1599,
    "brand" : "LG"
}
}
...

```

As you can see, the results have products that are not in the specified price range (for example, the third product in the returned result is \$1,599 (way beyond what we've asked for); however, it is a match because the second criteria, 4K Ultra HD, matched. This indicates the `should` clause operates on the OR condition.

There's a bit more than using the OR condition on queries when executing the search for a `should` clause. Although we just ran the `should` query, usually it is expected to be joined with other clauses like `must` and `must_not`. The advantage of using a `should` clause with a `must` clause is that the results that match the query in the `should` clause get a boosted score. Let's discuss this in detail in the next section, using an example.

BOOSTING THE SCORE WITH SHOULD

The query in listing 11.8 demonstrates the `should` clause, which returns positive results by using an OR condition. The `should` clause adds weight to the relevance scoring when used in conjunction with the `must` clause. Let's say we issue a `must` clause that matches an LG TV as the following listing shows.

Listing 11.9 Fetching TVs using a must query

```

GET products/_search
{
  "_source": ["product","brand"],
  "query": {
    "bool": {
      "must": [#A
        {
          "match": {
            "product": "TV"
          }
        },
        {
          "match": {
            "brand": "LG"
          }
        }
      ]
    }
  }
}

```

```
}
```

#A The must clause with two individual match queries

There's not much to dissect in this query except the `_score` mentioned in the result, which is 4.4325914. The following snippet shows our score:

```
"hits" : [
  {
    "_index" : "products",
    "_id" : "5",
    "_score" : 4.4325914,
    "_ignored" : [
      "overview.keyword"
    ],
    "_source" : {
      "product" : "TV",
      "brand" : "LG"
    }
  }
]
```

Now, let's add a `should` clause to this query. The following listing demonstrates the effect on the score.

Listing 11.10 Adding the `should` clause to boost the score

```
GET products/_search
{
  "_source": ["product","brand"],
  "query": {
    "bool": {
      "must": [ #A
        {
          "match": {
            "product": "TV"
          }
        },
        {
          "match": {
            "brand": "LG"
          }
        }
      ],
      "should": [ #B
        {
          "range": {
            "price": {
              "gte": 500,
              "lte": 1000
            }
          }
        },
        {
          "match_phrase_prefix": {
            "overview": "4K Ultra HD"
          }
        }
      ]
    }
  }
}
```

```

        }
    }
}
```

#A The must clause that searches for LG TVs

#B The should clause that checks the price range or match phrase of the resulting TVs

The result of this query, using a `must` clause with a `should` clause, boosts the score of the matching documents. The earlier score of 4.4325914 is now a whopping 14.9038105 as you can see from the following snippet:

```

"hits" : [
  {
    "_index" : "products",
    "_id" : "5",
    "_score" : 14.9038105,
    "_ignored" : [
      "overview.keyword"
    ],
    "_source" : {
      "product" : "TV",
      "brand" : "LG"
    }
  }
]
```

The score increased because the query not only was successful in the `must` clause, but it also matched in the `should` clause. Hence, the takeaway is that if the query in a `should` clause matches (in addition to the positive matches in the `must` clause), the score increases. Having said that, how many queries should it match? There could be a handful of leaf queries in a `should` clause, right? Do all leaf queries need to match or can we ask Elasticsearch to check if at least one of the leaf queries matches. This can be achieved by a `minimum_should_match` attribute, which is discussed in the next section.

THE MINIMUM SHOULD MATCH SETTING

When we run a set of queries in a `must` clause, along with some queries in a `should` clause, the following rules are applied implicitly:

- All the results must match with the ones that were declared in the `must` clause (the query will not return positive search results if one of the `must` queries fails to match the criteria).
- There is no need for any of the results to match with the queries declared in the `should` clause. If they match, well, the `_score` is boosted; otherwise, there's no effect on the score.

Sometimes, however, we may need at least one of the `should` criteria to be matched before sending the results to the client. We'll want the score to be boosted, based on these `should` query matches. This is achieved by using the `minimum_should_match` attribute.

In listing 11.10, we can declare the query to be successful *if and only if at least one* of the matches is positive; it returns positive results (with a boosted score) only if one of the many queries that were declared match. The following listing demonstrates this.

Listing 11.11 Using the `minimum_should_match` parameter

```
GET products/_search
{
  "_source": ["product", "brand", "overview", "price", "colour"],
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "product": "TV"
          }
        },
        {
          "match": {
            "brand": "LG"
          }
        }
      ],
      "should": [
        {
          "range": {
            "price": {
              "gte": 500,
              "lte": 2000
            }
          }
        },
        {
          "match": {
            "colour": "silver"
          }
        },
        {
          "match_phrase_prefix": {
            "overview": "4K Ultra HD"
          }
        }
      ],
      "minimum_should_match": 1 #A
    }
  }
}
```

#A Matches at least one leaf query in the `should` clause

In the listing, we set `minimum_should_match` to 1. This means that the query tries to match the criteria defined in the leaf queries of the `should` clause but with a condition that at least one of the queries must be a positive match. We get a positive match (for example, reading from listing 11.11) if a product is silver OR a 4K Ultra HD OR the price range is between £500 to £2,000. However, if none of the criteria in the `should` clause match, the query fails as we are asking the query to satisfy the `minimum_should_match` parameter.

As you may have guessed, the `bool` query can be declared with a `should` clause but not without a `must` clause. Based on whether a `bool` query comprises a `must` clause along with the `should` clause, the default value of `minimum_should_match` attribute varies. The default

value of `minimum_should_match` is set to 0 if the bool query consists of a `should` with a `must` clause; otherwise, it is set to 1 with only a `should` clause. This is represented in table 11.3.

Table 11.3 The default values of the `minimum_should_match` attribute

Combination of the clauses	Default value
should clause only	1
should with a must clause	0

So far, we looked at `must`, `must_not`, and `should` clauses. Although `must_not` runs in a filter context, `must` and `should` run in a query context. We discussed the query context versus filter context in chapter 8, but for completeness, let's touch base with these concepts here.

Queries run in a query context execute the appropriate relevancy algorithm so we can expect relevancy scores associated with the resultant documents. Whereas queries in a filter context do not output scores and are pretty performant because the execution of the scoring algorithm isn't needed. That leads to a clause that doesn't use relevance scoring but works in filter context—the `filter` clause. Let's check that out in the next section.

11.3.7 The filter clause

The `filter` clause fetches all the documents that match the criteria, similar to the `must` clause. The only difference is that the `filter` clause runs in a filter context, and thus, the results are not scored. Remember, running a query in filter context speeds up the query performance because it caches the query results returned by the Elasticsearch. The following listing demonstrates the filter query in action.

11.12 The filter clause in action

```
GET products/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "term": {
            "product.keyword": "TV"
          }
        },
        {
          "range": {
            "price": {
              "gte": 500,
              "lte": 1000
            }
          }
        }
      ]
    }
  }
}
```

```

        ]
    }
}
```

#A Executes the query in a filter context implicitly

The following code snippet shows the results for this query. Note the scoring of the results; it is zero.

```

"hits" : [
    ...
    "_score" : 0.0,
    "_source" : {
        "product" : "TV",
        "colour" : "silver",
        "brand" : "Samsung"
    }
],
{
    ...
    "_score" : 0.0,
    "_source" : {
        "product" : "TV",
        "colour" : "black",
        "brand" : "Samsung"
    }
}
...
]
```

As you can see from the results, the `filter` clause provides no scoring. Because the scoring is not required for the output, Elasticsearch may cache the query/results, so this benefits the application's performance.

We usually combine a `filter` clause with a `must` clause. The results of the `must` clause are fed through the `filter` clause, which filters nonfitting data. The example in the next listing demonstrates this approach.

11.13 Filter clause in action

```

GET products/_search
{
    "_source": ["brand","product","colour","price"],
    "query": {
        "bool": {
            "must": [
                {
                    "match": {
                        "brand": "LG"
                    }
                }
            ],
            "filter": [
                {
                    "range": {
                        "price": {
                            "gte": 500,
                            "lte": 1000
                        }
                    }
                }
            ]
        }
    }
}
```

```

        }
    ]
}
}
```

Here, we fetch all LG products (we have 1 TV and 3 Fridges manufactured by LG in our stash). Then we filter them by price, leaving only 2 Fridges that fall in our price range (both are \$900) as this code snippet shows:

```
"hits" : [{  
    ..  
    "_score" : 2.6820748,  
    "_source" : {  
        "product" : "Fridge",  
        "colour" : "Matte Black",  
        "price" : 900,  
        "brand" : "LG"  
    }  
,{  
    ..  
    "_score" : 2.6820748,  
    "_source" : {  
        "product" : "Fridge",  
        "colour" : "Matte Black",  
        "price" : 900,  
        "brand" : "LG"  
    }  
}]
```

Note that both of the returned documents now carry a score, meaning that the query was executed in the query context. As we discussed, the filter query is similar to a must query except that it's not run in a query context (the filter query is run in a filter context). Therefore, adding filters does not affect the scoring of the documents.

So far, we worked with bool queries employing individual clauses with individual leaf queries. We can combine all these clauses to construct a complex advanced query. Let's look at that in the next section.

11.3.8 All clauses combined

Let's combine all the `must`, `must_not`, `should`, and `filter` clauses together. The requirement is to find products manufactured by LG that are not silver, they should either be a Fridge Freezer or have an energy rating of A++, and the products should meet a specific price range.

The query in listing 11.12 fetches the LG products with a `match` query in a `must` clause, ignores the color silver in a `must_not` clause, and queries for a Fridge Freezer OR a certain energy rating using a `should` clause. Finally, we use a `filter` clause to check the price of the products to see if they fit in a specific price range.

Listing 11.14 All clauses combined

```
GET products/_search
```

```
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "brand": "LG"
          }
        }
      ],
      "must_not": [
        {
          "term": {
            "colour": "silver"
          }
        }
      ],
      "should": [
        {
          "match": {
            "energy_rating": "A++"
          }
        },
        {
          "term": {
            "type": "Fridge Freezer"
          }
        }
      ],
      "filter": [
        {
          "range": {
            "price": {
              "gte": 500,
              "lte": 1000
            }
          }
        }
      ]
    }
  }
}
```

This listing combines the four clauses using a must query, matching LG TVs but not those that are silver(`must_not` clause). However, if we have a product with an A++ energy rating or the product is a Fridge Freezer (`should` clause), that's even better. Finally, we filter the products by price (`filter` clause).

We can enhance the query with further clauses and leaf queries as more complex requirements are added to the list. There is no restriction on how many queries that a clause can contain because it depends entirely on your discretion. Having said that, how do we know which query of all the leaf queries in various clauses really matched to get the results? Surely, a few of the queries must've been omitted, based on the requirement.

Knowing the exact leaf query that was picked up for the final outcome is a great way to identify the source queries. We can do this by setting a name to each and every query, which is what we briefly go over in the next section.

11.3.9 Named queries

Sometimes, we might have dozens of queries built for a complex query; however, we really have no clue how many of those queries were actually used in a match to get the final results. We can name our individual queries so that Elasticsearch outputs the results along

with the names of the queries that it uses during the query match. Let's look at this in an example. The code the next listing demonstrates a complex query with all clauses and a few leaf queries.

Listing 11.15 Complex queries given individual names

```
GET products/_search
{
  "_source": ["product", "brand"],
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "brand": {
              "query": "LG",
              "_name": "must_match_brand_query" #A
            }
          }
        },
        {
          "must_not": [
            {
              "match": {
                "colour.keyword": {
                  "query": "black",
                  "_name": "must_not_colour_query" #B
                }
              }
            }
          ]
        },
        "should": [
          {
            "term": {
              "type.keyword": {
                "value": "Frost Free Fridge Freezer",
                "_name": "should_term_type_query" #C
              }
            }
          },
          {
            "match": {
              "energy_rating": {
                "query": "A++",
                "_name": "should_match_energy_rating_query" #D
              }
            }
          }
        ],
        "filter": [
          {
            "range": {
              "price": {
                "gte": 500,
                "lte": 1000,
                "_name": "filter_range_price_query" #E
              }
            }
          }
        ]
      }
    }
  }
}
```

```

        }
    }
}
```

```

#A Query's name for matching the brand in a must clause
#B Query's name for not matching a particular color
#C Query's name for matching the type in a should clause
#D Query's name for matching the energy rating in a should clause
#E Query's name for matching the price range in a filter clause
```

The individual leaf queries are tagged with `_name` property with a value of our choice. Once executed, the response contains a `matched_queries` object attached to the individual result. Enclosed in this `matched_queries` is the set of queries that were matched to fetch the document. See the following snippet, which demonstrates this:

```

"hits" : [
  {
    ...
    "_source" : {
      "product" : "Fridge",
      "brand" : "LG"
    },
    "matched_queries" : [
      "filter_range_price_query",
      "should_match_energy_rating_query",
      "must_match_brand_query",
      "should_term_type_query"
    ],
  },
  {
    ...
    "_source" : {
      "product" : "Fridge",
      "brand" : "LG"
    },
    "matched_queries" : [
      "filter_range_price_query",
      "should_match_energy_rating_query",
      "must_match_brand_query",
      "should_term_type_query"
    ]
  }
]
```

Both the results are matched on the same four queries mentioned in the `matched_queries` block. The real benefit of naming the queries is to remove redundant queries that are not associated with the outcome. This way, you can reduce the size of the query and concentrate on tweaking the queries that were part of fetching the results.

That's pretty much how we conclude the most popular advanced query, the bool query, which is one of the most important and sophisticated compound queries. In the next few sections, we will look at other compound queries, starting with constant scores.

11.4 Constant scores

Previously, we looked at a filter query within a `bool` clause. For completeness, let's rerun a sample filter query for fetching products with a user rating between 4 and 5. The following listing shows the query.

Listing 11.16 The filter clause declared in a bool query

```
GET products/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "user_ratings": {
              "gte": 4,
              "lte": 5
            }
          }
        }
      ]
    }
  }
}
```

As you can expect, the query results in all products matching the criteria of user ratings. The only point of interest is that the query is executed in a filter context, and hence, no score (zero) is associated with all the results. However, there could be a need to set a nonzero score, especially when we want to boost a particular search criteria over another. This is where a new query type, called `constant_score`, comes into the picture.

As the name suggests, `constant_score` wraps a filter query and produces the results with a predefined (boosted) score. The query in the following listing demonstrates this in action.

Listing 11.17 A constant_score query that produces a static score

```
GET products/_search
{
  "query": {
    "constant_score": { #A
      "filter": { #B
        "range": {
          "user_ratings": {
            "gte": 4,
            "lte": 5
          }
        }
      },
      "boost": 5.0 #C
    }
  }
}
```

#A Declares the `constant_score` query

```
#B Wraps up a filter query
#C Boosts the results using a predefined score
```

The `constant_score` query in this listing wraps a filter query. It also has another attribute, `boost`, which enhances the score with the value set in this attribute. All the resulting documents are therefore stamped with a score of 5 rather than zero.

If you are wondering about the practical use of this `constant_score` function, then look no further. The following listing shows a `bool` query where we wrap a `constant_score` function into a `must` query alongside a `match` query.

Listing 11.18 A bool query with a constant score

```
# Bool query with a constant score
GET products/_search
{
  "query": {
    "bool": {
      "must": [{ #A
        "match": { #B
          "product": "TV"
        }
      },
      {
        "constant_score": { #C
          "filter": {
            "term": {
              "colour": "black"
            }
          },
          "boost": 3.5
        }
      }
    }
  }
}
```

```
#A The must clause with two queries: a match and a constant_query
#B The match query that searches for TVs
#C The constant_score that boosts the score by 3.5 if the color of the TV found is black
```

If you check the query, the `must` clause in this `bool` query houses two queries: a `match` and a `constant_score`. The `constant_score` query filters all the TVs based on a certain rating as expected but with a tweak: it boosts the score by 3.5 for all black TVs. Here, we ask the Elasticsearch engine to take our input when scoring the result by setting the `boost` value to our choice in a filtered query wrapped under `constant_score`.

In this section, we saw query results painted with a static score using the `constant_score` function. However, what if we want part of the results to be scored higher with part of the results set at the bottom of the results page? That's exactly what a boosting query does, which we discuss in the following section.

11.5 The boosting query

There are times where we want to have biased answers. For example, we may want the result set to have LG TVs on the top and at the same time another brand, say Sony, to go at the bottom of the list. This sort of biased manipulation of the scoring so the list consists of favored items at the top is done by a boosting query. A boosting query works with two sets of query parts: a *positive* part, where any number of queries produce a positive match and a *negative* part, which matches to queries to negate the score by a negative boost.

Let's consider an example where we want to search for LG TVs, but if the price of them is greater than \$1,500, we drop them to the bottom of the list with a score calculated by the value specified by the negative boost of the negative query. Let's see how this is demonstrated in this listing.

Listing 11.19 A bool query with a constant score

```
GET products/_search
{
  "size": 50,
  "_source": ["product", "price", "colour"],
  "query": {
    "boosting": { #A
      "positive": { #B
        "term": {
          "product": "tv"
        }
      },
      "negative": { #C
        "range": {
          "price": {
            "gte": 2500
          }
        }
      },
      "negative_boost": 0.5 #D
    }
  }
}
```

#A A boosting query in action

#B The boosting query's positive part

#C The boosting query's negative part

#D The negative boost

The boosting query as shown has two parts: a positive part and negative part. In the positive part, we simply create a query (a term query in this case) to fetch TVs. On the other hand, we don't want TVs that are too pricey, so we want to suppress (move to the bottom of the list) the pricey TVs from the results by negating the scores of those that are matched in the negative part of the query. The value set by the `negative_boost` attribute is used to recalculate the score to those that were of a match in the negative part. This makes the results from the negative part to be pushed down the list.

11.5.1 Boosting query combined with bool queries

The boosting query we saw a moment ago is a simple one: it uses leaf queries in the positive and negative parts. However, we can have a compound query such as boosting query scripted with other compound queries too. We can declare the boosting query with `bool` or `constant_score` or other compound queries including top-level leaf queries. The query in the next listing shows exactly this: a boosting query with embedded bool queries (unlike the code script in listing 11.19).

Listing 11.20 Bool query with a constant score

```
GET products/_search
{
  "size": 40,
  "_source": ["product", "price", "colour", "brand"],
  "query": {
    "boosting": { #A
      "positive": {
        "bool": { #B
          "must": [
            {
              "match": {
                "product": "TV"
              }
            }
          ]
        }
      },
      "negative": { #C
        "bool": {
          "must": [
            {
              "match": {
                "brand": "Sony"
              }
            }
          ]
        }
      },
      "negative_boost": 0.5 #D
    }
  }
}
```

#A Declares a boosting query

#B Defines the positive part of the boating query in a bool query, which is further coded with a must clause within a match query

#C Defines the negative part of the query with an embedded bool query

#D Sets the negative_boost to 0.5 on successful matches for queries from negative part

This boosting query consists of both positive and negative parts as expected, and the negative part has a `negative_boost` value set to `0.5`. What it does is this: once the TVs are searched (as shown in the positive block) and then negated by `0.5` if any of the TV brands are Sony, the Sony TVs, although they may be excellent, are sent to the bottom of the result

set. This is because we manipulated the score of Sony TVs using the `negative_boost` setting.

The boosting query therefore helps us downplay a certain type of document by using a negative score. This way, we can prepare the results by manipulating the score based on the negative query and negative boost. With that discussion on a boosting query, let's now jump to another compound query, called the disjunction max (or `dis_max`) query.

11.6 Disjunction max (`dis_max`) query

In the last chapter on full text searching, we worked with a query called `multi_match`, which searched for words across multiple fields. If we want to search for a smart TV across two fields, `type` and `overview`, we can use `multi_match`. For completeness, here's the resulting `multi_match` query.

Listing 11.21 Searching across multiple fields using a `multi_match` query

```
GET products/_search
{
  "query": {
    "multi_match": {
      "query": "smart tv",
      "fields": ["type", "overview"]
    }
  }
}
```

The reason we bring up the `multi_match` query under the heading of a disjunction max (`dis_max`) query is that the `multi_match` query uses the disjunction max query behind the scenes. That brings us to the topic of this section: the `dis_max` query.

The `dis_max` query wraps a number of queries and expects at least one of the queries to match. If more than one query matches, it returns the document(s) with the highest relevance score. Let's replay the same query from listing 11.19 but this time using `dis_max` (search for the words *smart tv* in the `type` and `overview` fields).

Listing 11.22 Disjunction maximum (`dis_max`) in action

```
GET products/_search
{
  "_source": ["type", "overview"],
  "query": {
    "dis_max": { #A
      "queries": [{ #B
        "match": {
          "type": "smart tv"
        }
      },
      {
        "match": {
          "overview": "smart tv"
        }
      }]
    }
  }
}
```

```

    }
}

#A Declares the dis_max query that wraps a set of queries
#B Declares a set of queries with match conditions

```

The `dis_max` query is a compound query that expects a number of leaf queries defined in the `queries` object. As the query in listing 11.20 shows, we declare two match queries, searching for multiple words in two different fields, the `type` and `overview` fields.

When searching for multiple words across multiple fields, Elasticsearch uses the *best_fields strategy*: a strategy that favors the document that has all the words present in the given fields. For example, let's say we are searching for "smart TV" across two fields, `overview` and `type`. We can expect that the document that consists of having this phrase in both fields is highly relevant, rather than the document with "smart" in the `overview` field and "TV" in the `type` field. This strategy is called the `best_fields` strategy.

Having said that, when executing the `dis_max` query on multiple queries, we can consider the scores from other matching queries too. In this case, we use a tie breaker to add the scores from other field matches, not just best fields. Let's just add the `tie_breaker` attribute to the following query as this listing shows.

Listing 11.23 Disjunction maximum (dis_max) with a tie-breaker

```

GET products/_search
{
  "_source": ["type", "overview"],
  "query": {
    "dis_max": {
      "queries": [
        {
          "match": {
            "type": "smart tv"
          }
        },
        {
          "match": {
            "overview": "smart TV"
          }
        },
        {
          "match": {
            "product": "smart TV"
          }
        }
      ], "tie_breaker": 0.5
    }
  }
}

```

In this case, we multiply the other non best fields' scores with this tie breaker value and add that to the score of those documents matching multiple fields. The `tie_breaker` value is a positive floating-point number between 0.0 and 1.0 (default is 0.0).

The final compound query we'll look at is the function score query, where we have much more flexibility when assigning the scores as per our needs by using some predefined functions. We discuss this in the next section.

11.7 The function score queries

At times, we may want to assign a score to a returned document from a search query that's based on some in-house requirements such as giving weightage on a particular field or randomly splashing a sponsor's advertisement based on the random relevance score. The function score (`function_score`) queries help create a score based on user-defined functions, including random, script-based, or some decay functions (such as gauss, linear, etc.).

Before we start working with the function score query, let's execute the query as shown in the following listing. It provides a simple and straightforward term query that returns documents, where the first document has a score of 1.6376086.

Listing 11.24 Term search with a standard term query

```
GET products/_search
{
  "query": {
    "term": {
      "product": {
        "value": "TV"
      }
    }
  }
}
```

The query isn't doing much other than searching for a TV using a standard term query. The only point you need to note is the score of the top document returned by this query: 1.6376086.

Although this query is purposely a simple one, in reality, there would be some queries that may have to undergo heavy processing to compute the relevance score. Maybe we aren't really interested in fetching the actual score computed by Elasticsearch's BM25 relevance algorithm (or any custom one) because we would like to create a score based on a self requirement. This is where we can use function score queries to generate a score based on a few user-defined functions. We can wrap the same query in a compound query called function score as listing 11.25 shows, although there's not much going on with it other than the query being wrapped up in a shiny `function_score` construct.

Listing 11.25 Term search wrapped up in a function_score

```
GET products/_search
{
  "query": {
    "function_score": { #A
      "query": {
        "term": {
          "product": "TV"
        }
      }
    }
  }
}
```

```
#A A function_score wraps up a query to generate a user-defined score.
```

The function score query expects a few attributes: a query, a set of functions, how the score is expected to be applied to the documents, and so on. Rather than learning them in theory now, we will look at these as we work with hands-on examples in this section.

The user-defined functions lets us modify and replace the score with our own custom score. We can do so by plugging in a function that allows us to tweak the score as per our requirement. For example, if we want a randomly generated score, there's a simple random_score function query for that purpose. Or, we may want to compute the score based on a few field values and parameters; in which case, we can use a script_score function query. There are a couple more functions that we will have a look at in this section, but first let's start with the random_score one.

11.7.1 Random_score function

As the name suggests, the random_score function creates a randomly generated score for the resulting documents. We can run the same query mentioned in listing 11.25, wrapped up in a function_score query, but this time, we specifically assign a random_score function to the query. The following listing demonstrates this.

Listing 11.26 Term search wrapped up in a random_score function

```
GET products/_search
{
  "query": {
    "function_score": { #A
      "query": {
        "term": {
          "product": "TV"
        }
      },
      "random_score": {} #B
    }
  }
}
```

#A A function_score with a term query and a function

#B A random_score function that generates and assigns a random score for each call

This function_score query is made of a term query and a random_score function. Every time you execute this query, you get a different score for the same returned document. The random scores are, well, random and, unfortunately, can't be reproduced. When we re-execute the query, expect the score to change.

What if we have a requirement of reproducing the random scores so that no matter how many times we execute the same query, the randomly generated score always returns exactly the same? For this purpose, we can tweak the random_score function by priming it with seed and field values. The example in the following listing shows the query with a random_score function initialized with a seed.

Listing 11.27 Tweaking random scores by setting seed on the function

```
GET products/_search
{
  "query": {
    "function_score": {
      "query": {
        "term": {
          "product": "TV"
        }
      },
      "random_score": {
        "seed": 10, #A
        "field": "user_ratings" #B
      }
    }
  }
}
```

#A Initializes the customized random_score with a seed

#B Computes the random score

As you can see, `random_score` is initialized with a `seed` and a `user_ratings` field value. If you execute this query more than once, you are guaranteed to get the exact (albeit random) score back. The algorithm and the mechanics to deduce the random score is beyond the scope of this book, but I advise you to consult the Elasticsearch documentation if you want to understand more about random scoring mechanics.

While the `random_score` function is one way to generate a random score, generating a static score using a scripting function is also interesting. Let's look at how we can use the `script_score` function in the next section.

11.7.2 Script_score function

Let's say we want to triple-boost (multiplying the field's value with the factor) the score of a document based on a field's value (for example, the `user_rating` of a product). In this instance, we can use a `script_score` function to compute the score based on the values of other fields (such as `user_ratings`) in the document.

Listing 11.28 Multiplying the field's value with an external parameter

```
GET products/_search
{
  "query": {
    "function_score": {
      "query": {
        "term": {
          "product": "TV"
        }
      },
      "script_score": { #A
        "script": { #B
          "source": "_score * doc['user_ratings'].value * params['factor']", #C
          "params": { #D
            "factor": 3
          }
        }
      }
    }
  }
}
```

```

        }
    }
}
```

#A The script_score function holds the key to generate a score based on the script defined in it.
#B The script object
#C The source is where we define our logic.
#D Passes the external parameters to the script

The `script_score` function is expected to produce a score, and it calculates the score based on a simple script calculation: find the `user_ratings` to multiply it by the original score and factor (the `factor` is passed via `external params`). We can construct a complicated query based on a fully fledged script if the need arises.

Scripts can create a complex scenario with parameters and field values as well as mathematical functions; for example, the square root of the average of user ratings multiplied by a given boosting factor. However, not every requirement needs such a complicated script. If the requirement is just to use a field's value, a simple way to get the same result is by using another function, called `field_value_factor`, rather than using a complex script. This is discussed in the next section.

11.7.3 Field_value_factor function

The `field_value_factor` function helps achieve the scoring by using fields without the complexity of scripting involved. The following listing demonstrates the mechanism.

Listing 11.29 Deriving the score from a field without scripting

```
GET products/_search
{
  "query": {
    "function_score": {
      "query": {
        "term": {
          "product": "TV"
        }
      },
      "field_value_factor": { #A
        "field": "user_ratings"
      }
    }
  }
}
```

#A The `field_value_factor` object that declares the field (`user_ratings` in this case)

As the script shows, the `field_value_factor` function works on a field (`user_ratings` in the listing) to produce a new relevancy score. You can add additional attributes to the `field_value_score` function. For example, you can multiply the score by using a `factor` attribute as well as applying a mathematical function such as a square root or logarithmic calculation on a field. The code in the following listing shows this in action.

Listing 11.30 Additional attributes on the field_value_factor function

```
GET products/_search
{
  "query": {
    "function_score": {
      "query": {
        "term": {
          "product": "TV"
        }
      },
      "field_value_factor": {
        "field": "user_ratings",
        "factor": 2,
        "modifier": "square"
      }
    }
  }
}
```

This script fetches the value of user_ratings from the document. It then multiplies the value by a factor 2 and then squares it.

11.7.4 Combining function scores

Although we looked at individual functions in the last few sections, we can also combine these functions together to produce an even better score. For example, the following listing demonstrates a function_score query that produces a score employing two functions: weight and field_value_factor.

Listing 11.31 Two functions producing a unified score

```
GET products/_search
{
  "query": {
    "function_score": {
      "query": {
        "term": {
          "product": "TV"
        }
      },
      "functions": [ #A
        {
          "filter": {
            "term": {
              "brand": "LG"
            }
          },
          "weight": 3 #B
        },
        {
          "filter": {
            "range": {
              "user_ratings": {
                "gte": 4.5,
                "lte": 5
              }
            }
          }
        }
      ]
    }
  }
}
```

```
        },
        "field_value_factor": {
            "field": "user_ratings",
            "factor": 5,
            "modifier": "square"
        }
    ],
    "score_mode": "avg",
    "boost_mode": "sum"
}
}
```

#A The `functions` object, which expects an array of leaf functions

#B The weight function

#C The field value factor based on the user_ratings field.

The `functions` object in this query expects a few functions (such as `weight` and `field_value_factor`), which combine to produce a unified score. The `weight`, which is the weighting function, expects a positive integer, which is used in further calculations. The original score of fetching a TV using a term query is complemented by the following:

- If the brand is LG, increase the score by a weight of 3.
 - If the user ratings are in a range of 4.5 to 5, use the `user_rating` field's value and square it by a factor of 5.

As more functions match, the score's final value can increase, thus the document may appear on the top of the list. Did you notice the two fields, `score_mode` and `boost_mode`, at the end of the script? These two attributes of the `function_score` query allow us to achieve a combined score from the original query and the score emitted by a single or many functions.

By default, the scores produced by these functions are all multiplied to get to a single score, the final score. However, we can change that behavior by setting the `score_mode` attribute in the function `score_query` to say, for example, `avg` or `max` or a few others.

The `score_mode` attribute defines how the individual scores are computed. For example, if the `score_mode` of a query is set to `sum`, the scores emitted by the individual functions are all be summed up. The `score_mode` attribute can be any of the modes such as `multiply` (default), `sum`, `avg`, `max`, `min`, and `first`.

The score from these functions will then be added (or multiplied by or averaged, etc.) to the original score of the query (the term query that finds the TVs) from the document based on the `boost_mode` parameter. The `boost_mode` parameter can be one of `multiply` (default), `min`, `max`, `replace`, `avg`, and `sum`. Should you want to learn more about the modes and the mechanics involved in function scoring, consult Elasticsearch's official documentation.

That's a wrap! This chapter introduced compound queries that are very advanced, most useful, and practical. The bool query is the Swiss army knife of all queries, and it helps build complex and complicated search queries. In the next chapter, we'll look at a few other advanced searches such as geospatial searches and join queries, stay tuned!

11.8 Summary

- Compound queries combine leaf queries to create advanced queries that satisfy multiple search criteria.
- The `bool` query is the most popular compound query, which is made of four clauses: `must`, `must_not`, `should`, and `filter`.
- The `queries_in_must_not_and_filter` clauses will not contribute to the overall relevance score. Whereas the `queries_in_must` and `should` clauses will always improve the scoring.
- The `constant_score` query wraps a filter query and produces a constant score set by the user.
- The boosting query increases the score of a positive clause while suppressing the score on the queries that aren't a match (the negative clause).
- The `dis_max` query, used by the `multi_match` query, wraps the queries and executes them individually.
- The `function_score` query sets a custom score based on a user-defined function such as a field's value, weight, or a random value.

12

Advanced search

This chapter covers

- **Geo data types**
- **Searching locations and addresses with geoqueries**
- **Using `geo_shape` to search for 2D shapes**
- **Using span queries to work with low-level positional tokens**
- **Specialized queries such as percolators**

In the last chapters, we covered searching data using term-level and full-text queries. We've also looked at some advanced queries like `bool`, `boosting`, and others. To continue building what we've learned so far and to advance the query landscape, this chapter introduces you to other types of queries; these under the umbrella of specialized queries.

We begin by looking at the searches aimed toward geolocations. The common and popular use cases involving geoqueries include searching nearby restaurants for a delivery order, finding the directions to a friend's house, popular schools within a 10 km range, and so on. Elasticsearch has first-class support for satisfying such location-related searches. It also provides a handful of geospatial queries in the name of `geo_bounding_box`, `geo_distance`, and `geo_shape` queries.

We'll then look at searching two-dimensional (2D) shapes using shape queries. Design engineers, game developers, and others can search 2D shapes in an index consisting of 2D shapes. We'll look at these in the second part of this chapter.

Then we'll delve into low-level positional queries called span queries. Although leaf queries, such as full-text and term-level, help us search data, they aren't helpful for searching at a level where we want to find the words in a particular order, their position, the exact (or

approximate) distance between the words, and so on. This is where span queries come into play.

Finally, we'll wrap up the chapter by looking at specialized queries such as `distance_feature`, `percolator`, `more_like_this`, and `pinned`. The `distance_feature` query boosts the results if our search result is nearer to a particular location; for example, searching for schools within a 10 km radius, but we want those schools with nearby parks to be given a higher priority. To append the organically found search results in a list of sponsored results, we use `pinned` queries. The `more_like_this` queries help find similarly looking documents. The final specialized query we'll look at in this section is the `percolator` query. Percolator queries help in alerting and notifying the users for their unyielded results from the past.

Let's begin by understanding the need for geospatial queries and the data types that support these queries. We'll then look at the queries provided by Elasticsearch out of the box for such search criteria.

12.1 Introducing location search

In this day and age of the internet and information, it is a common requirement to enable location-based search in apps and applications. Location-based search fetches venues or places based on proximity such as nearby restaurants, houses for sale not farther than 1 km radius, and so on. We also use location-based searches for finding the directions to a place or point of interest.

The good news is that geospatial support is a first-class citizen in Elasticsearch. Dedicated data types allow us to define a schema for indexing geospatial data, thus enabling a focussed search. The out-of-the-box data types that support geospatial data are `geo_point` and `geo_shape`, which we will look at in the next section.

Elasticsearch also provides a set of geospatial search queries such as `bounding_box`, `geo_distance`, and `geo_shape`, depending on the given use case. These queries suffice for most of the use cases, which we'll see in the sections that follow. Each of these queries satisfy a set of requirements briefly discussed in this section; however, they are discussed at great length in the sections that follow.

THE BOUNDING BOX QUERY

We, at times, may want to find out a set of locations such as restaurants, schools, or universities in a surrounding area; let's say in a square or a rectangular. We can construct a rectangle, often called a *georectangle*, by taking the set of coordinates of the top-left and bottom-right corners. These coordinates consist of a pair of longitude and latitude measurements, representing these corners.

Elasticsearch provides a `bounding_box` query that lets us search required addresses fitting in a georectangle. This query fetches the points of interest (as query criteria) inside the georectangle constructed by our set of coordinates. For example, figure 12.1 indicates the addresses enclosed in one such georectangle.

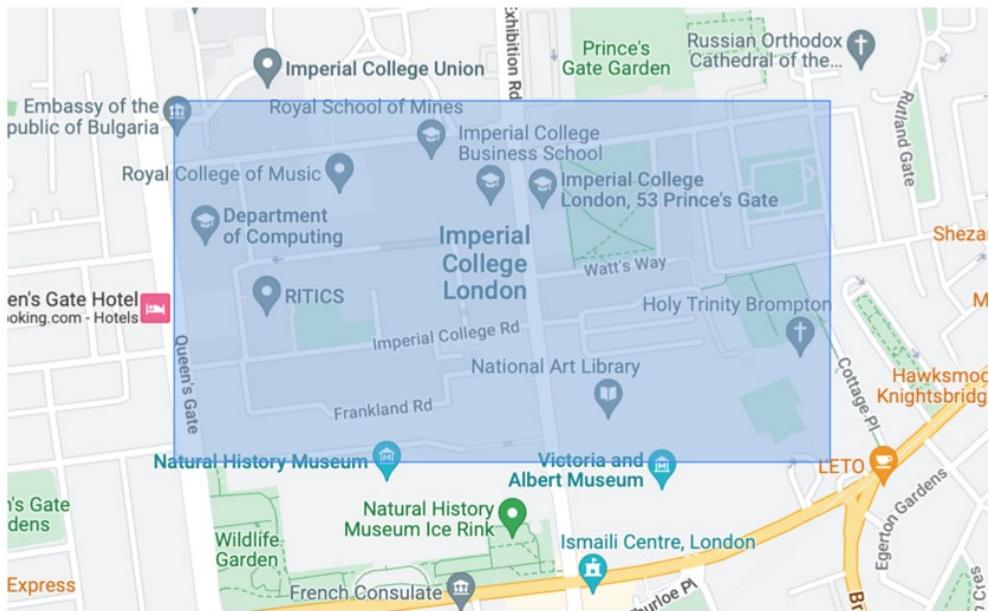


Figure 12.1 The georectangle constructed with an appropriate set of longitude and latitude coordinates.

As marked by the rectangle in figure 12.1, we are searching for addresses in central London in an area highlighted in the georectangle. Addresses intersecting this rectangle are returned as positive results. We will learn and run through some `bounding_box` queries in detail shortly.

THE GEO_DISTANCE QUERY

You may have watched Hollywood movies where an FBI agent is trying to pin down the fugitive in an area drawn as a circle around a central focal point. That's exactly what the `geo_distance` query does!

Elasticsearch provides the `geo_distance` query to fetch the addresses in an area enclosed by a circle. The given center is defined by longitude and latitude and a radius as the distance. Figure 12.2 demonstrates the geodistance concept pictorially.

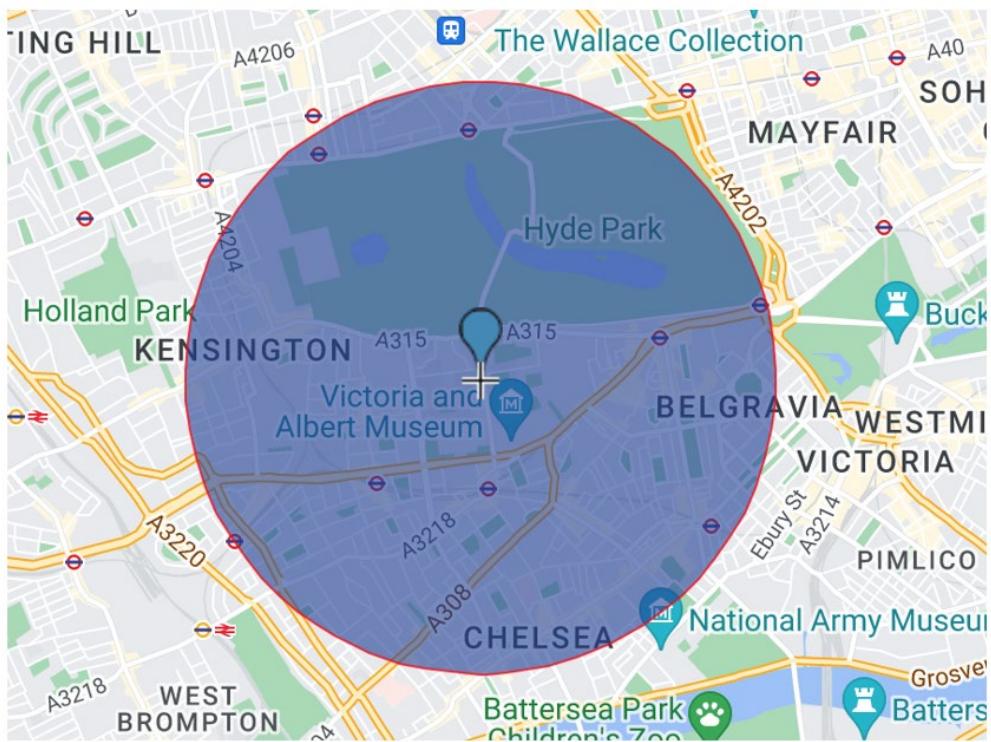


Figure 12.2 Addresses enclosed in a circular area constructed by a `geo_distance` query.

As figure 12.2 demonstrates, we have a central location (shown as the dropped pin on the map) and a circular area covering the addresses that we are looking for. The focus (or central location) is a point on the map that's dictated by latitude and longitude coordinates.

THE `GEO_SHAPE` QUERY

There's also another type of query, a `geo_shape` query. This query fetches a list of geographical points (addresses) in a given geometrically constructed geo-envelope. The envelope could be a three-sided triangle or a multi-sided polygon (except that the envelope must not be an open ended one). Figure 12.3 shows this concept pictorially.

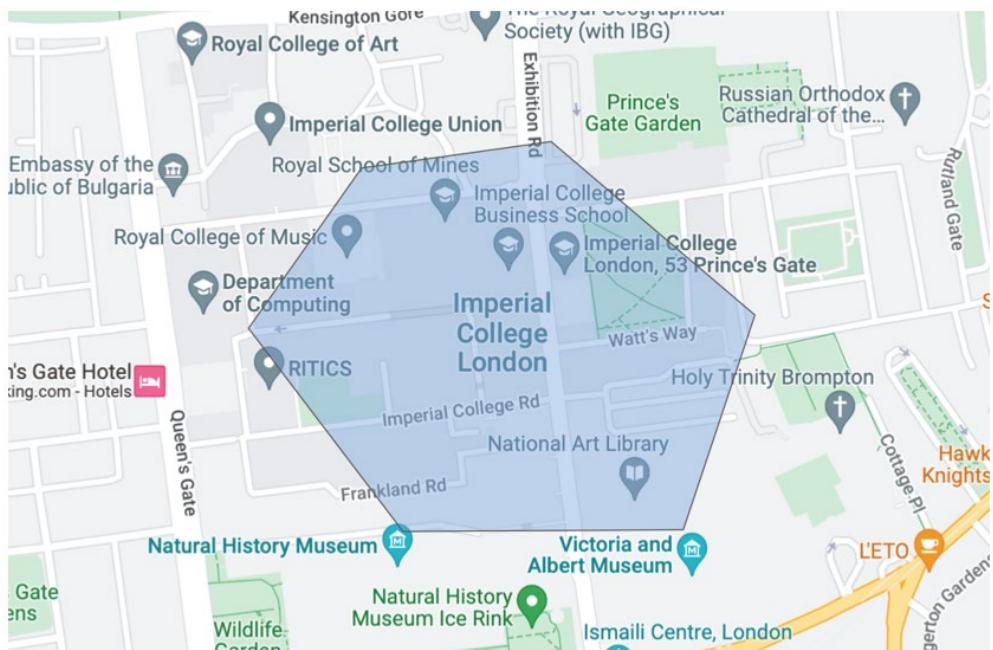


Figure 12.3 Finding the addresses in a polygonal shape with a `geo_shape` query

Figure 12.3 shows an hexagonal envelope constructed on a map with a given six pairs of coordinates (each pair is a geopoint with latitude and longitude). The `geo_shape` search finds the locations fitting inside this polygon.

Before we jump into experimenting and learning the geospatial queries in full swing, we will need to understand the mapping schema of geospatial data: the data types that support the geodata and the mechanics for indexing that data. In the next section, we will go through the `geo_point` types first, then on to the `geo_shapes`.

12.2 Geospatial data types

Similar to how the textual data is represented by the `text` data type, Elasticsearch provides two dedicated data types to work with spatial data: the `geo_point` and `geo_shape`. The `geo_point` data type expresses a longitude and latitude that works on location-based queries. The `geo_shape` type, on the other hand, lets us index geoshapes such as points, multi lines, polygons, and a few others. Let's look at these spatial data types in the following sections.

12.2.1 The geo_point data type

A location on a map is expressed universally by longitude and latitude. Elasticsearch supports the representation of such location data using a dedicated `geo_point` data type. We've looked at the `geo_point` data type earlier (briefly in chapter 4). Let's recap how we can define a field as a `geo_point` in our mapping schema. Once the mapping is ready, we can index a document. The following listing demonstrates the code for creating a data schema for the `bus_stops` index with a couple of fields.

Listing 12.1 Creating a mapping with geo_point

```
PUT bus_stops
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "location": {
        "type": "geo_point" #A
      }
    }
  }
}
```

#A Defines the location attribute as a geo_point data type

The `bus_stops` index is defined with two properties: a `name` and a `location`. The `location` is represented by a `geo_point` data type, which means it would expect to be set with latitude and longitude values when indexing the document. The following query in the next listing indexes the London Bridge Station bus stop.

Listing 12.2 Indexing a bus stop with a location defined as a string

```
POST bus_stops/_doc
{
  "name": "London Bridge Station",
  "location": "51.07, 0.08" #A
}
```

#A Inputting the station as a string with latitude and longitude values

As the query shows, the `location` field is provided with stringified latitude and longitude values separated by a comma: "51.07, 0.08". Providing the coordinates in this string format is not the only way you can set the `location` field. Fortunately, there are a bunch of formats in addition to string, such as array, well-known-text (WKT) point, and geohash, that we can use to input the `location` field's geographic coordinates. The query in the following listing provides the mechanism of these types of inputs.

Listing 12.3 Indexing in various formats for the geo_point data type

```
# As WKT point (lat, lon)
POST bus_stops/_doc
{
  "text": "London Victoria Station",
  "location" : "POINT (51.49 0.14)"
}

# As location object
POST bus_stops/_doc
{
  "text": "Leicester Square Station",
  "location" : {
    "lon": -0.12,
    "lat": 51.50
  }
}

# As an array (lon, lat)
POST bus_stops/_doc
{
  "text": "Westminster Station",
  "location" : [51.54, 0.23]
}

# As a geohash
POST bus_stops/_doc
{
  "text": "Hyde Park Station",
  "location" : "gcpvh2bg7sff"
}
```

The queries in listing 12.3 index various bus stop locations using multiple formats. As you can see, one can use a string of latitude and longitude as in listing 12.2 or, as in listing 12.3, either an object, an array, a geohash, or a WKT-formatted `POINT` shape.

Now that we understand the `geo_point` data type, it's time to learn about the second type: the `geo_shape` data type. As the name indicates, the `geo_shape` type helps index and search data using a particular shape; for example, a polygon. Let's next look at the `geo_shape` data type to understand how we can index data for geoshapes.

12.2.2 The geo_shape data type

Similar to the `geo_point` type, which represents a point on the map, Elasticsearch provides a `geo_shape` data type to represent shapes such as points, multipoints, lines, and polygons. The shapes are represented by an open standard called GeoJSON (<http://geojson.org>) and, accordingly, is written in JSON format. The geometric shapes are mapped to a `geo_shape` data type.

Let's first create the mapping for an index of `cafes` with a couple of fields. One of them is the `address` field, which points to the location of a cafe, represented as a `geo_shape` type. The following listing demonstrates this.

Listing 12.4 Creating a mapping with the geo_shape field type

```
PUT cafes
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "address": {
        "type": "geo_shape" #A
      }
    }
  }
}
```

#A Sets the address type as geo_shape

The code creates an index called `cafes` to house local restaurants. The notable field is the `address` field, which is defined as a `geo_shape` type. This type now expects inputs of shapes in GeoJSON or WKT. For example, to represent a point on a map, we can input the field using `Point` in GeoJSON or `POINT` in WKT as the code in this listing demonstrates.

Listing 12.5 Inputting geo_shape using WKT and GeoJSON formats

```
# Inputting the address in GeoJSON format
PUT cafes/_doc/1
{
  "name": "Costa Coffee",
  "address": {
    "type": "Point", #A
    "coordinates": [0.17, 51.57] #B
  }
}

# Inputting the address in WKT format
PUT /cafes/_doc/2
{
  "address": "POINT (0.17 51.57)" #C
}
```

#A Sets the address type as geo_shape

#B The coordinates (longitude and latitude) representing the Point

#C Sets the address type as POINT in WKT format

This code declares two ways to input a `geo_shape` field: using GeoJSON or WKT. GeoJSON expects a `type` attribute of an appropriate shape (`"type": "Point"`) and the corresponding coordinates (`"coordinates": [0.17, 51.57]`) as in the example. The second example in listing 12.5 shows the mechanics of creating a point using a WKT format (`"address": "POINT (0.17 51.57)"`).

NOTE There is a subtle difference when representing the coordinates using a string format versus other formats. The string format expects the values in the order of latitude and longitude separated by a comma; for example, "(51.57, 0.17)". However, the coordinates are interchanged for GeoJSON or WKT formats as longitude and latitude; for example, "POINT (0.17 51.57)".

We can build various shapes using these formats. Table 12.1 provides a brief description of few of them. I suggest that you consult the Elasticsearch documentation about how you can index and search documents to understand the concepts and examples in detail.

Table 12.1 Various shapes supported by the geo_shape data type

Shape	Description	GeoJSON representation	WKT representation
Point	A point represented by latitude and longitude	Point	POINT
Multipoint	An array of points	MultiPoint	MULTIPOINT
Polygon	A multi-edge shape	Polygon	POLYGON
Multipolygon	A list of multiple polygons	MultiPolygon	MULTIPOLYGON
Line string	A line between two points	LineString	LINESTRING
Multiline string	A list of multiline strings	MultiLineString	MULTILINESTRING

Now that we know how to work with the indexing side of geodata using the `geo_point` and `geo_shape` fields, we're now ready to search through our documents. We'll discuss this in the next section.

12.3 Geospatial queries

To locate geospatial data, our next task is to search the documents for a given geocriterion. For example, the place where you get your coffee is represented by longitude and latitude, which, in turn, is called a *point* on the map. As another example, we can search for the restaurants nearest my house, which returns each of the matching cafes represented as a point. On the other hand, a plot of land on a map can be represented by a shape, representing a country or your local school's playground.

Elasticsearch provides a set of geospatial queries specifically suited to search for these use cases (such as finding nearby addresses or searching for all the interesting spots in a given area and so on). The following briefly lists a few of these queries, but we will learn about them individually in the following sections:

- A `geo_bounding_box` query finds the documents enclosed in a rectangle constructed by geopoints. For example, finding all the restaurants located inside a georectangle.
- A `geo_distance` query finds the addresses within a certain distance from a point. For example, finding all ATMs within 1 km from the London Bridge.
- A `geo_shape` query finds addresses represented as shapes within a shape constructed by a set of coordinates. For example, finding agricultural farms contained in a green belt. The farms are represented by individual shapes as well as the green belt.

We will discuss queries for both geopoints and geoshapes in detail in the following sections. Let's begin with queries executed on `geo_point` fields.

12.4 The `geo_bounding_box` query

When we search for a list of addresses, we can use an area of interest. This area can be represented by a circle of a certain radius or an area enclosed by a shape such as a rectangle, or a polygon from a central point (a landmark).

Elasticsearch provides a `geo_bounding_box` query that lets us search locations that fall inside these areas. For example, as figure 12.4 shows, we can construct a rectangle using latitude and longitude coordinates and search if our address exists in that area.

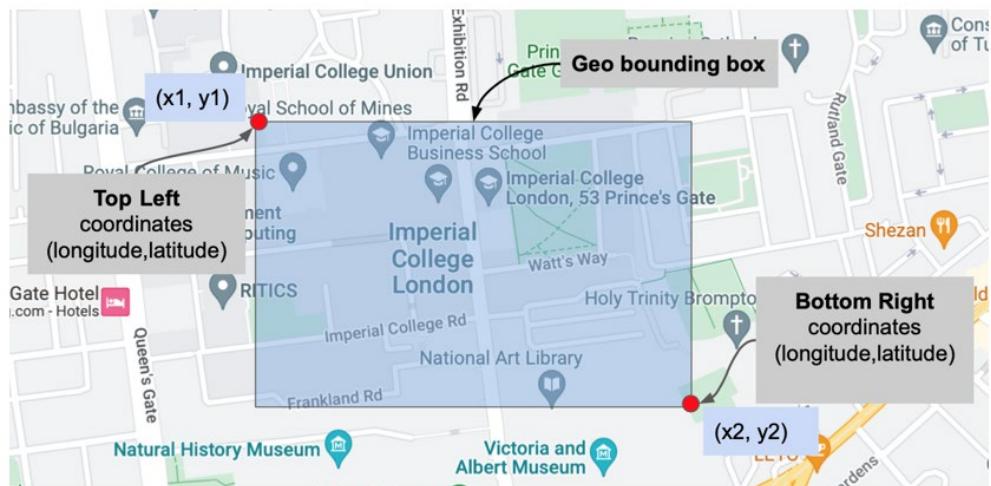


Figure 12.4 Georectangle from a set of latitude and longitude coordinates

The `top_left` and `bottom_right` fields are the coordinates of latitude and longitude that make up our georectangle. Once we define a georectangle, we can check if the points of interest (the Imperial College London, for example) are present inside this rectangle.

Before we discuss the `geo_bounding_box` query in detail, let's write the query first and then we can come back to dissect it. The following query searches all the documents (locations) that would fit in the rectangle constructed with `top_left` and `bottom_right` coordinates.

Listing 12.6 Matching restaurant locations in a georectangle

```
GET restaurants/_search
{
  "query": {
    "geo_bounding_box": { #A
      "location": { #B
        "top_left": { #C
          "lat": 52,
          "lon": 0.2
        },
        "bottom_right": { #D
          "lat": 49,
          "lon": 0.1
        }
      }
    }
  }
}
```

#A Constructs a georectangle

#B Sets the document's `geo_point` field

#C Defines the `top_left` point, formed by a pair of latitude and longitude

#D Defines the `bottom_right` point, formed by a pair of latitude and longitude

The query searches all the documents that intersect (fit in) a georectangle made by the two coordinates, `top_left` and `bottom_right`, as in figure 12.4. The user can provide these two coordinates so that we can construct a rectangular shape with them.

The restaurants that fall inside this rectangle are returned as search results, whereas the rest are dropped. In listing 12.6, we are essentially creating an area represented by a rectangle and searching for our restaurants in that rectangle.

NOTE You can also represent the vertices of the bounded rectangle as `top_right` and `bottom_left` (as opposed to `top_left` and `bottom_right`). Or, if you want, you can break it down even further by simply having the corresponding coordinates named as `top`, `left`, `bottom`, and `right`.

We provided the values of longitude and latitude as an object in listing 12.6; however, there are other ways of setting these values. We'll discuss this in the next section.

12.4.1 Working with geoshape data

We executed the previous `geo_bounding_box` query on the schema that has fields declared as `geo_point` data types. But, can we use the same query if our document consists of a field defined as `geo_shape` instead? We learned in section 12.3.3 that geospatial data can also be represented using a `geo_shape` data type, remember?

Fortunately, we can use the same query (listing 12.6) for geoshape data too, except that we must swap the URL pointing to the appropriate index. Hence, the same query can be used for geoshapes. For example, with the `cafes` index and geoshape data at hand, all we need to do is to construct the same `geo_bounding_box` query but change the URL to reflect the geoshapes index (`cafes`). The following query shows this.

Listing 12.7 Matching locations in a georectangle with geoshape data

```
GET cafes/_search #A
{
  "query": {
    "geo_bounding_box": {
      "address": { #B
        "top_left": {
          "lat": 52,
          "lon": 0.04
        },
        "bottom_right": {
          "lat": 49,
          "lon": 0.2
        }
      }
    }
  }
}
```

#A The `cafes` index with `geo_shape` data type fields
#B The `address` field is defined as a `geo_shape` data type.

This query searches for `cafes` in a given georectangle constructed with top-left and bottom-right parameters. Here, we use the `cafes` index in the URL (GET) when invoking the `geo_bounding_box` query. Other than swapping the geopoints index (`restaurants`) with an index consisting of geoshapes (`cafes`), there's no difference whatsoever in the query!

12.4.2 Multiple formats of longitude and latitude

The values of latitude and longitude can be set in multiple ways. The ones we saw in the previous `geo_bounding_box` queries provided the `top_left` and `bottom_right` attributes as an object of "lat" and "lon" as this snippet of code shows:

```
"top_left": {
  "lat": 52.00,
  "lon": 0.20
}
```

Let's look at the multiple formats that these location values can be set to in the next section. For these formats, we can use either an array or WKT values. Let's look at arrays first.

LONGITUDE AND LATITUDE AS AN ARRAY

Instead of providing the longitude and latitude as an object, we can set them as an array. There's one gotcha, however, that you need to consider when setting the values as an array: the values in the array *must be reversed*. They should be `lon` and `lat` (as opposed to `lat`

and `lon` in the previous examples). The following listing shows the same `geo_bounding_box` query, but this time with longitude and latitude (highlighted as bold) provided as an array.

Listing 12.8 Geoquery with a geopoint specified as an array

```
GET restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match_all": {}
        }
      ],
      "filter": [
        {
          "geo_bounding_box": {
            "location": {
              "top_left": [0, 52.00], #A
              "bottom_right": [0.10, 49] #B
            }
          }
        ]
      }
    }
  }
}
```

#A The `top_left` attribute with the lon and lat values
#B The `bottom_right` attribute with the lon and lat values

As you can see, we defined the `top_left` and `bottom_right` attributes as an array of two geopoints: latitude and longitude. Now, let's look at the WKT format for longitude and latitude.

LONGITUDE AND LATITUDE AS A VECTOR OBJECT

WKT is a text markup language that's standard for representing vector objects on a map. For example, to represent a point in WKT, we would write `POINT(10, 20)`, which represents a point on a map with x- and y-coordinates as 10 and 20, respectively. Elasticsearch provides WKT markup for bounding-box queries as `BBOX` with the respective values. The following query demonstrates this.

Listing 12.9 Geoquery with location represented as WKT

```
GET restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match_all": {}
        }
      ],
      "filter": [
        {
          "geo_bounding_box": {
            "location": {
              "wkt": "BBOX(0.08, 0.04, 52.00, 49.00)" #A
            }
          }
        }
      ]
    }
  }
}
```

#A wkt sets the coordinates in a WKT format

The `get_bounding_box` filter's `location` field accepts the coordinates in the form of `BBOX` with the corresponding longitude and latitude values. The `BBOX` creates a georectangle from the pair to make up `top_left` and `bottom_right` points.

Other than your own preference, there's no difference between the WKT and array formats. If you are building an application that uses WKT standards for the geodata, it makes sense to lean toward using WKT-based indexing and searching in Elasticsearch.

In this section, we learned how to find a set of locations inside a georectangle by employing the `geo_bounding_box` query. There are times where we may want to find a set of restaurants in the vicinity of a central location; say, all restaurants 10 km from the city's center. This where we can employ another query: the `geo_distance` query. This query fetches all the available locations within a circle with a central focal point. We'll discuss the `geo_distance` query in detail in the next section.

12.5 The `geo_distance` query

When we want to find a list of addresses surrounding a central point, the `geo_distance` query comes in handy. It works by circling an area with a radius of a given distance from a focal point. For example, as figure 12.5 shows, we may want to find nearby schools in the vicinity of a 10 km radius.



Figure 12.5 Returning schools with a `geo_distance` query

Let's look at the `geo_distance` query in action. The following listing defines a `geo_distance` query that fetches all the restaurants within 175 km from the given central coordinates.

Listing 12.10 Searching for restaurants within a given radius

```
GET restaurants/_search
{
  "query": {
    "geo_distance": { #A
      "distance": "175 km", #B
      "location": { #C
        "lat": 50.00,
        "lon": 0.10
      }
    }
  }
}
```

#A Declares the `geo_distance` query

#B Sets the vicinity of an area to search (distance from a central point)

#C Sets the central location, defined as a point on the map

As the listing shows, the `geo_distance` query expects two attributes: the `distance` attribute, which provides the radius of the geocircle, and the `geo` field `location`, which defines the central point of the geocircle. The query returns all the restaurants that are 175 km from the defined point.

NOTE The `distance` field accepts the distance measured in kilometers or miles provided as `km` or `mi`, respectfully. For example, Elasticsearch honors the value given as "`350 mi`" as well as "`350mi`" (with the space removed).

It shouldn't surprise you if I tell you that the field for defining the central point can be input using the latitude and longitude in the form of strings, arrays, WKT, and other formats. The queries can also be run on geoshapes too, though I will leave it to you to experiment with them.

The `geo_bounding_box` and `geo_distance` queries cater for searching our addresses represented as point-based locations in rectangular shapes and circles. However, we often require searching for an address defined as a shape inside another shape, preferably a polygonal shape. This is where we use a `geo_shape` query, discussed in the following section.

12.6 The `geo_shape` query

Not all the locations we have are presented as point-based locations (coordinates with latitude and longitude values). There are times when we want to find if a set of shapes are present inside (or outside) the boundaries of another shape or if they intersect the boundaries. For example, figure 12.6 indicates a few land plots on our London map.

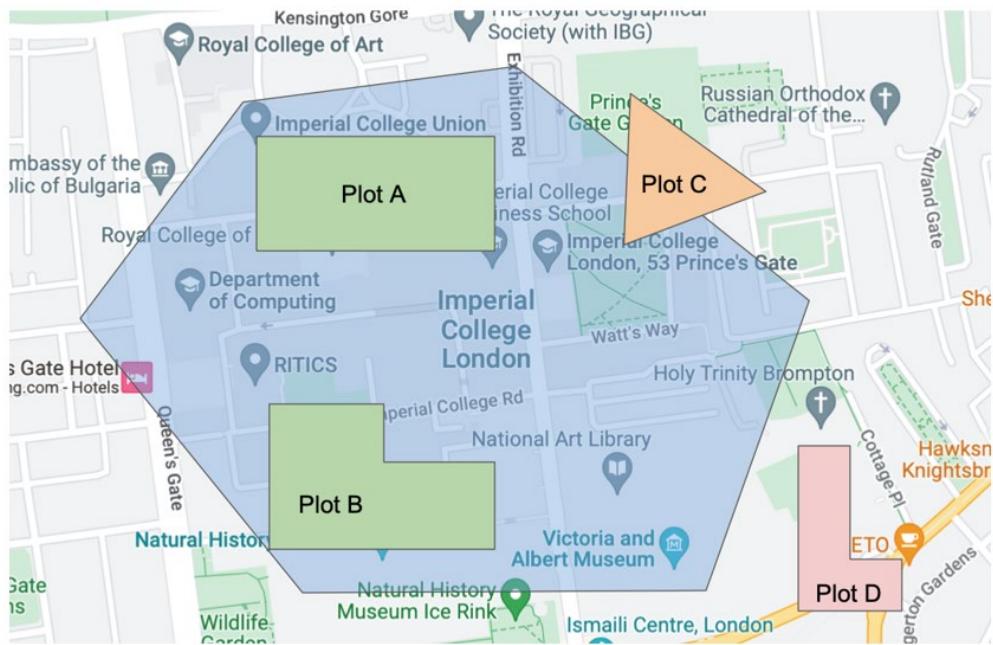


Figure 12.6 Plots of farm land in an area of London

In the figure, our geoshape is represented by the hexagon. Plots A and B are *within* the boundaries of this shape; plot C intersects the hexagon; plot D is outside the boundaries of the geoshape. Elasticsearch provides the capability via a `geo_shape` query to search these plots of various shapes in an envelope built by a set of coordinates.

The `geo_shape` query retrieves locations or addresses that are represented as shapes present in another shape such as the hexagon shape in the above figure. We construct this hexagonal shape by providing the coordinates as values to an `envelope` field.

In our `cafes` index, we have a couple of documents with point-based shapes, so let's write a query to understand how to retrieve this data. (If you are still curious, do index a few more documents with other shapes too.) The query in the following listing searches for all the cafes that fall within the shape defined by the `envelope` field with longitude and latitude pairs.

Listing 12.11 Searching for cafes within a given shape

```
GET cafes/_search
{
  "query": {
    "geo_shape": {
      "address": {
        "shape": {
          "type": "envelope", #A
          "coordinates": [
            [0.1,55],
            [1,45]
          ]
        },
        "relation": "within" #C
      }
    }
  }
}
```

#A Defines a geo_shape query that expects a field and the shape

#B Sets the shape attribute type that expects an envelope built by using a set of coordinates

#C Defines the relationship between the envelope and the resulting geoshape.

The query in the listing 12.11 fetches the documents (cafes) that fall within the envelope constructed by the given pair of longitude and latitude values. In our case, the search returns the cafe (Costa coffee) found in the given envelope. There's one last thing that we need to understand: the `relation` attribute.

The `relation` attribute defines the relationship of the documents that need to be found for the given shape. The default value of `relation` is `intersects`, which means that the query returns the documents that intersect the given shape. Table 12.2 mentions all the given values for the `relation` attribute.

Table 12.2 Relationships between the document and the envelope shape

The values for <code>relation</code>	Description
<code>intersects</code> (default)	Returns the documents that intersect with the given geometrical shape
<code>within</code>	Matches the documents if they exist within the boundaries of the given geometrical shape
<code>contains</code>	Returns the documents if they contain the geometrical shape
<code>disjoint</code>	Returns the documents that do not exist in the given geometrical shape

In our land example (figure 12.6), if we specifically set `relation=intersects`, the expected plots would be A, B, and C (as C intersects the main envelope). If we set the `relation=within`, the plots A and B are returned because they are within the bounded envelope. Setting `relation=contains` returns plots A and B because they are pretty much contained in the envelope, and it should not be a surprise if I say that plot D is a result of `relation=disjoint`.

So far, we've looked at the geospatial queries: queries on geodata represented by `geo_point` and `geo_shape` fields. These queries enable us to search the data on a map for various use cases. We shift gears and look at a different set of queries to search through two-dimensional (2D) shapes. They are called *shape queries*.

In the next section, we'll explore shape queries in detail. We use shape queries for indexing and searching 2D shapes. For example, a civil engineer's blueprint data, a machine operator's CAD (computer-aided-designs) designs, and others fit in this criteria.

12.7 The shape query

We build 2D shapes, such as lines, points, polygons, etc., using x and y Cartesian coordinates. Elasticsearch provides indexing and searching 2D objects using `shape queries`. In this section, we will briefly index and search geoshapes such as lines, rectangles, triangles, and others.

When we work with 2D data, we use the dedicated data type, `shape`. We create the fields of this `shape` type for indexing and searching 2D data. This is easy to understand by going over the following examples.

The query in the listing 12.12 demonstrates the mapping for the index `myshapes` with two properties, `name` and `myshape`. As you may notice, the `myshape` attribute is defined as the `shape` data type.

Listing 12.12 The index mapping with a shape type

```
PUT myshapes
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "myshape": {
        "type": "shape"
      }
    }
  }
}
```

Now that we have the mapping, the next step is to index a bunch of documents with different shapes. The following query indexes two documents with a point and line shape.

Listing 12.13 Indexing point and multipoint shapes

```
# Indexing a point
PUT myshapes/_doc/1
{
  "name": "A point shape",
  "myshape": {
    "type": "point",
    "coordinates": [12,14]
  }
}

# Indexing a multipoint
PUT myshapes/_doc/2
{
  "name": "A multipoint shape",
  "myshape": {
    "type": "multipoint",
    "coordinates": [[12,14],[13,16]]
  }
}
```

As you can see, the listing places a point and a multipoint, both 2D shapes, into the `myshapes` index. Figure 12.7 illustrates the two shapes: the line and the point.

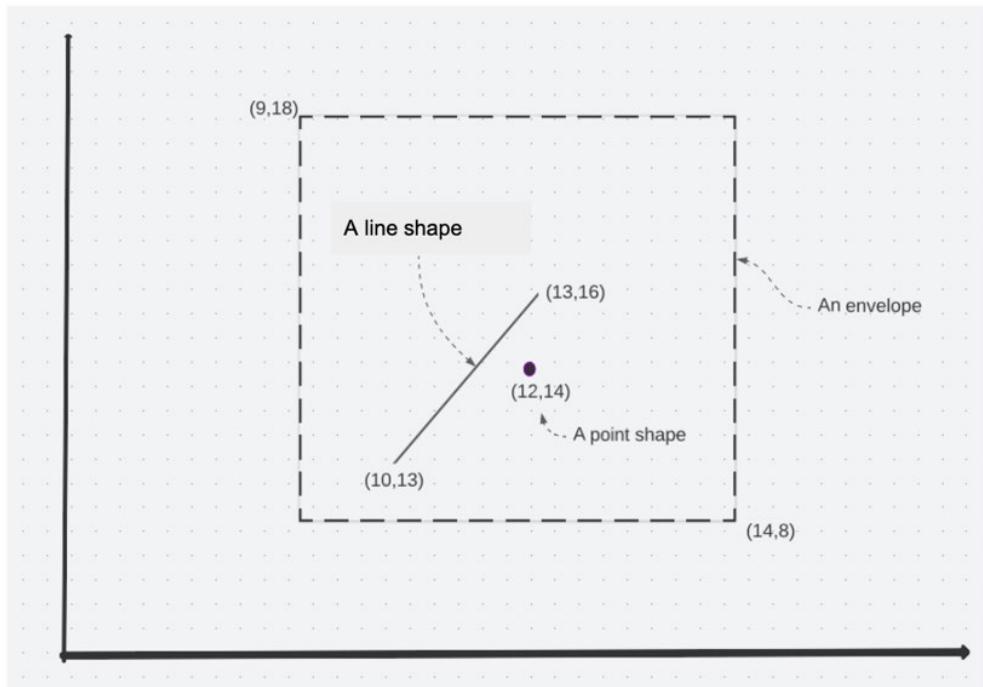


Figure 12.7 Searching for 2D shapes in a bounded envelope

We can search all the documents for shapes that fall in the geoshape enclosed in the envelope. The following query shows the code for this search.

Listing 12.14 Searching for all the shapes in a given envelope

```
GET myshapes/_search
{
  "query": {
    "shape": {
      "myshape": {
        "shape": {
          "type": "envelope",
          "coordinates": [[10,16],[14,10]]
        }
      }
    }
  }
}
```

#A Specifies the shape query

#B The field on which the query is run

#C The shape we want to construct

#D The envelope constructed by the given coordinates

The `shape` query defined in the listing searches for the documents that are contained in the envelope created from the given coordinates, `[10,16], [14,10]`. (Refer to figure 12.7 to see the bounded envelope created by this query and the shapes that are contained within.)

As the query in listing 12.14 shows, we can create a polygonal envelope using the required coordinates (make sure that the end coordinates meet because open polygons are not supported in Elasticsearch). Keep in mind that shape queries are useful when working with 2D Cartesian coordinates for drawings and designs.

We now jump to a completely different set of queries under the umbrella of specialized queries: the span queries. Span queries support searching for the terms at a particular location in the document unlike the normal search queries, where the token's position is ignored. This is best understood by running some queries; we'll do this next.

12.8 The span queries

The term-level and full-text queries work the magic of searching the token (word) level. They are not focused on the positions of the tokens (words) or even their order. Consider the following text, a famous quote by Isaac Newton:

“Plato is my friend. Aristotle is my friend. But my greatest friend is truth.”

We want to find a document (quote) where Plato and Aristotle are both mentioned but in the same order (not Aristotle and Plato), and Aristotle should be at least four positions away from Plato. Figure 12.8 shows this relationship.

Plato is my friend. Aristotle is my friend. But my greatest friend is truth

Isaac Newton

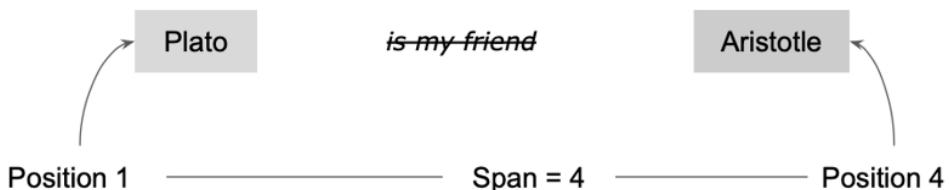


Figure 12.8 Finding a quote using a positional query

As figure 12.8 shows, Plato is at position 1 and Aristotle is at position 4, and their span is 4. Our requirement is to fetch all the quotes where Plato and Aristotle meet this specification. We cannot satisfy this requirement with the use of full-text (or term-level) queries. Although a prefix query can do a bit of justice to some extent, it cannot satisfy a few other sophisticated criteria that we will look at shortly in the coming sections.

This example demonstrates why the span queries come handy. Span queries are a set of low-level queries that help us to find documents with tokens specified by their positions. When working with legal documents, research articles, or technical books, where sentences with the words exact positions are required, we use the span queries. There are a half dozen of span queries. We'll take a peek at a few of them here, but as always, refer to documentation should you want to learn more about other span queries.

12.8.1 Sample data

Before we work with span queries, let's get the Elasticsearch primed with a `quotes` index and a couple of documents. The code in the following listing does this for us.

Listing 12.15 Priming Elasticsearch for a span query

```
# Creating a quotes index with a couple of properties
PUT quotes
{
  "mappings": {
    "properties": {
      "author": {
        "type": "text"
      },
      "quote": {
        "type": "text"
      }
    }
  }
}

# Index Newton's quote
PUT quotes/_doc/1
{
  "author": "Isaac Newton",
  "quote": "Plato is my friend. Aristotle is my friend. But my greatest friend is the
            truth."
}
```

We create the `quotes` index with a couple of properties: the `author` and `quote`, which are both text fields (`"type": "text"`). This indexes the famous quote by Isaac Newton as defined in the listing. Now that we have primed the `quotes` index, let's see some span queries in action.

There are a handful of span queries: `span_first`, `span_within`, `span_near`, and so on. Each one has a specific use case as you can imagine. We will discuss the details of most of these span queries in the next few subsections, beginning with the `span_first` query.

12.8.2 Looking at the `span_first` query

Suppose we want to find a particular word in the first n number of tokens. For example, we want to know if *Aristotle* exists in the first 5 positions of our documents as figure 12.9 illustrates.

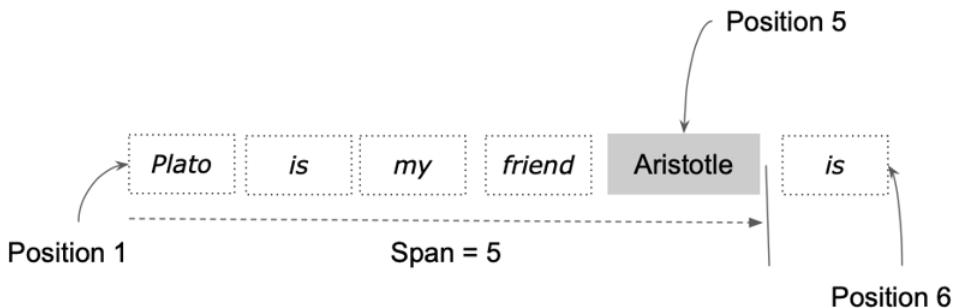


Figure 12.9 Searching for a document with a term in the first n number of tokens

As you can deduce from the figure, Aristotle is at the fifth position and, therefore, exists in the first 5 positions. This sort of use case can be satisfied by the `span_first` query. We see the query in action in the following listing.

Listing 12.16 Searching for the given term in the first 5 positions

```
GET quotes/_search
{
  "query": {
    "span_first": { #A
      "match": {
        "span_term": { #B
          "quote": "aristotle"
        }
      },
      "end": 5 #C
    }
  }
}
```

#A Fetches the document in the first n number of spans

#B The term we want to search for

#C The first of n positions to find a match

The `span_first` query expects a `match` query where we are expected to provide other span queries. In listing 12.16, the `span_first` query is wrapped in a `span_term` query. Although the `span_term` query is equivalent to a `term` query, it is commonly wrapped in other span query blocks. The `end` attribute indicates the maximum number of positions permitted when searching for the match term from the beginning of the field (in this case, `end` equals 5).

CALLOUTS The `end` attribute naming is a bit confusing. It is the end position of the token that was allowed when searching the match terms. I would've expected "`n_position_from_beginning`" to be better suited :)

In the example in listing 12.16, we search for Aristotle in the first 5 positions from the start of the quote (refer to figure 12.9 for clarity). Because Aristotle's position is indeed fifth, the document consisting of the quote is successfully returned. If you change the `end` attribute to anything less than 5, the query will not return the search term. Yes, your guess is correct! The query returns successfully if the value for the `end` attribute is anything above 5 (6, 7, 10...).

12.8.3 Looking at the `span_near` query

In the `span_first` query, the query word is always counted from the starting position (position 1). Instead of finding the word if it exists in the first n positions, sometimes we may want to find words that are nearer to each other. For example, continuing with Newton's quote, suppose we want to find if the words *Plato* and *Aristotle* are next to each other. Or, perhaps not further than 3 or 4 positions apart. Figure 12.10 depicts this requirement visually.

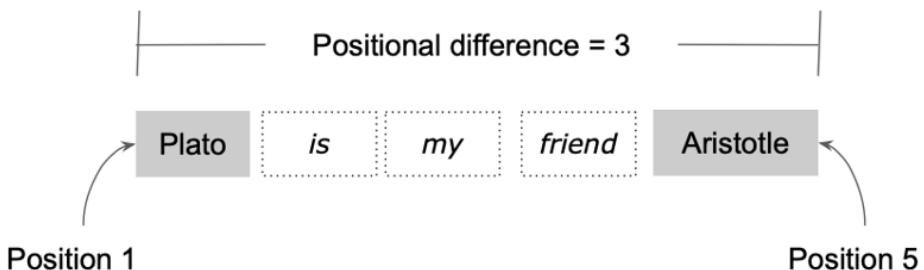


Figure 12.10 Words expected to be at a specified length.

As figure 12.10 illustrates, we will obtain positive results if we are searching for a quote where Plato and Aristotle differ in three positions. As well as finding the difference in distance between these words, we may want them to exist in the same order. The next listing shows a query to determine if Plato and Aristotle are nearer to each other (in about 3 positions from each other).

Listing 12.17 Searching for documents with terms nearer to each other

```
GET quotes/_search
{
  "query": {
    "span_near": {
      "clauses": [
        {
          "span_term": {
            "quote": "plato"
          }
        },
        {
          "span_term": {
            "quote": "aristotle"
          }
        }
      ],
      "slop": 3,
      "in_order": true
    }
  }
}
```

#A The span_near query definition with a couple of clauses

#B The clauses consisting of two independent span_terms to search individual words

#C The slop attribute permits the acceptable number of positions between the words.

#D The in_order attribute to set the order of the attributes.

There's a bit more going on in this query. The `span_near` query accepts multiple clauses; we also have `span_term` queries trying to match our terms. Additionally, because we know that these two words are three positions apart, we can provide this difference in the form of the `slop` attribute.

The `slop` attribute permits the acceptable maximum number of positional differences in between the words. For example, in the query, it specifies that the two words can be apart from each other by a maximum of three positions. You can increase the `slop` (say `"slop": 10`) if you want to drop the strict constraints. Then the query has a greater chance of success. However, the span queries are pretty good in finding the exact words at exact positions, so consider increasing the `slop` attribute judiciously.

In addition to the `slop` attribute, we may want to define the order of these words too. Going with the same example, if the order is not important, we could return a positive result even if we request a `span_near` between Aristotle and Plato (instead of Plato and Aristotle). If the order is important, we can set a Boolean for the `in_order` flag. The `in_order` attribute can be set to true or false; when set to true as in listing 12.17, the order in which the words are indexed will be taken in to account.

12.8.4 Looking at the span_within query

The next use case for span queries deals with when we want to find a word between two words. For example, we want to find documents where Aristotle is between two words (*friends* in this case). Figure 12.11 illustrates this use case.

Plato is my **friend**. Aristotle is my **friend**. But my greatest friend is truth

Figure 12.11 Finding a word if it exists between other words

As you can see from figure 12.11, we want to find if the word *Aristotle* (underlined in figure 12.17) exists between the highlighted words, *friend*. We can use the `span_within` query type for this purpose. Let's look at the query in the following listing and then come back to understand it.

Listing 12.18 Searching a word that exists between other words

```
GET quotes/_search
{
  "query": {
    "span_within": { #A
      "little": { #B
        "span_term": {
          "quote": "aristotle"
        }
      },
      "big": { #C
        "span_near": {
          "clauses": [
            {
              "span_term": {
                "quote": "friend"
              }
            },
            {
              "span_term": {
                "quote": "friend"
              }
            }
          ],
          "slop": 4,
          "in_order": true
        }
      }
    }
  }
}
```

#A A `span_within` query consisting of two blocks: `little` and `big`

#B Defines the search word

#C Encloses the `little` block

This `span_within` query consists of two blocks, `little` and `big`. The `little` block within the search expects to be enclosed in the `big` block. In this query, we want to find the documents where Aristotle is enclosed between the two words *friend*, which are defined in the `big` block.

Remember, the `big` block is nothing but a `span_near` query (we learned about this query in the previous section). There are no restrictions on how many clauses we can have in the `big` block. For example, we can extend the query in listing 12.18 with the code in the next listing, which has three clauses, each looking for the word `friend`.

Listing 12.19 Checking if a word exists in a set of words

```
GET quotes/_search
{
  "query": {
    "span_within": {
      "little": {
        "span_term": {
          "quote": "aristotle"
        }
      },
      "big": {
        "span_near": {
          "clauses": [
            {
              "span_term": {
                "quote": "friend"
              }
            },
            {
              "span_term": {
                "quote": "friend"
              }
            },
            {
              "span_term": {
                "quote": "friend"
              }
            ]
          ],
          "slop": 10, #A
          "in_order": true
        }
      }
    }
  }
}
```

#A Bumping up the slop attribute's value

This query now tries to find if Aristotle is in between a set of words (`friend`), which is defined in the `big` block. The notable change you may need to do is to bump up the `slop` value. Thus, the `span_within` queries help us identify a query within another query.

12.8.5 Looking at the `span_or` query

The last span query we'll look at is the one that satisfies the OR condition, returning results matching one *or* more given input criteria. Elasticsearch provides the `span_or` query for this. It finds the documents matching one or more span queries from a given set of clauses. For example, the following query finds the documents matching Plato or Aristotle but ignores the

word *friends* (note the plural; our document contains the word *friend* in the `quote` field and not *friends*).

Listing 12.20 Searching any matching word

```
GET quotes/_search
{
  "query": {
    "span_or": { #A
      "clauses": [ #B
        {
          "span_term": {
            "quote": "plato"
          }
        },
        {
          "span_term": {
            "quote": "friends"
          }
        },
        {
          "span_term": {
            "quote": "aristotle"
          }
        }
      ]
    }
  }
}
```

#A Defines the `span_or` query

#B Lists multiple clauses

This `span_or` query fetches the document with Newton's quote because it matches both Plato and Aristotle. Note that the `friends` query is not a match, but because the operator is `OR`, the query is happy to proceed on the basis that at least one of the words queried was a match. The query does not fail although the word *friends* is not a match.

Elastic search has other span queries such as `span_not`, `span_containing`, `span_multi_term`, and others, but unfortunately, we cannot discuss all of these types here. I advise you to consult the documentation to get a better understanding of these queries. The documentation is available here:

<https://www.elastic.co/guide/en/elasticsearch/reference/current/span-queries.html>

The next section deals with a set of specialized queries such as `distance_feature`, `percolator`, and others. Let's turn our attention to those now.

12.9 Specialized queries

In addition to the types of queries we saw so far, Elasticsearch has a handful of advanced queries dedicated to serving a specialized function. For example, boosting the score for cafes serving chilled drinks at a specified location (`distance_feature` query), alerting the user

when an item is back in stock (`percolate` query), finding similarly looking documents (`more_like_this` query), giving a few documents a bit more importance (`pinned` query), and so on. We dedicate the final section of this chapter to look at these specialized queries in detail.

12.9.1 Looking at the `distance_feature` query

When searching for classic literature, we may want to add a clause to find the books that were published in 1813. Along with returning all the books that are literature classics, we can expect to find *Pride and Prejudice* (Jane Austen's classic), but the idea is to show *Pride and Prejudice* at the top of the list because it was printed in 1813. Topping the list is nothing more than boosting the relevance score of the query results based on a particular clause; in this case, we specifically want the books published in 1813 to be given higher importance.

This sort of feature is available in Elasticsearch by using the `distance_feature` query. The query fetches the results and marks a few of them with a higher relevancy score if they are nearer to an origin date (1813 in this example).

The `distance_feature` query also provides similar support for locations. We can highlight the locations nearer a particular address boosted to the top of the list if we so desire. Say that we want to find all the restaurants serving fish and chips, but those topping the list should be near Borough Market by London Bridge. (Borough Market is a world-renowned thirteenth century artisan food market; see <https://boroughmarket.org.uk>.)

We can use the `distance_feature` query for such use cases, which works on finding the results nearer an origin location or date. The dates and locations are fields declared as `date` (we can declare it as `date_nanos` too) and `geo_point` data types respectively. The results that are closer to the given date or given location are rated higher in relevance scores. Let's look at a couple of examples to understand the concept in detail.

BOOSTING SCORE FOR NEARBY UNIVERSITIES USING GEOLOCATIONS

Let's just say we are searching for universities in the United Kingdom. While searching for those, we would like to give preference to universities that are closer to a place; say, all the universities within a 10 km radius of the Knightsbridge. We will boost the score for these.

To try out this scenario, let's create a mapping for the `university` index with a location declared as a `geo_point` field. The following listing creates the mapping as well as the indexes for four universities: two in London and two elsewhere in the country.

Listing 12.21 Creating a universities index

```

PUT universities
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "location": {
        "type": "geo_point"
      }
    }
  }
}

PUT universities/_doc/1
{
  "name": "London School of Economics (LSE)",
  "location": [0.1165, 51.5144]
}

PUT universities/_doc/2
{
  "name": "Imperial College London",
  "location": [0.1749, 51.4988]
}

PUT universities/_doc/3
{
  "name": "University of Oxford",
  "location": [1.2544, 51.7548]
}

PUT universities/_doc/4
{
  "name": "University of Cambridge",
  "location": [0.1132, 52.2054]
}

```

Now that the index and data is prepped, let's fetch universities, boosting the relevance scores so that those closer to London Bridge are at the top of the list. See the map of London in figure 12.12 with the approximate distances of these universities around said locations. We use the `distance_feature` query for this purpose, which matches the query criteria but boosts the relevance score based on the additional parameters provided in the query.



Figure 12.12 Map of London showing the universities around London Bridge

First, let's write the query and then dig into it to learn the details. The following listing uses a `distance_feature` query within a `bool` query to fetch the universities.

Listing 12.22 Boosting scores of the universities closer to London Bridge

```
GET universities/_search
{
  "query": {
    "distance_feature": { #A
      "field": "location", #B
      "origin": [-0.0860, 51.5048], #C
      "pivot": "10 km" #D
    }
  }
}
```

#A The `distance_feature` query declaration

#B The location to search for

#C The focal point where the distance is measured from an origin

#D The distance from the focal point

The query searches for all the universities returning our two universities, London School of Economics and the Imperial College London. Additionally, if any of these universities are in the vicinity of 10 km around the origin (-0.0860, 51.5048 represents London Bridge in UK), they are scored higher than the others.

The `distance_feature` query as defined in the listing expects these properties:

- `field`—The `geo_point` field in the document
- `origin`—The focal point (in longitude and latitude) from which to measure the distance
- `pivot`—The distance from the focal point

In the query in listing 12.22, London School of Economics university is closer to London Bridge than Imperial College; hence, LSE is returned at the top with a higher score. Now let's look at using the `distance_feature` query with dates.

BOOSTING THE SCORE USING DATES

In the last section, the `distance_feature` query helped us to search for universities, boosting the score for those that are nearer to a certain geolocation. There's also a similar requirement that can be satisfied by the `distance_feature` query: boosting the score of the results if they are pivoted around a date.

Let's say that we want to search for all the iPhone release dates, topping the list with those iPhones that were released within 30 days around December 1, 2020 (no particular reason, other than trying out the concept). We can write a similar query as we did in the last section, except the `field` attribute will be based on a date. Let's first create an `iphones` mapping and index a few iPhones into our index. The query in the following listing does that.

Listing 12.23 Creating an `iphones` index and priming it with a few documents

```
PUT iphones
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "release_date": {
        "type": "date",
        "format": "dd-MM-yyyy"
      }
    }
  }
}

# Indexing a few documents
PUT iphones/_doc/1
{
  "name": "iPhone",
  "release_date": "29-06-2007"
}

PUT iphones/_doc/2
{
  "name": "iPhone 12",
  "release_date": "23-10-2020"
}
PUT iphones/_doc/3
{
  "name": "iPhone 13",
  "release_date": "24-09-2021"
}
PUT iphones/_doc/4
{
  "name": "iPhone 12 Mini",
  "release_date": "13-11-2020"
}
```

Now that we have an index with a bunch of iPhones in it, let's develop a query to satisfy our requirement: we'll fetch all iPhones but prioritize the ones released 30 days around the first of December, 2020. The query in the next listing does this.

Listing 12.24 Fetching iPhones and boosting the ratings

```
GET iphones/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "name": "12"
          }
        }
      ],
      "should": [
        {
          "distance_feature": {
            "field": "release_date", #A
            "origin": "1-12-2020", #B
            "pivot": "30 d" #C
          }
        }
      ]
    }
  }
}
```

#A The field against which our query executes

#B Defines the focal date

#C Pivots the number of days to boost the scores

In this query, we wrap a `distance_feature` in a `bool` query with a `must` and a `should` clause (we learned about the `bool` query in the last chapter). The `must` clause searches for all the documents with 12 in the `name` field and returns iPhone 12 and iPhone 12 mini documents from our index. Our requirement is to prioritize the phones released 30 days around the first of December (so, potentially, all phones released between November to December, 2020).

To satisfy this requirement, the `should` clause uses the `distance_feature` query to enhance the scores for the matching documents closest to the pivoted date mentioned. The query fetches all the documents from the `iphones` index. Any iPhone released 30 days before or after December 1, 2020 (`origin`) is returned with a higher relevance score.

Remember the matches the `should` clause returns adds to the overall score. Hence, you should see iPhone 12 Mini topping the list because the release date ("`release_date`": "13-11-2020") of this iPhone is closer to the pivoted date ("`origin`": "01-12-2020" \pm 30 days). The results of the query are presented in the following snippet for completeness.

```

"hits" : [
  {
    "_index" : "iphones",
    "_id" : "4",
    "_score" : 1.1876879,
    "_source" : {
      "name" : "iPhone 12 Mini",
      "release_date" : "13-11-2020"
    }
  },
  {
    "_index" : "iphones",
    "_id" : "2",
    "_score" : 1.1217185,
    "_source" : {
      "name" : "iPhone 12",
      "release_date" : "23-10-2020"
    }
  }
]

```

As you can see, the iPhone 12 Mini scored higher than the iPhone 12 because it was released just 17 days prior to our pivot date, while the iPhone 12 was released a bit earlier than that (almost 5 weeks prior).

12.9.2 Looking at the pinned query

You may have seen a few sponsored search results appearing at the top of the result set when querying your favorite e-commerce website such as Amazon. Suppose we want to implement such functionality in our application using Elasticsearch. Well, fret not; a pinned query is at hand.

The `pinned` query helps to add chosen documents to the result set so they appear at the top of the list. This happens by making their relevance scores higher than others. Let's quickly look at an example query, given in the following listing, that demonstrates this functionality.

Listing 12.25 Modifying the search results by adding sponsored results

```
GET iphones/_search
{
  "query": {
    "pinned": { #A
      "ids": ["1", "3"], #B
      "organic": { #C
        "match": { #D
          "name": "iPhone 12"
        }
      }
    }
  }
}
```

#A Specifies the pinned query

#B The list of document IDs scored higher than the rest of the results.

#C Carries out the query search

#D A match query that searches for iPhone 12

The `pinned` query in the listing has a few moving parts. Let's look at the `organic` block first. It is the query block that houses the search query; in this case, we are searching for iPhone 12 in our `iphones` index. This query ideally should return the two documents, iPhone 12 and iPhone 12 Mini. However, what you see in the output is two documents (iPhone and iPhone 13) in addition to iPhone 12 and iPhone 12 Mini. The reason for this is the `ids` field. This field encloses the additional documents that must be appended to the results and shown at the top of the list (the sponsored results), thus creating higher relevance scores synthetically.

The `pinned` query helps add additional high-priority documents with the results sets. These documents trump others in the list position to create sponsored results.

You may be wondering if the pinned results have any scoring: can one or some of the pinned results be prioritized over the other(s)? Unfortunately, the answer is no. These documents are presented in the order of IDs as input by us in the query: `"ids": ["1", "3"]`, for example.

12.9.3 Looking at the more_like_this query

You may have noticed on Netflix or Amazon Prime Video (or one of your favorite streaming apps) showing you More Like This movies when you browse one of them. For example, figure 12.13 shows all More Like This movies when I visit Paddington 2.

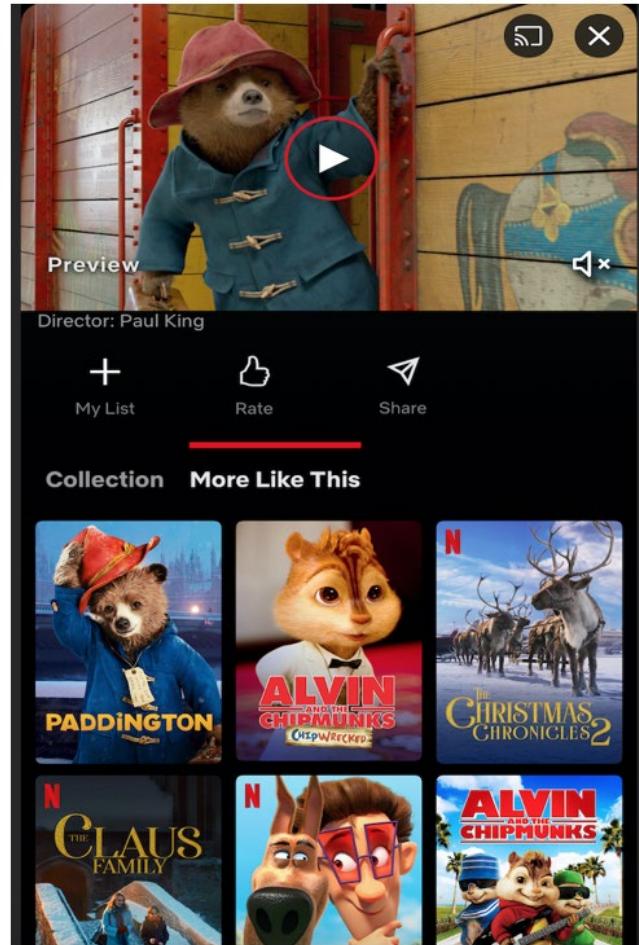


Figure 12.13 Viewing More Like This movies

One of the requirements for users is to search “similar” or “like ” in some documents. For example, finding quotes similar to Newton’s Friends and Truth, researching papers similar to COVID and SARS, or querying movies like *The Godfather*. Let’s jump right in to an example to understand the use case better.

Let’s say that we are collecting a list of profiles about some people. To create a set of profiles, we index sample documents into the profiles index as the code in the following listing demonstrates.

Listing 12.26 Indexing sample profiles

```
PUT profiles/_doc/1
{
  "name": "John Smith",
  "profile": "John Smith is a capable carpenter"
}

PUT profiles/_doc/2
{
  "name": "John Smith Patterson",
  "profile": "John Smith Patterson is a pretty plumber"
}

PUT profiles/_doc/3
{
  "name": "Smith Sotherby",
  "profile": "Smith Sotherby is a gentle painter"
}
PUT profiles/_doc/4
{
  "name": "Frances Sotherby",
  "profile": "Frances Sotherby is a gentleman"
}
```

There's nothing surprising about these documents; they're just profiles about a bunch of routine people. Now that we have these documents indexed, let's find out how we can ask Elasticsearch to fetch documents that are similar to the text *gentle painter* or to *capable carpenter* or even retrieve documents with the similar name, Sotherby. That's exactly what the `more_like_this` query helps us with. The next listing creates a query to search profiles more like Sotherby.

Listing 12.27 Searching for More Like This documents

```
GET profiles/_search
{
  "query": {
    "more_like_this": { #A
      "fields": ["name", "profile"], #B
      "like": "Sotherby", #C
      "min_term_freq": 1, #D
      "max_query_terms": 12, #E
      "min_doc_freq": 1
    }
  }
}
```

#A Defines the `more_like_this` query
#B Searches the given input in fields
#C Defines the query criteria
#D Sets the term frequency (defaults to 2)
#E Sets the number of terms to be selected

The `more_like_this` query accepts text in a `like` parameter, where this input text is matched against the given fields mentioned in the `fields` parameter. The query accepts a few tuning parameters such as minimum term and document frequency (`min_term`) and the

maximum number of terms (`max_query_terms`) that the query should select. If we want to give the user a better experience when showing similar documents, the `more_like_this` query is the right choice.

12.9.4 Looking at the percolate queries

Searching for a set of documents given an input is straightforward. All we need to do is to return search results from the index if there are any matches to the given criteria. This satisfies the requirement of searching user's criteria, and this is what we've done so far when querying for results.

There's another requirement that Elasticsearch satisfies: the requirement of notifying the user when their present search yields negative results, but the outcome will become available at a future date. Say, for example, that a user searches for a *Python in Action* book in an e-commerce book seller site, but unfortunately, we do not have the book in stock. The dissatisfied customer leaves the site. However, after a day or two, we've get new stock in, and the book is added to the inventory. Now, as the book re-appears in our inventory, we want to notify the user so the user can purchase it.

Elasticsearch supports this sort of use case by providing a special query called the percolate query, which uses the `percolator` field type. The percolate query is opposite to our normal search query mechanism in that instead of running the query against the documents, we search for a query given a document. This is a bit of a strange concept to understand at first glance, but we'll demystify that in this section. Figure 12.14 shows the differences between the normal versus percolate query.

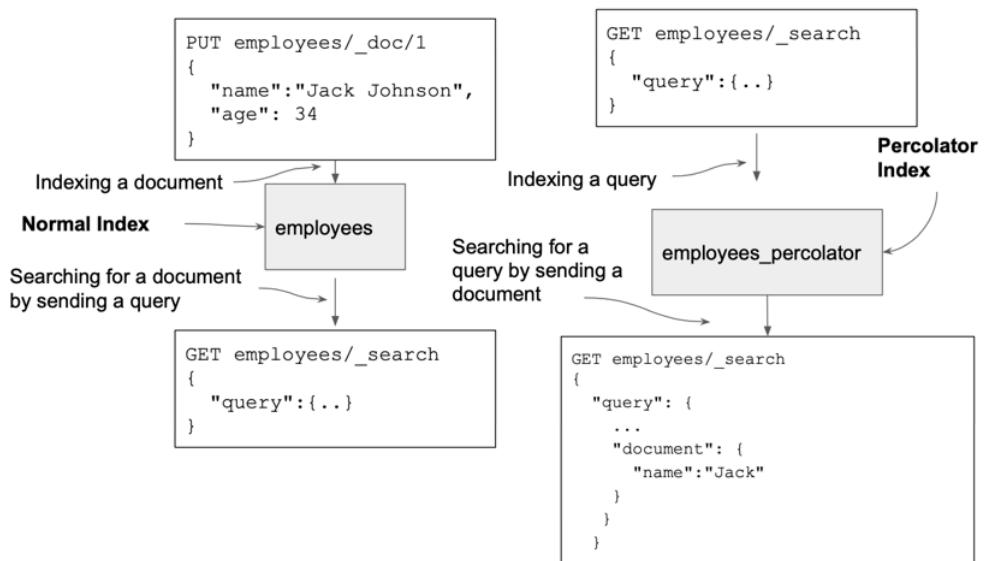


Figure 12.14 Normal vs. percolate query

Let's check the percolate queries in action by first indexing a set of documents. The following listing indexes three technical books in to the `tech_books` index. Note that we do not include a Python book yet.

Listing 12.28 Indexing technical books

```
PUT tech_books/_doc/1
{
  "name": "Effective Java",
  "tags": ["Java", "Software engineering", "Programming"]
}

PUT tech_books/_doc/2
{
  "name": "Elasticsearch crash course",
  "tags": ["Elasticsearch", "Software engineering", "Programming"]
}

PUT tech_books/_doc/3
{
  "name": "Java Core Fundamentals",
  "tags": ["Java", "Software"]
}
```

Now that we've seeded our books' inventory index with a few books, we can expect users to search for books using the simple `match/term` queries. (I'm omitting these queries in this discussion because we've already mastered them.) However, not all user queries yield

results; for example, someone searching for the *Python in Action* book won't find one. The following listing demonstrates this.

Listing 12.29 Searching for a nonexistent book

```
GET tech_books/_search
{
  "query": {
    "match": {
      "name": "Python"
    }
  }
}
```

From the user's perspective, the search ends with a disappointing result: not returning the book when searched. We can take our queries to the next level by notifying the user when the out-of-stock Python book becomes available. This is exactly where we can put percolators to work.

Just as we index documents into an index, percolators have their own index for a set of queries and expect the queries to be indexed. We need to define a schema for a percolator index. Let's call it `tech_books_percolator`, which the following listing shows.

Listing 12.30 Creating a percolator index

```
PUT tech_books_percolator
{
  "mappings": {
    "properties": {
      "query": { #A
        "type": "percolator" #B
      },
      "name": { #C
        "type": "text"
      },
      "tags": { #D
        "type": "text"
      }
    }
  }
}

#A Sets the field name to query
#B Sets the query field's data type to percolator
#C Sets the name field as it appears in the tech_books index
#D Sets the tags field as it appears in the tech_books index
```

This listing defines the index for holding the percolator queries. A few things to note are

- It contains a `query` field to hold the users' (failed) queries.
- The data type of the `query` field must be `percolator`.

The rest of the schema consists of the schema definition borrowed from the original `tech_books` index.

Just as we define the fields with various data types such as `text`, `long`, `double`, `keyword`, etc., Elasticsearch provides a `percolator` type too. The `query` field is defined as `percolator` in listing 12.30, and it expects a query as the field value, which we will see shortly.

Now that we have our percolator index (`tech_books_percolator`) mapping ready, the next step is to store queries. The queries are usually the ones that don't return results to the users (like the Python example).

In the real world, the user's query that doesn't yield a result will be indexed into this percolator index. The process of collating the users' failed queries into a percolator index can be done inside the search application, but unfortunately, it is out of scope to discuss it here. Just imagine that somehow the query in listing 12.30 doesn't yield a result and is now sent to the percolator's index to get it stored. The next listing provides the code to store the query.

Listing 12.31 Storing the query to search a book

```
PUT tech_books_percolator/_doc/1
{
  "query" : { #A
    "match": {
      "name": "Python"
    }
  }
}
```

#A The same query that the user tried to search but failed to get a positive result

As you can see, the listing shows the indexing of a query, which is unlike indexing a normal document. If you remember the document/indexing operations from the beginning chapters, any time we index a document, we use a JSON-formatted document with name-value pairs. This one, however, oddly has a `match` query.

This query (listing 12.31) is stored in the `tech_books_percolator` index with the document ID 1. As you can imagine, this index keeps growing with the failed searches. The only notable thing is that the JSON document consists of the query(ies) issued by the users that don't return positive results.

The final piece of the puzzle is to search the `tech_books_percolators` index when our stock gets updated. As a bookshop owner, we are expected to stock up and, perhaps, the next time we receive the new stock, let's just assume the Python book is in the new stock. We now can index it in to our `tech_books` index for users to search and buy as this listing shows.

Listing 12.32 Stocking up (indexing) a Python book

```
PUT tech_books/_doc/4
{
  "name": "Python in Action",
  "tags": ["Python", "Software Programming"]
}
```

Now that we have the Python book indexed, we need to rerun the user's failed query. But this time instead of running the query on the `tech_books` index, let's run it against the `tech_books_percolator` index. The query against the percolator index has a special syntax. Let's first write that and then come back to it to discuss more about it.

Listing 12.33 Searching for the queries in the percolator index

```
GET tech_books_percolator/_search
{
  "query": {
    "percolate": { #A
      "field": "query", #B
      "document": { #C
        "name": "Python in Action",
        "tags": ["Python", "Software Programming"]
      }
    }
  }
}
```

#A Specifies the percolate query

#B Sets the field's name as query

#C Specifies the document consisting of the original book indexed into `tech_books`

As the listing shows, the percolate query expects two bits of input: a `field` with a value of `query` (this coincides with the property defined in the percolator mapping in listing 12.30) and a `document`, which is the same document we indexed in our `tech_books` index. All we need to do is to check if there's any `query` that matches with the given `document`. Fortunately, *Python in Action* has a match (as you may recollect, we indexed a query in to our percolator index earlier).

Now that given a document (the Python document defined in listing 12.32), we can return a query from the `tech_books_percolator` index. This lets us inform the user that the book they were looking for is back in stock! Note that we could've extended the query that gets stored in the `tech_books_percolator` index with a specific user ID.

That's all for percolators. They are a little bit difficult to understand because there are a couple of moving parts, but once you understand the use case, it's not that difficult to implement it. Do keep in mind that there must always be an automate, semi-automated, or even manual process in place to sync the operations a user performs against the data stored in the percolator index.

That's a wrap! By discussing advanced queries in this chapter, we've pretty much covered the search part of Elasticsearch. The final part is aggregating data, which is the subject of the next chapter. Stay tuned to learn more about how we can find intelligence in the data we hold by analyzing it with various mathematical and statistical functions.

12.10 Summary

- Elasticsearch supports the `geo_point` and `geo_shape` data types to work with geodata.
- The geospatial queries fetch locations and addresses using a set of coordinates formed from longitude and latitude.
- The `geo_bounding_box` query fetches the addresses in a georectangle, which is constructed using a pair of longitude and latitude values as the top left and bottom right coordinates.
- The `geo_distance` query finds the locations inside a circular area with a center provided as a pivot and the radius as the distance from the pivot.
- The `geo_shape` query fetches all the eligible locations inside a given envelope formed by a set of coordinates.
- The `geo_shape` query searches two-dimensional (2D) shapes in a rectangular coordinate system (Cartesian plane).
- The span queries are a type of advanced queries that work with the lower-level positions of the individual tokens or words. Elasticsearch supports a handful of span queries such as `span_first`, `span_within`, `span_near`, and more.
- The `distance_feature` query is a specialized query where the proximity of documents to a given focal point increases the relevance score and, thus, gets a higher priority.
- The `pinned` query lets you bundle additional (even unmatched) documents with the original result set, thus potentially creating sponsored search results.
- The `more_like_this` query gets related or similar looking results.
- The `percolate` query lets you notify users about their negative searches at a future date.

13

Aggregations

This chapter covers

- Aggregation basics
- Working with metric aggregations
- Categorizing data using bucket aggregations
- Chaining metric and bucket aggregations in pipeline aggregations

Search and analytics are two sides of a coin, and Elasticsearch delivers both with absolute detail and countless features. Elasticsearch is a market leader in analytics by providing the feature-rich functions for querying and analyzing data, thus enabling organizations to find insights and deep intelligence from their data. Although a search finds results for certain criteria, analytics, on the other hand, helps organizations derive statistics and metrics from it. So far, we've looked at searching for documents from a given corpus of documents. With analytics, we take a step back and look at the data from a high level to draw conclusions about it.

In this chapter, we'll look at Elasticsearch's aggregations in detail. Elasticsearch boasts a large number of aggregations, predominantly categorized into one of these types: metric, bucket, and pipeline. Metric aggregations allow us to use analytical functions such as sum, min, max, or average for calculations on the data; bucket aggregations help us to categorize data into buckets or ranges. Finally, pipeline aggregations permit us to chain aggregations, meaning that they take metric or bucket aggregations and create new aggregations.

There are a lot of aggregations provided by Elasticsearch out of the box. We will get familiar with each of the types before jumping into a few to get our hands dirty. Do note that it is impractical to go over all of them individually due to page-count restrictions. Learning the concepts and applying them on a handful of common aggregations is more important in my view rather than documenting each and every one of them.

Having said that, the source code with as many aggregations as possible , especially those that may not have been discussed in this chapter. Do keep a look out for the code snippets on my GitHub page: https://github.com/madhusudhankonda/elasticsearch-in-action/blob/main/kibana_scripts/ch13_aggregations.txt

13.1 Overview

Aggregations let companies understand their amassed data. It helps to understand customers and their relationships, and to evaluate the performance of the products, forecasting sales as well as answering a wide range of questions such as performance of the application over a period of time, security threats, and more. Aggregations in Elasticsearch primarily fall into three categories:

- *Metrics aggregations*—These aggregation types generate metrics such as sum, average, minimum, maximum, top hits, mode, and many others. They include the popular, single-value metrics that help in understanding the data over a large set of documents. These answer questions like what is the sum of all product sales overall in the last month? What are the minimum number of API errors, the top search hits, and so on?
- *Bucket aggregations*—Bucketing is a process of collecting data in interval buckets. These aggregations split the data in to given sets; for example, splitting cars in to years of registration, students in to various grades, and so on. Histograms, range, terms, filters, and others fall into these categories.
- *Pipeline aggregations*—These aggregations work on the output of other aggregations to provide a complex set of statistical analytics such as moving averages, derivatives, and others.

We will look at these aggregations with examples in the following sections. First, let's go over the syntax and the endpoint that's used to carry out the aggregations.

13.1.1 The endpoint and the syntax

By now, after working with search queries, we are pretty much familiar with the `_search` endpoint. The good news is that we can use the same `_search` endpoint for running aggregations too. The body of the request, however, uses a new object in the request, which is called `aggregations` (or, in short, `aggs`) instead of the usual `query` object. The following snippet shows the aggregation query syntax.

```
GET <index_name>/_search
{
  "aggregations|aggs": {
    "NAME": {
      "AGG_TYPE": {}
    }
  }
}
```

As you can see from the code snippet, Elasticsearch aggregations use the `_search` endpoint similar to the search queries we've used earlier on in the book. The `aggregations` (or `aggs`,

which we prefer in this book) object tells Elasticsearch that the query invocation is of an aggregation type. The `NAME` attribute, provided by the user, gives a suitable name as deemed fit to the aggregation. Finally, `AGG_TYPE` is the type of aggregation such as `sum`, `min`, `max`, `range`, `terms`, `histogram`, and more. We will dedicate this chapter in learning the fundamental concepts and work with a handful of them.

13.1.2 Combining searches and aggregations

We can also combine aggregations with queries too. For example, we can run a query to fetch a set of results and then run through aggregations on that result set. This is called *scoped aggregations* because the input for the aggregations is the query's result. As the following snippet shows, the syntax gets slightly extended from the previous aggregation query.

```
GET <index_name>/_search
{
  "query": {
    "QUERY_TYPE": {
      "FIELD": "TEXT"
    }
  },
  "aggs": {
    "NAME": {
      "AGG_TYPE": {}
    }
  }
}
```

As you can see in the code snippet, the scope of the aggregation is defined by the query's output. If you don't associate a query with an aggregation request, it is assumed that it will work on all the documents of the index (or indices) defined in the request URL.

13.1.3 Multiple and nested aggregations

In addition to running solo aggregations, we could also execute multiple aggregations on the given set of data. This feature is extremely handy if we need to extract analytics on various fields with multiple conditions. For example, we may want to create a histogram of the iPhone 14 sales per day as well as a `sum` aggregation of the total sales for the whole month. For this, we can employ the bucketing type aggregation, `histogram`, and the metric type aggregation, `sum`.

Additionally, there are times when we need to nest the aggregations; for example, the bucketed data in a histogram may need further categorizing (bucketing) of data by a date or by finding the minimum and maximum for each bucket. In this case, the aggregated data in each of the top-level buckets is fed into the next level bucket for further aggregations. Nested aggregations can be further categorized using more buckets or using single-value metrics (like `sum`, `avg` etc). We will look at nested aggregations using some examples in the coming sections.

13.1.4 Ignoring the results

Search (or aggregation) queries tend to return source documents in the response if not asked to suppress them in the query. We looked at how to manipulate the responses in chapter 8, where we configured the `_source`, `_source_includes`, and `_source_excludes` parameters with appropriate settings.

When working with aggregations, the actual documents in the response are most likely to be unimportant because we'd be more interested in the aggregations and not the actual source. Under normal circumstances, unfortunately, the source documents get tagged along with the aggregations even when we run the aggregation query. If this is not our intention (and usually it is not when running aggregations), we can tweak the query by setting the `size` parameter to zero. The following snippet highlights this approach. We will use this handy parameter (`size=0`) throughout this chapter as we execute aggregation queries to suppress the source documents.

```
GET tv_sales/_search
{
  size = 0 #A
  "aggs": {
    <<your query goes here>>
  }
}
```

#A Sets the size parameter to zero

As pointed out earlier, aggregations are broadly classified into three types: metric, bucket and pipeline. We will look at the metric aggregations in the next section with sample data specifically tailored for metric calculations. The following sections will deal with the remaining aggregations.

13.2 Metric aggregations

Metric aggregations are the simple aggregations that one uses often in our daily lives. For example,

- What is the average height and weight of the students across a class?
- What is the minimum hedge trade?
- What's the gross earnings of a best bookseller?

Elasticsearch provides a handful of metric functions to calculate most of the single-valued as well as multi-valued metrics. If you are wondering about single- versus multi-valued aggregations, this is a simple concept based on the number of outputs.

A single value metric aggregation is an aggregation on given a set of data that outputs a single value such as min, max, average, and so. They work on a set of input documents to produce these single-valued output data. On the other hand, there are a couple of aggregations that produce multiple values as output: the `stats` and `extended_stats` aggregations. In this case, both aggregations produce multiple values under the said

umbrella. For example, the `stats` aggregation output consists of `min`, `max`, `sum`, `avg`, and a couple more for the same set of documents.

Most metric aggregations are self-explanatory. For example, a `sum` aggregation sums up all the given values, whereas `avg` averages the values. If you need to work with other metrics not discussed in this chapter, see the documentation on the Elasticsearch site for their usage: <https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics.html>

13.2.1 Sample data

In the next few sections, we will discuss popular aggregations. Before we do that, let's prime Elasticsearch with a few documents.

Because we are working through the aggregations at a high level, it's sufficient to index a handful of documents. The following listing primes our data store with a list of TV sales by creating a new index, `tv_sales`. You can also download the same sample set from my GitHub page: https://github.com/madhusudhankonda/elasticsearch-in-action/blob/main/datasets/tv_sales.txt

Listing 13.1 Indexing TV sales data

```
PUT tv_sales/_bulk
{"index": {"_id": "1"}}
{"brand": "Samsung", "name": "UHD TV", "size_inches": 65, "price_gbp": 1400, "sales": 17}
{"index": {"_id": "2"}}
{"brand": "Samsung", "name": "UHD TV", "size_inches": 45, "price_gbp": 1000, "sales": 11}
{"index": {"_id": "3"}}
 {"brand": "Samsung", "name": "UHD TV", "size_inches": 23, "price_gbp": 999, "sales": 14}
 {"index": {"_id": "4"}}
 {"brand": "LG", "name": "8K TV", "size_inches": 65, "price_gbp": 1499, "sales": 13}
 {"index": {"_id": "5"}}
 {"brand": "LG", "name": "4K TV", "size_inches": 55, "price_gbp": 1100, "sales": 31}
 {"index": {"_id": "6"}}
 {"brand": "Philips", "name": "8K TV", "size_inches": 65, "price_gbp": 1800, "sales": 23}
 {"index": {"_id": "7"}}
 {"name": "8K TV", "size_inches": 65, "price_gbp": 2000, "sales": 23}
 {"index": {"_id": "9"}}
 {"name": "8K TV", "size_inches": 65, "price_gbp": 2000, "sales": 23, "best_seller": true}
 {"index": {"_id": "10"}}
 {"name": "4K TV", "size_inches": 75, "price_gbp": 2200, "sales": 14, "best_seller": false}
```

This listing indexes a bunch of documents representing a few attributes such as `brand`, `size`, `price`, and others to a `tv_sales` index. Take a particular note of the `best_seller` field: it is only set on the last two records. Now that we have a sample data set, let's run a few common metric aggregations.

13.2.2 The value count metric

The `value_count` metric counts the number of values present for a field in a set of documents. If our requirement is to fetch the number of values existing in Elasticsearch for a given field, this `value_count` aggregation satisfies that requirement. For example, running

the following query returns the number of values we have in our stash for the `best_seller` field.

Listing 13.2 Finding the number of values for a field

```
GET tv_sales/_search
{
  size = 0
  "aggs": {
    "total-number-of-values": { #A
      "value_count": { #B
        "field": "best_seller" #C
      }
    }
  }
}
```

#A Names the aggregation results
#B Names the aggregation (value_count)
#C The field on which the value_count is performed.

As you can see from this listing, the `value_count` aggregation is carried out on the `best_seller` field. Note that aggregations by default are not executed on text fields. With our sample data, the `best_seller` field, a `boolean` data type, is a good candidate for the `value_count` aggregation metric. Running the query in listing 13.2 outputs the following:

```
"aggregations" : {
  "total-values" : {
    "value" : 2
  }
}
```

There are two values (two documents) for that `best_seller` field. Note that `value_count` doesn't pick unique values; it does not remove duplicate values contained for the specified field across the document set.

Aggregations on text fields are not optimized

Text fields do not support sorting, scripting, and aggregations. Aggregations are ideally carried out on nontext fields such as number, keyword, Boolean, and so on. Because text fields are not optimized for aggregations, by default Elasticsearch stops us from creating aggregation queries on them. If you are curious, run the aggregations on a text field like `name` field, for example, and see what exception Elasticsearch throws:

```
"root_cause" : [
{
  "type" : "illegal_argument_exception",
  "reason" : "Text fields are not optimised for operations that require per-document field data like aggregations and sorting, so these operations are disabled by default. Please use a keyword field instead. Alternatively, set fielddata=true on [name] in order to load field data by uninverting the inverted index. Note that this can use significant memory."
```

```
}
```

As the error says, running the aggregations on a text field is prohibited by default, but if you want to perform aggregations on text fields, you need to enable `fielddata` on the respective fields. You can do this setting `"fielddata": true` when defining the mapping:

```
# Enabling field data on a text field
PUT tv_sales_with_field_data
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text",
        "fielddata": true
      }
    }
  }
}
```

Note that enabling `fielddata` may lead to performance issues because the data is stored in memory on the nodes. Rather than taking a hit on performance by enabling `fielddata`, we can instead create a multi-field data type with keyword as the second type. That's because keyword data types are allowed for aggregation.

Finding the average of a set of numbers is a frequent operation as part of analytics. As expected, Elasticsearch provides a handy function called `avg` to find averages, which is the topic of the next section.

13.2.3 The average metric

Finding the average of a certain set of numbers is a pretty basic statistical function that we often require. Elasticsearch provides the `avg` metric aggregation out of the box for running an average calculation over a set of numbers. For example, the next query fetches the average TVprice by employing `avg`.

Listing 13.3 Average price of all TVs

```
GET tv_sales/_search
{
  "size": 0,
  "aggs": {
    "tv_average_price": { #A
      "avg": { #B
        "field": "price_gbp" #C
      }
    }
  }
}
```

```
#A Names the aggregation
#B Calculates the average price
#C The field on which the average is calculated.
```

The `tv_average_price` is a user-defined name given to this average aggregation. The `avg` declaration in the code represents the average function. The data field that we need our single-field metric `avg` to run the average calculation on is called `field`. Once the query is executed, we should get the following results:

```
"aggregations" : {
  "tv_average_price" : {
    "value" : 1555.333333333333
  }
}
```

The average price of all TVs is calculated by the engine and, as you can see, is returned to the user. The average of the TV price across all six documents is about £1555.

13.2.4 The sum metric

The single value `sum` metric adds the values of the field in question and produces the end result. For example, if we want to find the total value of the TVs sold, we can issue the following query.

Listing 13.4 Total sum of all the TVs sold

```
GET tv_sales/_search
{
  "size": 0,
  "aggs": {
    "tv_total_price": {
      "sum": {
        "field": "price_gbp"
      }
    }
  }
}
```

The `sum` metric adds all the prices and produces a single figure of £13,998 when you run the query. In the same vein, let's look at the minimum and maximum metric functions in the next section.

13.2.5 The minimum (min) and maximum (max) metrics

There will be times where we need to find the minimum and maximum quantities from a set of values, say the *minimum* number of available speakers for a conference, or the session with the highest number of attendees. Elasticsearch exposes the corresponding metrics in the form of `min` and `max` for producing these extremes of a data set. These metrics are self explanatory, but in the interest of completeness, let's go over them briefly.

MINIMUM METRIC

Let's say we want to find the cheapest priced TV in our stock. This clearly is a candidate for employing minimum metric on the data values. The following listing defines the `min` metric on the `price_gbp` field.

Listing 13.5 Cheapest price for the TVs

```
GET tv_sales/_search
{
  "size": 0,
  "aggs": {
    "cheapest_tv_price": {
      "min": { #A
        "field": "price_gbp" #B
      }
    }
  }
}
```

#A Calculates the minimum value

#B Applies the min function to this field

The `min` keyword fetches this metric, which works on the `price_gbp` field to produce the expected result: the field's minimum value derived from all the documents. Here we are fetching the lowest priced TV (£999) present in our stock by executing the query.

MAXIMUM METRIC

You can use similar logic to fetch the best selling TV: a TV with a *maximum* number of sales. The following query fetches the TV with the maximum number of sales (the best selling TV).

Listing 13.6 Best selling TV

```
GET tv_sales/_search
{
  "size": 0,
  "aggs": {
    "best_seller_tv_by_sales": {
      "max": {
        "field": "sales"
      }
    }
  }
}
```

Once we execute the query, we should receive a TV that sells super fast (maximum sales). From the results, it seems to be LG's 8K TV with 48 sales.

13.2.6 The common stats metric

While the previous metrics are single-valued (meaning they work only on a single field), the `stats` metric fetches all common statistical functions. It is a multi-value aggregation that

fetches a few metrics (`avg`, `min`, `max`, `count`, and `sum`) all in one go. The query in the next listing applies the `stats` aggregations for the `price_gbp` field.

Listing 13.7 All common stats in one go

```
GET tv_sales/_search
{
  "size": 0,
  "aggs": {
    "common_stats": {
      "stats": { #A
        "field": "price_gbp" #B
      }
    }
  }
}
```

#A The `stats` function
#B Applies the `stats` to this field

As the query demonstrates, we use the `stats` function on the `price_gbp` field to fetch the common statistics. Once executed, this query returns the following results:

```
"aggregations" : {
  "common_stats" : {
    "count" : 6,
    "min" : 999.0,
    "max" : 1800.0,
    "avg" : 1299.666666666667,
    "sum" : 7798.0
  }
}
```

As you can see, the `stats` metric returns all the other five metrics in one go. This makes it a handy metric if you want to see the basic aggregations all in one place.

13.2.7 The extended stats metric

Although `stats` is a useful common metric, it doesn't provide us with advanced statistical analytics such as variance, standard deviation, and other statistical functions. Elasticsearch provides another metric called `extended_stats` out of the box, which is the cousin of `stats` by dealing with these advanced statistical metrics.

The `extended_stats` metric provides three additional stats in addition to the standard statistical metrics: the `sum_of_squares`, `variance`, and `standard_deviation` metrics. The following listing illustrates how we can extract various variance flavors and standard deviations using the `extended_stats` metric.

Listing 13.8 Advanced (extended) stats on the `price_gbp` field

```
GET tv_sales/_search
{
  "size": 0,
  "aggs": {
    "additional_stats": {
```

```

        "extended_stats": { #A
            "field": "price_gbp" #B
        }
    }
}

```

#A: Applying extended_stats function fetches advanced statistical measures.
#B: Employs the extended_stats function on this field

As the code demonstrates, we invoke the `extended_stats` function on the `price_gbp` field. This retrieves a whole lot of statistical data as figure 13.1 illustrates.

```

,
"aggregations" : {
    "extended_stats" : {
        "count" : 6,
        "min" : 999.0,
        "max" : 1800.0,
        "avg" : 1299.666666666667,
        "sum" : 7798.0,
        "sum_of_squares" : 1.0655002E7,
        "variance" : 86700.2222222232,
        "variance_population" : 86700.2222222232,
        "variance_sampling" : 104040.2666666668,
        "std_deviation" : 294.4490146395846,
        "std_deviation_population" : 294.4490146395846,
        "std_deviation_sampling" : 322.5527347065388,
        "std_deviation_bounds" : {
            "upper" : 1888.5646959458359,
            "lower" : 710.7686373874975,
            "upper_population" : 1888.5646959458359,
            "lower_population" : 710.7686373874975,
            "upper_sampling" : 1944.7721360797443,
            "lower_sampling" : 654.5611972535892
        }
    }
}

```

Figure 13.1 The extended statistics on the `price_gbp` field

The query in listing 13.8 calculates a lot of advanced statistical information on the `price_gbp` field. Note that the result also includes the common metrics (`avg`, `min`, `max`, and so on) in addition to the various variances and standard deviations.

13.2.8 The cardinality metric

The `cardinality` metric returns unique values for the given set of documents. It is a single value metric that fetches occurrences of distinct values from our data. For example, the query in the next listing retrieves the unique TV brands that we have in our index.

Listing 13.9 Fetching unique TV brands

```
GET tv_sales/_search
{
  "size": 0,
  "aggs": {
    "unique_tvs": {
      "cardinality": { #A
        "field": "brand.keyword" #B
      }
    }
  }
}
```

#A The cardinality metric fetches the unique values.

#B Applies cardinality to the brand.keyword field

The query fetches the number of unique brands that we have in our `tv_sales` index. Because we have four unique brands (Samsung, LG, Phillips, and Panasonic), the result should show us 4 in the `unique_tvs` aggregation as in the following snippet:

```
"aggregations" : {
  "unique_tvs" : {
    "value" : 4
  }
}
```

Because the data is distributed in Elasticsearch, trying to fetch exact counts of cardinality may lead to performance issues. In order to fetch the exact number, the data must be retrieved and loaded into a hashset of some sort in the in-memory cache. And because this is an expensive operation, the cardinality runs as an approximation. Hence, we should not expect exact counts for unique values, but they are pretty close.

In addition to the metric aggregations we discussed previously, there are a few more metric aggregations that Elasticsearch exposes (and they seem to be growing in number as a new product release comes out). As you can imagine, going over all of these aggregations in this chapter is not practical. I strongly recommend that you go over the Elasticsearch documentation to learn those not covered in this chapter.

For now, there's another type of metric that produces a bucket of documents rather than creating a metric across all the documents. These are called *bucket aggregations*, a subject discussed in the next section.

13.3 Bucket aggregations

One of the requirements on data is to run some grouping operations. Elasticsearch calls these grouping actions *bucket aggregations*. Their sole aim is to categorize data into groups, commonly called *buckets*.

Bucketing is a process of collecting data into interval buckets. For example,

- Grouping runners for a marathon according to their age bracket (21-30, 31-40, 41-50).
- Categorizing schools based on their inspection ratings (good, outstanding, exceptional).
- Getting the number of new houses constructed each month, each year, and so on.

Before we start playing with bucketing aggregations, let's reuse a data set we've worked with in the past: the books data. Pick up the dataset from my GitHub page and index it: <https://github.com/madhusudhankonda/elasticsearch-in-action/blob/main/datasets/books.txt>

```
This sample snippet provides a quick reminder. (Note that this is not a full dataset.)
POST _bulk
{"index":{"_index":"books","_id":"1"}}
{"title": "Core Java Volume I – Fundamentals", "author": "Cay S. Horstmann", "edition": 11,
 "synopsis": "Java reference book that offers a detailed explanation of various
 features of Core Java, including exception handling, interfaces, and lambda
 expressions. Significant highlights of the book include simple language,
 conciseness, and detailed examples.", "amazon_rating": 4.6, "release_date": "2018-08-
 27", "tags": ["Programming Languages, Java Programming"]}
 {"index":{"_index":"books","_id":"2"}}
 {"title": "Effective Java", "author": "Joshua Bloch", "edition": 3, "synopsis": "A must-have
 book for every Java programmer and Java aspirant, Effective Java makes up for an
 excellent complementary read with other Java books or learning material. The book
 offers 78 best practices to follow for making the code better.", "amazon_rating": 4.7, "release_date": "2017-12-27", "tags": ["Object Oriented Software Design"]}
```

Now that we've primed our server with book data, let's run some common bucketing aggregations. There are at least two dozen of these aggregations out of the box; each one has its own bucketing strategy.

As I mentioned earlier in the chapter, it is quite a boring and repetitive task to document all the aggregations in this book. Once you have the concept and get the idea of working with bucketing, you should be good to go with others by following the documentation. For now, let's start with the common bucket aggregation: histograms.

13.3.1 Histograms

Histograms are pretty neat bar charts, representing grouped data. Most analytical software tools provide visual as well as data representation of histograms. Elasticsearch exposes a histogram bucket aggregation out of the box.

We may have worked with histograms where the data is split into multiple categories based on the appropriate interval it falls under. The histograms in Elasticsearch are no different: they create a set of buckets over all the documents on a predetermined interval.

Let's take an example of categorizing books by ratings. We want to find the number of books in each of the ratings such as 2-3, 3-4, or 4-5, and so on. We can create a histogram aggregation using the set interval 1 so the books fall in respective buckets of 1-step ratings. The query in the following listing demonstrates this.

Listing 13.10 Histogram aggregation for books

```
GET books/_search
{
  "size": 0,
  "aggs": {
    "ratings_histogram": { #A
      "histogram": { #B
        "field": "amazon_rating", #C
        "interval": 1 #D
      }
    }
  }
}
```

```

        }
    }
}

#A Names our aggregation
#B Categories the data into buckets
#C Applies the aggregation on this field
#D Specifies the bucket interval (1 unit)

```

As the listing shows, the `histogram` aggregation expects the field on which we want to aggregate the buckets as well as the interval of the buckets. In the listing, we split the books based on the `amazon_rating` field with an interval of 1. This fetches all books that fall between 3-4, 4-5, and so on. Figure 13.2 provides the response.

```

"aggregations" : {
  "ratings_histogram" : {
    "buckets" : [
      {
        "key" : 3.0,
        "doc_count" : 2
      },
      {
        "key" : 4.0,
        "doc_count" : 35
      },
      {
        "key" : 5.0,
        "doc_count" : 2
      }
    ]
  }
}

```

Figure 13.2 Book ratings aggregation as a histogram

As you can see from the result, running the query fetches 2 books that fall in the bucket of 2-3 ratings, 35 books with the rating 3-4, and so on. The response shown in figure 13.2 indicates that the buckets have two fields: `key` and `doc_count`. The `key` field represents the bucket classification, whereas the `doc_count` field indicates the number of documents that fits in the bucket.

HISTOGRAM AGGREGATION WITH KIBANA

In listing 13.10, we developed an aggregation query and executed it in Kibana's console. The results in JSON format aren't visually appealing as you can see in figure 13.2. It is up to the client who receives that data to represent it as a visual chart. Kibana, however, has a rich set of visualizations to aggregate data. While working with the Kibana visualizations is out of scope for the discussions in this chapter, figure 13.3 shows the same data represented as histogram in Kibana's Dashboard but this time with an interval of 0.5.

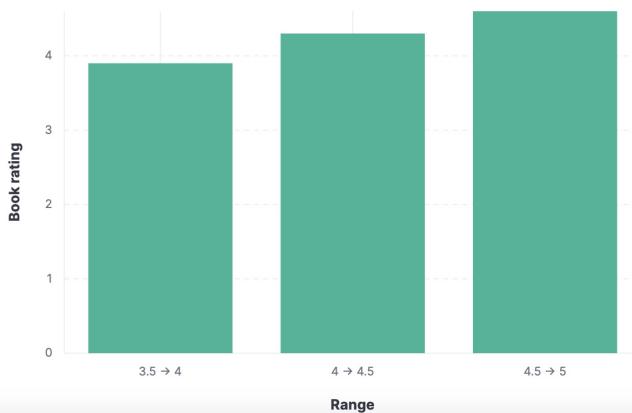


Figure 13.3 A histogram categorizing books by ratings viewed as a bar chart in Kibana's Dashboard

As you can see from the bar chart in figure 13.3, the data is categorized into buckets based on the interval 0.5 and filled with the documents that fit in them. To learn all about Kibana visualizations, refer to the documentation at <https://www.elastic.co/guide/en/kibana/current/dashboard.html>.

THE DATE HISTOGRAM

At times, we may want to group the data based on not necessarily numbers but on dates. For example, we might want to find all the books released each year, get the weekly sales of an iPhone product, or determine the daily server threat attempts every hour, and so forth. This is where the `date_histogram` aggregation comes in handy.

Although the histogram's bucketing strategy we looked at in the last section was based on numerical intervals, Elasticsearch also provides a histogram based on dates, aptly called `date_histogram`. Let's say we want to categorize books based on their release dates. Here's the query that applies bucketing based on a book's release date.

Listing 13.11 Date histogram query

```
GET books/_search
{
  "size":0,
  "aggs": {
```

```

    "release_year_histogram": {
      "date_histogram": { #A
        "field": "release_date", #B
        "calendar_interval": "year" #C
      }
    }
}

```

#A Declares the histogram type (date_histogram)

#B Applies the aggregation to this field

#C Defines the bucket interval

This query uses a `date_histogram` aggregation, which requires the field on which the aggregation is expected to run and the bucket interval. In the example, we use `release_date` as the date field with a `year` interval.

CALLOUTS We can set the bucket's interval value to any of year, quarter, month, week, day, hour, minute, second, and millisecond, based on your requirements

Running the query in listing 13.11 produces the individual buckets for each year and the number of documents in that bucket. The following snippet shows the results at a glance:

```

...
{
  "key_as_string" : "2020-01-01T00:00:00.000Z",
  "key" : 1577836800000,
  "doc_count" : 5
},
{
  "key_as_string" : "2021-01-01T00:00:00.000Z",
  "key" : 1609459200000,
  "doc_count" : 6
},
{
  "key_as_string" : "2022-01-01T00:00:00.000Z",
  "key" : 1640995200000,
  "doc_count" : 3
}
...

```

As you can deduce from the results, each key (expressed as `key_as_string`) represents a year: 2020, 2021, 2022. As the results show, there are 5 books released in 2020, 6 books in 2021, and 3 in 2022.

INTERVAL SET UP FOR THE DATE HISTOGRAM

In the code for listing 13.11, we set the interval to `year` in the `calendar_interval` attribute. In addition to `calendar_interval`, there's another type of interval: the `fixed_interval`. We can set this interval in either of these two ways: as a calendar interval or as a fixed interval. There's a subtle difference between these two types, so to understand, let's look at the differences in the following subsections.

Calendar interval

The calendar interval, declared as `calendar_interval`, is calendar-aware, meaning that the hours and days in a month are adjusted according to the daylight settings of the calendar. The following units are acceptable values: `year`, `quarter`, `month`, `week`, `day`, `hour`, `minute`, `second`, and `millisecond`. They can also be represented as single units like `1y`, `1q`, `1M`, `1w`, `1d`, `1h`, and `1m`, respectively. For example, we could write the query in listing 13.11 as `"calendar_interval": "1y"` instead of using `"year"`.

Note that we can't use multiples like `5y` (five years) or `4q` (four quarters) when setting the interval using `calendar_interval`. For example, setting the interval as `"calendar_interval": "4q"` results in a parser exception: "The supplied interval [4q] could not be parsed as a calendar interval".

Fixed interval

The `fixed_interval` allows setting time intervals as a fixed number of units such as 365d (365 days), 12h (12 hours), and so on. When we don't need to worry about the calendar settings, we can use these fixed intervals. The accepted values are `days` (`d`), `hours` (`h`), `minutes` (`m`), `seconds` (`s`), and `milliseconds(ms)`.

Because `fixed_interval` does not know about the calendar, unlike `calendar_interval`, there are no units to support month, year, quarter, and so on. These attributes depend on the calendar (every month has a certain number of days and so on). As an example, the following listing fetches all the documents for 730 days (2 years).

Listing 13.12 Histogram with a fixed interval of 2 years (730 days)

```
GET books/_search
{
  "size":0,
  "aggs": {
    "release_date_histogram": {
      "date_histogram": {
        "field": "release_date",
        "fixed_interval": "730d" #A
      }
    }
  }
}
```

#A Sets a fixed interval of 2 years (730 days)

As you can see, the query uses a `fixed_interval` of `730d` (or 2 years). The results should show all books in buckets of exactly 730 days. The following snippet demonstrates this:

```
{
  "key_as_string" : "2017-12-20T00:00:00.000Z",
  "key" : 1513728000000,
  "doc_count" : 11
},
{
  "key_as_string" : "2019-12-20T00:00:00.000Z",
  "key" : 1576800000000,
  "doc_count" : 11
}
```

```
},
{
  "key_as_string" : "2021-12-19T00:00:00.000Z",
  "key" : 1639872000000,
  "doc_count" : 3
}
```

If you are curious, run the same query with two different settings: `"calendar_interval": "1y"` and `"fixed_interval": "365d"`. You can refer to my GitHub page for the executable code when experimenting with these settings.

Once the queries successfully run, check the keys. In the former (the one with `fixed_interval: 730d`), the keys start exactly on the 1st of January (`"key_as_string" : "2005-01-01"`); in the latter (the one with `fixed_interval: 365d`), they start on the first release date, 23rd of December 2004 (`"key_as_string" : "2004-12-23"`). The second bucket then simply adds 365 days from the first release date, thus yielding the 23rd of December 2005 (`"key_as_string" : "2005-12-23"`).

NOTE When we use a `fixed_interval`, the range starts from the first document's available date. Going forward, `fixed_interval` is added to that. For example, if the `publish_date` of a document is 25-12-2020 and if you set the interval as `"month"`, the range starts with 25-12-2020 and goes to 25-01-2021, 25-02-2021, and so on.

13.3.2 Child-level aggregates

We looked at how to categorize the data into date buckets in the last section. In addition to creating the buckets with the respective ranges, we may want to aggregate the data inside those buckets too. For example, we may want to find the average rating of a book for each bucket.

To satisfy such requirements, we can use a *sub-aggregation*: an aggregation that works on the bucket's data. With bucketing aggregations, we get support for both metric or bucket aggregations, applied at the child level. The code in the following listing demonstrates the query that fetches books released yearly plus the average rating for each bucket.

Listing 13.13 Average metric on books categorized per year

```
GET books/_search
{
  "size":0,
  "aggs": {
    "release_date_histogram": { #A
      "date_histogram": {
        "field": "release_date",
        "calendar_interval": "1y"
      },
      "aggs": { #B
        "avg_rating_per_bucket": { #B
          "avg": { #C
            "field": "amazon_rating"
          }
        }
      }
    }
  }
}
```

```

        }
    }
}
}

#A The bucketing histogram that categorizes books by year
#B Names the sub-aggregation
#C Applies a single-valued metric across individual buckets

```

As you can see from the listing, there are two blocks of aggregation, one weaved inside the other. The outer aggregation (`release_date_histogram`) produces the data as a histogram, based on the calendar interval of one year. The results of this aggregation are then fed to the next level of aggregation: the inner aggregation (`avg_rating_per_bucket`). The inner aggregation considers each of the buckets as its scope and runs the average (`avg`) aggregation on that data. This produces the average rating of a book per bucket. Figure 13.4 shows the expected result from the aggregation execution.

```

{
  "key_as_string" : "2013-01-01T00:00:00.000Z",
  "key" : 1356998400000,
  "doc_count" : 2,
  "avg_rating_per_bucket" : {
    "value" : 4.200000047683716
  }
},
{
  "key_as_string" : "2014-01-01T00:00:00.000Z",
  "key" : 1388534400000,
  "doc_count" : 6,
  "avg_rating_per_bucket" : {
    "value" : 4.383333285649617
  }
}

```

Figure 13.4 Finding the average rating per bucket (sub-aggregation)

As you can see in figure 13.4, the keys are the dates that honor the calendar year with a set of documents present in that bucket. The notable thing about this query is that an additional object in the bucket, `avg_rating_per_bucket`, consists of our average book rating.

13.3.3 Custom range aggregation

The histogram provides an automatic set of ranges for the given intervals. There are times we may want the data to be segregated in to a certain range, which is not dictated by a strict interval (for example, we may want to classify people into three groups by age: 18-21, 22-

49, and 50+). The standardized interval does not cut this requirement. All we need is a mechanism to customize the ranges. That's exactly what we use the `range` aggregation for.

The `range` aggregation aggregates the documents in a user-defined custom range. Let's look at it in action by writing a query to fetch books that fall in just two categories of ratings: above and below a value of 4 (1-4 and 4-5). The query in the following listing does exactly that.

Listing 13.14 Classifying books into just two baskets

```
GET books/_search
{
  "size": 0,
  "aggs": {
    "book_ratings_range": {
      "range": { #A
        "field": "amazon_rating", #B
        "ranges": [ #C
          {
            "from": 1,
            "to": 4
          },
          {
            "from": 4,
            "to": 5
          }
        ]
      }
    }
  }
}
```

#A Declares the range aggregation
#B Applies the aggregation to this field
#C Sets the customized ranges

The query constructs an aggregation with a custom range defined by an array (`ranges`) with only two buckets: from 1-4 and from 4-5. The following response indicates that there are 2 books that fall in to the 1-4 rating and the rest in to the 4-5 rating:

```
"aggregations" : {
  "book_ratings_range" : {
    "buckets" : [
      {
        "key" : "1.0-4.0",
        "from" : 1.0,
        "to" : 4.0,
        "doc_count" : 2
      },
      {
        "key" : "4.0-5.0",
        "from" : 4.0,
        "to" : 5.0,
        "doc_count" : 35
      }
    ]
  }
}
```

```
}
```

The `range` aggregation is a slight variation from histogram aggregations and, hence, well suited for special or custom ranges that a user needs sometimes. Of course, if you want to go with the system-provided categories and don't need customization, a histogram is the suitable one for that purpose.

NOTE The `range` aggregation is made up of `from` and `to` attributes with the `from` value included while the `to` value is excluded when calculating the bucket items that fit this range.

By the same principle, we can also classify IP addresses in a custom range using a dedicated `ip_range` aggregation. The code in listing 13.15 demonstrates exactly that. Note that this code is for demonstrative purposes only because we do not have a `networks` index primed with data consisting of the `localhost_ip_address` field.

Listing 13.15 Classifying IP addresses in to two baskets

```
GET networks/_search
{
  "aggs": {
    "my_ip_addresses_custom_range": {
      "ip_range": { #A
        "field": "localhost_ip_address", #B
        "ranges": [ #C
          {
            "to": "192.168.0.10",
            "from": "192.168.0.20"
          },
          {
            "to": "192.168.0.20",
            "from": "192.168.0.100"
          }
        ]
      }
    }
  }
}
```

#A The `ip_range` bucketing segregates certain IPs in to given buckets.

#B Runs the range aggregation on this field (must be of type ip)

#C Defines the custom range of IP addresses that are expected to be categorized

As you can see from this sample aggregation, we can segregate the IP addresses as per our custom range. The query produces two ranges: one that includes 192.168.0.10 to 192.168.0.20 and, in the second bucket, 192.168.0.20 to 192.168.0.100.

13.3.4 The terms aggregation

When we want to retrieve an aggregated count of a certain field, say authors and their book count, the `terms` aggregations comes in handy. The `terms` aggregation collects data in the

buckets for each occurrence of the term. For example, in the following query, the `terms` aggregation creates a bucket for each author and the number of books they've written.

Listing 13.16 Aggregated count of books by authors

```
GET books/_search?size=0
{
  "aggs": {
    "author_book_count": {
      "terms": { #A
        "field": "author.keyword" #B
      }
    }
  }
}
```

#A Declares the terms aggregation type
#B Applies the aggregation to this field

The query uses the `terms` aggregation to fetch the list of authors in the `books` index as well as their book count. The response indicates the `key` as author, and the `doc_count` shows the number of books for each author:

```
"buckets" : [
  {
    "key" : "Herbert Schildt",
    "doc_count" : 2
  },
  {
    "key" : "Mike McGrath",
    "doc_count" : 2
  },
  {
    "key" : "Terry Norton",
    "doc_count" : 2
  },
  {
    "key" : "Adam Scott",
    "doc_count" : 1
  }
..]
```

As you can see from this response, each bucket represents an author with the number of books that the author wrote. By default, the `terms` aggregation only returns the top 10 aggregations, but you can tweak this return size by setting the `size` parameter in the `terms` aggregation as the following listing shows.

Listing 13.17 Terms query with a custom size

```
GET books/_search?size=0
{
  "aggs": {
    "author_book_count": {
      "terms": {
        "field": "author.keyword",
        "size": 100
      }
    }
  }
}
```

```

        "size": 25 #A
    }
}
}
```

#A Sets the aggregation size

Here, setting `size` to 25 fetches 25 aggregations (25 authors and their book count).

13.3.5 Multi-terms aggregation

The `multi_terms` aggregation resembles the `terms` aggregation we saw in the last section with an additional feature: it aggregates the data based on multiple keys. For example, rather than just finding the number of books written by an author, we might want to find the number of books with a specific title and author. The following listing shows the query to get the author and their book's title(s) as a map.

Listing 13.18 Aggregation for authors and their book titles as a map

```
GET books/_search?size=0
{
  "aggs": {
    "author_title_map": {
      "multi_terms": { #A
        "terms": [ #B
          {
            "field": "author.keyword"
          },
          {
            "field": "title.keyword"
          }
        ]
      }
    }
  }
}
```

#A Declares the aggregation type

#B The set of terms with which to form the author/title map

As you can see in listing 13.18, `multi_terms` accepts a set of terms. In the example, we expect Elasticsearch to return a book count using the `author/title` keys. The response indicates that we were able to retrieve such information:

```
{
  "key" : [
    "Adam Scott",
    "JavaScript Everywhere"
  ],
  "key_as_string" : "Adam Scott|JavaScript Everywhere",
  "doc_count" : 1
},
{
  "key" : [
    "Al Sweigart",
    "Automate the Boring Stuff with Python"
  ],
  "key_as_string" : "Al Sweigart|Automate the Boring Stuff with Python",
  "doc_count" : 1
}
```

```

    "Automate The Boring Stuff With Python"
],
"key_as_string" : "Al Sweigart|Automate The Boring Stuff With Python",
"doc_count" : 1
},
...

```

This response shows two variations of the `key` representation: as a set of fields (both `author` and `title`) and as a string (`key_as_string`), which is nothing more than stitching both fields together by a `|` delimiter. The `doc_count` indicates the number of documents (books) that are present in the index for that key.

If you are curious, rerun the query in 13.18, this time using the tags and the title as terms. You should get multiple books under the same tags as we would expect (the code is available in my GitHub repository).

Before we jump into the third type of aggregations, the pipeline aggregations, we need to understand the concept of parent and sibling aggregations. These form the basis of pipeline aggregations. We'll discuss parent and sibling aggregations in the next section and then jump to our final section of the chapter, the pipeline aggregations.

13.4 Parent and sibling aggregations

Broadly speaking, we can group aggregations into two types: *parent* and *sibling* aggregations. You may find them a bit confusing, so let's see what they are and how they can be used.

13.4.1 Parent aggregations

Parent aggregations are a group of aggregations that work on the input from the parent aggregation to produce new buckets, which are then added to the existing buckets. Take a look at the following code listing.

Listing 13.19 Parent aggregations

```

GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "coffee_sales_by_day": {
      "date_histogram": {
        "field": "date",
        "calendar_interval": "1d"
      },
      "aggs": {
        "cappuccino_sales": {
          "sum": {
            "field": "sales.cappuccino"
          }
        }
      }
    }
  }
}

```

```
}
```

If you look closely (see figure 13.5), the `cappuccino_sales` aggregation is created as a child of the parent `coffee_sales_by_day` aggregation. It is at the same level as the `date_histogram`.



Figure 13.5 Parent aggregations visualized

The result of such an aggregation produces a set of buckets inside the existing bucket. Figure 13.6 shows this result. As you can see in figure 13.5, the `cappuccino_sales` aggregation produces the new buckets that are tucked away under the main `date_histogram` bucket.

```

"aggregations" : {
  "coffee_sales_by_day" : {
    "buckets" : [
      {
        "key_as_string" : "2022-09-01T00:00:00.000Z",
        "key" : 1661990400000,
        "doc_count" : 1,
        "cappuccino_sales" : {
          "value" : 23.0
        }
      },
      {
        "key_as_string" : "2022-09-02T00:00:00.000Z",
        "key" : 1662076800000,
        "doc_count" : 1,
        "cappuccino_sales" : {
          "value" : 40.0
        }
      }
    ]
  }
}

```

The new buckets are added to the existing buckets

Figure 13.6 New buckets created inside the existing buckets

13.4.2 Sibling aggregations

Sibling aggregations are those that produce a new aggregation at the same level of the sibling aggregation. The code in the following listing creates an aggregation with two queries at the same level (hence, we call them siblings).

Listing 13.20 Sibling aggregation in action

```

GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "coffee_date_histogram": {
      "date_histogram": {
        "field": "date",
        "calendar_interval": "1d"
      }
    },
    "total_sale_of_americanos": {
      "sum": {
        "field": "sales.americano"
      }
    }
  }
}

```

```
}
```

In the listing, the `coffee_date_histogram` and `total_sales_of_americanos` aggregations are defined at the same level. If we take a snapshot of the query with the aggregations collapsed, we'd see them pictorially as shown in figure 13.7.

```
GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "coffee_date_histogram": { },
    "total_sale_of_americanos": { }
  }
}
```

Figure 13.7 Sibling aggregations on the query side

When we execute the sibling queries, new sets of buckets are produced; however, unlike parent aggregations, where the buckets are created and added to the existing buckets, with sibling aggregations, new aggregations or new buckets are created at the root aggregation level. The query in listing 13.20 produces the aggregated results in figure 13.8 with newly created buckets for each of the sibling aggregations.

```
"aggregations" : {
  "total_sale_of_americanos" : {
    "value" : 28.0
  },
  "coffee_date_histogram" : {
    "buckets" : [
      { },
      { }
    ]
  }
}
```

Figure 13.8 Sibling queries output aggregations at same level

13.5 Pipeline aggregations

In the last few sections, we looked at creating aggregations by generating metrics on the data, by bucketing data, or by a combination of both. But sometimes we may want to chain a few aggregations to produce another level of metric or a bucket. For example, say that we want to find the maximum and minimum of all the buckets that are produced during an aggregation or that we want to find the moving average of a sliding window of data, such as the hourly average sales during a cyber-Monday sale. The metric or bucket aggregations won't allow us to deal with the requirement of chaining aggregations.

Elasticsearch provides a third set of aggregations called *pipeline aggregations* that permit chaining the aggregations. These aggregations work on the output of other aggregations rather than the individual documents or fields of the documents. That is, we create a pipeline aggregation by passing the output of a bucket or a metric aggregation. Before we dig into getting our hands dirty, let's look at the types of pipelines, their syntax, and some other details in the next section.

13.5.1 Pipeline aggregation types

Broadly speaking, we can group pipeline aggregations into two types: *parent* and *sibling*. As mentioned previously, the parent pipeline aggregations are a group of aggregations that work on the input from the parent aggregation to produce new buckets or new aggregations, which are then added to the existing buckets. The sibling pipeline aggregations produce a new aggregation at the same level of the sibling aggregation. Revisit section 13.4 to reinforce the basics on parent and sibling aggregations.

13.5.2 ample data

We will look at both parent and sibling aggregation types in detail as we execute some examples in this section. We'll use the `coffee_sales` data as a sample data set for running these pipeline aggregations. Follow the usual process of indexing the data using the `_bulk` API as listing 13.21 shows. You can fetch the sample data from the book's repository on Github at: https://github.com/madhusudhankonda/elasticsearch-in-action/blob/main/datasets/coffee_sales.txt

Listing 13.21 Indexing data using the `_bulk` API

```
PUT coffee_sales/_bulk
{"index":{"_id":"1"}}
{"date":"2022-09-
    01","sales":{"cappuccino":23,"latte":12,"americano":9,"tea":7},"price":{"cappuccino":2.50,"latte":2.40,"americano":2.10,"tea":1.50}}
 {"index":{"_id":"2"}}
 {"date":"2022-09-
    02","sales":{"cappuccino":40,"latte":16,"americano":19,"tea":15},"price":{"cappuccino":2.50,"latte":2.40,"americano":2.10,"tea":1.50}}
```

Executing this query indexes two sales documents into our `coffee_sales` index. Now that we have a couple of documents in `coffee_sales`, the next step is to create a set of pipeline aggregations to help us understand them in detail.

13.5.3 Syntax for the pipeline aggregations

Pipeline aggregations, as discussed, work on the input from other aggregations. That means, when declaring the pipeline, it is expected that we provide a reference to the metric or bucket aggregations. For our example, we can set this reference as `buckets_path`, which is made of the aggregation names with an appropriate separator in the query. The `buckets_path` variable is a mechanism to identify the input to the pipeline query.

For example, figure 13.9 indicates the parent aggregation `cappuccino_sales`, whereas the pipeline aggregation `cumulative_sum` as defined by `total_cappuccinos` refers to the parent aggregation via the `buckets_path`, which is set with a value referring to the name of the parent aggregation.

```
GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "sales_by_coffee": {
      "date_histogram": { },
      "aggs": {
        "cappucino_sales": {
          "sum": { }
        },
        "total_cappuccinos": {
          "cumulative_sum": {
            "buckets_path": "cappucino_sales"
          }
        }
      }
    }
  }
}

The cumulative_sum is referring to the
parent aggregation (defined by
cappucino_sales) by setting the
buckets_path to cappucino_sales
```

Figure 13.9 Parent pipeline aggregation bucket path setting

The `buckets_path` setting becomes a bit more involved if the aggregation that's in play is a sibling aggregation. Figure 13.10 shows the aggregation.

```

GET coffee_sales/_search
{
  "size": 0,           Sibling aggregations
  "aggs": {
    "sales_by_coffee": {           "max_bucket" (a sibling aggregation) is referring
      "date_histogram": {          to the constituents of a sibling aggregations (defined
        "ags": {                  by sales_by_coffee and cappuccino_sales)
          "cappuccino_sales": {}   by setting the buckets_path to
        }                         sales_by_coffee>cappuccino_sales
      },
      "highest_cappuccino_sales_bucket": {
        "max_bucket": {
          "buckets_path": "sales_by_coffee>cappuccino_sales"           The ">" operator is the aggregation
        }                           separator
      }
    }
  }
}
  
```

The `max_bucket` (a sibling aggregation) is referring to the constituents of a sibling aggregations (defined by `sales_by_coffee` and `cappuccino_sales`) by setting the `buckets_path` to `sales_by_coffee>cappuccino_sales`

The `>` operator is the aggregation separator

The `buckets_path` for a sibling pipeline aggregation

Figure 13.10 Sibling pipeline aggregation bucket path setting

The `max_bucket` in the aggregation in figure 13.10 is a sibling pipeline aggregation (defined under the `highest_cappuccino_sales_bucket` aggregation), which calculates the result by taking input from the other aggregations set by the `buckets_path` variable. In this case, it is fed by the aggregation called `cappuccino_sales`, which lives under the `sales_by_coffee` sibling aggregation.

If you are puzzled with `buckets_path` or even with pipeline aggregations, hang in there. We will go over them in practice in the next few sections.

13.5.4 List of pipeline aggregations

Knowing if the pipeline aggregation falls in which type of aggregation, parent or sibling, helps us develop these aggregations with ease. Tables 13.1 and 13.2 show the list of pipeline aggregations and their definitions.

Table 13.1 Parent pipeline aggregations

Name	Description
Bucket script (<code>buckets_script</code>) aggregation	Executes a script on multi-bucket aggregations
Bucket selector (<code>bucket_selector</code>) aggregation	Executes a script to select the current bucket for its place in the multi-bucket aggregation

Bucket sort (bucket_sort) aggregation	Sorts the buckets
Cumulative cardinality (cumulative_cardinality) aggregation	Checks the recently added unique (cumulative cardinality) values
Cumulative sum (cumulative_sum) aggregation	Finds the cumulative sum of a metric
Derivative (derivative) aggregation	Finds the derivative of a metric in a histogram or date histogram
Inference (inference) aggregation	Finds the inference on a pretrained model
Moving function (moving_function) aggregation	Executes a custom script on a sliding window
Moving percentiles (moving_percentiles) aggregation	Similar to moving_function except it calculates in percentiles
Normalize (normalize) aggregation	Calculates the normalized value of a given bucket
Serial difference (serial_diff) aggregation	Calculates the serial difference on a metric

Table 13.2 Sibling pipeline aggregations

Name	Description
Average (avg_bucket) aggregation	Calculates the average value of the sibling metric
Bucket count (bucket_count_ks_test) aggregation	Calculates the Kolmogorov–Smirnov statistic over a distribution
Bucket correlation (bucket_correlation) aggregation	Executes a correlation function
Change point (change_point) aggregation	Detects the dips and changes in a metric
Extended stats (extended_stats) aggregation	Calculates multiple statistical functions
Max bucket (max_bucket) aggregation	Finds the maximum valued bucket
Min bucket (min_bucket) aggregation	Finds the minimum valued bucket

Percentiles bucket (percentiles_bucket) aggregation	Calculates the percentiles of a metric
Stats bucket (stats_bucket) aggregation	Calculates common stats for a metric
Sum bucket (sum_bucket) aggregation	Calculates the sum of a metric

We will not be able to go over all of the pipeline aggregations in this section, but we can learn and understand the basics of pipeline aggregations by working through a few common ones. To begin, let's suppose we want to find our cumulative coffee sales: how many cappuccinos are sold daily, for example. Instead of having a daily score, we want to have the total number of cappuccinos sold from the first day of operation, accumulated daily. The `cumulative_sum` aggregation is a handy parent pipeline aggregation that keeps a sum total for the current day as well as tracks the sum for the next day and so on. Let's see it in action in the next section.

13.5.5 Cumulative sum parent aggregation

To collect the cumulative sum of coffees sold each day, we can chain the coffee sales by day and pass the results to the `cumulative_sum` pipeline aggregation. The code snippet in the following listing fetches the cumulative sum of cappuccinos sold each day by using this pipeline aggregation.

Listing 13.22 Cumulative sales (sum) of coffees sold daily

```
GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "sales_by_coffee": {
      "date_histogram": {
        "field": "date",
        "calendar_interval": "1d"
      },
      "aggs": {
        "cappuccino_sales": {
          "sum": {
            "field": "sales.cappuccino"
          }
        },
        "total_cappuccinos": {
          "cumulative_sum": { #A
            "buckets_path": "cappuccino_sales"
          }
        }
      }
    }
  }
}
```

#A Parent aggregation that calculates the cumulative total of coffee sales

Let's dissect the aggregation in listing 13.22. We have a `sales_by_coffee` aggregation, which is a `date_histogram` that brings all the dates and the documents that fall within those dates (so far, we only have two dates). We also have a sub-aggregation (`cappuccino_sales`) that sums the sales figures for cappuccinos for that bucket.

The highlighted portion of the code is the parent pipeline aggregation (`total_cappuccinos`). It fetches the cumulative coffee sales per day. This is called a parent pipeline aggregation because it is applied in the scope of its parent, the `cappuccino_sales` aggregation. The following code snippet shows the result of the aggregation:

```
"aggregations" : {
    "sales_by_coffee" : {
        "buckets" : [
            {
                "key_as_string" : "2022-09-01T00:00:00.000Z",
                "key" : 1661990400000,
                "doc_count" : 1,
                "cappuccino_sales" : {
                    "value" : 23.0
                },
                "total_cappuccinos" : {
                    "value" : 23.0
                }
            },
            {
                "key_as_string" : "2022-09-02T00:00:00.000Z",
                "key" : 1662076800000,
                "doc_count" : 1,
                "cappuccino_sales" : {
                    "value" : 40.0
                },
                "total_cappuccinos" : {
                    "value" : 63.0
                }
            }
        ]
    }
}
```

Let's go over the result for a moment. As you can see, the buckets are segregated by dates (check `key_as_string`) due to the `date_histogram` aggregation at the top of the query. We also created a sub-aggregation (`cappuccino_sales`) that fetches the number of cappuccinos sold daily (per bucket). The final part of the result is the cumulatively totalled sum of cappuccinos (`total_cappuccinos`) added to the existing bucket. Notice that on day 2, the total cappuccinos were 63 (23 from the first day and 40 from the second day).

While the cumulative sum total of cappuccinos is an existing parent bucket level, finding the maximum or minimum coffees sold in a set of buckets is at a sibling level. For that, we will need to create an aggregation at the same level as the main aggregation, which is why the aggregation is called a sibling aggregation.

Let's say we want to find which day the most cappuccinos were sold or, conversely, on which day the least number of cappuccinos were sold. To do this, we need to utilize the

pipeline aggregation's `max_bucket` and `min_bucket` aggregations, which the next section covers.

13.5.6 Max and min sibling pipeline aggregations

Elasticsearch provides a pipeline aggregation called `max_bucket` to fetch the top bucket from the given set of buckets fetched from the other aggregations. Remember, the pipeline aggregation takes the input of other aggregations to calculate its own aggregation.

THE MAX_BUCKET AGGREGATION

The query in the following listing enhances the aggregation we performed in the last section. It does this by adding a `max_bucket` function.

Listing 13.23 Pipeline aggregation to find to sales of cappuccinos

```
GET coffee_sales/_search
{
  "size": 0,
  "aggs": {
    "sales_by_coffee": {
      "date_histogram": {
        "field": "date",
        "calendar_interval": "1d"
      },
      "aggs": {
        "cappuccino_sales": {
          "sum": {
            "field": "sales.cappuccino"
          }
        }
      }
    },
    "highest_cappuccino_sales_bucket": {
      "max_bucket": {
        "buckets_path": "sales_by_coffee>cappuccino_sales"
      }
    }
  }
}
```

As you can see in the highlighted code, the `highest_cappuccino_sales_bucket` is the custom name given to the sibling pipeline aggregation that we are about to perform. We declare the `max_bucket` aggregation at the same level as the `sales_by_coffee` aggregation; hence, it is called a sibling aggregation. This expects a `buckets_path`, which is the combination of the aggregations `sales_by_coffee` and `cappuccino_sales`. (These two were the result of bucket and metric aggregations on the data.) Once executed, we get this response:

```
"aggregations" : {
  "sales_by_coffee" : {
    "buckets" : [
      {
        "key_as_string" : "2022-09-01T00:00:00.000Z",
        "key" : 1661990400000,
```

```

"doc_count" : 1,
"cappuccino_sales" : {
  "value" : 23.0
},
{
  "key_as_string" : "2022-09-02T00:00:00.000Z",
  "key" : 1662076800000,
  "doc_count" : 1,
  "cappuccino_sales" : {
    "value" : 40.0
  }
}
},
"highest_cappuccino_sales_bucket" : {
  "value" : 40.0,
  "keys" : [
    "2022-09-02T00:00:00.000Z"
  ]
}
}

```

In this snippet, the highlighted portion indicates the `highest_cappuccino_sales_bucket` information. The date `2022-09-02` (September 2, 2022) is the one where the most cappuccinos were sold.

THE MIN_BUCKET AGGREGATION

We can also fetch the days where the cappuccinos were not sold as much. To do this, we need to use the `min_bucket` pipeline aggregation. Replace the highlighted code in listing 30.23 with the code in the following snippet:

```

..
"lowest_cappuccino_sales_bucket":{
  "min_bucket": {
    "buckets_path": "sales_by_coffee>cappuccino_sales"
  }
}

```

This should yield the lowest number of cappuccinos that were sold on a particular day (Sept 1, 2022, in this case). The following response demonstrates this:

```

"lowest_cappuccino_sales_bucket" : {
  "value" : 23.0,
  "keys" : [
    "2022-09-01T00:00:00.000Z"
  ]
}

```

There are a handful of pipeline aggregations like the metric and bucket aggregations. Because it is impractical to discuss all of these in this chapter, I will try to put up the code samples on my GitHub for most of the aggregations. Refer to the code there if you are interested. I also advise you to check the official documentation when you work with a particular aggregation. Here's the link to pipeline aggregations: <https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-pipeline.html>

That's pretty much it in the world of aggregations! Let's wrap it up here.

13.6 Summary

- Although a search finds the answers in the amassed data based on a search criteria, an aggregation compiles patterns, insights, and information for the data collected by our organizations.
- Aggregations find intelligence in the data organizations hold.
- Elasticsearch allows us to perform nested and sibling aggregations on our data.
- Elasticsearch classifies aggregations into three types: metrics, buckets, and pipelines.
- The metric aggregations fetch single-value metrics such as average, min and max, sum, and so on.
- Bucket aggregations classify the data into various buckets based on a certain bucketing strategy. With a bucketing strategy, we can ask Elasticsearch to split the data into buckets as deemed fit.
- We can either let Elasticsearch create predefined buckets based on the interval we provide or create custom ranges
 - If the interval is 10 for an age group, for example, Elasticsearch splits the data into steps of 10.
 - If we want to create a range like 10-30 and 30-100, where the interval differs, we can do so by creating a custom range.
- Pipeline aggregations work on the output from the other metric and bucket aggregations to create new aggregations or new buckets.