

# Elasticsearch IN ACTION

Radu Gheorghe  
Matthew Lee Hinman



MANNING



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

Copyright 2014 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# Welcome

Thanks for purchasing the MEAP for *Elasticsearch in Action*. We're excited to see the book reach this stage, and we're looking forward to its continued development and eventual release. This is an intermediate level book, designed for anyone writing applications using Elasticsearch, or responsible for managing Elasticsearch in a production environment.

We've worked to make the content both approachable and meaningful, and to explain not just how to do things with Elasticsearch, but also *why* things are done the way they are. We feel it's important to know Elasticsearch's foundations prior to diving into all of the different ways you can leverage it.

We're releasing the first five chapters to start. Chapter 1 explains what Elasticsearch is: what a typical use-case looks like, the tasks it's good at, and the challenges it faces. By the end of chapter one, you'll know if Elasticsearch is likely to be a good fit for you and what you can expect from it. You will also learn various ways to install it.

Chapter 2 is about getting your hands dirty with Elasticsearch's functionality. You will understand how data is organized, both logically and physically. By the end, you'll know how to index and search for documents, as well as how to configure Elasticsearch.

Chapter 3 is all about indexing, updating and deleting documents. You will understand the types of fields you can have in your documents – all of which goes into your mapping definition, as well as how updating and deleting data works under the covers.

Chapter 4 is all about searching. You will understand most types of queries, and when you'll use which. You will also get a clear idea about the difference between queries and filters: how they work and where you'd use which.

Chapter 5 explains analysis, which is a big chunk of how Elasticsearch can make your searches fast, and also return the right results in the right order. Analysis is the process of making tokens out of your text and by the end, you'll understand how it works and how to configure analysis for your use-case.

Looking ahead in part 2, other chapters will deal with more advanced functionality: how to make searches more relevant – where you'll discover some more query types; how to work with relational data, and how to extract statistics from your data in real-time, using aggregations. We'll also show how to make Elasticsearch perform to your production standards: we'll talk about scaling out, tuning indexing and search performance, as well as administering your cluster.

As you're reading, we hope you'll take advantage of the Author Online forum. We'll be reading your comments and responding. We appreciate any feedback, as it's very helpful in the development process. Thanks again!

Matthew Lee Hinman and Radu Gheorghe

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<http://www.manning-sandbox.com/forum.jspa?forumID=871>

Licensed to zhailiang UNKNOWN <[zhailiang@belink.com](mailto:zhailiang@belink.com)>

# *brief contents*

---

## **PART 1: CORE ELASTICSEARCH FUNCTIONALITY**

- 1 Introducing Elasticsearch*
- 2 Diving into the functionality*
- 3 Indexing, updating and deleting data*
- 4 Searching your data*
- 5 Analyzing your data*
- 6 Searching with relevancy*
- 7 Exploring your data with Aggregations*
- 8 Relations among documents*

## **PART 2: ADVANCED FUNCTIONALITY**

- 9 Scaling out*
- 10 Improving performance*
- 11 Administering your cluster*

## **APPENDICES:**

- Appendix A Working with geo-spatial data*
- Appendix B Plugins*
- Appendix C Highlighting*
- Appendix D Introduction to the Java API*
- Appendix E JVM and JC tuning 101*

# 1

## *Introducing Elasticsearch*

### ***This chapter covers***

- Understanding search engines and the issues they address
- How Elasticsearch fits in the context of search engines
- Typical scenarios for Elasticsearch
- Features Elasticsearch provides
- Installing Elasticsearch

We use search everywhere these days. And that's a good thing, because search helps you finish tasks quickly and easily. Whether you're buying something from an online shop or visiting a blog, you expect to have a search box somewhere to help you find what you're looking for, without scanning the entire website. Maybe it's me, but when I wake up in the morning, I wish I could enter the kitchen and type in "bowl" in a search box somewhere and have my favorite bowl highlighted.

We've also come to expect those search boxes to be smart. I don't want to have to type the entire word "bowl"; I expect the search box to come up with suggestions, and I don't want results and suggestions to come to me in random order. I want the search to be smart and give me the most relevant results first: to guess what I want, if that's possible. For example, if I search for "laptop" from an online shop but have to scroll through laptop accessories before I get to a laptop, I'm likely to go somewhere else after the first page of results. And this isn't only because we're in a hurry and spoiled with good search interfaces, it's also because there's increasingly more stuff to choose from. For example, a friend asked me to help her buy a new laptop. Typing "best laptop for my friend" in the search box of an online store that sells

thousands of laptops wouldn't be effective. Good keyword search is often not enough; you need aggregated data so you can narrow down the results to what the user is interested in. I narrowed down my laptop search by selecting the size of the screen, the price range, and so on, until I had five or so laptops to choose from.

Finally, there's the matter of performance—because nobody wants to wait. I've seen websites where you search for something and get the results in few minutes. *Minutes!* For a search.

If you want to provide search for your data, you'll have to deal with all these issues: returning relevant search results, returning statistics, and doing all that quickly. This is where search engines like Elasticsearch come into play because they're built to meet exactly those challenges.

You can deploy a search engine on top of a relational database to create indexes and speed up the SQL queries. Or, you can index data from your NoSQL data store to add search capabilities there. You can do that with Elasticsearch, and it works well with document-oriented stores like MongoDB because data is represented in Elasticsearch as documents, too.

Modern search engines like Elasticsearch also do a good job storing your data, so you can use it as a NoSQL data store with powerful search capabilities. Elasticsearch is open-source, distributed, and it's built on top of Apache Lucene, an open-source search engine library, which allows you to implement search functionality in your own Java application. Elasticsearch takes this functionality and extends it to make storing, indexing, and searching faster, easier, and, as the name suggests, elastic. Also, your application doesn't need to be written in Java to work with Elasticsearch; you can send data over HTTP in JSON to index, search, and manage your Elasticsearch cluster.

In this chapter, we expand on these searching and data features, and you'll learn to use them throughout this book. First, let's take a closer look at the challenges search engines are typically confronted with and Elasticsearch's approach to solving them.

## 1.1 Elasticsearch as a search engine

To get a better idea of how Elasticsearch works, let's look at an example. Imagine that you're working on a website that hosts blogs, and you want to let users search across the entire site for specific posts. Your first task is to implement keyword search. For example, if a user searches for "elections", you'd better return all posts containing that word.

A search engine will do that for you, but for a *robust* search feature, you need more than that: results need to come in quickly, and they need to be relevant. And it's also nice to provide features that help users search when they don't know the exact words of what they're looking for. Those features include detecting typos, providing suggestions, and breaking down results into categories.

**TIP** In most of this chapter, you'll get an overview of Elasticsearch's features. If you want to get practical and jump to installing it, skip to section 1.5. You'll find the installation procedure surprisingly easy. And you can always come back here for the high-level overview.

### 1.1.1 Providing quick searches

If you have a huge number of posts on your site, searching through all of them for the word “elections” can take a long time. And you don’t want your users to wait. That’s where Elasticsearch helps because it uses Lucene, a high-performance search engine library, to index all your data by default.

An index is a data structure, which you create along with your data and is meant to allow faster searches. You can add indexes to fields in most databases, and there are several ways to do it. Lucene does it with *inverted indexing*, which means it creates a data structure where it keeps a list of where each word belongs. For example, if you need to search for blog posts by their tags, using inverted indexing might look like table 1.1.

Table 1.1 Inverted index for blog tags

Raw data		Index data	
Blog Post ID	Tags	Tags	Blog Post IDs
1	elections	elections	1, 3
2	peace	peace	2, 3, 4
3	elections, peace		
4	peace		

If you search for blog posts that have an `elections` tag, it’s much faster to look at the index rather than looking at each word of each blog post, because you only have to look at the place where tags is “`elections`”, and you’ll get all the corresponding blog posts. This speed gain makes sense in the context of a search engine. In the real world, you’re rarely searching for one word only. For example, if you’re searching for “Elasticsearch in Action”, three-word look ups imply multiplying your speed gain by three. All this may seem a bit complex at this point, but we’ll clear up the details when we discuss indexing in chapter 3 and searching in chapter 4.

An inverted index is appropriate for a search engine when it comes to relevance, too. For example, when you’re looking up a word like “`peace`”, not only will you see which document matches, but you’ll also get the number of matching documents for free. This is important because if a word occurs in most documents, it’s probably less relevant. Let’s say you search for “Elasticsearch in Action”, and a document contains the word “`in`”—along with a million other documents. At this point, you know that “`in`” is a common word, and the fact that this document matched doesn’t say much about how relevant it is to your search. In contrast, if it

contains “Elasticsearch”, along with a hundred others, you know you’re getting closer to relevant documents. Although, it’s not “you” who has to know you’re getting closer, Elasticsearch does that for you. You’ll learn all about tuning data and searches for relevancy in chapter 6.

That said, the tradeoff for improved search performance and relevancy is that the index will take up disk space, and adding new blog posts will be slower because you have to update the index after adding the data itself. On the upside, tuning can make Elasticsearch faster, both when it comes to indexing and searching. We’ll discuss these topics in great detail in chapter 10.

### **1.1.2 Ensuring relevant results**

Then there’s the hard part: how do you make the blog posts that are about elections appear before the ones that merely contain that word? With Elasticsearch, you have a few algorithms for calculating the relevancy score, which is used, by default, to sort the results.

The relevancy score is a number assigned to each document that matches your search criteria and indicates how relevant the given document is to the criteria. For example, if a blog post contains “elections” more times than another, it’s more likely to be about elections.

By default, the algorithm used to calculate a document’s relevancy score is *tf-idf*. We’ll discuss scoring and tf-idf more in chapters 4 and 6, which are about searching and relevancy, but here’s the basic idea: *tf-idf* comes from **term frequency-inverse document frequency**, which are the two factors that influence relevancy score:

- *Term frequency*—The more times the words you’re looking for appear in a document, the higher the score
- *Inverse document frequency*—The weight of each word is higher if the word is uncommon across other documents

For example, if you’re looking for “bicycle race” on a cyclist’s blog, the word “bicycle” counts much less for the score than “race.” But the more times both words appear in a document, the higher that document’s score.

In addition to choosing an algorithm, Elasticsearch provides many other built-in features to influence the relevancy score to suit your needs. For example, you can “boost” the score of a particular field, such as the title of a post, to be more important than the body. This gives higher scores to documents that match your search criteria in the title, compared to similar documents that match only the body. You can make exact matches count more than partial matches, and you can even use a script to add custom criteria to the way the score is calculated. For example, if you let users like posts, you can boost the score based on the number of likes, or you can make newer posts have higher scores than similar, older posts.

Don’t worry about the mechanics of any of these features right now; we discuss relevancy in great detail in chapter 6. For now, let’s focus on what you can do with Elasticsearch and when you’d want to use those features.

### **1.1.3 Searching beyond exact matches**

Finally, with Elasticsearch you have options to make your searches intuitive and go beyond exactly matching what the user types in. These options are handy when the user enters a typo or uses a synonym or a derived word different than what you've stored. And they're also handy when the user doesn't know exactly what to search for in the first place.

#### **HANDLING TYPOs**

You can configure Elasticsearch to be tolerant of variations instead of looking for only exact matches. A fuzzy query can be used so a search for "bicycel" will match a blog post about bicycles. We explore fuzzy queries and other features that make your searches relevant in chapter 6.

#### **SUPPORTING DERIVATIVES**

You can also use analysis, covered in chapter 5, to make Elasticsearch understand that a blog with "bicycle" in its title should also match queries that mention "bicyclist" or "cycling".

#### **USING STATISTICS**

When users don't know what to search for, you can help them in a number of ways. One way is to present statistics through aggregations, which we cover in chapter 8. Imagine that upon entering your blog, users see popular topics listed on the right-hand side. One topic may be cycling. Those interested in cycling would click that topic to narrow the results. Then, you might have another facet to separate cycling posts into "bicycle reviews," "cycling events," and so on.

#### **PROVIDING SUGGESTIONS**

Once users start typing, you can help them discover popular searches and popular results. You can use suggestions to predict their searches as they type, like most search engines on the web do. You can also show popular results as they type, using special query types that match prefixes, wild cards, or regular expressions.

Let's look now at how Elasticsearch is typically used in production.

## **1.2 Typical setups using Elasticsearch**

We've already established that storing and indexing your data in Elasticsearch is a good way to provide quick and relevant results to your searches. But in the end, Elasticsearch is just a search engine, and you'll never use it on its own. Like any other data store, you need a way to feed data into it, and you probably need to provide an interface for the users searching that data.

To get an idea of how Elasticsearch might fit into a bigger system, let's consider three typical scenarios.

- *Elasticsearch as the primary back end for your website*

As we discussed, you may have a website that allows people to write blog posts, but you also want the ability to search through the posts. You can use Elasticsearch to store all the data related to these posts and serve queries as well.

- *Adding Elasticsearch to an existing system*

You may be reading this book because you already have a system that's crunching data, and you want to add search. We'll look at a couple of overall designs on how that might be done.

- *Elasticsearch as the back end of a ready-made solution built around it*

Because Elasticsearch is open-source and offers a straightforward HTTP interface, a big ecosystem supports it. For example, Elasticsearch is popular for centralizing logs; given the tools already available that can write to and read from Elasticsearch, other than configuring those tools to work the way you want, you don't need to develop anything.

### **1.2.1 One-stop shop for storing, searching, and statistics**

Traditionally, search engines are deployed on top of well-established data stores to provide fast and relevant search capability. That's because, historically, search engines haven't offered durable storage or other features that are often needed, such as statistics. Elasticsearch is one of those modern search engines that provides durable storage, statistics, and many other features you've come to expect from a data store.

If you're starting a new project, we recommend that you consider using Elasticsearch as the only data store to help keep your design as simple as possible. You can also use Elasticsearch on top of another data store, as we discuss later.

Let's return to the blog example: you can store newly written blog posts in Elasticsearch. Similarly, you can use Elasticsearch to retrieve, search, or do statistics through all that data, as shown in figure 1.1.

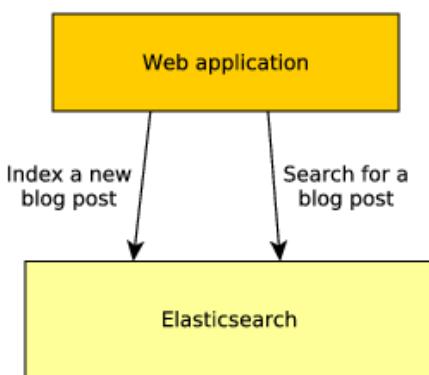


Figure 1.1 Elasticsearch as the only data store

What happens if a server goes down? You can get fault tolerance by replicating your data to different servers. Many other features make Elasticsearch a tempting NoSQL data store. It can't be great for everything, but you should weigh whether including another data store in your overall design is worth the extra complexity.

### 1.2.2 Plugin search in a complex system

By itself, Elasticsearch may not always provide all the functionality you need from a data store. These situations may require you to use Elasticsearch in addition to another data store.

For example, transaction support and complex relationships are features that Elasticsearch doesn't currently support, at least in version 1.0. If you need those features, consider using Elasticsearch along with a different data store.

Or, you may already have a complex system that works, but you want to add search. It might be risky to redesign the entire system for the sole purpose of using Elasticsearch alone (though you might want to do that over time).

Either way, if you have two data stores, you'll have to find a way to keep them synchronized. Depending on what your primary data store is and how your data is laid out, you can deploy an Elasticsearch plugin to keep the two entities synchronized, as illustrated in figure 1.2.

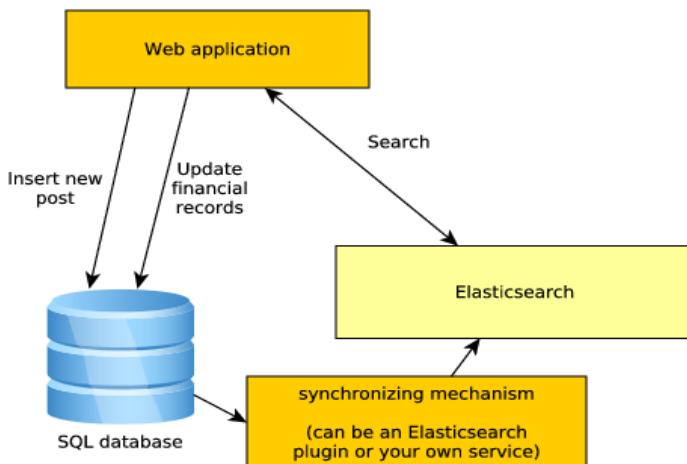


Figure 1.2 Elasticsearch in the same system with another data store

Typically, when you synchronize two entities, Elasticsearch also stores the original data. For example, suppose you have an online retail store with product information stored in an SQL database. You need fast and relevant searching, so you install Elasticsearch. To index the data, you need to deploy a synchronizing mechanism, which can be an Elasticsearch plugin or a custom service that you build. You'll learn more about plugins in appendix B and about dealing with indexing and updating from your own application in chapter 3. This synchronizing

mechanism could pull all the data corresponding to each product and index it in Elasticsearch, where each product is stored as a document. When a user types in search criteria on the web page, the storefront web application queries Elasticsearch for that criteria. Elasticsearch returns a number of product documents that match the criteria, sorted in the way you prefer. Sorting can be based on a relevance score that indicates how many times the words people searched for appear in each product, or anything stored in the product document, such as how recently the product was added, or the average rating, or even a combination of those.

Inserting or updating information can still be done on the “primary” SQL database, so you can use Elasticsearch solely for handling searches. It’s up to the synchronizing mechanism to keep Elasticsearch up to date with the latest changes.

### 1.2.3 Use it with existing tools

In some use cases, you don’t have to write a single line of code to get a job done with Elasticsearch. Many tools are available that work with Elasticsearch, so you don’t have to write yours from scratch.

For example, say you want to deploy a large-scale logging framework to store, search, and analyze a large number of events. As shown in figure 1.3, to process logs and output to Elasticsearch, you can use logging tools such as rsyslog<sup>1</sup>, Logstash<sup>2</sup> or Apache Flume<sup>3</sup>. To search and analyze those logs in a visual interface, you can use Kibana.<sup>4</sup>

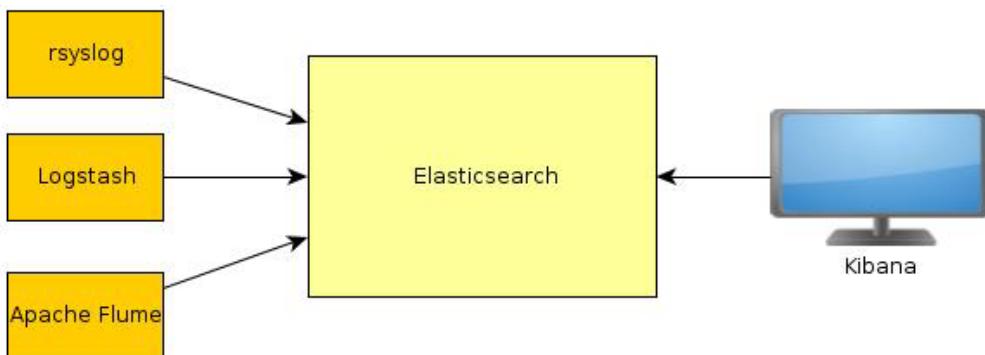


Figure 1.3 Elasticsearch in a system of logging tools, which support Elasticsearch out of the box

The fact that Elasticsearch is open source, under the Apache 2 license to be precise, isn’t the only reason that so many tools support it. Even though Elasticsearch is written in Java,

<sup>1</sup> [www.rsyslog.com/](http://www.rsyslog.com/)

<sup>2</sup> [www.elasticsearch.org/overview/logstash/](http://www.elasticsearch.org/overview/logstash/)

<sup>3</sup> <https://flume.apache.org/>

<sup>4</sup> [www.elasticsearch.org/overview/kibana/](http://www.elasticsearch.org/overview/kibana/)

there's more than a Java API that lets you work with it. It also exposes an HTTP API, which any application can access, no matter the programming language it was written in.

What's more, the HTTP requests and replies are typically in JSON (JavaScript Object Notation) format. Every HTTP request has its payload in JSON, and every reply is also a JSON document.

## JSON and YAML

JSON is a format for expressing data structures. A JSON object typically contains keys and values, where values can be strings, numbers, true/false, null, another object or an array. For more details about the JSON format, visit <http://json.org>

JSON is easy for applications to parse and generate. YAML (YAML Ain't Markup Language), is also supported for the same purpose. To activate YAML, add the `format=yaml` parameter to the HTTP request. For more details on YAML, visit <http://yaml.org>

Although JSON is typically used for HTTP communication, the configuration files are usually written in YAML. In this book, we stick with the popular formats: JSON for HTML communication and YAML for configuration.

For example, a log event might look like this when you index it in Elasticsearch:

```
{
  "message": "logging to Elasticsearch for the first time", #A
  "timestamp": "2013-08-05T10:34:00Z", #B
}
```

**#A A field with a string value**

**#B A string value can be a date, which Elasticsearch evaluates automatically**

**NOTE** Throughout this book, JSON field names are shown in blue and their values are in red to make the code easier to read.

And a search request for log events with a value of `first` in the `message` field would look like this:

```
{
  "query": {
    "match": {
      "message": "first"
    }
  }
}
```

**#A The value of the field query is an object containing the field match**

**#B The match field contains another object in which first is the value of message**

Sending data and running queries by sending JSON objects over HTTP makes it easy for you to extend anything, from a syslog daemon like rsyslog, to a connecting framework like

Apache ManifoldCF to interact with Elasticsearch. If you're building a new application from scratch, or want to add search to an existing application, the REST API is one of the features that makes Elasticsearch appealing. In the next section, we'll look at other such features.

### 1.3 Data structure and interaction

Whether you're ready to start using Elasticsearch from your application or you're still in the stage of evaluating it, you may have two sets of essential questions:

- How do I index, search, and run statistics on a set of data? What are the features that let me do that, and how do they work?

When it comes to indexing and searching, much of Elasticsearch's functionality maps to that of Apache Lucene because Elasticsearch is built on the Lucene search engine library. Elasticsearch exposes most of its functionality through its REST API (and extends it in several places) and has a different implementation of some Lucene concepts.

- How should I organize my data? What are my options in terms of data structure?

In terms of structuring your data, the main unit of indexing and searching is a document. A *document* could be a blog post with all its metadata, a user with all its metadata, or any other type of data you plan to search for. Elasticsearch is document-oriented, as are many NoSQL data stores in which documents are organized in document types, and document types in indices. In SQL terms, you can roughly think of an Elasticsearch index as a database and a document type as a table in that database.

Finally, there's the matter of extending the existing functionality of Elasticsearch. Whether you want to synchronize Elasticsearch with an external data store, such as MongoDB, or you want additional features, there are many plugins you can choose from. We talk more about plugins in appendix B; for now, we'll look at the core functionality.

#### 1.3.1 Understanding indexing and search functionality

Being a search engine library, anyone could use Lucene to implement search in their own Java projects. And if you don't work with Java, there are a few ports of Lucene in other languages. Given that Lucene can do all the heavy lifting for you, from indexing and storing documents to searching them in various ways, why use Elasticsearch when you can use Lucene directly? The answer: Elasticsearch offers functionality you'll want to use in your application that isn't available in Lucene because it's out of its scope. Elasticsearch offers the following features:

- Robust caching
- An HTTP API you can use from applications written in any language
- Backward-compatibility

Elasticsearch makes most of Lucene's features available to your application through the REST API we mentioned previously. You can index documents over HTTP, as you'll see in

chapter 3, and you can run searches the same way, as you'll see in chapter 4, but we already established that Elasticsearch offers much more than good keyword search. We'll discuss statistics via aggregations in chapter 8, and you'll see various ways to make searches more relevant in chapter 6.

### **What about Apache Solr?**

If you've already heard about Lucene, you've probably also heard about Solr, which is an open-source, distributed search engine based on Lucene. In fact, Lucene and Solr merged as a single Apache project in 2010, so you might wonder how Elasticsearch compares with Solr.

Both search engines provide similar functionality, and features evolve quickly with each new version. You can search the web for comparisons, but we'd recommend taking them with a grain of salt. Besides being tied to particular versions, which makes such comparisons obsolete in a matter of months, few authors of such comparisons have in-depth knowledge of both solutions.

That said, a few historical facts help explain the origins of the two products. Solr was created in 2004 and Elasticsearch in 2010. When Elasticsearch came around, its distributed model, which is discussed later in this chapter, made it much easier to scale out than any of the competitors, which suggests the "elastic" part of the name. In the meantime, however, Solr added sharding with version 4.0, which makes the "distributed" argument debatable, like many other aspects.

At the time of this writing, both Elasticsearch and Solr have features that the other one doesn't, and choosing between them may come down to the specific functionality you need at a given point in time. For many use cases, the functionality you need is covered by both, and, as is often the case with competitors, choosing between them becomes a matter of taste. If you want to read more about Solr, we recommend Manning Publication's *Solr in Action* (<http://solrinaction.com>).

### **1.3.2 Analysis**

Another key feature of any Lucene-based search engine is analysis. Through *analysis*, the words from the text you're indexing become terms in Elasticsearch. For example, if you index the text "bicycle race," analysis may produce the terms "bicycle," "race," "cycling," "racing," and when you search for any of those terms, the corresponding document is included in the results.

The same analysis process applies when you search, as illustrated in figure 1.4. If you enter "bicycle race," you probably don't want to search for only the exact match. Maybe a document that contains both those words somewhere will do.

The default analyzer first breaks text into words by looking for common word separators, such as a space or a comma. Then, it lowerscases those words, so that "Bicycle Race" generates "bicycle" and "race." We talk more about analysis in chapter 5.

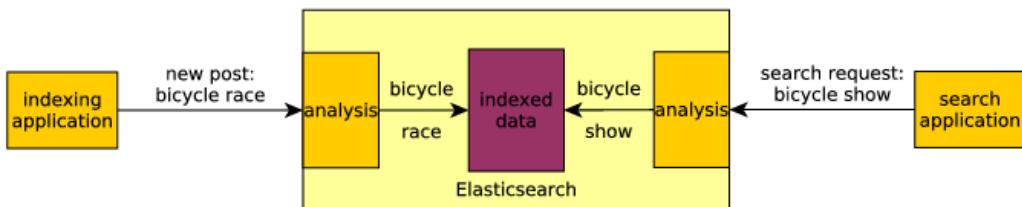


Figure 1.4 Analysis breaks text into words, both when you're indexing and searching.

At this point you might want to know more about what's in that "indexed data" box shown in figure 1.4 because it sounds quite vague. As we'll discuss next, data is organized in documents. By default, Elasticsearch stores your documents as they are, and it also puts all the terms resulting from analysis into the inverted index to enable the all-important fast and relevant searches. We go into more detail about indexing and storing data in chapter 3.

### 1.3.3 Structuring your data in Elasticsearch

Unlike a relational database, which stores data in records or rows, Elasticsearch stores data in documents. Yet, to some extent, the two concepts are similar. With a table, you have columns, and for each column, each row has a value. With a document you have keys and values, in much the same way.

The difference is that a document is more flexible than a record mainly because, in Elasticsearch at least, a document can be hierarchical. For example, the same way you associate a key with a string value, such as "author": "Joe," a document can have an array of strings, such as "tags": ["cycling", "bicycles"], or even key-value pairs, such as "author": {"first\_name": "Joe", "last\_name": "Smith"}.

This flexibility is important because it encourages you to keep all the data that belongs to a logical entity in the same document as opposed to keeping it in different rows in different tables. For example, the easiest (and probably fastest) way of storing blog articles is to keep all the data that belongs to a post in the same document. This way, searches are fast because you don't need joins or any other relational work.

If you have an SQL background, you might miss the ability to use joins. Unfortunately, they're not supported, at least in version 1.0, because it's difficult to do complex relational work in a distributed environment. When data goes back and forth between many nodes, the query time increases dramatically.

You can define relationships in Elasticsearch (we explore these in chapter 7), and although this functionality is limited compared to what you'd expect in a relational database, it can still prove useful. For example, if you index access logs from your HTTP server, you can define a relationship between each log and the blog post that was accessed. This way, you can let users search for blog posts that match certain criteria and show the most accessed first.

But how would this data be organized? Suppose you have two blogs: one about cycling and one about elections, and you put all the posts of each particular blog in its own *index* (see

figure 1.5). An index is much like a database in the sense that it can have its own completely different settings, such as the way it's stored on disk or whether it's read-only. Indices are stored on disk in different sets of files, making them physically separated, although you can search across multiple indices in the same way you search one index.

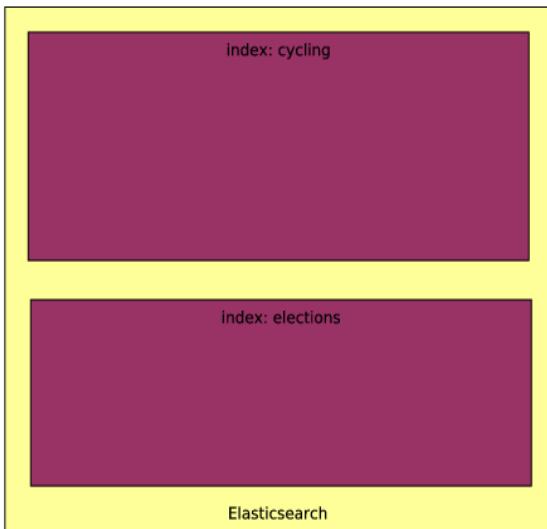


Figure 1.5 Data organized in multiple indices

Inside each index, each document goes into a *type*. The purpose of types is to logically separate different kinds of documents. In this case, let's assume you have blog posts and access logs. Figure 1.6 shows the two types defined in each index.

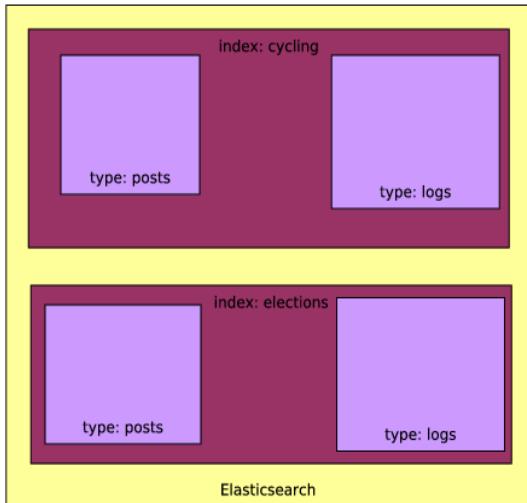


Figure 1.6 Types provide logical separation in the same index.

Types are also used to specify relationships between documents as well. You can define a one-to-many relationship between documents of one type, and documents of another, which we cover in chapter 7. In this case, “posts” are the parent of “logs” because each post can have many corresponding access logs (children), but an access log can only hit one post (parent). The resulting data structure is illustrated in figure 1.7.

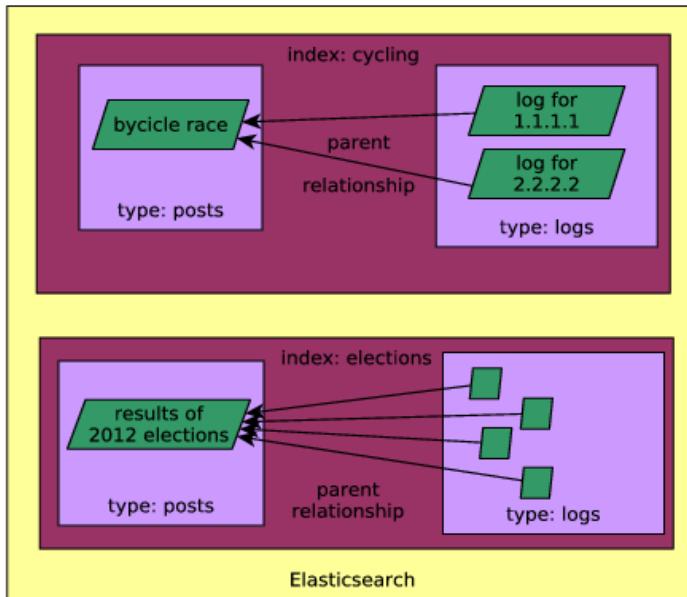


Figure 1.7 Data organized in indices and types

Next, let's look at how this data organization is stored on your physical servers running Elasticsearch. Because it's distributed, your data can be spread on multiple such servers, which form a cluster.

## 1.4 Performance and scaling

Performance is often the main reason you want a search engine. If you have large amounts of data, you need to be able to index it quickly and get results from your searches quickly. If you have many users, you need your search engine to support many concurrent searches.

Elasticsearch provides several tuning options to make indexing operations fast, from sending many operations in the same bulk request, to changing the way your data is stored on disk. Searches can also be tuned for speed; for example, some search types are faster than others, depending on the use case. We show you how to make your indexing and searching faster in chapter 10.

As your data grows, an important feature is the ability to split your data across multiple servers, also known as *sharding*. Fortunately, Elasticsearch is sharded by default, as you'll come to understand, which makes it easy to spread your data across a cluster of multiple instances. You can also store multiple copies of the same data on multiple machines, which is good for availability.

Another feature, which is useful when you have multiple instances, is the ability to change the configuration on the fly through the REST API. The same API that you use for indexing and

searching can be used to modify most configuration options. Those changes can apply to all nodes of a cluster, when you're running a single command.

### SCALING, SHARDING, AND PERFORMANCE

A cluster is made up of one or more nodes, where each node is an Elasticsearch process that typically runs on a separate server. Elasticsearch is clustered by default: when you start your first process, you have a cluster of one node, which you can expand without making any configuration changes.

This configuration works because Elasticsearch divides every index into multiple chunks called *shards*. By default, each index has five shards. In the blog example, when you start your first node and index your first blog posts, your one-node cluster works as illustrated in figure 1.8.

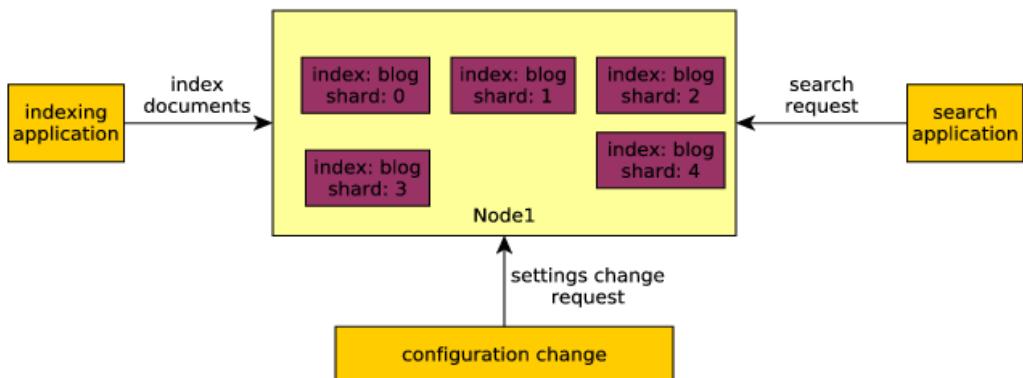


Figure 1.8 One-node cluster with one index divided into five shards

"How does this help?" you might ask. If you stay on one node forever, it doesn't—you might as well have only one shard. But if you add new nodes to your cluster, Elasticsearch can move shards from your initial node to the new ones. Figure 1.9 shows how a three-node cluster might look with your initial five shards spread across the available nodes.

Spreading out the indexing and searching load across multiple machines gives you more performance and capacity. Each node can receive all kinds of requests, so, from the application's point of view, talking to any node is the same as talking to the single entity that is your Elasticsearch cluster.

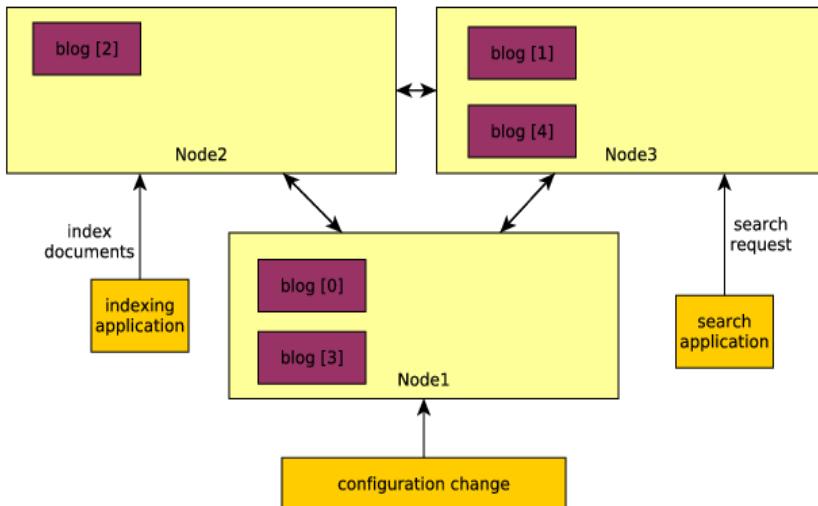


Figure 1.9 Five shards spread across a three-node cluster

Shards can be moved around, allowing you to expand or shrink your cluster at any time. By default, Elasticsearch tries to balance the number of shards across your nodes so the load is evenly spread.

The problem with the setup in figure 1.9 is that if a node goes down, part of your data becomes unavailable. To increase availability, you can create one or more copies (called *replicas*) for each of your initial shards (called *primaries*).

Primaries differ from replicas in that they're the first to receive new documents. Other than that, they're the same: both index the same documents eventually, and both can serve searches. This helps in two ways:

- Because searches also run on replicas, you can increase the number of concurrent searches you serve by adding more nodes and replicas to your cluster.
- Replicas and primary shards index documents in the same way, and replicas can be promoted to primaries. That's exactly what Elasticsearch does in the event that a node hosting a primary shard goes down.

The number of replicas per shard can be changed on the fly. Figure 1.10 shows the cluster from figure 1.9 with one replica added for each shard.

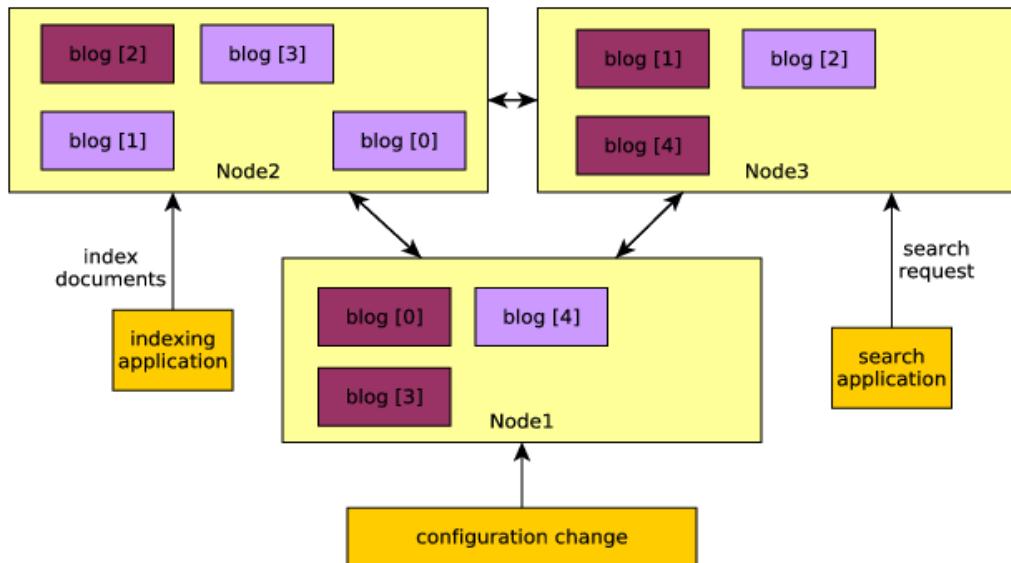


Figure 1.10 The three-node cluster with a set of replicas

With this cluster, if a node goes down, you still have a complete set of shards because of the replicas. From the application's point of view, the cluster continues to work as expected. In the background, Elasticsearch automatically promotes the needed replicas to primaries, and creates additional replicas to get back to the configuration you requested. You'll learn more about this process in chapter 9, which is all about scaling.

At this point, we bet you're eager to install Elasticsearch and start exploring all this functionality. In the next section, you'll install Elasticsearch and its requirements, and then run your first request through its HTTP API.

## 1.5 Getting started with Elasticsearch

To install your own single-server Elasticsearch cluster, you need to have at least Java 6 installed. Once that's in place, you're typically only a download away from getting Elasticsearch ready to start.

### 1.5.1 Installing Java

If you don't have a Java Runtime Environment (JRE) already, you'll have to install it first. Any JRE should work, but typically you install the one from Oracle (<https://www.java.com/en/download/index.jsp>) or the open-source implementation, OpenJDK (<http://download.java.net/openjdk/>).

## Troubleshooting “no Java found” errors

With Elasticsearch, as with other Java applications, it might happen that you’ve downloaded and installed Java, but the application refuses to start, complaining that it can’t find Java.

Elasticsearch’s script looks for Java in two places: the `JAVA_HOME` environment variable and the system path.

To check if it’s in `JAVA_HOME`, use the `env` command on UNIX-like systems and the `set` command on Windows.

To check if it’s in the system path, run the following command:

```
% java -version
```

If it works, then Java is in your path. If it doesn’t, either configure `JAVA_HOME`, or add the Java binary to your path. The Java binary is typically found wherever you installed Java (which should be `JAVA_HOME`), in the `bin` directory.

### 1.5.2 Downloading and starting Elasticsearch

With Java set up, you need to get Elasticsearch and start it. Download the package that best fits your environment. The following package options are available from [www.elasticsearch.org/download/](http://www.elasticsearch.org/download/): Tar, Zip, RPM, and DEB.

#### ANY UNIX-LIKE OPERATING SYSTEM

If you’re running on Linux, Mac, or any other UNIX-like operating system, you can get Elasticsearch from the `tar.gz` package. Then, you can unpack it and start Elasticsearch with the shell script from the archive:

```
% tar zxf elasticsearch-*.tar.gz
% cd elasticsearch-*
% bin/elasticsearch
```

#### MAKING ELASTICSEARCH A SYSTEM SERVICE

In production, you probably want to run Elasticsearch as a service instead of starting it manually. To do that, you need to install the service wrapper, which is available on GitHub. To download it, run the following command:

```
% git clone https://github.com/elasticsearch/elasticsearch-servicewrapper.git
```

Then copy the service/ directory in the bin/ directory of your Elasticsearch installation:

```
% cp -r elasticsearch-servicewrapper/service/ bin/
```

Finally, you’ll have your init script as a symbolic link to the “elasticsearch” service wrapper script:

```
% ln -s `pwd`/bin/service/elasticsearch /etc/init.d/elasticsearch
```

Now you’re done! To restart Elasticsearch, run the following command:

```
/etc/init.d/elasticsearch restart
```

### **HOMEBREW PACKAGE MANAGER FOR OS X**

If you need an easier way to install Elasticsearch on your Mac, you can install Homebrew. Instructions for doing that can be found at <http://brew.sh>. With Homebrew installed, getting Elasticsearch is a matter of running the following command:

```
% brew install elasticsearch
```

Then you start it in a similar way to the tar.gz archive:

```
% elasticsearch
```

### **ZIP PACKAGE**

If you're running on Windows, download the ZIP archive. Unpack it, then run elasticsearch.bat from the bin/ directory, much as you run Elasticsearch on UNIX:

```
% bin\elasticsearch.bat
```

### **RPM PACKAGE**

If you're running on Red Hat Linux, CentOS, SUSE, or anything else that works with RPMs, get the RPM package, and then install it with the following command:

```
% rpm -Uvh elasticsearch-*.*.noarch.rpm
```

And that's it! Elasticsearch is installed and started. Restart it the same way as any other service. For example:

```
% systemctl restart elasticsearch.service
```

### **DEB PACKAGE**

If you're running on Debian, Ubuntu, or anything else that works with DEBs, download the DEB package, and then install it with the following command:

```
% dpkg -i elasticsearch-*.*.deb
```

And again, that's it! If you need to restart it, you can use the init script:

```
% /etc/init.d/elasticsearch restart
```

If you want to see what Elasticsearch is doing, look up the logs in `/var/log/elasticsearch/`. This is the same for the RPM package.

### **1.5.3 Verifying that it works**

Now that you have Elasticsearch installed and started, let's take a look at the logs generated during startup and connect to the REST API for the first time.

#### **EXAMINING THE STARTUP LOGS**

When you first run Elasticsearch, you see a series of log lines telling you what's going on. Let's take a look at some of those lines and what they mean.

The first line typically provides statistics about the node you started:

```
[node] [Basilisk] version[1.1.0], pid[6011], build[77bc5d5/2013-11-06T14:40:44Z]
```

By default, Elasticsearch gives your node a random name, in this case Basilisk, which you can modify from the configuration. You can see details on the particular version you're running, along with the PID of the Java process that started.

```
[plugins] [Basilisk] loaded [], sites []
```

For more information about plugins, see appendix B.

```
[transport] [Basilisk] bound_address {inet[/0.0.0.0:9300]}, publish_address
{inet[/192.168.1.8:9300]}
```

If you use the native Java API instead of the HTTP API, this is the point where you need to connect.

In the next line, a *master node* was elected and it's the node you started named Basilisk:

```
[cluster.service] [Basilisk] new_master [Basilisk][YPHC_vWiQVuSX-
ZIJlMhg][inet[/192.168.1.8:9300]], reason: zen-disco-join
(elected_as_master)
```

We discuss master election in chapter 9, which covers scaling out. The basic idea is that each cluster has a master node, responsible for knowing which nodes are in the cluster and where all the shards are located. Each time the master is unavailable, a new one is elected. In this case, you started the first node in the cluster, so this is your master.

Port 9200 is used for HTTP communication by default. This is where applications using the REST API connect:

```
[http] [Basilisk] bound_address {inet[/0.0.0.0:9200]}, publish_address
{inet[/192.168.1.8:9200]}
```

The next line indicates that your node is now started:

```
[node] [Basilisk] started
```

At this point, you can connect to it and start issuing requests.

Gateway is the component of Elasticsearch responsible for persisting your data to disk so you don't lose it if the node goes down:

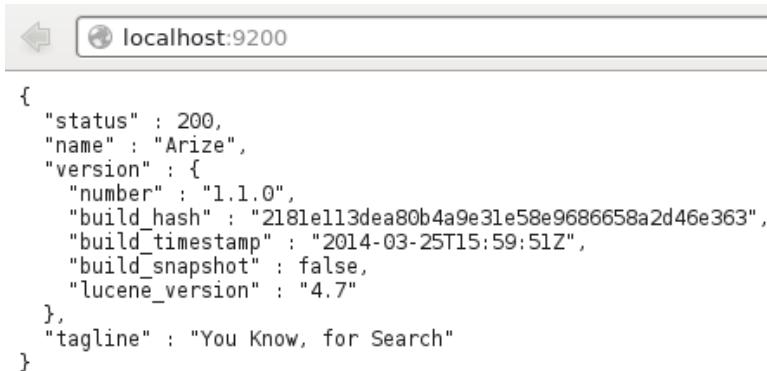
```
[gateway] [Basilisk] recovered [0] indices into cluster_state
```

When you start your node, Gateway looks on the disk to see if any data is saved so it can restore it. In this case, there's no index to restore.

Much of the information we've looked at in these log lines is configurable from the node name to the Gateway settings. We talk about configuration options, and the concepts around them, as the book progresses. You can expect such configuration options to appear in part 3, which is all about performance and administration. Until then, you won't need to configure much because the default values are developer-friendly.

## USING THE REST API

The easiest way to connect to the REST API is by pointing your browser to <http://localhost:9200>. If you didn't install Elasticsearch on your local machine, replace localhost with the IP address of the remote machine. By default, Elasticsearch listens for incoming HTTP requests on port 9200 of all interfaces. If the request works, you should get a JSON reply, as shown in figure 1.11.



```
{
  "status" : 200,
  "name" : "Arize",
  "version" : {
    "number" : "1.1.0",
    "build_hash" : "218lell3dea80b4a9e31e58e9686658a2d46e363",
    "build_timestamp" : "2014-03-25T15:59:51Z",
    "build_snapshot" : false,
    "lucene_version" : "4.7"
  },
  "tagline" : "You Know, for Search"
}
```

Figure 1.11 Checking out Elasticsearch from your browser

Let's look at each field of the JSON and see what it's about:

- *Status*—Displays the HTTP error code that resulted from the request; 200 means OK.
- *Name*—Displays the name of your Elasticsearch instance, which you can also see from the logs.
- *Version*—Contains an object that demonstrates the hierarchical nature of documents, which we discussed previously. The object includes a number of fields that tell you about the installed version: the version number, its hash, the time it was built, whether it's an official release or a build from a snapshot of a branch from GitHub, and the underlying Lucene version.
- *Tagline*—Displays the first tagline of Elasticsearch, "You Know, for Search."

## 1.6 Summary

Now that you're all set up, let's review what we explored in this chapter.

- Elasticsearch is an open-source, distributed search engine built on top of Apache Lucene.
- The typical use case for Elasticsearch is to index large amounts of data so you can run full-text searches and real-time statistics on it.
- Elasticsearch provides features that go well beyond full-text search, for example, you can tune the relevance of your searches and offer search suggestions.
- To get started, download the package, unpack it if necessary, and run the Elasticsearch

start script.

- For indexing and searching data, as well as for managing your cluster's settings, use the JSON over HTTP API and get back a JSON reply.
- You can also look at Elasticsearch as a NoSQL data store with real-time search and analytics capabilities. It's document-oriented and scalable by default.
- Elasticsearch automatically divides data into shards, which get balanced across the available servers in your cluster. This makes it easy to add and remove servers on the fly. Shards are also replicated, making your cluster fault-tolerant.

In chapter chapter 2, you'll get to know Elasticsearch even better by indexing and searching real data.

# 2

## *Diving into the functionality*

### ***This chapter covers***

- Defining documents, mapping types, and indices
- Understanding Elasticsearch nodes and primary and replica shards
- Indexing documents with curl and a data set
- Searching and retrieving data
- Setting Elasticsearch configuration options
- Working with multiple nodes

Now you know what kind of search engine Elasticsearch is, and you've seen some of its main features in chapter 1. Let's switch to the practical side and see how it does what it's good at. Imagine you're tasked with creating a way to search through millions of documents, like a website that allows people to build common interest groups and get together. You need to implement this in a fault-tolerant way, and you need your setup to be able to accommodate more data and more concurrent searches, as your get-together site becomes more successful.

In this chapter, we'll show you how to deal with such a scenario, by explaining how Elasticsearch data is organized. Then, you'll get practical and start indexing and searching some real data for a get-together website using the code samples provided for this chapter. We'll revisit this data and example throughout the book, as we explore working with data in Elasticsearch. All operations will be done using *cURL*, a nice little command-line tool for HTTP requests. Later, you can translate what cURL does into your preferred programming language if you need to. Toward the end of the chapter, you'll make some configuration changes and start new instances of Elasticsearch, so you can experiment with a cluster of multiple nodes.

We'll get started with data organization. To understand how data is organized in Elasticsearch, we'll look at it from two angles:

- *Logical layout*—What your search application needs to be aware of

The unit you'll use for indexing and searching is a document, and you can think of it like a row in a relational database. Documents are grouped into *mapping types*, sometimes called types, which contain documents in a similar way to how tables contain rows.

Finally, one or multiple types live in an *index*, the biggest container, similar to a database in the SQL world.

- *Physical layout*—How Elasticsearch handles your data in the background

Elasticsearch divides each index into *shards*, which can migrate between servers that make up a cluster. Typically, applications don't care about this because they work with Elasticsearch in much the same way, whether it's one or more servers. But when you're administering the cluster, you care because the way you configure the physical layout determines its performance, scalability, and availability.

Figure 2.1 illustrates the two perspectives:

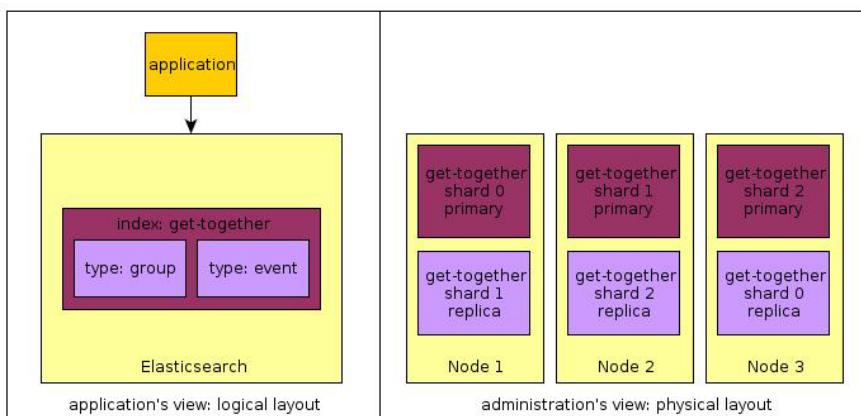
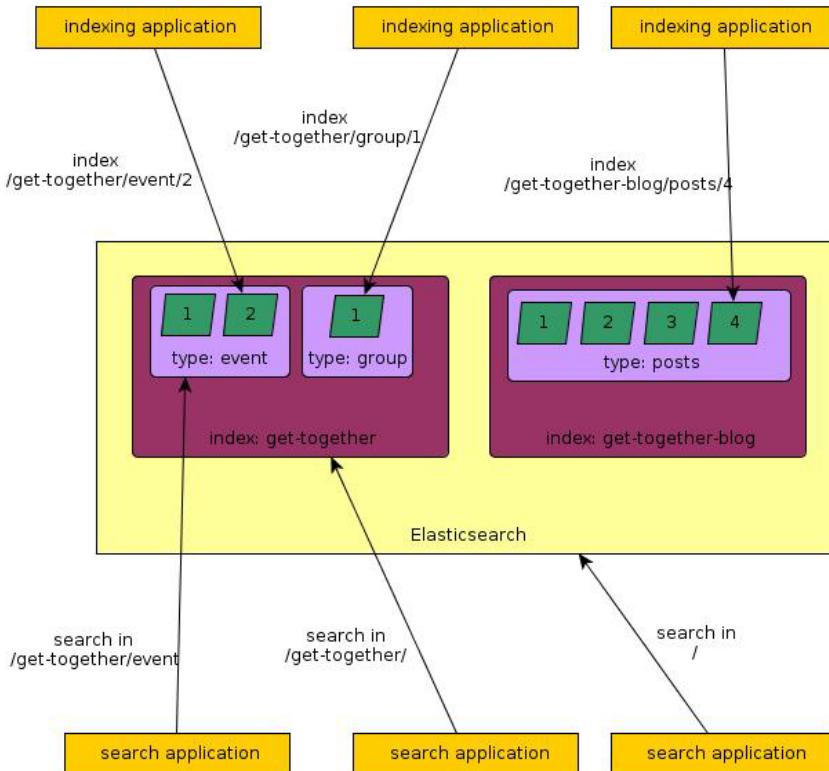


Figure 2.1 An Elasticsearch cluster from the application's and administrator's points of view

## 2.1 Understanding the logical layout: documents, types, and indices

When you index a document in Elasticsearch, you put it in a type that belongs to a specific index. You can see this idea in figure 2.2, where the *get-together* index contains two types: *event* and *group*. Those types contain documents, such as the one labeled "1." The label "1" is that document's ID. The index-type-ID combination uniquely identifies a document in your Elasticsearch setup. When you search, you can look for documents in that specific type, of that specific index, or you can search across multiple types or even multiple indices.



#A (Add arrow to the left of /get-together/event/1) Index name + type name + document id = uniquely identified document

Figure 2.2 Logical layout of data in Elasticsearch: how an application sees data

At this point you might be asking: what exactly is a document, type, and an index? That's exactly what we're going to discuss next.

### 2.1.1 Documents

We said in chapter 1 that Elasticsearch is *document-oriented*, where a document is that smallest unit of data you index or search for. There are a few important properties of a document in Elasticsearch:

- It's *self-contained*. A document contains both the fields (name) and their values (Elasticsearch Denver).
- It can be *hierarchical*. Think of this as documents within documents. A value of a field can be simple, like the value of the location field can be a string. It can also contain other fields and values. For example, the location field might contain both a city and a street-address within it.

- It has a *flexible structure*. Your documents don't depend on a predefined schema. For example, not all events need description values, so that field can be omitted altogether. But it might bring new fields, like the latitude and longitude of the location.

A document is normally be a JSON representation of your data. As we discussed in chapter 1, JSON over HTTP is the most widely used way to communicate with Elasticsearch, and it's the method we use throughout the book. For example, an event in your get-together site can be represented in the following document:

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
  "location": "Denver, Colorado, USA"
}
```

**NOTE** Throughout the book, we'll use different colors for the field names and values of the JSON documents, to make them easier to read. Field names are *darker/blue*, and values are in *lighter/red*.

You can also imagine a table with three columns: name, organizer, and location. The document would be a row containing the values. But there are some differences that make this comparison inexact.

The main difference between documents like this and rows in a table is that a single document contains the names of all the fields it has a value for. So, although it uses up more space in its raw form, you can easily understand which value belongs to which field by looking at one document.

Another difference is that, unlike rows, documents can be hierarchical. For example, the location can contain a name and a geo-location:

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
  "location": {
    "name": "Denver, Colorado, USA",
    "geolocation": "39.7392, -104.9847"
  }
}
```

A single document can also contain arrays of values. For example:

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
  "members": ["Lee", "Mike"]
}
```

Finally, documents in Elasticsearch are said to be *schema-free*, in the sense that not all your documents need to have the same fields, so they're not bound to the same schema. For example, you can omit the location altogether, in case the organizer needs to be called before every gathering:

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
  "members": [ "Lee", "Mike" ]
}
```

Although you can add or omit fields at will, the type of each fields matters: some are strings, some are integers, and so on. Because of that, Elasticsearch keeps a mapping of all your fields and their types, and other settings. This mapping is specific to every type of every index. That's why types are also called mapping types in Elasticsearch terminology.

### **2.1.2 Mapping types**

*Mapping types* are logical containers for documents, similar to how tables are containers for rows. They're often called simply *types*, because you'd put different types of documents in different mapping types.

For example, you can have a type that defines the get-together groups, and a type for the events when people gather. These could be different types of documents because they'd have different structures.

We call them *mapping types* because they're typically used as containers for different types of documents—documents with different structures. The definition of fields in each type is called a *mapping*. For example, name would be mapped as a string, but the geolocation field under location would be mapped as a special geo\_point type. (We explore working with geospatial data in appendix A). Each kind of fields is handled differently. For example, you search for a word in the name field, and you search for groups that are located near where you live.

**TIP** Whenever you're searching in a field that isn't at the root of your JSON document, you must specify its path. For example, the geolocation field under location is referred to as location.geolocation.

You may ask yourself: If Elasticsearch is schema-free, why does each document belong to a type, and each type contains a mapping, which is like a schema?

We say *schema-free* because documents are not bound to the schema. They aren't required to contain all the fields defined in your mapping and may come up with new fields. How does it work? First, the mapping contains all the fields of all the documents indexed so far in that type. But not all documents have to have all fields. Also, if a new document gets indexed with a field that's not already in the mapping, Elasticsearch automatically adds that new field to your mapping. To add that field, it has to decide what type it is, so it guesses it. For example, if the value is 7, it assumes it's a long type.

This autodetection of new fields has its downside because Elasticsearch might not guess right. For example, after indexing 7, you might want to index 7.5, which will fail because it's a float and not a long. In production, the safe way to go is to define your mapping before indexing data. We talk more about defining mappings in chapter 3.

Mapping types only divide documents logically. Physically, documents from the same index are written to disk regardless of the mapping type they belong to.

### **2.1.3 Indices**

Indices are containers for mapping types. An Elasticsearch *index* is an independent chunk of documents, much like a database is in the relational world: each index is stored on the disk in the same set of files; it stores all the fields from all the mapping types in there, and it has its own settings.

For example, each index has a setting called `refresh_interval`, which defines the interval at which newly indexed documents are made available for searches. This *refresh* operation is quite expensive in terms of performance, and this is why it's done occasionally—by default, every second—instead of doing it after each indexed document. If you've read that Elasticsearch is *near-real-time*, this refresh process is what it refers to. Even though you can refresh after each new document, it's not worth it for many use cases. We talk more about indexing performance in chapter 10.

A nice feature of Elasticsearch is that you can search across indices like you can search across mapping types. This gives you flexibility in terms of how you can organize your documents. For example, you can put your get-together events and the blog posts about them in different indices or in different types of the same index. Because Elasticsearch is schema-free, you can even put them in the same type. You can organize your documents in various ways, but some ways are more efficient than others, depending on your use case. We talk more about how to organize your data for efficient indexing in chapter 4.

#### **Elasticsearch index vs. Lucene index**

You'll see the word "index" used frequently as we discuss Elasticsearch; here's how the terminology works.

An Elasticsearch *index* is broken down into chunks: *shards*. A shard is a Lucene index. So an Elasticsearch index is made up of multiple Lucene indices. This makes sense because Elasticsearch uses Apache Lucene as its core library to index your data and search through it.

Throughout this book, whenever you see the word "index" by itself, it refers an Elasticsearch index. If we're digging into the details of what's in a shard, we'll specifically use the term "Lucene index."

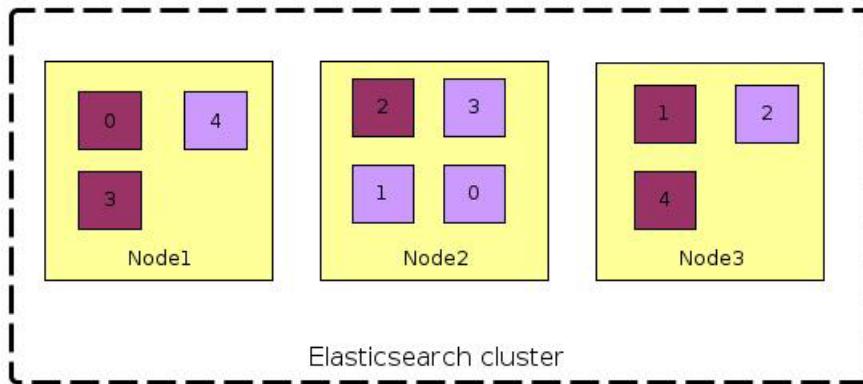
Another index-specific setting is the number of shards. You saw in chapter 1 that an index can be made up of one or more chunks called shards. This is good for scalability: you can run Elasticsearch on multiple servers and have shards of the same index live on multiple servers.

From a search or an indexing application's point of view, the way you shard your index doesn't matter in terms of how data is organized logically. It's about how your data is physically laid out, and we'll look at that next.

## 2.2 Understanding the physical layout: nodes and shards

Understanding how data is physically laid out boils down to understanding how Elasticsearch scales. In this section, we'll explain how scaling works by looking at how multiple nodes work together in a cluster, how data is divided in shards and replicated, and how indexing and searching works with multiple shards and replicas.

To understand the big picture, let's review what happens when an Elasticsearch index is created. By default, each index is made up of five primary shards, each with one replica, for a total of ten shards. Each shard is a part of your index, so roughly 20% of your data is found on each shard. Those five primary shards are replicated into five more shards, as illustrated in figure 2.3.



#A Arrow from the left to Node1: A node is an instance of Elasticsearch  
#B Arrow from the top to primary shard 2 of Node 2: A primary shard is a chunk of your index  
#C Arrow from the right to replica 2 of Node 3: A replica is a copy of a primary shard

Figure 2.3 A three-node cluster with an index divided into five shards, with one replica per shard

As we'll explore next, replicas are good for reliability and search performance. Technically, a shard is a directory of files where Lucene stores the data for your index. A shard is also the smallest unit that Elasticsearch moves from node to node.

### 2.2.1 Creating a cluster of one or more nodes

A *node* is an instance of Elasticsearch. When you start Elasticsearch on your server, you have a node. If you start Elasticsearch on another server, it's another node. You can even have more nodes on the same server, by starting multiple Elasticsearch processes.

Multiple nodes can join the same *cluster*. With a cluster of multiple nodes, the same data can be spread across multiple servers. This helps performance because Elasticsearch has more resources to work with. It also helps reliability: if you have at least one replica per shard, any node can disappear, and Elasticsearch will still serve you all the data. For an application that's

using Elasticsearch, having one or more nodes in a cluster is transparent. Applications typically care only about documents, types and indices, not shards and nodes.

### Nodes in a cluster are like dancers in a show

You might prefer skaters, or actors, or any other type of performer. Either way, the same way you can have one or more dancers in a show, you can have one or more nodes in a cluster.

Similar to shows, the minute you add a second node in a cluster, you need to make sure that they can communicate to each other. Only then can they share the work. The same way a show remains a single show, designed to entertain the viewer, a cluster remains a cluster, designed to serve the same piece of data.

With Elasticsearch, you typically start with one node so you can test your application and add more nodes as your data grows and you need more performance. We take a deeper look at how you can add nodes to your cluster in chapter 9.

### WHAT HAPPENS WHEN YOU INDEX A DOCUMENT?

By default, when you index a document, it's first sent to one of the primary shards, which is chosen based on a hash of the document's ID. Then, the document is sent to be indexed in all of that primary shard's replicas (see left side of figure 2.4). This keeps replicas in sync with data from the primary shards. Being in sync allows replicas to serve searches and to be automatically promoted to primary shards in case the original primary becomes unavailable.

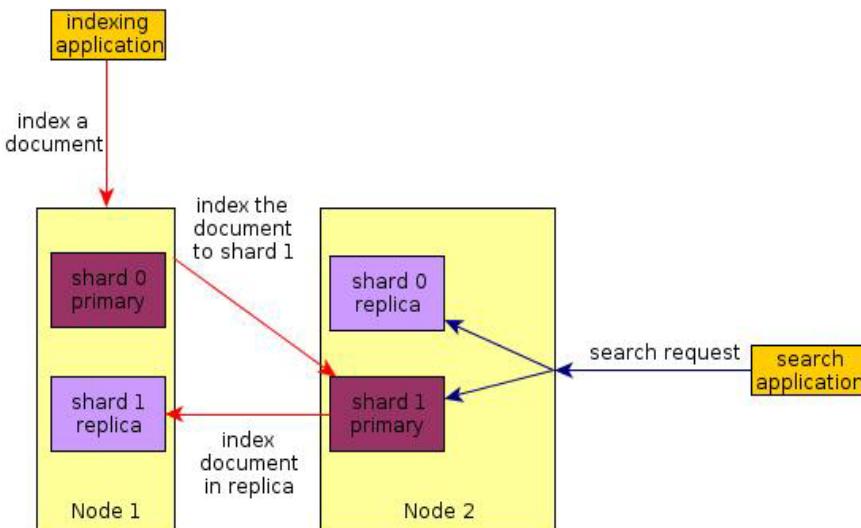


Figure 2.4 Documents are indexed to random primary shards and their replicas. Searches run on complete sets of shards, regardless of their status as primaries or replicas.

## WHAT HAPPENS WHEN YOU SEARCH AN INDEX?

When you search an index, Elasticsearch has to look in a complete set of shards for that index (see right side of figure 2.4). Those shards can be either primary or replicas because primary and replica shards typically contain the same documents. Elasticsearch distributes the search load between the primary and replica shards of the index you're searching, making replicas useful for both search performance and fault tolerance.

Next, we'll look at the details of what primary and replica shards are and how they're allocated in an Elasticsearch cluster.

### 2.2.2 Understanding primary and replica shards

Let's start with the smallest unit Elasticsearch deals with, a shard. A *shard* is a Lucene index: a directory of files containing an inverted index. An *inverted index* is a structure that enables Elasticsearch to tell you which document contains a term (a word) without having to look at all the documents.

In Figure 2.5, you can see what sort of information the first primary shard of your get-together index may contain. The shard `get-together0`, as we'll call it from now on, is a Lucene index—an inverted index. By default, it stores the original document's content plus additional information, such as *term dictionary* and *term frequencies*, which helps searching.

The *term dictionary* maps each term to identifiers of documents containing that term (see figure 2.5). When searching, Elasticsearch doesn't have to look through all the documents for that term—it uses this dictionary to quickly identify all the documents that match.

*Term frequencies* give Elasticsearch quick access to the number of appearances of a term in a document. This is important for calculating the relevancy score of results. For example, if you search for "denver", documents that contain "denver" many times are typically more relevant. Elasticsearch gives them a higher score, and they appear higher in the list of results.

get-together0 shard		
Inverted index		
Term	Documents	Frequencies
elasticsearch	id1	1 occurrence: id1->1 time
denver	id1,id3	3 occurrences: id1->1 time, id3->2 times
clojure	id2,id3	5 occurrences: id2->2 times, id3->3 times
data	id2	2 occurrences: id2->2 times

# Arrow from the top to the black title: A shard is a Lucene index

Figure 2.5 Term dictionary and frequencies in a Lucene index

A shard can be either a primary or a replica shard, with *replicas* being exactly that—copies of the primary shard. A replica is used for searching, or it becomes a new primary shard if the original primary shard is lost.

An Elasticsearch index is made up of one or more primary shards and zero or more replica shards. In Figure 2.6, you can see that the Elasticsearch get-together index is made up of six total shards: two primary shards (darker boxes), and two replicas for each shard (lighter boxes) for a total of four replicas.

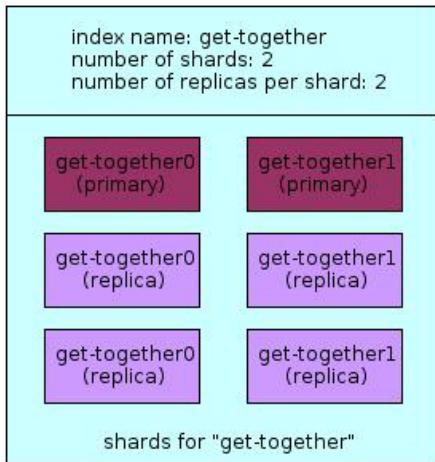


Figure 2.6 Multiple primary and replica shards make up the get-together index

**NOTE** You can change the number of replicas per shard at any time because replicas can always be created or removed. This doesn't apply to the number of primary shards an index is divided into: you have to decide on the number of shards before creating the index. Keep in mind that too few shards limit how much you can scale, and too many shards impact performance. The default setting of five is typically a good start, and we expand the subject in chapter 9, which is all about scaling.

All the shards and replicas you've seen so far are distributed to nodes within an Elasticsearch cluster. Next, we'll look at some details about how Elasticsearch distributes shards and replicas in a cluster having one or more nodes.

### 2.2.3 Distributing shards in a cluster

The simplest Elasticsearch cluster is one having one node: one machine running one Elasticsearch process. When you first installed Elasticsearch in chapter 1 and started it, you created a one-node cluster.

As you add more nodes to the same cluster, the existing shards get balanced between all the nodes. As a result, both indexing and search requests that work with those shards benefit from the extra power of your added nodes. Scaling this way (by adding nodes to a cluster) is called *horizontal scaling*; you add more nodes, and requests are then distributed so they all share the work.

The alternative to horizontal scaling is to scale vertically; you add more resources to your Elasticsearch node, perhaps by dedicating more processors to it if it's a virtual machine, or adding RAM to a physical machine. Although vertical scaling helps performance almost every time, it's not always possible or cost-effective. Using shards enables you to scale horizontally.

Suppose you want to scale your get-together index, which currently has two shards and no replicas. The first option is to scale vertically by upgrading the node: for example, adding more RAM, more CPUs, faster disks and so on. The second option is to scale horizontally by adding another node and having your data distributed between the two nodes.

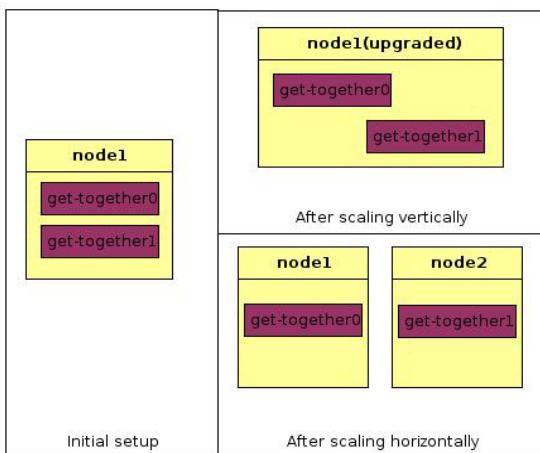


Figure 2.7 To improve performance, scale vertically (top left) or scale horizontally (lower right).

We discuss more about performance in chapter 10. For now, let's see how indexing and searching work across multiple shards and replicas.

#### 2.2.4 Distributed indexing and searching

At this point you might wonder how indexing and searching works with multiple shards spread across multiple nodes.

Let's take indexing, as shown in figure 2.8. The Elasticsearch node that receives your indexing request first selects the shard to index the document to. By default, documents are distributed evenly between shards<sup>5</sup>.

Once the target shard is determined, the current node forwards the document to the node holding that shard. Subsequently, that indexing operation is replayed by all the replicas of that

<sup>5</sup> By default, for each document, the shard is determined by hashing its ID string. Each shard has an equal chunk of the total hash range and receives, in normal conditions, an equal chunk of documents.

shard. The indexing command successfully returns after all the available replicas finish indexing the document.

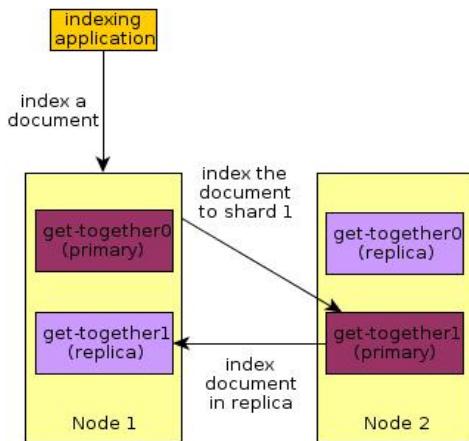


Figure 2.8 Indexing operation is forwarded to the responsible shard, and then to its replicas

With searching, the node that receives the request forwards it to a set of shards containing all your data, it doesn't matter if those shards are primaries or replicas. Elasticsearch uses a round-robin format to forward the request to the cluster's nodes and shards. As shown in figure 2.9, Elasticsearch then gathers results from those shards, aggregates them into a single reply, and forwards the reply back to the client application.

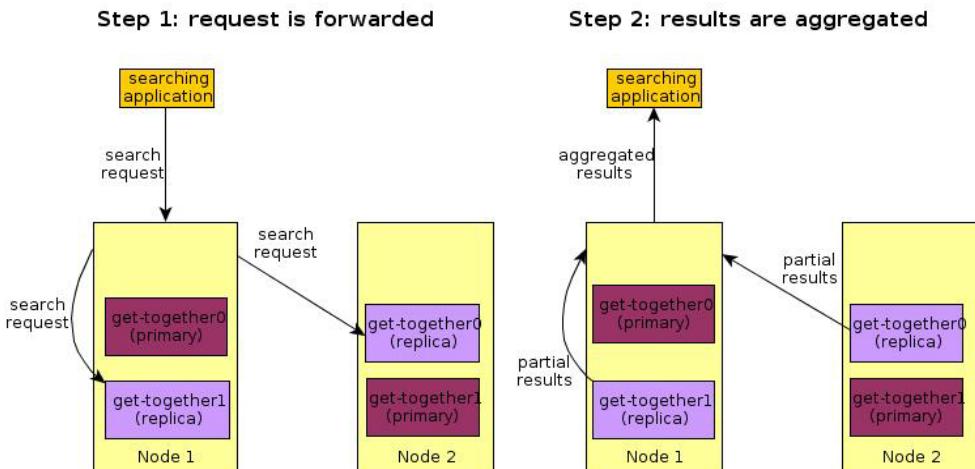


Figure 2.9 Search request is forwarded to shards/replicas containing a complete set of data. Then, results are aggregated and sent back to the client.

Given that requests are sent to primary shards and replicas in round-robin fashion, Elasticsearch assumes that all nodes in your cluster are equally fast. You typically achieve this with identical hardware and software configurations. If that's not the case, you can organize your data or configure your shards to prevent the slower nodes from becoming a bottleneck. We explore more about such options in chapter 9. For now, let's start indexing documents in the single-node Elasticsearch cluster that you started in chapter 1.

## 2.3 Indexing new data

Although chapter 3 gets into the details of indexing, here the goal is to give you a feel for what indexing is about. In this section, we'll discuss the following processes:

- Indexing a document with cURL. To send your first document, you'll use the HTTP API to send a JSON document to be indexed with Elasticsearch. Then, we'll have a look at the JSON reply that comes back.
- Looking at how Elasticsearch automatically creates the index and type to which your document belongs if they don't exist already.
- Running a script to index additional documents. You'll run the code samples for this chapter to quickly index additional files. This way, you have a bunch of documents ready to search through.

You'll index your first document by hand, so let's start by looking at how to issue an HTTP PUT request to a URI. A sample URI shown in figure 2.10 with each part labeled.

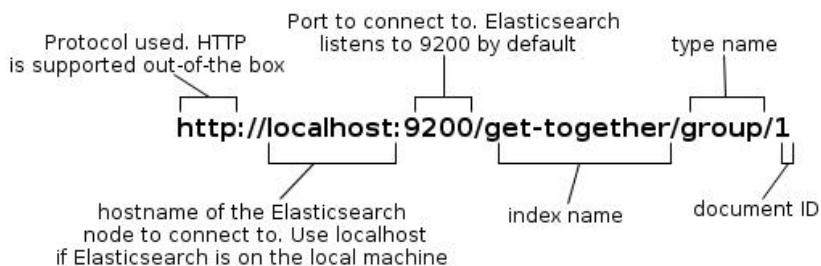


Figure 2.10 URI of a document in Elasticsearch

Let's walk through how you issue the request.

### 2.3.1 Indexing a document with cURL

For most snippets in this book we'll use the curl binary. curl is a command-line tool for transferring data over HTTP. You'll use the `curl` command to make HTTP requests, as it has become a convention to use curl for Elasticsearch code snippets. That's because it's easy to translate a curl example into any programming language. In fact, if you ask for help on the official mailing list for Elasticsearch, it's recommended that you provide a *curl re-creation* of your problem. A curl re-creation is a command or a sequence of curl commands that

reproduces the problem you're experiencing, and anyone who has Elasticsearch installed locally can run it.

## Installing cURL

If you're running on a UNIX-like operating system, like Linux or Mac OS X, then you're likely to have the `curl` command available. If you don't have it already, or if you're on Windows, you can download it from <http://curl.haxx.se/>. You can also install Cygwin, and then select cURL as part of the Cygwin installation, which is the approach we recommend.

Using Cygwin to run `curl` commands on Windows is preferred because you can copy-paste the commands that work on UNIX-like systems. If you choose to stick with the Windows shell, take extra care because single quotes behave differently on Windows. In most situations, you must replace single quotes ('') with double-quotes ("") and escape double quotes with a backslash (\"). For example, a UNIX command like this

```
curl 'http://localhost' -d '{"field": "value"}'  
looks like this on Windows  
curl "http://localhost" -d "{\"field\": \"value\"}"
```

Assuming you can use the `curl` command and you have Elasticsearch installed with the defaults settings on your local machine, you can index your first CD document with the following command:

```
% curl -XPUT 'localhost:9200/get-together/group/1?pretty' -d '{  
  "name": "Elasticsearch Denver",  
  "organizer": "Lee"  
}'
```

You should get the following output:

```
{  
  "ok" : true,  
  "_index" : "get-together",  
  "_type" : "group",  
  "_id" : "1",  
  "_version" : 1,  
  "created" : true  
}
```

The reply tells you whether the request succeeded or failed. If it worked, you should get back the index, type, and ID of the indexed document. In this case, you get the ones you specified, but it's also possible to rely on Elasticsearch to generate IDs, as you'll learn in chapter 3. You also get the version of the document, which begins at 1 and is incremented with each update. You'll learn all about updates in chapter 3.

There are many ways to use `curl` to make HTTP requests; run `man curl` to see all of them. Throughout this book, we use the following `curl` usage conventions:

- The *method*, which is typically `GET`, `PUT` or `POST`, is the argument of the `-X` parameter.

You can add a space between the parameter and its argument, but we don't add one. For example, we use `-XPUT` instead of `-X PUT`. The default method is `GET`, and when we use it, we skip the `-X` parameter altogether.

- In the URI, we skip specifying the protocol; it's always `http`, and curl uses `http` by default when no protocol is specified.
- We put single quotes around the URI because it can contain multiple parameters, and you have to separate the parameters with an ampersand (`&`), which normally sends the process to the background.
- True values of Boolean parameters can be expressed as `pretty=true` or simply `pretty`.

We use the latter. The `pretty` parameter in particular makes the JSON reply look more readable than the default, which is to return the reply all in one line.

- The data that we send through HTTP is typically JSON, and we surround it with single quotes because the JSON itself contains double quotes.

If single quotes are needed in the JSON itself, we first close the single quotes, and then surround the needed single quote with double quotes as shown in this example:

```
'{"name": "Scarlet O'****Hara"}'
```

## Using Elasticsearch from your browser via Head, kopf or Marvel

If you prefer graphical interfaces to the command line, several tools are available.

*Elasticsearch Head*—You can install this tool as an Elasticsearch plugin, a standalone HTTP server, or a web page that you can open from your file system. You can send HTTP requests from there, but Head is most useful as a monitoring tool to show you how shards are distributed in your cluster. You can find Elasticsearch Head at <https://github.com/mobz/elasticsearch-head>.

*Elasticsearch kopf*—Similar to Head in that it's good for both monitoring and sending requests, this tool runs as a web page from your file system or as an Elasticsearch plugin. Both Head and kopf evolve quickly, so any comparison might become obsolete quickly as well. You can find Elasticsearch kopf at <https://github.com/Imenezes/elasticsearch-kopf>.

*Marvel*—This tool is a monitoring solution for Elasticsearch. We discuss more about monitoring in chapter 11, which is all about administering your cluster. For now, the thing to remember is that Marvel also provides a graphical way to send requests to Elasticsearch, and it provides an autocomplete feature, which is a useful learning aid. You can download Marvel at <http://www.elasticsearch.org/overview/marvel/download/>.

### 2.3.2 Creating an index and mapping type

If you installed Elasticsearch and ran the `curl` command to index a document, you might be wondering why it worked given the following factors:

- The index wasn't there before. You didn't issue any command to create an index named get-together.
- The mapping wasn't previously defined. You didn't define any mapping type called group in which to define the fields from your document.

The curl command works because Elasticsearch automatically adds the get-together index for you and also creates a new mapping for the type group. That mapping contains a definition of your field as strings. Elasticsearch handles all this for you by default, which enables you to start indexing without any prior configuration. You can change this default behavior if you need to as you'll explore in chapter 3.

### CREATING AN INDEX MANUALLY

You can always create an index with a PUT request similar to the request used to index a document:

```
% curl -XPUT 'localhost:9200/get-together?pretty'
{
  "acknowledged" : true
}
```

Creating the index itself takes more time than creating a document, so you might want to have the index ready beforehand. Another reason to create indices in advance is if you want to specify different settings than the ones Elasticsearch defaults to, for example, you may want a specific number of shards.

### VIEWING THE MAPPING TYPE

As we mentioned, the mapping is automatically created with your new document, and it automatically detects your name and organizer fields as strings. If you add a new document with yet another new field, Elasticsearch guesses its type, too and appends the new field to the mapping.

To view the current mapping, issue an HTTP GET to the \_mapping endpoint of the type's URL:

```
% curl 'localhost:9200/get-together/group/_mapping?pretty'
{
  "group" : {
    "properties" : {
      "name" : {
        "type" : "string"
      },
      "organizer" : {
        "type" : "string"
      }
    }
  }
}
```

The response contains the following relevant data:

- *Type name*—group
- *Property list*—name and organizer

- *Property options*—The type is string for both properties

We talk more about indices, mappings, and mapping types in chapter 3. For now, let's define a mapping, and then index some documents by running a script from the code samples that came with this book.

### 2.3.3 Indexing documents from the code samples

Before we look at searching through the indexed documents, let's do some more indexing by running `populate.sh` from the code samples for chapter 2.

**NOTE** To download the source code, visit <https://github.com/dakrone/elasticsearch-in-action>, and then follow the instructions from there.

The script first deletes the get-together index you created. Then, it re-creates it and creates the mapping that's defined in `mapping.json`. The mapping file specifies options other than those you've seen so far, and we explore them in the rest of the book, mostly in chapter 3. Finally, the script indexes documents in two types: group and event. There is a parent-child relationship between those types (events belonging to groups), which we explore in chapter 8. For now, ignore this relationship.

Running the `populate.sh` script should look similar to the following listing.

#### Listing 2.1 Indexing data with the `populate.sh` script

```
% ./populate.sh
WARNING, this script will delete the 'get-together' index and re-index all data!
Press Control-C to cancel this operation.
Press [Enter] to continue.
Creating 'get-together' index...
{"acknowledged":true}
Done creating 'get-together' index. #A
Indexing data...
Indexing groups...
{"_index":"get-together","_type":"group","_id":"1","_version":1}
#more replies like this, one for each document
Done indexing groups. #A
Indexing events...
{"_index":"get-together","_type":"event","_id":"10","_version":1}
#more replies like this, one for each document
Done indexing events. #A
{"_shards":{"total":4,"successful":2,"failed":0}}
Done indexing data. #A
```

#A JSON replies, which come from Elasticsearch to acknowledge indexing

After running the script, you'll have a handful of groups that meet and the events planned for those groups. Let's have a look at how you can search through those documents.

## 2.4 Searching for and retrieving data

As you might imagine, there are many options around how to search. After all, searching is what Elasticsearch is for.

**NOTE** We look at the most common ways to search in chapter 4; you learn more about getting relevant results in chapter 6, and you learn all about search performance in chapter 10.

To take a closer look at the pieces that make up a typical search, let's search for groups that contain the word "elasticsearch" but ask only for the name and location fields of the most relevant document. The following listing shows the GET request and response.

### Listing 2.2 Search for “elasticsearch” in groups

```
% curl "localhost:9200/get-together/group/_search?\n#A\nq=elasticsearch\n#B\n&fields=name,location\n#B\n&size=1\n#B\n&pretty"\n#C\n{\n    \"took\" : 2,\n    \"timed_out\" : false,\n    \"_shards\" : {\n        \"total\" : 2,\n        \"successful\" : 2,\n        \"failed\" : 0\n    },\n    \"hits\" : {\n        \"total\" : 2,\n        \"max_score\" : 0.9066504,\n        \"hits\" : [ {\n            \"_index\" : \"get-together\",\n            \"_type\" : \"group\",\n            \"_id\" : \"3\",\n            \"_score\" : 0.9066504,\n            \"fields\" : {\n                \"location\" : \"San Francisco, California, USA\", #D\n                \"name\" : \"Elasticsearch San Francisco\" #D\n            }\n        } ]\n    }\n}
```

#A URL indicates where to search: in the group type of the get-together index

#B URI parameters give the details of the search: find documents containing “elasticsearch”, but return only the name and location fields for the top result

#C Flag to print the JSON reply in a more readable format

#D JSON reply

The following items are the three most important pieces of a search request:

- *Where to search*
- *Contents of the reply*

- How to search

In this section, we'll give you a brief overview of each item (we provide more details in chapter 4), and you'll also see how to retrieve documents by ID.

### **2.4.1 Where to search**

You can tell Elasticsearch to look in a specific type of a specific index, as in listing 2.2, but you can also search in multiple types in the same index, in multiple indices, or in all indices.

To search in multiple types, use a comma-separated list. For example, to search in both group and event types, run a command like this:

```
% curl "localhost:9200/get-together/group,event/_search\"  
?q=elasticsearch&pretty"
```

You can also search in all types of an index by sending your request to the \_search endpoint of the index's URL:

```
% curl 'localhost:9200/get-together/_search?q=sample&pretty'
```

Similar to types, to search in multiple indices, separate them with a comma:

```
% curl "localhost:9200/get-together,other-index/_search\"  
?q=elasticsearch&pretty"
```

To search in all indices, omit the index name altogether:

```
% curl 'localhost:9200/_search?q=elasticsearch&pretty'
```

**TIP** If you need to search in all indices, you can also use an alias called \_all as the index name. This comes in handy when you need to search in a single type across all indices as in this example: [http://localhost:9200/\\_all/event/](http://localhost:9200/_all/event/).

This flexibility regarding where to search allows you to organize data in multiple indices and types, depending on what makes sense for your use case. For example, log events are often organized in time-based indices, such as “logs-2013-06-03”, “logs-2013-06-04”, and so on. Such a design implies that today's index is hot: all new events go here, and most of the searches are in recent data. The hot index contains only a fraction of all your data, making it easier to handle and faster. And you can still search in older data or in all data if you need to. You'll learn more about such design patterns in part 3, which is all about performance and administration.

### **2.4.2 Contents of the reply**

In addition to the documents that match your search criteria, the reply of a search contains information that's useful for checking the performance of your search or the relevance of the results.

You might have some questions about listing 2.2 regarding what the reply from Elasticsearch contains. What's the score about? What happens if not all shards are available? Let's look at each part of the reply shown the following listing.

### Listing 2.3 Search reply returning two fields of a single resulting document

```
{
  "took" : 2,                                #A
  "timed_out" : false,                         #A
  "_shards" : {
    "total" : 2,                                #B
    "successful" : 2,                            #B
    "failed" : 0                                #B
  },
  "hits" : {
    "total" : 2,                                #C
    "max_score" : 0.9066504,                     #C
    "hits" : [ {
      "_index" : "get-together",                 #D
      "_type" : "group",                         #D
      "_id" : "3",                               #D
      "_score" : 0.9066504,                      #D
      "fields" : {
        "location" : "San Francisco, California, USA",   #D
        "name" : "Elasticsearch San Francisco"           #D
      }
    } ]
  }
}
```

#A How long your request took and if it timed out

#B How many shards were queried

#C Statistics on all documents that matched

#D The results array

As you can see, the JSON reply from Elasticsearch includes information on time, shards, hits statistics, and the documents you asked for. We'll look at each of these in turn.

#### TIME

The first items of a reply look something like this:

```
"took" : 2,
"timed_out" : false,
```

The took field tells you how long Elasticsearch needed to process your request. The time is in milliseconds. The timed\_out field indicates whether your search timed out. By default, searches never time out, but you can specify a limit via the timeout parameter. For example, the following search times out after three seconds:

```
% curl "localhost:9200/get-together/group/_search\
?q=elasticsearch\
&pretty\
&timeout=3"
```

If a search times out, the value of timed\_out is true, and you get only results that were gathered until the search timed out.

## SHARDS

The next bit of the response is information about shards involved in the search:

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

This might look natural to you because you searched in one index, which by default has five shards. All shards replied, so successful is 5, which leaves failed with 0.

You might wonder what happens when a node goes down and a shard can't reply to a search request. Take a look at figure 2.11, which shows a cluster of three nodes, each with only one shard and no replicas. If one node goes down, some data would be missing. In this case, Elasticsearch gives you the results from shards that are up and reports the number of shards unavailable for search in the failed field.

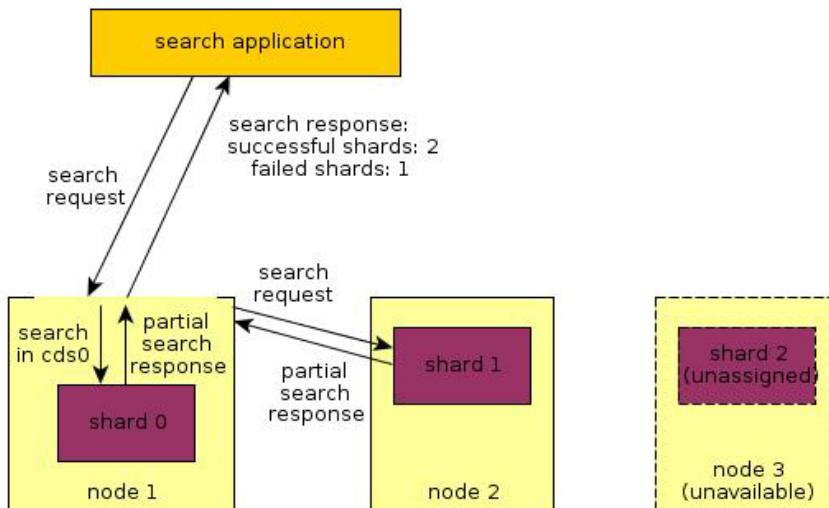


Figure 2.11 Partial results can be returned from shards that are still available

## HITS STATISTICS

The last element of the reply is called hits and is quite lengthy because it contains an array of the matching documents. But before that array, it contains a couple of statistics:

```
"total" : 2,
"max_score" : 0.90178301
```

In total, you see the total number of matching documents, and in max\_score, you see the maximum score of those matching documents.

**DEFINITION** The score of a document returned by a search is the measure of how relevant that document is for the given search criteria. By default, the score is calculated with the tf-idf (term frequency-inverse document frequency) algorithm. *Term frequency* means for each term (word) you search, the document's score is increased if it has more occurrences of that term. *Inverse document frequency* means the score is increased more if the term is rare across all documents because it's considered more relevant. If the term occurs often in other documents, it's probably a common term, thus less relevant. We'll show you how to make your searches more relevant in chapter 6.

The total number of documents may not match the number of documents you see in the reply. By default, Elasticsearch limits the number of results to 10, so if you can have more than 10 results, look at the value of total for the precise number of documents that match your search criteria. As you saw previously, to change the number of results returned, use the size parameter.

## RESULTING DOCUMENTS

The array of hits is usually the most interesting information in a reply:

```
"hits" : [ {
  "_index" : "get-together",
  "_type" : "group",
  "_id" : "3",
  "_score" : 0.9066504,
  "fields" : {
    "location" : "San Francisco, California, USA",
    "name" : "Elasticsearch San Francisco"
  }
}]
```

Each matching document is shown with the index and type it belongs to, its ID, and its score. The values of the fields you specified in your search query are also shown. In listing 2.2, you used fields=name,location. If you don't specify which fields you want, the \_source field is shown. Like \_all, \_source is a special field, where, by default, Elasticsearch stores the original JSON document. You can configure what gets stored in the source, and we explore that in chapter 3.

### 2.4.3 How to search

So far, you've searched through what's called a *URI request*, so named because all your search options go into the URI. This is good for simple searches you run on the command line, but in production, a URI request can get lengthy and be hard to maintain.

Another option is to put your query in the data part of your request. Elasticsearch allows you to specify the search criteria in JSON format, which is much easier to read and write,

particularly for complex searches. For example, to search for all groups that are about Elasticsearch, you could do this:

```
% curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "query": {
    "query_string": {
      "query": "elasticsearch"
    }
  }
}'
```

In plain English, this translates to “run a query of type query\_string, where the string is elasticsearch.” It might seem like too much boilerplate to type in elasticsearch, but this is because JSON provides many more options than a URI request. As you’ll see in chapter 4, using a JSON query makes sense when you start to combine different types of queries: squeezing all those options in a URI would be more difficult to handle. Let’s explore each field.

### SETTING QUERY STRING OPTIONS

At the last level of the JSON request, you have `"query": "elasticsearch"`, and you might think the `"query"` part is redundant because we already know it’s a query. But a query\_string provides more options than the string itself.

For example, if you search for “elasticsearch san francisco”, Elasticsearch looks in the `_all` field by default. If you wanted to look in the group’s name instead, you’d specify

```
"default_field": "name"
```

Also by default, Elasticsearch returns documents matching any of the specified words. If you wanted to match all the words, you’d specify

```
"default_operator": "AND"
```

The revised query looks like this:

```
% curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "query": {
    "query_string": {
      "query": "elasticsearch san francisco",
      "default_field": "name",
      "default_operator": "AND",
    }
  }
}'
```

Another way to achieve the same results is to specify the field and the operator in the query string itself:

```
"query": "name:elasticsearch AND name:san AND name:francisco"
```

The query string is a powerful tool to specify your search criteria. Elasticsearch parses the string to understand the terms you're looking for and your other options, such as fields and operators, and then runs the query. This functionality is inherited from Lucene.<sup>6</sup>

### CHOOSING THE RIGHT QUERY TYPE

If the query\_string query type looks intimidating, the good news is there are many other types of queries, most of which we cover in chapter 4. For example, if you're looking only for the term "elasticsearch" in the name field, a term query would be faster and more straightforward:

```
% curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "query": {
    "term": {
      "name": "elasticsearch"
    }
  }
}'
```

### USING FILTERS

So far, all the searches you've seen have been queries. Queries give you back results, and each result has a score. If you're not interested in the score, you can run a filter instead. Filters care only whether a result matches the search or not, and as a result, they're faster and easier to cache than their query counterparts. For example, the following filter looks for the term "elasticsearch" in the name of groups:

```
% curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "filter": {
    "term": {
      "name": "elasticsearch"
    }
  }
}'
```

The results are the same as the ones you get with the equivalent term query, but filter results aren't sorted by score (because the score is 1.0 for all results). Like queries, we cover filters in chapter 4.

### APPLYING AGGREGATIONS

In addition to queries and filters, you can do all sorts of statistics through aggregations. We look at aggregations in chapter 8, but let's look at a simple example here.

Suppose a user is visiting your get-together website and wants to explore the kinds of groups that are available. You might want to show who the group organizers are. For example,

<sup>6</sup> If you want to find out more about the query string syntax, visit [http://lucene.apache.org/core/4\\_4\\_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package\\_description](http://lucene.apache.org/core/4_4_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package_description).

if “Lee” comes up in the results as the organizer of seven meetings, a user who knows Lee might click his name to filter only those seven meetings.

To return people who are group organizers, you can use a *terms* aggregation. This shows counters for each term that appears in the field you specify—in this case, organizer. The aggregation might look like this:

```
% curl localhost:9200/get-together/group/_search?pretty -d '{
  "aggs" : {
    "genders" : {
      "terms" : { "field" : "organizer" }
    }
  }
}'
```

In plain English this request translates to “give me an aggregation named *tags*, which is of type *terms* and is looking at the *organizer* field.” The following results display at the bottom of the reply:

```
"aggregations" : {
  "genders" : {
    "buckets" : [ {
      "key" : "lee",
      "doc_count" : 2
    }, {
      "key" : "andy",
      "doc_count" : 1
    }, {
      "key" : "daniel",
      "doc_count" : 1
    }, {
      "key" : "mik",
      "doc_count" : 1
    }, {
      "key" : "tyler",
      "doc_count" : 1
    } ]
  }
}
```

The results show you that, out of the six total terms, “lee” appears two times, “andy” one time, and so on. We have two groups organized by Lee. You could then search for the groups for which Lee is the organizer to narrow down your results.

Aggregations are useful when you can’t search for what you need because you don’t know what that is. What kind of groups are available? Are there any events hosted near where I live? You can use aggregations to drill down in the available data and see real-time statistics.

Other times, you have the opposite scenario. You know exactly what you need, and you don’t want to run a search at all. That’s when it’s useful to retrieve a document by ID.

#### **2.4.4 Getting documents by ID**

To retrieve a specific document, you must know the index and type it belongs to and its ID. You then issue an HTTP GET request to that document’s URI:

```
% curl 'localhost:9200/get-together/group/1?pretty'
{
  "_index" : "get-together",
  "_type" : "group",
  "_id" : "1",
  "_version" : 1,
  "exists" : true, "_source" : {
    "name": "Denver Clojure",
    "organizer": ["Daniel", "Lee"],
    "description": "Group of Clojure enthusiasts from Denver who want to hack on
      code together and learn more about Clojure",
    "created_on": "2012-06-15",
    "tags": ["clojure", "denver", "functional programming", "jvm", "java"],
    "members": ["Lee", "Daniel", "Mike"],
    "location": "Denver, Colorado, USA"
  }
}
```

The reply contains the index, type, and ID you specified. If the document exists, you'll see that the exists field is true, in addition to its version and its source. If the document doesn't exist, exists is false:

```
% curl 'localhost:9200/get-together/group/doesnt-exist?pretty'
{
  "_index" : "get-together",
  "_type" : "group",
  "_id" : "doesnt-exist",
  "exists" : false
}
```

As you might expect, getting documents by ID is much faster and less expensive in terms of resources than searching. It's also done in real time: as soon as an indexing operation is finished, the new document can be fetched through this GET API.

Remember that newly indexed documents only appear in searches after a refresh. When you index something, you must wait for the automatic refresh to apply—which occurs once per second by default. Alternatively, you can refresh the index manually, by sending an HTTP POST request to the URL of that index:

```
% curl -XPOST 'localhost:9200/get-together/_refresh?pretty'
{
  "ok" : true,
  "_shards" : {
    "total" : 4,
    "successful" : 2,
    "failed" : 0
  }
}
```

Unlike searching, when you get documents by ID, you don't need to wait for a refresh unless you set `action.get.realtime` to `false` in your `elasticsearch.yml` configuration file. In fact, let's take a closer look at how to configure Elasticsearch.

## 2.5 Configuring Elasticsearch

One of Elasticsearch's strong points is that it has developer-friendly defaults, making it easy to get started. As you saw in the previous section, you can do indexing and searching on your own test server without making any configuration changes. Elasticsearch automatically creates an index for you and detects the type of new fields in your documents.

Elasticsearch also scales easily and efficiently, which is another important feature when you're dealing with large amounts of data or requests. In the final section of this chapter, you'll start a second Elasticsearch instance, in addition to the one you already started in chapter 1, and let them form a cluster. This way, you'll see how Elasticsearch scales out and distributes your data throughout the cluster.

Although scaling out can be done without any configuration changes, we'll tweak a few knobs in this section to avoid surprises later when you add a second node. We'll make the following changes in three different configuration files:

- *Specify a cluster name in elasticsearch.yml*—This is the main configuration file where Elasticsearch-specific options go.
- Edit logging options in logging.yml—The logging configuration file is for logging options of log4j, the library that Elasticsearch uses for logging.
- *Adjust memory settings* in environment variables or elasticsearch.in.sh—This file is for configuring the Java virtual machine (JVM) that powers Elasticsearch.

There are a few others, and we'll point them out as they appear, but those listed are the most commonly used. Let's walk through each of these configuration changes.

### 2.5.1 Specifying a cluster name in elasticsearch.yml

The main configuration file of Elasticsearch can be found in the config/ directory of the unpacked tar.gz or zip archive.

**TIP** The file is in /etc/elasticsearch/ if you installed it from the RPM or DEB package.

Like the REST API, the configuration can be in JSON or YAML. Unlike the REST API, the most popular format is the YAML. It's easier to read and use, and all the configuration samples in this book are based on elasticsearch.yml.

By default, new nodes discover existing clusters via multicast—by sending a ping to all hosts listening on a specific multicast address. If a cluster is discovered, the new node joins it only if it has the same cluster name. Let's customize the cluster name to prevent instances of the default configuration from joining our cluster. To change the cluster name, add the following line to your .yml file:

```
cluster.name: elasticsearch-in-action
```

After you update the file, stop Elasticsearch by pressing Control-C, and then start it again with the following command:

bin/elasticsearch

## 2.5.2 Specifying verbose logging via `logging.yml`

When something goes wrong, application logs are the first place to look for clues. They're also useful when you just want to see what's going on. If you need to look in Elasticsearch's logs, the default location is the `logs/` directory under the path where you unpacked the zip/tar.gz archive.

**TIP** If you installed it from the RPM or DEB package, the default path is `/var/log/elasticsearch/`.

Elasticsearch log entries are organized in three types of files:

- *Main log (cluster-name.log)*— Here you can find general information about what happens when Elasticsearch is running, for example, whether a query failed or a new node joined the cluster.
- *Slow-search log (cluster-name\_index\_search\_slowlog.log)*— This is where Elasticsearch logs when a query runs too slow. By default, if a query takes more than half a second, it logs an entry here.
- *Index-slow log (cluster-name\_index\_indexing\_slowlog.log)*—This is similar to the slow-search log, but, by default, it writes an entry if an indexing operation takes more than half a second.

To change logging options, you edit the `logging.yml` file, which is located in the same place as `elasticsearch.yml`. Elasticsearch uses log4j (<http://logging.apache.org/log4j/>), and the configuration options in `logging.yml` are specific to this logging utility.

As with other settings, the defaults are sensible, but if, for example, you need more verbose logging, a good first step is to change the `rootLogger`, which influences all the logging. We'll leave the defaults for now, but if you wanted to make it log everything, you'd change the first line of `logging.yml` to this:

```
rootLogger: TRACE, console, file
```

By default, the logging level is `INFO`, which writes all events with a severity level of `INFO` or above.

## 2.5.3 Adjusting JVM settings

As a Java application, Elasticsearch runs in a JVM, which, like a physical machine, has its own memory. . The JVM comes with its own configuration, and the most important one is how much memory you allow it to use. Choosing the correct memory setting is important for Elasticsearch's performance and stability.

Most of the memory used by Elasticsearch is called *heap*. The default setting lets Elasticsearch allocate 256MB of your RAM for its heap, initially, and expand it up to 1GB. If your searches or indexing operations need more than 1GB of RAM, those operations will fail

and you'll see out of memory errors in your logs. Conversely, if you run Elasticsearch on an appliance that has only 256MB of RAM, the default settings might allocate too much memory.

To change the default values, set the `ES_MIN_MEM` and `ES_MAX_MEM` environment variables. Alternatively, you can use `ES_HEAP_SIZE` to set the same value for both. Set these environment variables on the command line before starting Elasticsearch.

On Unix-like systems, use the `export` command:

```
export ES_HEAP_SIZE=500m; bin/elasticsearch
```

On Windows, use the `SET` command:

```
SET ES_HEAP_SIZE=500m & bin\elasticsearch.bat
```

A more permanent way to set these variables is by changing `bin/elasticsearch.in.sh` (and `elasticsearch.bat` on Windows). Add `ES_HEAP_SIZE=500m` at the beginning of the file, after `#!/bin/sh`.

**TIP** If you installed Elasticsearch though the DEB package, change these variables in `/etc/default/elasticsearch`. If you installed from the RPM package, the same settings can be configured in `/etc/sysconfig/elasticsearch`.

For the scope of this book, the default values should be adequate. If you run more extensive tests, you may need to allocate more memory. If you're on a machine with less than 1GB of RAM, lowering those values to something like `200m` should also work.

### How much memory to allocate in production

Start with half of your total RAM as `ES_HEAP_SIZE`, if you run Elasticsearch only on that server. Try with less if other applications need significant memory. The other half is used by the operating system for caches, which make for faster access to your stored data. Beyond that rule of thumb, you'll have to run some tests while monitoring your cluster to see how much memory Elasticsearch needs. We talk more about performance tuning and monitoring in part 2 of the book.

Now that you've gotten your hands dirty with Elasticsearch configuration options and you've indexed and searched through some data, let's get a taste of the "elastic" part of Elasticsearch: the way it scales—we cover this topic in depth in chapter 9. You could just as well work through all chapters with a single node, but to get an overview of how scaling works, let's add more nodes to the same cluster.

## 2.6 Adding nodes to the cluster

In chapter 1, you unpacked the tar.gz or zip archive and started up your first Elasticsearch instance. This created your one-node cluster. Before you add a second node, let's check the cluster's status to visualize how data is currently allocated. You can do that with a graphical

tool such as Elasticsearch Kopf or Elasticsearch Head, which we mentioned previously (see section 2.1.4) when you indexed a document. Figure 2.12 shows the cluster in Kopf.



Figure 2.12 One-node cluster shown in Elasticsearch Kopf

If you don't have either of these plugins installed, you can always get this information from the command line via the Cluster Health API:<sup>7</sup>

```
% curl 'localhost:9200/_cluster/health?pretty'
{
  "cluster_name" : "elasticsearch-in-action",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 2,
  "active_shards" : 2,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 2
}
```

Either way, you should see the following information:

- Cluster name, as you defined it previously in `elasticsearch.yml`.
- There's only one node.
- The `get-together` index has two primary shards, which are active. The unassigned shards represent a set of replicas that were configured for this index. Because there's only one node, those replicas remain unallocated.

The unallocated replica shards cause the status to be yellow. This means all the primaries are there, but not all the replicas. If primaries were missing, the cluster would be red, to signal at least one index being incomplete. If all replicas would be allocated, the cluster would be green, to signal that everything works as expected.

<sup>7</sup> [www.elasticsearch.org/guide/en/elasticsearch/reference/current/cluster-health.html](http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/cluster-health.html)

### 2.6.1 Starting a second node

From a different terminal, run bin/elasticsearch or elasticsearch.bat. This starts another Elasticsearch instance on the same machine. You'd normally start new nodes on different machines to take advantage of additional processing power, but for now we'll run everything locally.

In the terminal or log file of the new node, you should see a line that begins like this:

```
[INFO ][cluster.service] [Raman] detected_master [Hammond, Jim]
```

Hammond, Jim is the name of your first node. What happened was that your second node detected the first one via multicast and joined the cluster. The first node is also the master of the cluster, which means it's responsible for keeping information such as which nodes are in the cluster and where shards are located. This information is called cluster state and it's replicated to other nodes. If the master goes down, another node can be elected to take its place.

If you look at your cluster's status in figure 2.13, you can see that the set of replicas was allocated to the new node, making the cluster green.

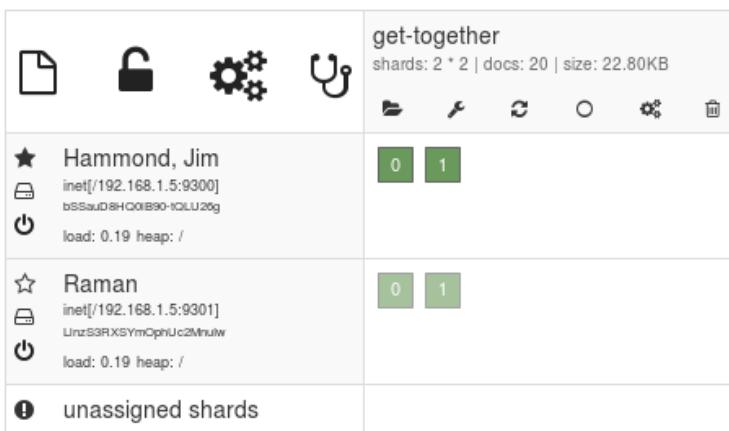


Figure 2.13 Replica shards are allocated to the second node

### 2.6.2 Adding additional nodes

If you run bin/elasticsearch or elasticsearch.bat again, to add a third node, and then a fourth, you'll see that they detect the master via multicast and join the cluster in the same way. Additionally, as shown in figure 2.14, the four shards of the get-together index automatically get balanced across the cluster.

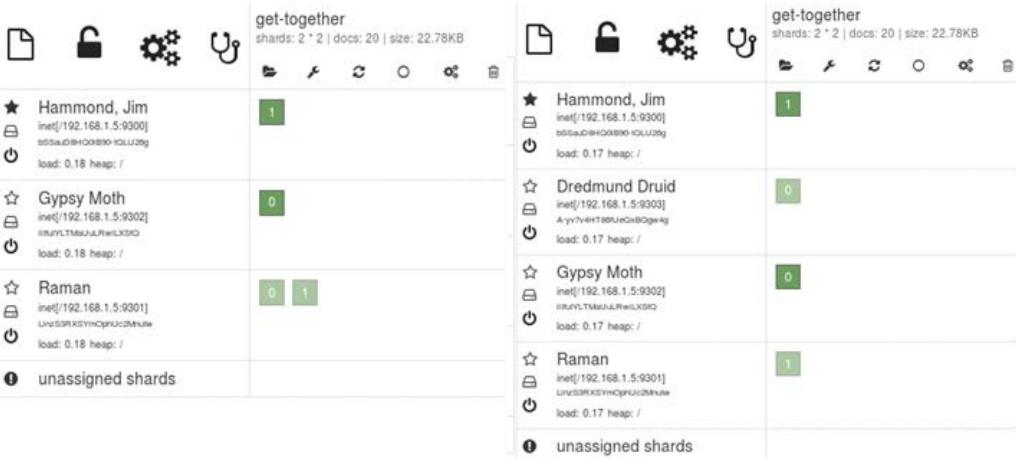


Figure 2.14 Elasticsearch automatically distributes shards across the growing cluster.

At this point you might wonder what happens if you add more nodes. By default, nothing happens because you have four total shards that can't be distributed to more than four nodes. That said, if you need to scale, you have a few options:

- *Change the number of replicas.* Replicas can be updated on the fly, but scaling this way increases only the number of concurrent searches your cluster can serve. The indexing throughput as well as the performance of isolated searches remains the same.
- *Create an index with more shards.* This implies reindexing your data because the number of shards can't be changed on the fly.
- *Add more indices.* Some data can be easily designed to use more indices. For example, if you index logs, you can put each day's logs in its own index.

We discuss these patterns for scaling out in chapter 9. Again, scaling out for more concurrent searches isn't a problem because you can change the number of replicas. The real challenge is in making indexing and individual searches run fast; a topic we discuss in chapter 10.

## 2.7 Summary

- Elasticsearch is document-oriented, scalable, and schema-free by default.
- Although you can form a cluster with the default settings, you should adjust at least some of them before you go to production; for example, cluster name and heap size.
- Indexing requests are distributed among the primary shards and replicated to those primary shards' replicas.
- Searches are done using a round-robin approach between complete sets of data, no matter if those are made up of shards or replicas. The node that received the search

request then aggregates partial results from individual shards and returns those results to the application.

- Client applications may be unaware of the sharded nature of each index or what the cluster looks like. They care only about indices, types, and document IDs. They use the HTTP REST API to index and search for documents.
- You can send new documents and search parameters as the JSON payload of a HTTP request, and you'll get back a JSON reply with the results.

In the next chapter, you'll get the foundation you need to organize your data effectively in Elasticsearch, you'll learn what types of fields your documents can have, and you'll become familiar with all the relevant options for indexing, updating, and deleting.

Figure 2.1 An Elasticsearch cluster from the application's and administrator's point of view .....	25
Figure 2.2 Logical layout of data in Elasticsearch: how an application sees data.....	26
Figure 2.3 A three-node cluster with an index divided into five shards, with one replica per shard.....	30
Figure 2.4 Documents gets indexed to random primary shards and their replicas. Searches run on complete sets of shards, regardless of their status as primaries or replicas. ....	31
Figure 2.5 Term dictionary and frequencies in a Lucene index.....	32
Figure 2.6 Multiple primary and replica shards make up the "get-together" index.....	33
Figure 2.7 Obtaining more performance by scaling vertically compared to scaling horizontally .....	34
Figure 2.8 Indexing operation is forwarded to the responsible shard, then to its replicas.....	35
Figure 2.9 Search request is forwarded to shards/replicas containing a complete set of data. Then, results are aggregated and sent back to the client.....	35
Figure 2.10 URI of a document in Elasticsearch.....	36
Figure 2.11 Partial results can be returned from shards that are still available .....	44
Figure 2.12 One node cluster shown in Elasticsearch Kopf.....	53
Figure 2.13 Replica shards are allocated to the second node .....	54
Figure 2.14 Elasticsearch automatically distributes shards across the growing cluster. ....	55
Listing 2.1 Indexing data with the populate.sh script .....	40
Listing 2.2 Search for "elasticsearch" in groups.....	41
Listing 2.3 Search reply returning two fields of a single resulting document.....	43

# 3

## *Indexing, updating, and deleting data*

### ***This chapter covers***

- Using mapping types to define multiple types of documents in the same index
- Types of fields you can use in mappings
- Using predefined fields and their options
- Updating and deleting data

This chapter is all about getting data in and out of Elasticsearch: indexing, updating and deleting documents. In chapter 1, you learned that Elasticsearch is *document-based* and that documents are made up of fields and their values, which makes them self-contained, much like having the column names from a table contained in the rows. In chapter 2, you saw how you can index such a document via Elasticsearch’s REST API. Here, we’ll dive deeper into the indexing process, by looking at the fields in those documents and what they contain. For example, when you index a document that looks like this:

```
{ "name": "Elasticsearch Denver" }
```

the name field is a string because its value, `Elasticsearch Denver`, is a string. Other fields could be numbers, booleans, and so on. In this chapter, we’ll look at three types of fields:

- Core—These fields include strings and numbers.
- Arrays and multi fields—These fields help you store multiple values of the same core type, in the same field. For example, you can have multiple tag strings in your tags field.

- Predefined—Examples of these fields include `_ttl` (which stands for “time to live”) and `timestamp`.

Think of these field types as metadata that can be automatically managed by Elasticsearch to give you additional functionality. For example, you can store some fields in a way that make your indices smaller, you can configure Elasticsearch to automatically add new data to documents, such as a timestamp, or you can use the `_ttl` field to get your documents automatically deleted after a specified amount of time.

Once you know the field types that can be in your documents and how to index them, we’ll look at how you can update documents that are already there. Because of the way it stores data, when Elasticsearch updates an existing document, it retrieves it and applies changes according to your specifications. It then indexes the resulting document again and deletes the old one. Such updates can raise concurrency issues, and you’ll see how they can be solved automatically with document versions.

You’ll also see various ways of deleting documents. Some ways are faster than others. This is again due to the particular way Apache Lucene, the main library used by Elasticsearch for indexing, stores data on disc.

We’ll start with indexing, by looking at how you can manage fields from your documents. As you saw in chapter 2, fields are defined in mappings, so before we dive into how you can work with each type of field, we’ll look at how you can work with mappings in general.

### 3.1 Using mappings to define kinds of documents

Each document belongs to a type, which in turn belongs to an index. As a logical division of data, you can think of indices as databases, and types as tables. For example, the get-together website that we introduced in chapter 2 uses a different type for groups and events because those documents have different structures. Note that if you also had a blog on that website, you might keep blog entries and comments in a separate index, because it’s a completely different set of data.

Types contain a definition of each field in the *mapping*. The mapping includes all the fields that might appear in documents from that type and tells Elasticsearch how to index the fields in a document. For example, if a field contains a date, you can define which date format is acceptable.

---

#### Types provide only logical separation

With Elasticsearch, there’s no physical separation of documents that have different types. All documents within the same Elasticsearch index, regardless of type, end up in the same set of files belonging to the same shards. In a shard, which is a Lucene index, the name of the type is a field, and all fields from all mappings come together as fields in the Lucene index.

The concept of a type is a layer of abstraction specific to Elasticsearch, but not Lucene, which makes it easy for you to have different kinds of documents in the same index. Elasticsearch takes

care of separating those documents, for example, by filtering documents belonging to a certain type, when you search in that type only.

This approach creates a problem when the same field name occurs in multiple types. To avoid unpredictable results, two fields with the same name should have the same settings, otherwise Elasticsearch might have a hard time figuring out which of the two fields you're referring to. In the end, both those fields belong to the same Lucene index. For example, if you have a name field in both group and event documents, both should be strings, not one a string and one an integer. This is rarely a problem in real life, but it's worth remembering to avoid surprises.

In figure 3.1, groups and events are stored in different types. The application can then search in a specific type, such as events. Elasticsearch also allows you to search in multiple types at once. Or even in all types of an index, by specifying only the index name when you search.

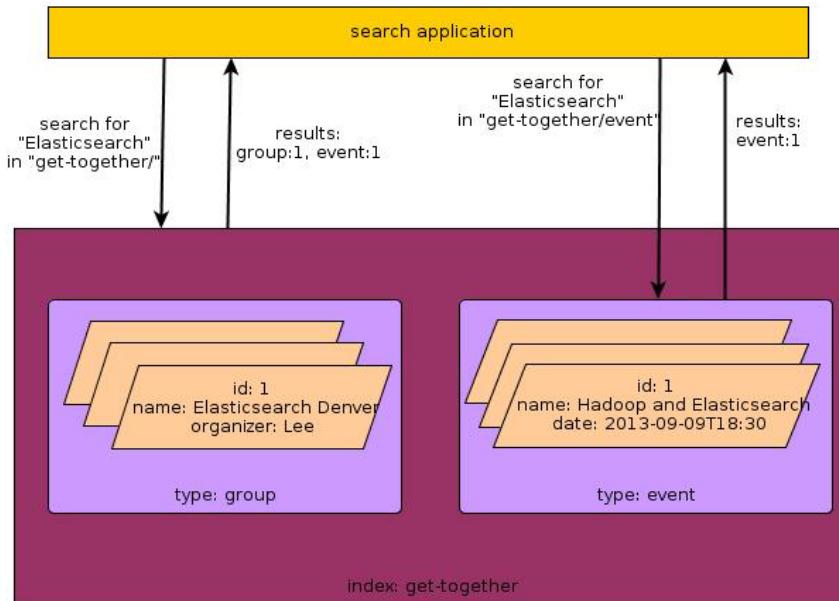


Figure 3.1 Using types to divide data in the same index; searches can run in one, multiple, or all types

Now that you know how mappings are used in Elasticsearch, let's have a look at how you can read the mapping of a type and how you can write one.

### 3.1.1 Retrieving and defining mappings

When you're learning Elasticsearch, you often don't need to worry about the mapping because Elasticsearch detects your fields automatically and adjusts your mapping accordingly. You'll have a look at how that works in listing 3.1.

## GETTING THE CURRENT MAPPING

To see the current mapping of a field type, issue an HTTP GET on `_mapping` under the type's URL:

```
curl 'localhost:9200/get-together/group/_mapping?pretty'
```

In the following listing, you first index a new document from your get-together website, specifying a new type called `new-events`, and Elasticsearch automatically creates the mapping for you. You then retrieve the created mapping, which shows you the fields from your document and the field types that Elasticsearch detected for each field.

### **Listing 3.1 Getting an automatically generated mapping**

```
% curl -XPUT 'localhost:9200/get-together/new-events/1' -d '{  
    "name": "Late Night with Elasticsearch",  
    "date": "2013-10-25T19:00"  
}'  
% curl 'localhost:9200/get-together/new-events/_mapping?pretty'  
# expected result:  
#{  
    "get-together" : {  
        "new-events" : {  
            "properties" : {  
                "date" : {  
                    "type" : "date",  
                    "format" : "dateOptionalTime"  
                },  
                "name" : {  
                    "type" : "string"  
                }  
            }  
        }  
    }  
}
```

#A Indexes a new document

#B Gets the mapping

#C Type name is included in the result

#D The two fields in the document were detected as well as the type of each field

## DEFINING A NEW MAPPING

To define a mapping, you use the same URL as previously, but do an HTTP PUT instead of GET. You need to specify the JSON mapping in the body using the same format that's returned

when you retrieve a mapping. For example, the following request puts a mapping that has only one `string` field:

```
% curl -XPUT 'localhost:9200/get-together/new-events/_mapping' -d '{
  "new-events" : {
    "properties" : {
      "host" : {
        "type" : "string"
      }
    }
  }
}'
```

You can define a new mapping after you create the index but before inserting any document into that type. Why does this `PUT` work if, as shown in listing 3.1, you already had a mapping in place? We'll explain why next.

### 3.1.2 Extending an existing mapping

When you put a mapping over an existing one, Elasticsearch merges the two. If you ask Elasticsearch for the mapping now, you should get something like this:

```
{
  "get-together" : {
    "new-events" : {
      "properties" : {
        "date" : {
          "type" : "date",
          "format" : "dateOptionalTime"
        },
        "host" : {
          "type" : "string"
        },
        "name" : {
          "type" : "string"
        }
      }
    }
}
```

As you can see, the mapping now contains the two fields from the initial mapping, plus the new field you defined. The initial mapping was extended with the newly added field, which is something you can do at any point. Elasticsearch calls this a merge between the existing mapping and the one you provide.

Unfortunately, not all merges work. For example, you can't change an existing field's data type, and, in general, you can't change the way a field is indexed. Let's take a closer look into why this happens. As shown in the following listing, if you try to change the host field to a `long`, it fails with a `MergeMappingException`.

### Listing 3.2 Trying to change an existing field type from string to long fails

```
% curl -XPUT 'localhost:9200/get-together/new-events/_mapping' -d '{
  "new-events" : {
    "properties" : {
      "host" : {
        "type" : "long"
      }
    }
  }
}'
# expected result
# {"error":"MergeMappingException[Merge failed with failures {[mapper [host] of
# different type, current_type [string], merged_type [long]]]}]","status":400}
```

The only way around this error is to reindex all the data in new-events, which involves the following steps:

- Removing all data from the new-events type—you’ll learn later in this chapter how to delete data. Removing data also removes the current mapping.
- Put the new mapping.
- Index all the data again.

To understand why reindexing might be required, imagine you’ve already indexed an event with a string in the host field. If you want the host field to be `long` now, Elasticsearch would have to change the way host is indexed in the existing document. As you’ll explore later in this chapter, editing an existing document implies deleting and indexing again.

To define correct mappings, that hopefully won’t need changes, only additions, let’s look at the core types you can choose for your fields in Elasticsearch, and what you can do with them.

## 3.2 Core types for defining your own fields in documents

With Elasticsearch, a field can be one of the *core types* (see table 3.1), such as a string or a number, or it can be a more complex type derived from core types, such as an array.

There are some additional types, not covered in this chapter. For example, there’s the *nested* type, which allows you to have documents within documents. Or the *geo\_point* type, which stores a location on Earth based on its longitude and latitude. We’ll discuss those additional types in chapter 7, where we cover relationships among documents, and in appendix A, where we discuss geospatial data.

**NOTE** In addition to the fields you define in your documents, such as name or date, Elasticsearch uses a set of predefined fields to enrich them. For example, there’s an `_all` field, where all the document’s fields are indexed together. This is useful when users search for something without specifying the field—you can search in all fields. These predefined fields have their own configuration options, and we’ll discuss them later in this chapter.

Table 3.1 Elasticsearch core field types

Core type	Example values
String	"Lee", "Elasticsearch Denver"
Numeric	17, 3.2
Date	2013-03-15T10:02:26.231+1:00
Boolean	Value can be either true or false

Let's look at each of these core types, so you can make good mapping choices when you index your own data.

### 3.2.1 String

*Strings* are the most straightforward: your field should be string if you're indexing characters. They're also the most interesting because you have so many options in your mapping about how to analyze them.

*Analysis* is the process of parsing the text to transform it and break it down into elements to make searches relevant. If it sounds too abstract, don't worry: chapter 5 explores the concept. But let's look at the basics now starting with the document you indexed in listing 3.1:

```
% curl -XPUT 'localhost:9200/get-together/new-events/1' -d '{
  "name": "Late Night with Elasticsearch",
  "date": "2013-10-25T19:00"
}'
```

With this document indexed, let's search for the word "late" in the name field, which is a string:

```
% curl 'localhost:9200/get-together/new-events/_search?pretty' -d '{
  "query": {
    "query_string": {
      "query": "late"
    }
  }
}'
```

And the search finds the "Late Night with Elasticsearch" document you indexed in Listing 3.1. Elasticsearch connects the strings "late" and "Late Night with Elasticsearch" through analysis. As you can see in figure 3.2, when you index "Late Night with Elasticsearch", the default analyzer lowers all letters, and then breaks the string into words.

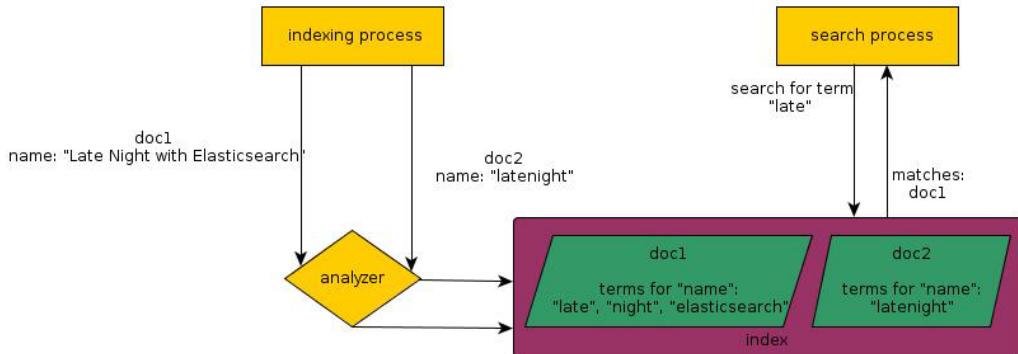


Figure 3.2 After the default analyzer breaks strings into terms, subsequent searches match those terms.

The analyzer removes the word "with" because it's so common it belongs to a list of stop words. By default, stop words are eliminated during analysis because they appear so frequently that they're irrelevant in searches.

The analysis produces three terms: "late", "night", and "elasticsearch". The same process is then applied to the query string, but this time, "late" produces the same string: "late." The document (doc1) matches the search because the "late" term that resulted from the query matches the "late" term that resulted from the document.

**DEFINITION** A *term* is a word from the text and is the basic unit for searching. In different contexts, this "word" can mean different things: it could be a name, or it could be an IP address, for example. If you want only exact matches on a field, the entire field should be treated as a word.

On the other hand, if you index "latenight", the default analyzer creates only one term: "latenight". Searching for "late" won't hit doc1 because it doesn't include the term "late".

#### MAPPING AND ANALYSIS INTERPLAY

This analysis process is where the mapping comes into play. You can specify many options around analyzing in your mapping. For example, you can configure stemming to take place during analysis. Stemming produces terms that are synonyms of your original terms, so queries for synonyms match as well. We'll dive into the details of analysis in chapter 5, as promised, but for now, let's look at the `index` option, which can be set to `analyzed` (the default), `not_analyzed` or `no`. For example, to set the `name` field to `not_analyzed`, your mapping might look like this:

```
% curl -XPUT 'localhost:9200/get-together/new-events/_mapping' -d '{
  "new-events" : {
    "properties" : {
      "name" : {
        ...
      }
    }
  }
}'
```

```

        "type" : "string",
        "index" : "not_analyzed"
    }
}
}'

```

Setting `index` to `analyzed` produces the behavior you saw previously: by default, the analyzer lowerscases all letters, breaks your string into words, and eliminates stop words. Use this option when your strings are long enough and you expect a single matching word to produce a match. For example, if users search for “elasticsearch,” they expect to see “Late Night with Elasticsearch” in the list of results.

Setting `index` to `not_analyzed` does the opposite: the analysis process is skipped, and the entire string is indexed as one term. Use this option when you want exact matches, such as when you search for tags. You probably want only “big data” to show up as a result when you search for “big data,” not “data”.

If you set `index` to `no`, then indexing is skipped and no terms are produced, so you won’t be able to search on that particular field. When you don’t need to search on a field, this option saves space and decreases the time it takes to index and search. For example, you might store reviews for events. Although storing and showing those reviews is valuable, searching through them might not be. In this case, disable indexing for that field, making the indexing process faster and saving space.

### Check if your query is analyzed when searching in fields that aren’t

For some queries, such as the `query_string` you used previously, the analysis process is applied to your search criteria. It’s important to be aware if this is happening, otherwise results might not be as expected.

For example, if you index “Elasticsearch,” and it’s not analyzed, it produces the term “Elasticsearch”. When you query for “Elasticsearch” like this:

```
curl 'localhost:9200/get-together/new-events/_search?q=Elasticsearch'
```

the URI request is analyzed, and the term “elasticsearch” (lowercased) is produced. But you don’t have the term “elasticsearch” in your index; you have only “Elasticsearch” (with a capital *E*), so you get no hits.

In chapter 4, which is all about searching, you’ll learn which query types analyze the input text and which don’t.

Next, let’s look at how you can index numbers. Elasticsearch provides many core types that can help you deal with numbers, so we’ll refer to them collectively as numeric.

#### 3.2.2 Numeric

*Numeric types* can be with or without a floating point. If you don’t need decimals, you can choose between `byte`, `short`, `integer` and `long`; if you do need them, your choices are `float`

and `double`. These types correspond to Java’s primitive data types, and choosing between them influences the size of your index and the range of values you can index. For example, whereas a `long` takes up 64 bits, a `short` takes up only 16 bits, but a `long` can store ranges up to several trillion times larger than the -32,768 to 32,767 that a `short` can store.

If you don’t know the range you need for your integer values or the precision you need for your floating point values, it’s safe to do what Elasticsearch does when it detects your mapping automatically: use `long` for integer values, and `double` for floating-point values. Your index might become larger and slower because these two types take up the most space, but at least you’re unlikely to get an “out of range” error from Elasticsearch when indexing.

Now that we’ve covered strings and numbers, let’s look at a type that’s more purpose-built: `date`.

### 3.2.3 Date

The `date` type is used for storing dates and times. It works like this: you normally provide a string with a date, as in `2013-12-25T09:00:00`. Then, Elasticsearch parses the string and stores it as a number of type `long` in the Lucene index. That `long` is the number of milliseconds that have elapsed since 00:00:00 UTC time on January 1, 1970 (UNIX epoch) and the time you provided.

When you search for documents, you still provide date strings, and Elasticsearch parses those strings and works with numbers in background. It does that because numbers are faster to store and work with than strings.

You, on the other hand, only have to consider whether Elasticsearch understands the date string you’re providing. The date format of your date string is defined by the `format` option, and Elasticsearch parses ISO 8601 timestamps by default.

#### ISO 8601

An international standard for exchanging date- and time-related data, ISO 8601 is widely used in timestamps due to RFC 3339 (<https://www.ietf.org/rfc/rfc3339.txt>). An ISO 8601 date looks like this:

`2013-10-11T10:32:45.453-3:0`

It has all the right ingredients of a good timestamp: information is read from left to right, from the most important to the least important; the year has four digits; and the time includes subseconds and time zone.

Much of the information in this timestamp is optional, for example, you don’t need to specify milliseconds, and you can skip the time altogether.

When you use the `format` option to specify a date format, you have two options:

- *Use a predefined date format.* For example, the “`date`” format parses dates as “`2013-02-25`.” Many predefined formats are available, and you can see them all here:

[www.elasticsearch.org/guide/reference/mapping/date-format/](http://www.elasticsearch.org/guide/reference/mapping/date-format/)

- *Specify your own custom format.* You can specify a pattern for timestamps to follow. For example, specifying “MMM YYYY” parses dates as “Jul 2001.” For a full reference on building date patterns, visit: <http://joda-time.sourceforge.net/api-release/org/joda/time/format/DateTimeFormat.html>

To put all this date information to use, let’s add a new mapping type called weekly-events, as shown in listing 3.3. Then, as also shown in the listing, add a title and date of the first event, and specify an ISO 8601 timestamp for that date. Also add a field with the date of the next event, and specify a custom date format for that date.

### Listing 3.3 Using default and custom time formats

```
% curl -XPUT 'localhost:9200/get-together/weekly-events/_mapping' -d '
{
  "weekly-events" : {
    "properties": {
      "next_event": {
        "type": "date",
        "format": "MMM DD YYYY"                                #A
      }
    }
  }
}

% curl -XPUT 'localhost:9200/get-together/weekly-events/1' -d '
{
  "name": "Elasticsearch News",
  "first_occurrence": "2011-04-03",                      #B
  "next_event": "Oct 25 2013"
}'
```

#A Defines the custom date format. Other dates are automatically detected and don’t need to be explicitly defined.

#B Specifies a standard date/time format. Only the date is included; the time isn’t specified.

We’ve talked about strings, numbers, and dates; let’s move on to the last core type: boolean. Like date, boolean is a type that’s more purpose-built.

#### 3.2.4 Boolean

The *boolean* type is used for storing true/false values from your documents. For example, you might want a field that indicates whether the event’s video is available for download. A sample document could be indexed like this:

```
% curl -XPUT 'localhost:9200/get-together/new-events/downloadable' -d '{
  "name": "Broadcasted Elasticsearch News",
  "downloadable": true
}'
```

The downloadable field is automatically mapped as boolean and is stored in the Lucene index as `T` for `true` or `F` for `false`. As with date fields, it parses the value you supply in the source document and transforms `true` and `false` to `T` and `F`, respectively. If you supply a number value, it transforms `0` to `F` and any other number to `T`:

```
% curl -XPUT 'localhost:9200/get-together/new-events/downloadable2' -d '{
  "name": "Broadcasted Big Data News",
  "downloadable": 0
}'
```

We've looked at the core types: string, numeric, date, and boolean, which you can use in your own fields; let's move on to arrays and multi-fields, which enable you to use the same core type multiple times.

### 3.3 Arrays and multi-fields

Sometimes having simple field-value pairs in your documents isn't enough. You might need to have multiple values in the same field. For example, if you're indexing blog posts, you might want to have a tag field with one or more tags in it. In this case, you need an array.

#### 3.3.1 Arrays

To index a field with multiple values, put those values in square brackets. For example:

```
% curl -XPUT 'localhost:9200/blog/posts/1' -d '{
  "tags": ["first", "initial"]
}'
```

At this point you might wonder, "How do you define an array field in your mapping?" The answer is: you don't. In this case, the mapping defines the `tags` field as `string`, as it does when you have a single value:

```
% curl 'localhost:9200/blog/posts/_mapping?pretty'
{
  "posts" : {
    "properties" : {
      "tags" : {
        "type" : "string"
      }
    }
  }
}
```

You can combine arrays with their core type counterparts without changing your mapping. For example, if the next blog post only has one tag, you can index it like this:

```
% curl -XPUT 'localhost:9200/blog/posts/2' -d '{"tags": "second"}'
```

Wherever you have a single value, it works the same as an array with a single element.

#### 3.3.2 Multi-fields

If arrays are all about indexing more data with the same settings, multi-fields are about indexing the same data multiple times using different settings.

For example, in listing 3.4, you configure the tags field from your blog type with two different settings: `analyzed`, for matches on every word; and `not_analyzed`, for exact matches on the full tag name.

**TIP** You can “upgrade” a single field to a multi-field configuration without needing to reindex your data. This is what happens if you’ve already created a `tags` `string` field before you run listing 3.4.

#### Listing 3.4 Multi-field for a string: once analyzed, once not\_analyzed

```
% curl -PUT 'localhost:9200/blog/posts/_mapping' -d '{
  "posts" : {
    "properties" : {
      "tags" : {
        "type": "string",
        "index": "analyzed"                                #A
        "fields": {
          "verbatim": {
            "type": "string",
            "index": "not_analyzed"                         #B
          }
        }
      }
    }
  }
}'
```

#A The default tags field is analyzed, which lowerscases and breaks the name into words

#B The second field, tags.verbatim, is not\_analyzed, which makes the original tag a single term

You search in the `analyzed` version of the `tags` field as you do with any other string. To search in the `not_analyzed` version (and get back only exact matches on the original tag), specify the full path: `tags.verbatim`.

Both multi field and array field types let you have multiple core types into a single field. Next, we’ll look at predefined fields, which are normally handled by Elasticsearch on its own, to add new functionality to your documents, such as automatically expiring them.

### 3.4 Using predefined fields

Elasticsearch provides a number of predefined fields you can use and configure to add new functionality. These predefined fields are different from the fields you’ve seen so far in three ways:

- Typically, you don’t fill the content of predefined field; Elasticsearch does it.

For example, you can use the `_timestamp` field to record the date when a document was indexed.

- They uncover field-specific functionality.

For example, the `_ttl` (time to live) field enables Elasticsearch to remove documents after a specified amount of time.

- Predefined field names always begin with an underscore (`_`).

These fields add new metadata to your documents, and Elasticsearch uses this metadata for various features from storing the original document to storing timestamp information for automatic expiry.

We'll divide the predefined fields in the following categories:

- *Control how to store and search your documents*—`_source` lets you store the original JSON document as you index it. `_all` indexes all your fields together.
- *Identify your documents*—These are special fields containing data about where your document was indexed: `_uid`, `_id`, `_type`, `_index`.
- *Add new properties to your documents*—You can index the size of the original JSON with `_size`. Similarly, you can index the time it was indexed with `_timestamp` and make Elasticsearch delete it after a specified amount of time with `_ttl`.
- *Control the shard where your documents are routed to*—These are `_routing` and `_parent`. We'll look at them in chapter 8, where we talk about relationships among documents.

### **3.4.1 Control how to store and search your documents**

Let's start by looking at `_source`, which lets you store the documents you index, and `_all`, which lets you index all their content in a single field.

#### **\_SOURCE FOR STORING THE ORIGINAL CONTENTS**

The `_source` field is for storing the original document, in the original format. This lets you see the documents that matched a search, not only their IDs.

`_source` can have `enabled` set to `true` or `false`, to specify whether you want to store the original document or not. By default it's `true`, and, in most cases, that's good because the existence of `_source` allows you to use other important features of Elasticsearch. For example, as you'll learn later in this chapter, updating document contents using the update API needs `_source`.

To see how this field works, let's look at what Elasticsearch typically returns when you retrieve a previously indexed document:

```
% curl 'localhost:9200/get-together/new-events/downloadable?pretty'
{
  "_index" : "get-together",
  "_type" : "new-events",
  "_id" : "downloadable",
  "_version" : 1,
  "exists" : true, "_source" : {
    "name": "Broadcasted Elasticsearch News",
    "downloadable": true
  }
}
```

You also get the \_source JSON back when you search, as it's returned there by default as well. If you disable \_source, you don't get the original document in the reply. This is typically done when you have a separate data store for your original content. In such a situation, you may want to index every entry in Elasticsearch with the same ID as in the data store. When you search, get the list of IDs from the results, and then go back to the data store to get the content. Such a process is illustrated in figure 3.3.

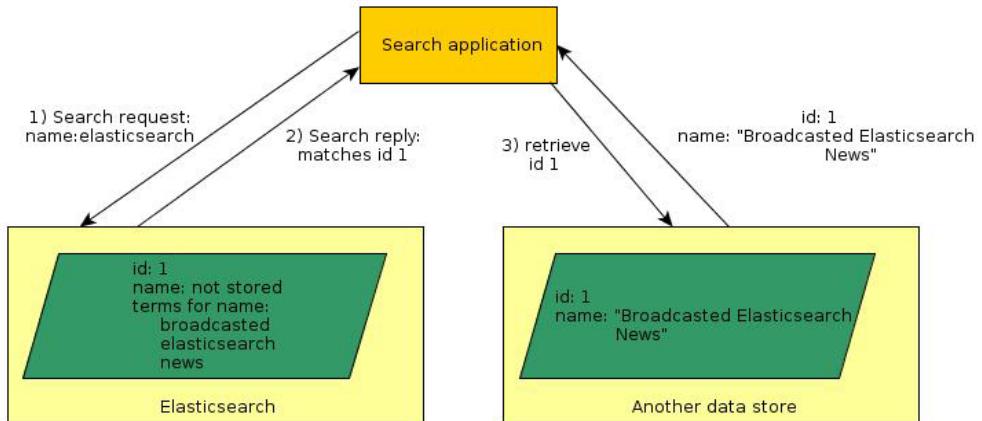


Figure 3.3 Using Elasticsearch for indexing only and using a different data store for document content

As you saw in sections 3.2 and 3.3, the fields you defined for your documents go under the properties field under the JSON mapping. Predefined fields, including \_source, go directly under the mapping name. This makes it clear that predefined fields have a special status, and they're not another property of your document. In this case, the \_source field is not content that you add to document but a way to control how Elasticsearch stores it. The document remains the same, it's the functionality around it that changes. To disable \_source, you can define a mapping as shown in the following listing:

#### **Listing 3.5 Disabling \_source**

```
% curl -XPUT localhost:9200/get-together/events_unstored/_mapping -d '{
  "events_unstored": {
    "_source": { "enabled": false},          #A
    "properties": {
      "name": {                                #B
        "type": "string"                      #B
      }
    }
  }
}'
```

#A Predefined field, defined at the root of the mapping type

#B Custom data field, defined under properties

## RETURNING ONLY SOME FIELDS OF THE SOURCE DOCUMENT

When you retrieve or search for a document, you can ask Elasticsearch to return only specific fields, and not the entire source. One way to do this is to give a comma-separated list of fields in the `fields` parameter. For example:

```
% curl -XGET 'localhost:9200/get-together/group/1?pretty&fields=name'
{
  "_index" : "get-together",
  "_type" : "group",
  "_id" : "1",
  "_version" : 1,
  "exists" : true,
  "fields" : {
    "name" : "Denver Clojure"
  }
}
```

When the source is stored, Elasticsearch automatically goes to the source, gets the required fields and returns them to you. When you have no source, there's nothing to return. For example, if you followed listing 3.5 and indexed a sample document, trying to retrieve the title won't get you anything:

```
% curl 'localhost:9200/get-together/events_unstored/1?pretty&fields=name'
{
  "_index" : "get-together",
  "_type" : "events_unstored",
  "_id" : "1",
  "_version" : 1,
  "exists" : true
}
```

If `_source` is disabled, you can store individual fields by settings the `store` option to `yes`. For example, to store only the `name` field, your mapping might look like this:

```
% curl -XPUT localhost:9200/get-together/events_unstored/_mapping -d '{
  "events_unstored": {
    "_source": { "enabled": false},
    "properties": {
      "name": {
        "type": "string",
        "store": "yes"
      }
    }
  }
}'
```

You can also choose to store both `_source` and individual fields. This might be useful when you often ask Elasticsearch for a particular field because retrieving a single stored field will be faster than retrieving the entire `_source` and extracting that field from it.

When you store `_source` and individual fields, you should take into account that the more you store, the bigger your index gets. And usually, bigger indices imply slower indexing and slower searching. The good news here is that since version 0.90, Elasticsearch automatically compresses both `_source` and any individual fields you might choose to store. This is useful

because it keeps your index size small, and lets the operating system keep more of your data in its caches. In most situations, the overhead of compressing and uncompressing fields is insignificant when you compare it to the benefit of having smaller indices.

#### **\_ALL FOR INDEXING EVERYTHING**

Just like `_source` is storing everything, `_all` is indexing everything. When you search in `_all`, Elasticsearch will return a hit regardless of which field matches. This is useful when users are looking for something without knowing where to look for, like searching for “elasticsearch” may match the group name “Elasticsearch Denver” as well as the tag “elasticsearch” on other groups.

Running a search from the URI without a field name will search on `_all` by default:

```
curl 'localhost:9200/get-together/group/_search?q=elasticsearch'
```

If you always search on specific fields, you can disable `_all` by setting `enabled` to `false`. Like with `_source`, doing so will reduce the total size of your index and will make indexing operations faster.

By default, each field is included in `_all`, by having `include_in_all` implicitly set to `true`. You can use this option to control what is and isn’t included in `_all`. In the next listing, you’ll create a mapping where you include in `_all` only two of the total of three data fields. Then you’ll search in specific fields and in `_all` and compare the results.

#### **Listing 3.6 Using `include_in_all` to store only some fields in `_all`**

```
curl -XPUT 'localhost:9200/get-together/custom-all/_mapping' -d '{
  "custom-all": {
    "properties": {
      "name": { "type": "string" },
      "tags": { "type": "string" },                                     #A
      "organizer": {
        "type": "string",
        "include_in_all": false                                         #B
      }
    }
  }
}'

curl -XPUT localhost:9200/get-together/custom-all/1 -d '{
  "name": "Elasticsearch Denver",
  "tags": "elasticsearch",
  "organizer": "Lee"
}'

curl 'localhost:9200/get-together/_refresh'
CUSTOM_ALL="localhost:9200/get-together/custom-all"
curl "$CUSTOM_ALL/_search?q=denver&pretty"                                #C
curl "$CUSTOM_ALL/_search?q=name:denver&pretty"                            #D
curl "$CUSTOM_ALL/_search?q=lee&pretty"
```

```

curl "$CUSTOM_ALL/_search?q=organizer:lee&pretty"          #E
                                                #F

#A These are included in _all by default
#B You explicitly say you don't need this in _all
#C Returns result because the name field is in _all
#D Returns result because you can search in the specific field
#E Doesn't return result because the organizer field isn't included in _all
#F Returns result when you search in the specific field

```

Using `include_in_all` gives you flexibility not only in terms of saving space but also regarding how your queries behave. As you saw in the previous example, if a user searches without specifying a field, you might want to give back matches from the name and tags fields and match organizer fields only when the user specifically asks for that. This prevents unexpected results from appearing, if most customers think of a name or a tag when they give a keyword.

The next set of predefined fields are those used to identify documents: `_index`, `_type`, `_id` and `_uid`.

### 3.4.2 Identify your documents

To identify a document within the same index, Elasticsearch uses a combination of the document's type and ID, in the `_uid` field. The `_uid` field is made up from the `_id` and `_type` fields that you always get when searching or retrieving documents:

```
% curl 'localhost:9200/get-together/group/1?fields&pretty'
{
  "_index" : "get-together",
  "_type" : "group",
  "_id" : "1",
  "_version" : 1,
  "exists" : true
}
```

At this point you might wonder, "Why does Elasticsearch store the same data in two places: you have `_id`, then `_type`, then `_uid`?"

Elasticsearch uses `_uid` internally for identification, and you don't have any useful options around it. In contrast, `_id` and `_type` are special fields you can search on, And you can change their settings. To make them stored, set `store` to `yes`; to make them indexed or even analyzed, change the `index` option. Table 3.2 shows the default settings for `_id` and `_type`:

Table 3.2 Default settings for `_id` and `_type` fields

Field name	store value	index value	Observations
<code>_id</code>	no	no	It's not indexed and not analyzed. You can search on it, but Elasticsearch uses <code>_uid</code> to give you the results.

```
_type      no          not_analyzed It's indexed, and it produces a single term. You can search
                                         on it, but you can't get it as a single field.
```

## PROVIDING IDs FOR YOUR DOCUMENTS

You've seen here, and in chapter 2, that when you index a document, you need to tell Elasticsearch the type and the index it belongs to. The document also needs an ID to uniquely identify it within the type. That's your `_id` field. There are three ways to specify IDs for your documents:

- *Manually add the ID when you index the document.*

So far, you've mostly provided IDs manually as part of the URI. For example, to index a document with ID `1st`, you run something like this:

```
% curl -XPUT 'localhost:9200/get-together/manual_id/1st&pretty' -d '{
  "name": "Elasticsearch Denver"
}'
```

And you get back something like this:

```
{
  "_index": "get-together",
  "_type": "manual_id",
  "_id": "1st",
  "_version": 1
}
```

You can see in the reply that the `_id` field returns the value you provided.

- *Configure Elasticsearch to take the ID from a field within your document.*

The second way to get IDs for your documents is to have Elasticsearch pick the ID from a field within your document. This is useful if you already have a field with unique values, like a barcode for items in an online shop. If you use that as the `_id` as well, you'll have a quick way of getting an item if you know the barcode, the index, and the type: you retrieve the document, and no search is required. Also, you have a reliable way of identifying items if you need to update their content. We'll look at updating documents later in this chapter.

To get IDs from the barcode field, you first need to put that field name in the `path` option of your `_id` field. This makes Elasticsearch look for an ID in the barcode field:

```
% curl -XPUT localhost:9200/online-shop/barcode_id/_mapping -d '{
  "barcode_id": {
    "_id": {
      "path": "barcode"
    }
  }
}'
```

**TIP** For the command to work without an error, create the online-shop index first: curl -XPUT 'localhost:9200/online-shop/'

To index an item with the barcode as the ID, omit the ID from the URI, and use an HTTP POST request instead:

```
% curl -XPOST 'localhost:9200/online-shop/barcode_id/?pretty' -d '{
  "barcode": "abcd",
  "name": "Promotional T-Shirt"
}'
```

And you get back a reply like this:

```
{
  "_index" : "online-shop",
  "_type" : "barcode_id",
  "_id" : "abcd",
  "_version" : 1
}
```

You can still use the reply to see that the \_id field is what you provided in the barcode field.

- *Configure Elasticsearch to automatically generate a unique ID for you.*

The final approach to creating document IDs is to rely on Elasticsearch to generate unique IDs for you. This is useful if you don't have an unique ID already, or you don't need to identify documents by a certain property. Typically, this is what you do when you index application logs: they don't have a unique property to identify them, and they're never updated.

To have Elasticsearch generate the ID, use HTTP POST and omit the ID, like you did with barcodes. The difference is that you don't need to configure the path property for the \_id field.

```
% curl -XPOST 'localhost:9200/logs/auto_id/?pretty' -d '{
  "message": "I have an automatic id"
}'
```

The reply should look similar to the following:

```
{
  "_index" : "logs",
  "_type" : "auto_id",
  "_id" : "RwdYYvcU8Rjyy8sJPobVqDQ",
  "_version" : 1
}
```

As was the case with the other methods, you can see the ID that was generated in the JSON reply.

### STORING THE INDEX NAME INSIDE THE DOCUMENT

To have Elasticsearch store the index name in the document, along with the ID and the type, use the \_index field.

As with `_id` and `_type`, you can see `_index` in the results of a search or a GET request, but, as with `_id` and `_type`, what you see there doesn't come from the field contents: `_index` is disabled by default.

Elasticsearch knows which index each result came from, so it can show an `_index` value there, but, by default, you can't search for `_index` yourself. The following command shouldn't find anything:

```
% curl 'localhost:9200/_search?q=_index:get-together'
```

To enable `_index`, set `enabled` to `true`. The mapping might look like this:

```
% curl -XPUT 'localhost:9200/get-together/with_index/_mapping' -d '{
  "with_index": {
    "_index": { "enabled": true }
  }
}'
```

Then, if you add documents to this type and rerun the previous search, you should find your new documents.

The `_id`, `_type` and `_index` fields help you search in properties that define your documents: each of them belongs to a type in an index and has an ID. Next, we'll look at predefined fields that add new properties to your documents, such as their size.

### 3.4.3 Adding new properties to your documents

Using `_size`, you can store the size of the original JSON document that you're indexing. This is useful if the size of your documents might be a search criterion. For example, if you're indexing blog articles, you might be interested in those that are small enough to fit in a single page.

By default, `_size` is disabled, but in the following listing you'll enable it, by setting `enabled` to `true`, to get it indexed and be able to search for documents of a certain size. You'll also store it, by setting `store` to `yes`, so you can see the size of each document.

In the following listing, you'll enable and store the `_size` field, so you can search for it and show its contents.

#### **Listing 3.7 Indexing and storing the JSON document size with the `_size` field**

```
curl -XPUT localhost:9200/blog/sized/_mapping -d '{
  "sized": {
    "_size": {
      "enabled": true,           #A
      "store": "yes"            #B
    }
  }
}'
echo '{"title": "First post"}' > test_size.json          #C
du -b /tmp/test_size.json                                #C
curl -XPUT localhost:9200/blog/sized/1 --data-binary @test_size.json #D
curl 'localhost:9200/blog/_refresh'
curl 'localhost:9200/blog/sized/1?fields=_size'          #E
curl 'localhost:9200/blog/_search?q=_size:24'            #F
```

```
#A Makes it indexed, thus searchable
#B Makes it stored, thus retrievable
#C Stores the JSON in a file, then checks its size to ensure it's 24 bytes
#D Sends the file contents as data to index the document
#E Retrieving _size should show 24
#F Searching for 24-byte documents should match _size
```

### \_TIMESTAMP

Like `_size`, you can enable the `_timestamp` field to provide additional data for categorizing your documents. Enabling it makes Elasticsearch write the time when the document was indexed. By setting `enabled` to `true`, you can get timestamp indexed, so you can search on it afterward. This helps if you want to search only for freshly defined events, for example:

```
% curl -XPUT 'localhost:9200/get-together/timed_events/_mapping' -d '{
  "timed_events": {
    "_timestamp": { "enabled": true }
  }
}'
```

You can also set `store` to `yes`, to store the timestamp value and make it retrievable. As with `_id`, you can point `_timestamp` to extract the time from a field within your document by using the `path` option.

As with any date field, you can specify a format for `_timestamp`. In the following listing, you define a mapping for events where you enable `_timestamp` and make it point to the `date` field of events. Finally, you index a sample album and search for it using the `_timestamp` field.

### **Listing 3.8 Using the `_timestamp` field**

```
curl -XPUT 'localhost:9200/get-together/timed_events/_mapping' -d '{
  "timed_events": {
    "_timestamp": {
      "enabled": true,
      "path": "date"
    }
  }
}'
curl -XPUT 'localhost:9200/get-together/timed_events/1' -d '{
  "name": "Old Event",
  "date": "2009-06-22"
}'
curl 'localhost:9200/get-together/_refresh'
TIMED_EVENTS="localhost:9200/get-together/timed_events"
curl "$TIMED_EVENTS/_search?q=_timestamp=2009-06-22" #B
```

**#A Takes the value from a field in the document**  
**#B Searching for the same date string should match the document**

### **\_TTL FOR DOCUMENT LIFESPAN**

In some situations, you might need to delete documents automatically after a specified amount of time. Think about application logs or old events of your get-together site that might not be relevant.

That's what the `_ttl` field is for: you can enable it and specify an interval after which documents are automatically deleted. A sample mapping looks like this:

```
curl -XPUT 'localhost:9200/get-together/expiring/_mapping' -d '{
  "expiring": {
    "_ttl": {
      "enabled": "true",
      "default": "1d"
    }
  }
}'
```

You can specify the expiry period in the `default` field in milliseconds, but Elasticsearch automatically parses strings containing a number and a multiplier. For example, `1d` for one day in this example. For the multiplier, you can also use `s` (seconds), `m` (minutes) or `h` (hours).

The value in the `default` field can be overridden by specifying a `_ttl` value inside the document:

```
% curl -XPUT 'localhost:9200/get-together/expiring/1' -d '{
  {
    "name": "Fresh Event",
    "_ttl": "10d"
  }
}'
```

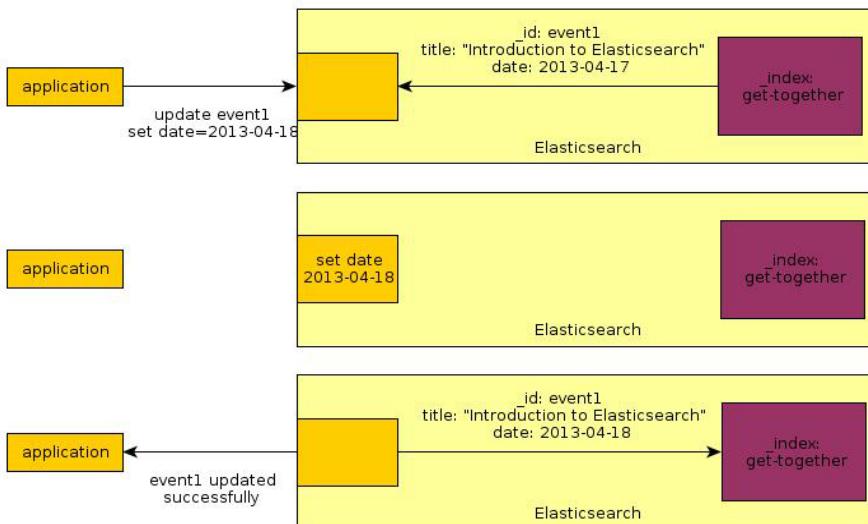
If you test this feature with a value such as `1s`, you may notice that documents aren't deleted immediately. That's because, by default, Elasticsearch looks for expired documents every minute, and then deletes them in bulk. To change the frequency of those searches, by change the value of `indices.ttl.interval` in the configuration file. You can also change the `indices.ttl.bulk_size` option to adjust the size of each bulk of expired documents that gets deleted at a time. Bulk processing allows you to send multiple operations to Elasticsearch with one request, making such operation faster. We talk about them in chapter 10, which is all about performance. And doing your index, update, and delete operations in bulk rather than one by one is one of the most important features for improving performance.

We've looked at how your documents are mapped in Elasticsearch so you can index them in a way that suits your use case. Next, we'll look at how you can modify documents that are already indexed.

## **3.5 Updating existing documents**

You may need to change an existing document for various reasons. Suppose you need to relocate a get-together event to a different venue. You could index a different document to the same "address" (index, type, and ID), but, as you might expect, you can update documents by sending the changes you want Elasticsearch to apply.

The update API in Elasticsearch allows you to send the changes you want to apply to a document, and the API returns a reply indicating whether the operation succeeded or not. The update process is shown in figure 3.4.



#A (top lane): Update request received; retrieve the existing document from the index

#B (middle lane): Make requested changes

#C (bottom lane): Reindex resulting document, and remove the old one

Figure 3.4 Updating a document involves retrieving it, processing it, reindexing it, and overwriting the previous document.

As figure 3.4 illustrates, Elasticsearch does the following (from the top down):

- *Retrieves the existing document*—For that to work, you must enable the `_source` field, otherwise Elasticsearch doesn't know what the original document looked like.
- *Applies the changes you specified*—For example, if your document was

```
{ "name": "Introduction to Elasticsearch", "date": "2013-04-17T19:00" }
```

and you wanted to shift the date forward by a day, the resulting document would be

```
{ "name": "Introduction to Elasticsearch", "date": "2013-04-18T19:00" }
```

- *Indexes the resulting document and removes the old one.*

In this section, we'll look at a few ways to use the update API and explore how to manage concurrency via Elasticsearch's versioning feature.

### 3.5.1 Using the update API

Let's look at how to update documents first. The update API exposes a few ways of doing that:

- *Send a partial document to add or replace the same part from the existing document.* This is straightforward: you send one or more fields with their values and, after the update is done, you expect to find them in the document.
- *When sending partial documents or scripts, make sure that the document is created if it doesn't exist.* You can specify the original contents of a document to be indexed if one isn't already there.
- *Send a script to update the document for you.* For example, in an online shop, you might want to increase the amount of T-shirts you have in stock by a certain amount, instead of setting it to a fixed number.

#### SENDING A PARTIAL DOCUMENT

The easiest way to update one or more fields is to send a partial document with the values you need to set for those fields. To do that, you need to send this info though an HTTP POST request to the `_update` endpoint of the document's URL. The following command will work after running `populate.sh` from the code samples:

```
% curl -XPOST 'localhost:9200/get-together/event/103/_update' -d '{
  "doc": {
    "date": "2013-04-18T19:00"
  }
}'
```

This sets the fields you specify under `doc` to the values you provide, regardless of the previous values or if these fields existed or not. If the entire document is missing, the update operation will fail, complaining that the document is missing.

**NOTE** When updating, you need to keep in mind that there might be conflicts. For example, if you're changing the event's date to the 18th of April, and a colleague changes it to the 19th of April, one of those updates will be overridden by the other one. To control this, you can use versioning, which will be covered later in this chapter.

#### CREATING DOCUMENTS THAT DON'T EXIST WITH UPSERT

To handle the situation when the updating document doesn't exist, you can use `upsert`. You might be familiar with this term from relational databases; the term is a portmanteau of `update` and `insert`.

If the document is missing, you can add an initial document to be indexed in the `upsert` section of the JSON. The previous command looks like this:

```
% curl -XPOST 'localhost:9200/get-together/event/103/_update' -d '{
  "doc": {
    "date": "2013-04-18T19:00"
  },
  "upsert": {
    "date": "2013-04-18T19:00"
  }
}'
```

```

    "name" : "Introduction to Elasticsearch",
    "date": "2013-04-18T19:00"
}
}

```

### UPDATING DOCUMENTS WITH A SCRIPT

Finally, let's look at how to update a document using the values from the previous example. Suppose you want to increment the price by 10. To do that, you use the same API, but instead of providing a document, you provide a *script*. A *script* is typically a piece of code in the JSON that you send to Elasticsearch, but it can also be an external script.

We'll talk more about scripting in chapter 6 because you'll most likely use scripts to make your searches more relevant. For now, let's look at three important elements of an update script:

- The default scripting language is *mvel* (<http://mvel.codehaus.org/>). Its syntax is similar to Java, but it's easier to use for scripting.
- Because updating gets the `_source` of an existing document, changes it, then reindexes the resulting document, your scripts alter fields within `_source`. To refer to `_source`, use `ctx._source`, and to refer to a specific field, use `ctx._source['field-name']`.
- If you need variables, it's recommended to define them separately from the script itself under `params`. That's because scripts need to be compiled, and once they're compiled, they get cached. Running the same script multiple times with different parameters requires the script to be compiled only once. Subsequent runs take the existing script from cache. This is faster than having different scripts because they all need compilation.

In the following listing, let's use an mvel script to increment the price of an Elasticsearch shirt by 10.

#### **Listing 3.9 Updating with a script**

```
% curl -XPUT 'localhost:9200/online-shop/shirts/1' -d '
{
  "caption": "Learning Elasticsearch",
  "price": 15
}'
% curl -XPOST 'localhost:9200/online-shop/shirts/1/_update' -d '{
  "script": "ctx._source.price += price_diff",           #A
  "params": {
    "price_diff": 10                                     #B
  }                                                       #B
}'

```

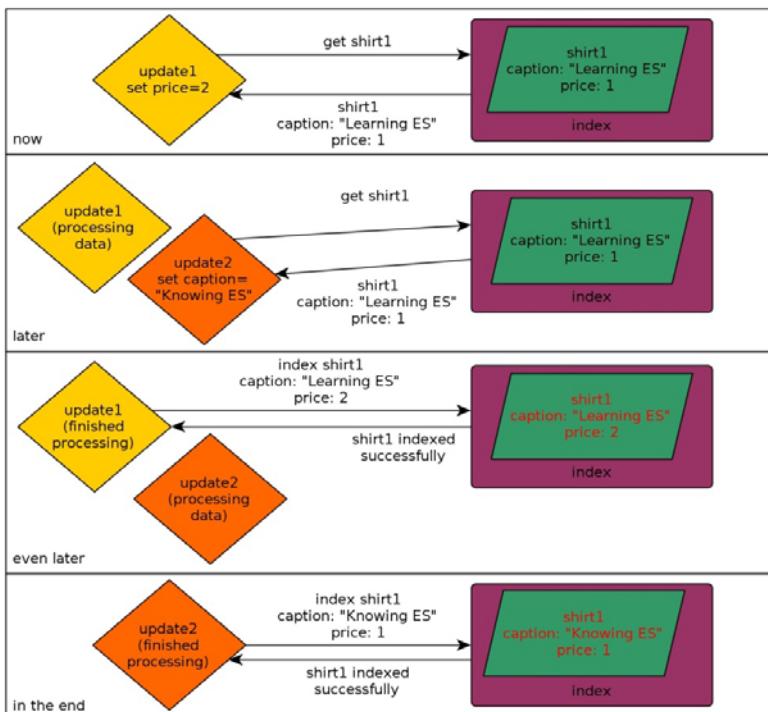
#A mvel script increments the price field with the value from the `price_diff` variable  
#B Optional params section for assigning values to variables used in the script

You can see that `ctx._source.price` was used instead of the expected `ctx._source['price']`. This is an alternative way to refer to the price field. It's easier to use with curl because escaping single quotes in shell scripts can be confusing.

Now that you've seen how you can update a document, let's look at how you can manage concurrency if multiple updates happen at the same time.

### 3.5.2 Implementing concurrency control through versioning

If multiple processes or threads are working with Elasticsearch at the same time, you could encounter concurrency issues. As illustrated in figure 3.5, if two updates run at the same time, it's possible that one of the processes reindexes the document between the time when the other process got the original document and applied its own changes. With no concurrency control, the second reindex will cancel the changes of the first update.



- #A (top lane) Update1 retrieves existing document (shirt1)
- #B (second lane) While update1 is applying changes, concurrent update2 is retrieving the document
- #C (third lane) Update1 finishes indexing the initial document with its changes
- #D (last lane) Without concurrency control, update2 is unaware of the update1 changes and overrides them

Figure 3.5 Without concurrency control, changes can get lost.

Fortunately, Elasticsearch supports concurrency control by using a version number for each document. The initially indexed document is version 1. When you reindex it, though an update or another index operation with the same ID, the version number is incremented.

To see how it works, let's replicate a process similar to the one shown in figure 3.5 using the code in listing 3.10:

- You index a document, and then update it (update1).
- Update1 starts in background and includes a waiting time (sleep).
- During that sleep, you issue another update (update2) command that modifies the document. This change occurs between update1's fetch of the original document and its reindexing operation.
- Instead of canceling the changes of update2, update1 fails, because the document is already at version 2.

### **Listing 3.10 Two concurrent updates managed with versioning**

```
% curl -XPOST 'localhost:9200/online-shop/shirts/1/_update' -d '{
  "script": "Thread.sleep(10000); ctx._source.price = 2"          #A
}' &
% curl -XPOST 'localhost:9200/online-shop/shirts/1/_update' -d '{
  "script": "ctx._source.caption = \"Knowing Elasticsearch\""
}'
```

#A Update1 waits 10 seconds and goes to background (&  
#B If update2 runs within 10 seconds, it forces update1 to fail because it increments the version number

Figure 3.6 is a graphical representation of what happens in this code.

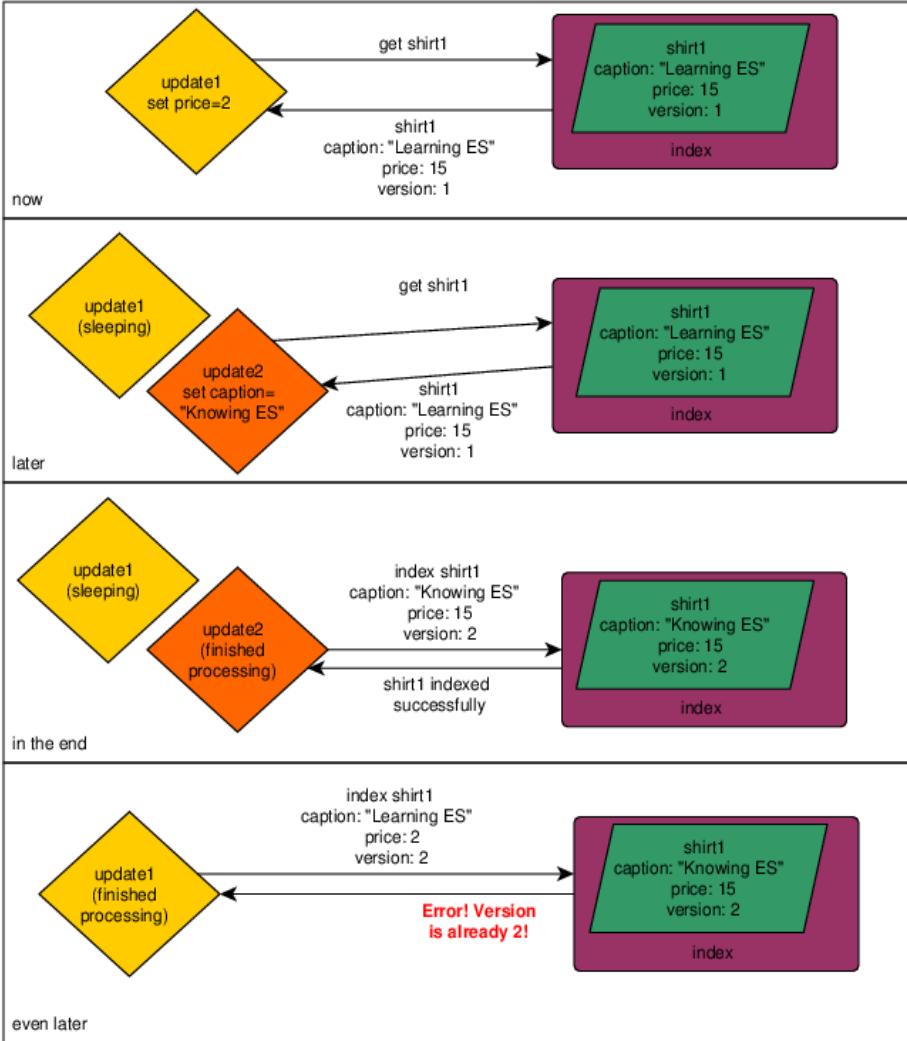


Figure 3.6 Concurrency control through versioning prevents one update from overriding another.

This kind of concurrency control is called *optimistic* because it allows parallel operations and assumes that conflicts appear rarely, throwing errors when they do appear. This is opposed to *pessimistic* locking, in which conflicts are prevented in the first place by blocking operations that might cause conflicts.

### AUTOMATICALLY RETRYING AN UPDATE WHEN THERE'S A CONFLICT

When a version conflict appears, you can deal with it in your own application. If it's an update, you can try applying it again. But you can also make Elasticsearch reapply it for you automatically by setting the `retry_on_conflict` parameter:

```
% $SHIRTS="localhost:9200/online-shop/shirts"
% curl -XPOST "$SHIRTS/_update?retry_on_conflict=3" -d '{
  "script": "ctx._source.price = 2"
}'
```

### USING VERSIONS WHEN YOU INDEX DOCUMENTS

Another way to update a document, without using the update API, is to index a new one to the same index, type, and ID. This overwrites the existing document, and you can still use the `version` field for concurrency control. To do that, set the `version` parameter in the HTTP request. The value should be the version you expect the document to have. For example, if you want to index a new T-shirt and also make sure you don't override anything by accident, set `version=0`:

```
% curl -XPUT 'localhost:9200/online-shop/shirts/2?version=0' -d '{
  "caption": "Learning Elasticsearch Versioning",
  "price": 2
}'
```

Elasticsearch throws an error if the document exists because the document has a version other than 0. For other such operations, specify the version you expect the document to have:

```
% curl -XPUT 'localhost:9200/online-shop/shirts/2?version=1' -d '{
  "caption": "I Know about Elasticsearch Versioning",
  "price": 5
}'
```

Again, the operation fails if the current version is different than 1—whether the document doesn't exist or it has a higher version number.

With versions, you can index or update your documents safely. Next, let's look at how you can remove documents.

## 3.6 Deleting data

Now that you know how to send data to Elasticsearch, let's look at what options you have for removing some of what was indexed. If you've worked through the listings throughout this chapter, you now have unnecessary data that's waiting to be removed. We'll look at a few ways to remove data—or at least get it out of the way of slowing down your searches or further indexing:

- *Delete individual documents or groups of documents.* When you do that, Elasticsearch only marks them to be deleted, so they don't show up in searches, and gets them out of the index later, in an asynchronous manner.
- *Delete complete indices.* This is a particular case of deleting groups of documents. But it differs in the sense that it's easy to do performance-wise. The main job is to remove

- all the files associated with that index, which happens almost instantly.
- *Close indices*. Although this isn't about removing, it's worth mentioning here. A closed index doesn't allow read or write operations, and its data isn't loaded in memory. It's similar to removing data from Elasticsearch, but it remains on disk, and it's easy to restore ; you open the closed index.

### 3.6.1 Deleting documents

There are a few ways to remove individual documents, and we'll discuss most of them here:

- *Remove a single document by its ID*. This is good if you have only one document to delete, provided that you know its ID.
- *Remove multiple documents in a single request*. If you have multiple individual documents that you want to delete, you can remove them all at once in a bulk request, which is faster than removing one document at a time. We'll cover bulk deletes in chapter 12, along with bulk indexing and bulk updating.
- *Remove a mapping type, with all the documents in it*. This will effectively search and remove all the documents you've indexed in that type, plus the mapping itself.
- *Remove all the documents matching a query*. This is similar to removing a mapping type, in the sense that internally, a search is run to identify the documents that need to be deleted. Only here you can specify any query you want, and the matching documents will be deleted.

#### REMOVE A SINGLE DOCUMENT

To remove a single document, you need to send an HTTP `DELETE` request to its URL. For example:

```
% curl -XDELETE 'localhost:9200/online-shop/shirts/1'
```

**TIP** You've used *versions* for indexing and updating, and you can use it for deletes to manage concurrency. For example, let's assume you sold all shirts of a certain type, and you want to remove that document so it doesn't appear in searches at all. But you might not know at that time if a new transport arrived and the stock data has been updated. To accomplish this, add `version=1` as a parameter to your `DELETE` request. The document will be deleted only if it's at version 1.

#### REMOVE A MAPPING TYPE AND DOCUMENTS MATCHING A QUERY

You can also remove an entire mapping type, which removes the mapping itself, plus all the documents indexed in that type. To do that, you provide the type's URL to the `DELETE` request:

```
% curl -XDELETE 'localhost:9200/online-shop/shirts'
```

The tricky part about removing types is that the type name is treated like another field in the documents. All documents of an index end up in the same shards regardless of the

mapping type they belong to. When you issue the previous command, Elasticsearch has to query for documents of that type, and then remove them. This is an important detail when it comes to performance for removing types versus removing complete indices because removing types typically takes longer and uses more resources.

In the same way you can query for all documents within a type and delete them, Elasticsearch allows you to specify your own query for documents you want to delete through an API called *delete by query*. Using the API is similar to running a query, except that the HTTP request is `DELETE`, and the `_search` endpoint is now `_query`. For example, to remove all documents that match “Elasticsearch” from the index `get-together`, you can run this command:

```
% curl -XDELETE 'localhost:9200/get-together/_query?q=elasticsearch'
```

Similar to regular queries, which we cover in more detail in chapter 4, you can run a delete by query on a specific type, on multiple types, everywhere in an index, in multiple indices or in all indices. When you search in all indices, be careful when you run a delete by query.

**TIP** Besides being careful, you can use backups. We talk about backups in chapter 14, which is all about administration.

### 3.6.2 Deleting indices

As you might expect, to delete an index, issue a `DELETE` request to the URL of that index:

```
% curl -XDELETE 'localhost:9200/get-together/'
```

You can also delete multiple indices, by providing a comma-separated list, or even delete all indices by providing `_all` as the index name.

Deleting an index is fast because it’s mostly about removing the files associated to all shards of that index. And deleting files on the file system happens fast. This is opposed to when you delete individual documents. When you do that, they’re only marked as deleted. They get removed when segments are merged. *Merging* is the process of combining multiple small Lucene segments into a bigger segment.

#### On segments and merging

A *segment* is a chunk of the Lucene index (or a shard, in Elasticsearch terminology), which is created when you’re indexing. Segments are never appended—only new ones are created as you index new documents. Data is never removed from them because deleting only marks documents as deleted. Finally, data never changes because updating documents implies reindexing.

When Elasticsearch is performing a query on a shard, Lucene has to query all its segments, merge the results, and send them back—much like the process of querying multiple shards within an index. As with shards, the more segments you have to go through, the slower the search.

As you may imagine, normal indexing operations create many such small segments. To avoid having an extremely large number of segments in an index, Lucene merges them from time to time.

Merging some documents implies reading their contents, excluding the deleted documents, and creating new and bigger segments with their combined content. This process requires resources—specifically, CPU and disk I/O. Fortunately, merges run asynchronously, and Elasticsearch lets you configure numerous options around them. We talk more about those options in chapter 12, where you learn how to improve the performance of index, update, and delete operations.

### **3.6.3 Closing indices**

Instead of deleting indices, you also have the option of closing them. If you close an index, you won't be able to read or write data from it with Elasticsearch until you open it again. This is useful when you have flowing data, such as application logs. You'll learn in chapter 12 that it's a good idea to store such flowing data in time-based indices, for example, creating one index per day.

In an ideal world, you'd hold application logs forever, in case you needed to look back a long time ago. On the other hand, having a large amount of data in Elasticsearch demands increased resources. For this use case, it makes sense to close "old" indices. You're unlikely to need that data, but you don't want to remove it, either.

To close the get-together index, send an HTTP POST request to its URL at the `_close` endpoint:

```
% curl -XPOST 'localhost:9200/get-together/_close'
```

To open it again, you run a similar command, only the endpoint becomes `_open`:

```
% curl -XPOST localhost:9200/get-together/_open
```

Once you close an index, the only trace of it in Elasticsearch's memory is its metadata, such as name, and where shards are located. If you have enough disk space and you're not sure whether you'll need to search in that data again, closing indices is better than removing them. Closing them gives you the peace of mind that you can always reopen a closed index and search in it again.

### **3.6.4 Reindexing sample documents**

In chapter 2, you used the book's code samples (<https://github.com/dakrone/elasticsearch-in-action>) to index documents. Running `populate.sh` from these code samples removes the get-together index you created in this chapter, and reindexes the sample documents.

If you look at both the `populate.sh` script and the mapping definition from `mapping.json`, you'll recognize various types of fields we discussed in this chapter.

Some of the mapping and indexing options, such as the analysis settings, are dealt with in upcoming chapters. For now, run `populate.sh` to prepare the get-together index for chapter 4, which is all about searches. The code samples provide you with sample data to search on.

### 3.7 *Summary*

- Mappings let you define fields in your documents and how those fields are indexed. We say Elasticsearch is schema-free because mappings are extended automatically, but in production you often need to take control over what is indexed, what is stored and how.
- Most fields in your documents are core types, such as strings and numbers. The way you index those fields has a big impact on how Elasticsearch performs and how relevant your search results are. For example, the analysis settings, which we cover in chapter 5.
- A single field can also be a container for multiple fields or values. We looked at arrays and multi fields, which let you have multiple occurrences of the same core type in the same field.
- Besides the fields that are specific to your documents, Elasticsearch provides predefined fields, such as `_source` and `_timestamp`. Configuring these fields changes some data that you don't explicitly provide in your documents but has a big impact on both performance and functionality. For example, you can decide whether you want the original document to be stored or not.
- Because Elasticsearch stores data in Lucene segments that don't change once they're created, updating a document implies retrieving the existing one, putting the changes in a new document that gets indexed, and marking the old one as deleted.
- The removal of documents happens when the Lucene segments are asynchronously merged. This is also why deleting an entire index is faster than removing one or more individual documents from it—it only implies removing files on disk with no merging.
- Throughout indexing, updating, and deleting, you can use document versions to manage concurrency issues. With updates, you can tell Elasticsearch to retry automatically if an update fails because of a concurrency issue.

# 4

## *Searching your data*

### ***This chapter covers***

- The structure of an Elasticsearch query and response
- Working with Elasticsearch filters and how they differ from queries
- Filter bitsets and caching
- Using queries and filters that Elasticsearch supports

Now that we've explored how you get data into Elasticsearch, let's cover how you get data out of Elasticsearch: by searching. After all, what good is indexing your data into a search engine if you can't search through it? Fortunately, Elasticsearch provides a rich API for searching through data, running the gamut of Lucene's search capability. Because of the format Elasticsearch allows for constructing queries, there are limitless possibilities for how queries can be built. The best way to tell which query to use for your data is to experiment, so don't be afraid to try out each query on your project's data to figure out which one suits your needs best.

### **Searchable data**

In this chapter, you'll again use the dataset formed around the get-together website we've touched on in previous examples. This dataset contains two different types of documents: groups and events. To follow along and perform the queries yourself, download and run the `populate.sh` script to populate an Elasticsearch index.

To download the script, see the source code for the book at  
<https://github.com/dakrone/elasticsearch-in-action>

To start off, we discuss the things common to all queries and results so you'll have an understanding of what a query and the result of a query look like in general. We then move on to discussing filters and how they differ from queries followed by a look at some of the most commonly used filters and queries. If you're wondering about the details of how Elasticsearch calculates the score for documents, don't worry, we discuss that in chapter 7, where we talk about searching with relevancy. Finally, we provide a quick-and-dirty guide to help you choose which type of query to use for a particular application. Make sure to check it out if there seem to be too many types of queries to keep straight!

Before we start, let's revisit what happens when you perform a search in Elasticsearch (see figure 4.1). The REST API search request is first sent to the node you choose to connect to, which, in turn, sends the search request to all shards (either primary or replica) for the index or indices being queried. When enough information has been collected from all shards to sort and rank results, only the shards containing the document content that will be returned are asked to return the relevant content.

Figure 4.1 How a search request is routed

This search routing behavior is configurable; the default behavior is shown in figure 4.1. We look at how to change it later in this chapter. For now, let's look at the basic structure that all Elasticsearch queries share.

## 4.1 Structure of a query

Elasticsearch queries are JSON requests that get sent to the server, and because all queries follow the same format, it's helpful to understand the things that can be changed for each query.

### 4.1.1 Specifying a search scope

All REST search requests use the `_search` REST endpoint and can be either a GET request or a POST request. A search can be limited to a number of indices, types, or the entire cluster by specifying the names of the indices or types with the URL. The following listing provides example search URLs that limit the scope of searches.

#### **Listing 4.1 Limiting the search scope in the URL**

```
% curl 'localhost:9200/_search' -d '....' #A
% curl 'localhost:9200/get-together/_search' -d '....' #B
% curl 'localhost:9200/get-together/event/_search' -d '....' #C
% curl 'localhost:9200/_all/event/_search' -d '....' #D
% curl 'localhost:9200/*/event/_search' -d '....' #D
% curl 'localhost:9200/get-together,other/event/group/_search' -d '....' #E
% curl 'localhost:9200/+get-together,-get-together/_search' -d '....' #F
```

**#A** Searches the entire cluster

**#B** Searches the get-together index

**#C** Searches the event type in the get-together index

**#D** Searches all event types in all indices

**#E** Searches the event and group types in the get-together and other indices

**#F** Searches all indices that start with get-toge, but not the get-together index

For the best performance, limit your queries to the smallest number of indices and types possible because anything Elasticsearch doesn't have to search means faster responses.

### 4.1.2 Specifying the body of the query

Once you've selected the indices to search, the next step is to determine the body, or the search part, of the query. Before we get into the different queries that Elasticsearch supports, let's talk about the four elements that most queries share:

- *From*—Specifies the result number to start from
- *Size*—Specifies page size
- *Fields*—Specifies one or more fields that should be returned with the results
- *Sort*—Specifies how to sort results

Together, these make up the *from*, *size*, *fields*, and *sort* keys in a JSON query.

#### **RESULTS START AND PAGE SIZE**

The aptly named *from* and *size* fields are sent to specify the offset to start results from and the size of each “page” of results. For example, if you send a *from* value of 7 and a *size* of 5, Elasticsearch will send the 8th, 9th, 10th, 11th, and 12th results back (because the *from* parameter starts at 0, specifying 7 starts at the 8th result). If these two parameters aren't sent, Elasticsearch defaults to starting at the first result (the “zero-th”), and sends 10 results with the response.

You can either send these parameters with the body of the request or specify them as URL parameters. Both methods are shown in the following listing, which searches for the second page of the get-together index where the organizer is "Lee".

#### **Listing 4.2 Paginating results using from and size**

```
% curl 'localhost:9200/get-together/_search' -d '{
  "query": {
    "match": {
      "organizer": "Lee"
    }
  },
  "from": 10,                                     #A
  "size": 10                                      #B
}'
```

```
% curl 'localhost:9200/get-together/_search?from=10&size=10' -d '{
  "query": {
    "match": {
      "organizer": "Lee"
    }
  }
}'
```

**#A Returns results starting from the tenth result**

**#B Returns a total of 10 results**

**#C The same request with from and size sent as parameters in the URL instead of the request body**

Other than noticing the "query" section, which is an object in every query, don't worry about the "match" section yet, we talk about it in section 4.3.

#### **FIELDS RETURNED WITH RESULTS**

The next element that all queries share is the list of fields Elasticsearch should return for each matching document. This is specified by sending the `fields` option with the request (either as a field in the JSON search request or as a URL parameter). If no fields are specified with the request, Elasticsearch returns either the entire `_source` of the document by default, or, if the `_source` isn't stored, only the metadata about the matching document: `_id`, `_type`, `_index`, and `_score`.

Here the previous query is used, returning the name and description fields of each matching group:

```
% curl 'localhost:9200/get-together/_search' -d '{
  "query": {
    "match": {
      "organizer": "Lee"
    }
  },
  "fields": [ "name", "description" ]                #A
}'
```

**#A Returns the name and description fields with the search response**

## SORT ORDER FOR RESULTS

The last element most searches include is the `sort` order for the results. If no sort order is specified, Elasticsearch returns matching documents sorted by the `_score` value descending, with the most relevant (highest scoring) documents first.

To sort fields in either ascending or descending order, specify a map instead of a field. You can sort on any number of fields by specifying a list of fields or field maps in the `sort` value. For example, using the previous organizer search, you can return results sorted first by creation date, starting with the oldest; then by the name of the get-together group, in reverse alphabetical order; and finally by the `_score` of the result, as shown in the following listing.

### Listing 4.3 Results sorted by date (ascending), name (descending), and `_score`

```
% curl 'localhost:9200/get-together/_search' -d '{
  "query": {
    "match": {
      "organizer": "Lee"
    }
  },
  "sort": [
    {"created_on": "asc"},          #A
    {"name": "desc"},               #B
    {"_score"}                      #C
  ]
}'
```

#A Sorts first by the creation date, starting from the oldest to newest  
#B Then sorts by name of the group, in reverse alphabetical order  
#C And, finally, sorts by the relevancy of the result (its `_score`)

To specify the sort order as parameters in the URL, specify multiple sort order fields in a comma-separated list in the `sort` parameter. For example, to specify the previous search with the fields as part of the URL, you'd use:

```
% curl 'localhost:9200/_search?sort=created_on:asc,name:desc,_score' ...
```

This is handy when testing things at the command line, but in most cases you'll want to send the `sort` option (as well as the `from`, `size`, and `fields` options) as part of the request.

## THE FOUR ELEMENTS IN ACTION

Now that we've covered the four query elements, here's an example of a query that uses them all.

### Listing 4.4 Query with all four elements: scope, pagination, fields, and sort order

```
% curl 'localhost:9200/get-together/group/_search' -d '{
  "query": {
    "match_all": {}
  },
  "from": 0,                         #A
  "size": 10,                         #B
  "fields": ["name", "organizer", "description"], #C
}'
```

```

    "sort": [{"created_on": "desc"}]                                #D
}

#A Starts from the first (zeroth) result
#B Returns a total of 10 results
#C Includes the name of the group, the organizer, and description of the group
#D Sorts by the created_on field, descending

```

Next, let's look at the structure of the search response.

#### 4.1.3 Understanding the structure of a response

Let's look at an example search and see what the response looks like. The next listing searches for groups about "elasticsearch".

##### **Listing 4.5 Example search request and response**

```
% curl 'localhost:9200/_search?q=elasticsearch&fields=_source,name,tags'
{
  "_shards": {
    "failed": 0,                                              #A
    "successful": 2,                                         #A
    "total": 2                                               #A
  },
  "hits": {
    "hits": [
      {
        "_id": "3",                                            #C
        "_index": "get-together",                               #D
        "_score": 0.9066504,                                    #E
        "_source": {
          "created_on": "2012-08-07",                         #F
          "description": "Elasticsearch group for ES users of all knowledge
levels",                                                 #F
          "location": "San Francisco, California, USA",       #F
          "members": [
            "Lee",                                              #F
            "Igor"                                              #F
          ],
          "name": "Elasticsearch San Francisco",               #F
          "organizer": "Mik",                                   #F
          "tags": [
            "Elasticsearch",                                     #F
            "big data",                                         #F
            "lucene",                                           #F
            "open source"                                       #F
          ]
        },
        "_type": "group",                                      #G
        "fields": {
          "name": "Elasticsearch San Francisco",             #H
          "tags": [
            "Elasticsearch",                                     #H
            "big data",                                         #H
            "lucene",                                           #H
            "open source"                                       #H
          ]
        }
      }
    ]
  }
}
```

```

        }
    },
    {
        ...
    },
    "max_score": 0.9066504, #I
    "total": 2 #J
},
"timed_out": false,
"took": 5 #K
}

```

#A The number of shards that responded to this request, successfully or unsuccessfully

#B The response contains a hits key that contains a hits array

#C The ID of the result document

#D Index of the result document

#E Relevancy score for this result

#F Source of the document is returned if \_source is stored and specified in the fields parameter

#G Elasticsearch type of the result document

#H Other fields that were requested (name and tags in this example)

#I Maximum score of all documents for this search

#J Total number of matching results for the search

#K Time the request took, in milliseconds

Usually, you wouldn't ask for both the `fields` and the `_source` together because you can easily get any field from the `_source` map, but to show how the response comes back, we asked for both in this case. Remember that if you don't store either the `_source` of the document or the `fields`, you won't be able to retrieve the value from Elasticsearch!

Now that you're familiar with the common elements of a search, there's one more topic we need to talk about before getting to types of searches, and that's filters.

## 4.2 Working with filters

Filters are similar to the queries we discuss in this chapter, but they differ in how they affect the scoring and performance of many searching actions. Rather than computing the score for a particular term as queries do, a filter on a search is a simple binary "does this document match this query" yes-or-no answer. Figure 4.2 shows the main difference between queries and filters.

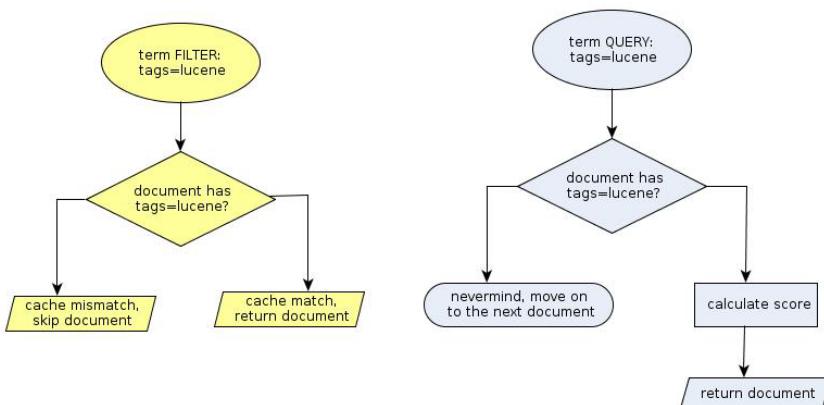


Figure 4.2 Filters require less processing and are cacheable because they don't calculate the score.

Because of this difference, filters can be faster than using a regular query, and they can also be cacheable. A query using a filter looks similar to a regular query, but the query is replaced with a "filtered" map that contains the original query and a filter to be applied, as shown in the next listing.

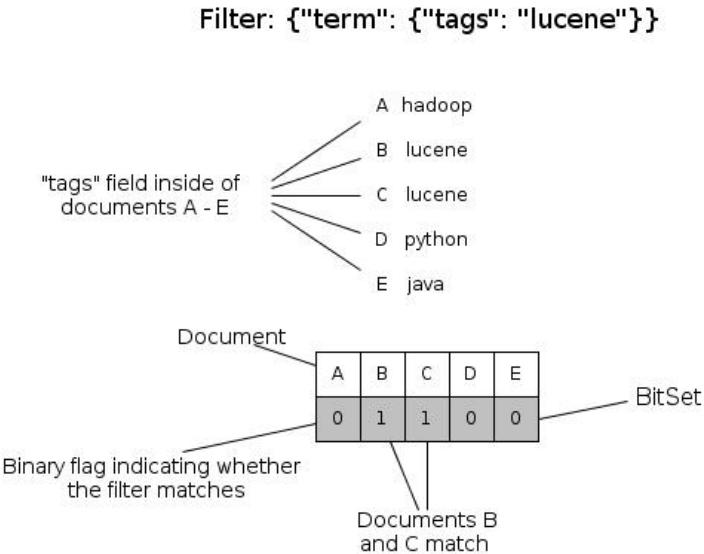
#### **Listing 4.6 Query using a filter**

```
% curl 'localhost:9200/get-together/group/_search' -d'
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "description": "search"
        }
      },
      "filter": {
        "term": {
          "tags": "lucene"
        }
      }
    }
  }
}
```

#A Query type, which, in this case, specifies a query with a filter attached  
#B The query searches for groups with “search” in the description  
#C The additional filter limits the query to documents that have the tag lucene

Here, a regular query for groups matching “search” is used as the query, but in addition to the query for the word “search”, a filter is used to limit the documents. Inside this particular filter section, a `term` filter is being applied for all documents that have the tag `lucene`. Behind the scenes, Elasticsearch constructs what is called a *bitset*, which is a binary set of bits

denoting whether the document matches this filter. Figure 4.3 shows what this bitset looks like.



**Figure 4.3 Filter results are cached in bitsets making subsequent runs much faster.**

After constructing the bitset, Elasticsearch can now use it to filter (hence the name!) out the documents that it shouldn't be searching based on the query part of the search. Because of this, filtering can be much faster than combining the entire query into a single search. Depending on what kind of filter is used also, Elasticsearch can cache the results in a bitset, so if the filter is used for another search, it doesn't have to be calculated! All of this translates into faster searches with filters. Therefore, make parts of your query into filters if you can.

We'll revisit bitsets to explain the details of how they work and how they affect performance in chapter 11, which discusses how to speed up searches.

**NOTE** How do you determine what should be a filter and what shouldn't? If the part of the query in question needs to affect the score (relevancy) of the result, it should be part of the regular query. Otherwise, it should be a filter.

### 4.2.1 Filter caching

As you learned, filters allow Elasticsearch to generate a bitset for the documents that match, which can be then cached. Some of these bitsets are automatically cached for you by default, when specific filters are used. Other types of filters aren't automatically cached. Additionally, Elasticsearch gives you the ability to manually specify whether a filter should be cached.

## FILTERS CACHED BY DEFAULT

The previous filter is a term filter, which filters based on a specific term in the document. The `term`, `terms`, `prefix`, and `range` filters are all cached by default.

## FILTERS NOT CACHED BY DEFAULT

The types of filters not cached by default include the following:

- any of the geo filters
- the `numeric_range` filter
- filters that use custom scripts

Additionally, filters can be combined with the `bool`, `and`, `or` and `not` filters, which aren't themselves cached, but each individual part may or may not be cached depending on its type.

To manually specify a filter that should be cached, set the `_cache` option to `true` when sending the filter, similar to the following:

```
% curl 'localhost:9200/get-together/group/_search' -d'
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "description": "search"
        }
      },
      "filter": {
        "script": {
          "script": "doc[\"tags\"].values.length > 2",
          "_cache": true
        }
      }
    }
  }
}
```

**#A Setting the `_cache` option to true tells Elasticsearch to cache this filter**

If the filter were to be gigantic, say, thousands of terms in a terms filter, you could also specify the `_cache_key` key at the same level as the `_cache` key, which is the key Elasticsearch should store the cache under. Normally Elasticsearch stores the cache with the same name as the contents of the filter (something like "terms filter for tags 'search,' 'lucene,' 'elasticsearch,' and so on"), which means a filter with a large number of terms could take up more memory than is needed. We discuss the `_cache` and `_cache_key` options in chapter 13 when we talk about getting more performance out of your queries.

Now that you understand what filters are, we'll cover several different types of filters and queries as well as run some searches against data.

## 4.3 Working with match and filter queries

Although there are a number of ways to query for things in Elasticsearch, some may be better than others, depending on how the data is stored in your index. In this section, you learn the

different types of queries Elasticsearch supports and try out an example of how to use each query. We assess the pros and cons of using each query, and provide any performance notes about each one so you can determine which query best fits your data.

Elasticsearch has queries ranging from the more basic, such as the `term` and `prefix` queries, to the more complex, such as the `query_string` and `match_phrase` queries. Elasticsearch also allows you to combine an arbitrary number of queries using the `bool` query by itself as well as with the `and`, `or`, and `not` filters.

You've already seen the `filtered` query type, and most (although not all!) queries also have a filter equivalent. We'll mention that for each query type as we go, but keep in mind that queries can be nested depending on whether you're using filters.

Let's start with the easiest queries, beginning with the `match_all` query.

### 4.3.1 Match\_all query

I'll give you a guess as to what this query does. That's correct! It matches all documents. The `match_all` query is useful when you want to use a filter instead of a query (perhaps if you don't care about the score of documents at all) or you want to return all documents among the indices and types you're searching. The query looks like this:

```
% curl 'localhost:9200/_search' -d '
{
  "query" : {
    "match_all" : {}
  }
}'
```

To use a filter for a search instead of any regular query parts, the query looks something like this (with the filters omitted):

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        ... filter details ...
      }
    }
  }
}'
```

Simple, huh? Not too useful, though, for a search engine, because users rarely search for everything. Let's look at a query that's a bit more useful, next.

### 4.3.2 Query\_string query

In chapter 2, you used the `query_string` query to see how easy it is to get an Elasticsearch server up and running, but we'll cover it again here in its entirety so you can see how it compares to the other queries.

As shown in the following listing, a `query_string` search can be performed either from the URL of the request or sent in a request body. In this example, you search for documents that contain "nosql".

#### **Listing 4.7 Example `query_string` search**

```
% curl -XGET 'localhost:9200/_search?q=nosql&pretty'                                #A
{
  "_shards": {
    "failed": 0,
    "successful": 2,
    "total": 2
  },
  "hits": {
    "hits": [
      {
        "_id": "4",
        "_index": "get-together",
        "_score": 0.3728585,
        "_source": {
          "created_on": "2010-04-02",
          "description": "Come learn and share your experience with nosql &
big data technologies, no experience required",
          "location": "Boulder, Colorado, USA",
          "members": [
            "Greg",
            "Bill"
          ],
          "name": "Boulder/Denver big data get-together",
          "organizer": "Andy",
          "tags": [
            "big data",
            "data visualization",
            "open source",
            "cloud computing",
            "hadoop"
          ]
        },
        "_type": "group"
      }
    ],
    "max_score": 0.3728585,
    "total": 1
  },
  "timed_out": false,
  "took": 2
}

% curl -XPOST "http://localhost:9200/_search?pretty" -d'
{
  "query" : {
    "query_string" : {
      "query" : "nosql"
    }
  }
}
{
  "took" : 135,
```

```

    "timed_out" : false,
    "_shards" : {
      "total" : 2,
      "successful" : 2,
      "failed" : 0
    },
    "hits" : {
      "total" : 1,
      "max_score" : 0.3728585,
      "hits" : [ ... exactly the same results ... ]
    }
}

```

**#A A query\_string search sent as a URL parameter**

**#B The same query\_string search sent as the body of a request**

By default, a `query_string` query searches the `_all` field, which, if you recall from chapter 3, is made up of all the fields combined together. This can be changed by either specifying a field with the query, such as `description:nosql`, or by specifying a `default_field` with the request, as seen in this next listing.

#### Listing 4.8 Specifying a default\_field for a query\_string search

```
% curl -XPOST 'localhost:9200/_search' -d'
{
  "query" : {
    "query_string" : {
      "default_field" : "description",          #A
      "query" : "nosql"                         #A
    }
  }
}
```

**#A Because no field is specified in the query, the default field (description) is used**

As you may have guessed, this syntax offers more than searching for a single word. Under the hood, this is the entire Lucene query syntax, which allows combining searching different terms with Boolean operators like `AND` and `OR`, as well as excluding documents from the results using the minus sign (`-`) operator. The following query searches for all groups with “nosql” in the name but without “mongodb” in the description:

```
name:nosql AND -description:mongodb;
```

To search for all search and lucene groups created between 1999 and 2001, you could use with the following:

```
(tags:search OR tags:lucene) AND created_on:[1999-01-01 TO 2001-01-01]
```

**NOTE** Refer to <http://www.luceneTutorial.com/lucene-query-syntax.html> for a full example of syntax the `query_string` query supports.

## QUERY\_STRING CAUTIONS

Although the `query_string` query is one of the most powerful queries available to you in Elasticsearch, it can sometimes be one of the hardest to read and easily extend. It may be tempting to allow your users the ability to specify their own queries with this syntax, but consider the difficulty in explaining the meaning of complex queries such as this:

```
name:search^2 AND (tags:ucene OR tags:"big data"~2) AND -description:analytics
AND created_on:[2006-05-01 TO 2007-03-29]
```

Suggested replacements for the `query_string` query include the `term`, `terms`, `match`, or `multi_match` queries, all of which allow you to search for strings within a field or fields in a document.

### 4.3.3 Term query

A `term` query and filter are some of the simplest queries that can be performed, allowing you to specify a field and term to search for within your documents. Note that because the term being searched for isn't analyzed, it must match a term in the document *exactly* for the result to be found. We'll cover how exactly *tokens*, which are individual pieces of text indexed by Elasticsearch, get analyzed in chapter 5. If you're familiar with Lucene, it might be helpful to know that the `term` query maps directly to Lucene's `TermQuery`.

The following listing shows a `term` query that searches for groups with the `elasticsearch` tag.

#### Listing 4.9 Example `term` query

```
% curl 'localhost:9200/get-together/group/_search' -d'
{
  "query": {
    "term": {
      "tags": "elasticsearch"
    }
  },
  "fields": [ "name", "tags" ]
}
{
  ...
  "hits": [
    {
      "_id": "3",
      "_index": "get-together",
      "_score": 1.0769258,
      "_type": "group",
      "fields": {
        "name": "Elasticsearch San Francisco",
        "tags": [
          "Elasticsearch", #A
          "big data",
          "ucene",
          "open source"
        ]
      }
    },
  ],
}
```

```
{
  {
    "_id": "2",
    "_index": "get-together",
    "_score": 0.8948604,
    "_type": "group",
    "fields": {
      "name": "Elasticsearch Denver",
      "tags": [
        "denver",
        "elasticsearch",
        "big data",
        "lucene",
        "solr"
      ]
    }
  },
  ...
}
```

**#A Because these two results contain the word elasticsearch in the tags, they're returned**

Like the `term` query, a `term` filter can be used when you want to limit the results to documents that contain the term, but without affecting the score. Compare the scores of the documents in the previous listing with the scores in the following listing: you'll notice that the filter doesn't bother calculating any score, setting all to 1.0:

#### Listing 4.10 Example term filter

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "term": {
          "tags": "elasticsearch"
        }
      }
    },
    "fields": [ "name", "tags" ]
  }
}

...
  "hits": [
  {
    "_id": "3",
    "_index": "get-together",
    "_score": 1.0,
    "_type": "group",
    "fields": {
      "name": "Elasticsearch San Francisco",
      "tags": [
        "Elasticsearch",
        "#B
        "elasticsearch"
      ]
    }
  }
]
```

```

        "big data",
        "lucene",
        "open source"
    ]
}
},
{
    "_id": "2",
    "_index": "get-together",
    "_score": 1.0,
    "_type": "group",
    "fields": {
        "name": "Elasticsearch Denver",
        "tags": [
            "denver",
            "elasticsearch",
            "big data",
            "lucene",
            "solr"
        ]
    }
}
]
...
}
```

#A The same query as before but using a filter this time

#B The document scores are now constant because a filter was used instead of a query

#### 4.3.4 Terms query

Similar to the term query, the terms query (note the s!) can search for multiple terms in a document's field. For example, the following listing searches for groups by a tag matching either jvm or hadoop.

##### Listing 4.11 Searching for multiple terms with the terms query

```
% curl 'localhost:9200/get-together/group/_search' -d'
{
    "query": {
        "terms": {
            "tags": [ "jvm", "hadoop" ]
        }
    },
    "fields": [ "name", "tags" ]
}
{
    ...
    "hits": [
        {
            "_id": "1",
            "_index": "get-together",
            "_score": 0.33779633,
            "_type": "group",
            "fields": {
                "name": "Denver Clojure",
                "tags": [
                    "closure",

```

```

        "denver",
        "functional programming",
        "jvm",
        "java"
    ]
}
},
{
    "_id": "4",
    "_index": "get-together",
    "_score": 0.22838624,
    "_type": "group",
    "fields": [
        "name": "Boulder/Denver big data get-together",
        "tags": [
            "big data",
            "data visualization",
            "open source",
            "cloud computing",
            "hadoop"
        ]
    ]
}
...
}
```

#### #A Multiple terms to search for

To force a minimum number of matching terms to be in a document before it matches the query, specify the `minimum_match` parameter:

```
% curl 'localhost:9200/get-together/group/_search' -d'
{
    "query": {
        "terms": {
            "tags": ["jvm", "hadoop", "lucene"],
            "minimum_match": 2
        }
    }
}'
```

If you're thinking, "Wait! That's pretty limited!" You're probably also wondering what happens when you need to combine multiple queries into a single query. Before we proceed too much further, let's discuss how to do that, which leads us straight into the `bool` query.

#### **4.3.5 Combining queries**

After learning about and using different types of queries, you'll likely find yourself needing to combine query types; this is where Elasticsearch's `Bool` query comes in.

##### **BOOL QUERY**

The `Bool` query allows you to combine any number of queries into a single query by specifying a `query` clause that indicates which parts `must`, `should`, or `must not` match the data in your Elasticsearch index:

- If you specify part of a `Bool` query `must` match, only results matching that query (or

- queries) are returned.
- Specifying a part of a query `should` means that a specified number of the clauses must match for a document to be returned.
  - If no `must` clauses are specified, at least one `should` clause has to match for the document to be returned.
  - Finally, the `must_not` clause causes matching documents to be excluded from the result set.

Table 4.1 lists the three clauses and their binary counterparts.

**Table 4.1 Bool query clause types**

Bool query clause	Binary equivalent	Meaning
<code>must</code>	To combine multiple clauses, use a binary <code>and (query1 AND query2 AND query3)</code> .	Any searches in the <code>must</code> clause must match the document;
<code>must_not</code>		Any searches in the <code>must_not</code> clause must not be part of the document; multiple clauses are combined in a binary <code>not</code> manner ( <code>NOT query1 AND NOT query2 AND NOT query3</code> )
<code>should</code>		Searches in the <code>should</code> clause may or may not match a document, but at least the <code>minimum_should_match</code> parameter number of them should match (defaults to 1); similar to a binary <code>or (query1 OR query2 OR query3)</code>

Understanding the difference between `must`, `should`, and `must_not` may be easier through example. In the following listing, we search for events that were attended by David, and must be attended by either Clint or Andy, and must not be older than June 30, 2013.

#### **Listing 4.12 Combining queries with a Bool query**

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "attendees": "david"
          }
        }
      ]
    }
  }
}
```

```

        ],
      "should": [
        {
          "term": {
            "attendees": "clint"
          }
        },
        {
          "term": {
            "attendees": "andy"
          }
        }
      ],
      "must_not": [
        {
          "range": {
            "date": {
              "lt": "2013-06-30T00:00"
            }
          }
        }
      ],
      "minimum_should_match": 1
    }
  }
}

"_shards": {
  "failed": 0,
  "successful": 2,
  "total": 2
},
"hits": {
  "hits": [
    {
      "_id": "110",
      "_index": "get-together",
      "_score": 0.56109595,
      "_source": {
        "attendees": [
          "Andy",
          "Michael",
          "Ben",
          "David"
        ],
        "date": "2013-07-31T18:00",
        "description": "Discussion about the Microsoft Azure cloud and
HDInsight.",
        "host": "Andy",
        "location": {
          "geolocation": "40.018528,-105.275806",
          "name": "Bing Boulder office"
        },
        "title": "Big Data and the cloud at Microsoft"
      },
      "_type": "event"
    }
  ],
  "max_score": 0.56109595,
  "total": 1
}

```

```

},
"timed_out": false,
"took": 67
}

#A Query that must match resulting documents
#B First query that should match documents
#C Second query that should match documents
#D Query that must not match resulting documents
#E Minimum number of should clauses that have to match a document to return it as a result

```

### BOOL FILTER

The filter version of the Bool query acts almost exactly like the query version, but instead of combining queries, it combines filters. The filter equivalent of the previous example is shown in the following listing.

#### **Listing 4.13 Combining filters with the Bool filter**

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "bool": {
          "must": [
            {
              "term": {
                "attendees": "david"
              }
            }
          ],
          "should": [
            {
              "term": {
                "attendees": "clint"
              }
            },
            {
              "term": {
                "attendees": "andy"
              }
            }
          ],
          "must_not": [
            {
              "range": {
                "date": {
                  "lt": "2013-06-30T00:00"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

```

        }
    }
}'
```

As you saw in the `Bool` query (listing 4.12), the `minimum_should_match` setting of the query version lets you specify the minimum number of `should` clauses that have to match for a result to be returned. In listing 4.12, the default value of 1 is used. This query is slightly contrived, but it includes all three of the boolean query options: `must`, `should`, and `must_not`. You could rewrite this query in a better form like so:

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "attendees": "david"
          }
        },
        {
          "range": {
            "date": {
              "gt": "2013-06-30T00:00"
            }
          }
        }
      ],
      "should": [
        {
          "terms": {
            "attendees": [ "clint", "andy" ]
          }
        }
      ]
    }
  }
}'
```

... same results as the previous query ...

Note that this query is smaller than the previous query. By inverting the `range` query from `lt` (less than) to `gt` (greater than), you can move it from the `must_not` section to the `must` section. You can also collapse the two separate `should` queries into a single `terms` query instead of two `term` queries. Elasticsearch has a flexible query language, so don't be afraid to experiment with how queries are formed as you're sending them to Elasticsearch!

With the `Bool` query and filter under your belt; you can combine any number of queries and filters together. We can now return to the other types of queries that Elasticsearch supports. You already know about the `term` query, but what if you want to analyze the data that you're sending to Elasticsearch? The `match` query is then exactly what you need.

### 4.3.6 Match and multi\_match queries

The `match` and `multi_match` queries behave similarly to the `term` query, except that they analyze the field being passed in. Don't worry if you're unsure what we mean by analyze, as we'll be covering analyzing your data in chapter 5.

#### MATCH QUERY

Similar to the `term` query, the `match` query is a hash map, containing the field you'd like to search as well as the string you want to search for, which can be either a field, or the special `_all` field to search all fields at once. Here's an example `match` query, searching for groups where name contains "elasticsearch":

```
% curl 'localhost:9200/get-together/group/_search' -d'
{
  "query": {
    "match": {
      "name": "elasticsearch"
    }
  }
}'
```

The `match` query can behave in a number of different ways; the two most important behaviors are *Boolean* and *phrase*.

#### BOOLEAN QUERY BEHAVIOR

By default, the `match` query uses Boolean behavior and the `OR` operator. For example, if you search for the text "Elasticsearch Denver," Elasticsearch searches for "Elasticsearch OR Denver," which would match get-togethers from both "Elasticsearch Amsterdam" and "Denver Clojure Group".

To search for results that contain *both* "Elasticsearch" and "Denver", change the operator by modifying the `match` field name into a map and set the `operator` field to `and`:

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "match": {
      "name": {
        "query": "elasticsearch denver",          #A
        "operator": "and"                        #B
      }
    }
  }
}'
```

#A Uses a map instead of a string for the name value  
#B Specifies search string in a query key  
#C Uses and operator instead of default or operator

The second important way a `match` query can behave is as a phrase query.

## PHRASE QUERY BEHAVIOR

A phrase query is useful when searching for a specific phrase within a document, with some amount of leeway between the positions of each word. This leeway is called *slop*, which is a number representing the distance between tokens in a phrase. Say you're trying to remember the name of a get-together group; you remember it had the words "Enterprise" and "London" in it, but you don't remember the rest of the name. Well, you could search for the phrase "enterprise london" with *slop* set to 2 or 3 instead of the default of 0 to find results containing that phrase without having to exactly know the title of the group:

```
% curl 'localhost:9200/get-together/groups/_search' -d'
{
  "query": {
    "match": {
      "name": {
        "type": "phrase",
        "query": "enterprise london",
        "slop": 1
      }
    }
  },
  "fields": ["name", "description"]
}
...
{
  "_id": "5",
  "_index": "get-together",
  "_score": 1.7768369,
  "_type": "group",
  "fields": {
    "description": "Enterprise search get-togethers are an opportunity to get
together with other people doing search.",
    "name": "Enterprise search London get-together"
  }
}
...
#A Instead of a regular match query, use a match phrase query
#B Specifies a slop of 2 to tell Elasticsearch to have leeway with the distance between the terms
#C The matching field with "enterprise" and "london" separated by a word
```

## PHRASE\_PREFIX QUERY

Similar to the `match phrase` query, the `match phrase_prefix` query allows you to go one step further and search for a phrase, but it allows prefix matching on the last term in the phrase. This behavior is extremely useful for providing a running autocomplete for a search box, where the user gets search suggestions while typing a search term. When using the search for this kind of use, it's a good idea to set the maximum number of expansions for the prefix by setting the `max_expansions` setting so the search returns in a reasonable amount of time.

In the following example, "elasticsearch den" is used as the `phrase_prefix` query. Elasticsearch takes the "den" text and looks across all the values of the `name` field to check for

those that start with “den” (“Denver”, for example). Because this could potentially be a large set, the number of expansions should be limited.

```
% curl 'localhost:9200/get-together/group/_search' -d'
{
  "query": {
    "match": {
      "name": {
        "type": "phrase_prefix",
        "query": "elasticsearch den", #A
        "max_expansions": 1 #B
      }
    },
    "fields": [ "name" ] #C
  }
}
...
{
  "_id": "2",
  "_index": "get-together",
  "_score": 2.7294521,
  "_type": "group",
  "fields": {
    "name": "Elasticsearch Denver"
  }
}
...

#A Uses a phrase prefix instead of a regular phrase query
#B Matches fields containing “Elasticsearch” and another term that starts with “den”
#C Specifies the maximum number of prefix expansions to try
```

The Boolean and phrase queries are a great choice for accepting user input; they allow you to pass in user input in a much less error-prone way, and unlike a `query_string` query, a `match` query won’t choke on reserved characters like `+`, `-`, `?`, and `!`.

### MATCHING MULTIPLE FIELDS WITH MULTI\_MATCH

Although it might be tempting to think that the `multi_match` query behaves like the `terms` query by searching for multiple matches in a field, its behavior is slightly different. Instead, it allows you to search for a value across multiple fields. This can be helpful in the get-together example, where you may want to search for a string across both the name of the group and the description:

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "multi_match": {
      "query": "elasticsearch hadoop",
      "fields": [ "name", "description" ]
    }
  }
}
```

Just like the `match` query can be turned into a `phrase` query, a `prefix` query, or a `phrase_prefix` query, the `multi_match` query can be turned into a `phrase` query or `phrase_prefix`

query as well, by specifying the `type` key. Consider the `multi match` query exactly like the `match` query, except that multiple fields can be specified for searching instead of a single field only.

With all the different `match` queries, it's possible to find a way to search for almost anything, which is why the `match` query and its relatives are considered the go-to query type for most uses. We highly recommended you use them whenever possible. For everything else, however, we'll cover some of the other types of queries that Elasticsearch supports.

## 4.4 Beyond match and filter queries

General-purpose queries that we've discussed so far, such as the `query_string` and the `match` queries, are particularly useful when the user is faced with a search box because you can run such a query with the words the user types in.

To narrow the scope of a search, some user interfaces also include other elements next to the search box, such as a calendar widget that allows you to search for newly created groups, or a checkbox for filtering events that have a location already established.

### 4.4.1 Range query and filter

The `range` query and filter are self-explanatory; they're used to query for values between a certain range and can be used for numbers, dates, and even strings.

To use the `range` query, you specify the top and bottom values for a field. For example, to search for all groups created between 2009 and 2011 in the index, use the following query:

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "range": {
      "created_on": {
        "gt": "2012-06-01",                                     #A
        "lt": "2012-09-01"                                      #A
      }
    }
  }
}'
```

**#A Specifies a date range using gt (greater than) and lt (less than)**

Or, you could use a filter instead:

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "range": {
          "created_on": {
            "gt": "2012-06-01",                                     #A
            "lt": "2012-09-01"                                      #B
          }
        }
      }
    }
  }
}'
```

```

        }
    }
}
}

#A Searches for a created_on date after June 1 ...
#B ... as well as a created_on date before September 1

```

See table 4.2 for the meaning of the parameters `gt`, `gte`, `lt`, and `lte`.

**Table 4.2 Range query parameters**

Parameter	Meaning
<code>Gt</code>	Search for fields greater than the value, not including the value
<code>Gte</code>	Search for fields greater than the value, including the value
<code>Lt</code>	Search for fields less than the value, not including the value
<code>Lte</code>	Search for fields less than the value, including the value

The range query also supports ranges of strings; so if you wanted to search for all the groups in get-togethers between "c" and "e", you could search using the following:

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "range": {
      "name": {
        "gt": "c",
        "lt": "e"
      }
    }
  }
}
```

When you use the `range` query, think long and hard about whether a filter would be a better choice. Because documents that fall into the range of the query have a binary match ("Yes, this document is in the range," or "No, this document isn't in the range"), the `range` query doesn't need to be a query. In fact, for better performance, it should be a filter. If you're unsure whether to make it a query or a filter, make it a filter. In 99% of cases, making a `range` query a filter is the right thing to do.

#### 4.4.2 Prefix query and filter

Similar to the term query, the prefix query and filter allow you to search for a term containing the given prefix, where the prefix isn't analyzed while searching. For example, to search the index for all events that start with "liber", the following query is used:

```
% curl 'localhost:9200/get-together/event/_search' -d'
{
  "query": {
    "prefix": {
      "title": "liber"
    }
  }
}'
```

And, similarly, you can use a filter instead of a regular query, which has almost the same syntax:

```
% curl 'localhost:9200/get-together/event/_search' -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "prefix": {
          "title": "liber"
        }
      }
    }
  }
}'
```

But wait! What happens if you were to send the same request, but with "Liber" instead of "liber"? Well, because the search prefix isn't analyzed before being sent, it won't find the terms that have been lowercased in the index. This is because of the way Elasticsearch analyzes documents and queries, which we cover in much more depth in chapter 5. Because of this behavior, the prefix query is a good choice for autocomplete of a partial term that a user enters if the term is part of the index. For example, you could provide a categories input box when existing categories are already known. You could take the text entered into a search box by the user, lowercase it, and use a prefix query to see what other results show up, allowing you to show the exact prefix matches to a user. But if you need to analyze the term or want an amount of fuzziness in the results, it's probably better to stick with the `match_phrase_prefix` query for autocomplete functionality.

#### 4.4.3 Wildcard query

You may be tempted to think of the wildcard query as a way to search with regular expressions, but in truth, the wildcard query is closer to the way shell wildcard *globbing* works, for example, running

```
ls *foo?ar
```

matches words such as “myfoobar”, “foocar” and “thefoodar”.

Using a string, you can allow Elasticsearch to substitute either any number of characters (including none of them) for the \* wildcard, or a single character for the ? wildcard. For example, a query for “ba\*n” would match “bacon”, “barn”, “ban”, and “baboon” because the \* can be any character sequence. Whereas, a query for “ba?n” would match only “barn” because ? must match a single character at all times. Listing 4.1 demonstrates these wildcard queries using a new index called wildcard-test.

You can also mix and match with multiple \* and ? characters to match a more complex wildcard pattern, but keep in mind that when a string is analyzed, spaces are stripped out by default, so ? can’t match a space if spaces aren’t indexed.

#### **Listing 4.14 Example wildcard query**

```
% curl -XPOST 'localhost:9200/wildcard-test/doc/1' -d'
{"title":"The Best Bacon Ever"}'
% curl -XPOST 'localhost:9200/wildcard-test/doc/2' -d'
{"title":"How to raise a barn"}'

% curl 'localhost:9200/wildcard-test/_search' -d'
{
  "query": {
    "wildcard": {
      "title": {
        "wildcard": "ba*n" #A
      }
    }
  }
}
{
  ...
  "hits" : [ {
    "_index" : "wildcard-test",
    "_type" : "doc",
    "_id" : "1",
    "_score" : 1.0, "_source" : {"title":"The Best Bacon Ever"}
  }, {
    "_index" : "wildcard-test",
    "_type" : "doc",
    "_id" : "2",
    "_score" : 1.0, "_source" : {"title":"How to raise a barn"}
  } ]
  ...
}

% curl 'localhost:9200/wildcard-test/_search' -d'
{
  "query": {
    "wildcard": {
      "title": {
        "wildcard": "ba?n" #B
      }
    }
  }
}
```

```

} '
{
...
  "hits" : [ {
    "_index" : "wildcard-test",
    "_type" : "doc",
    "_id" : "2",
    "_score" : 1.0, "_source" : {"title":"How to raise a barn"}
  } ]
}
...

```

**#A ba\*n matches both bacon and barn**

**#B ba?n matches only barn, not bacon**

Something to note when using this query—the `wildcard` query isn’t as lightweight as other queries like the `match` query; the sooner a wildcard character (`*` or `?`) occurs in the query term, the more work Lucene and Elasticsearch have to do to match it. Take, for example, the search term `"h*`"; Elasticsearch must now match every term starting with `"h"`. If the term were `"hi*`, Elasticsearch would only have to search through every term starting with `"hi"`, which is a smaller subset of all terms starting with `"h"`. Because of this overhead and performance considerations, be careful to test the `wildcard` query on a copy of your data before putting these queries into production! We talk more about a similar query, the `regexp` query, in chapter 6, where we talk about searching with relevancy.

#### 4.4.4 Querying for field existence with filters

Sometimes when querying Elasticsearch, it can be helpful to search for all the documents that don’t have a field, or are missing a value in the field. In the `get-together` index, for example, you might want to search for all groups that don’t have a review. On the other hand, you may also want to search for all the documents that have a field, regardless of what the content of the field is. This is where the `exists` and `missing` filters come in, both of which act only as filters, not as regular queries.

##### EXISTS FILTER

As the name suggests, the `exists` filter allows you to filter any query to documents that have a value in a particular field, whatever that value may be. Here’s what the `exists` filter looks like:

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "exists": { "field": "location.geolocation" }
      }
    }
  }
}
```

```
'  
... only documents with the location.geolocation field are returned ...
```

On the opposite side, you can use the `missing` filter.

### MISSING FILTER

The `missing` filter allows you to search for documents that have no value, or where the value is a default value (also called the “null value,” or `null_value` in the mapping) that was specified during the mapping. To search for documents that are missing the `reviews` field, you’d use a filter like this:

```
% curl 'localhost:9200/get-together/_search' -d'  
{  
  "query": {  
    "filtered": {  
      "query": {  
        "match_all": {}  
      },  
      "filter": {  
        "missing": {  
          "field": "reviews",  
          "existence": true,  
          "null_value": true  
        }  
      }  
    }  
  }  
'  
  
#A Finds documents missing the reviews field entirely
```

If you wanted to expand that filter to also match documents that are missing the field entirely and that might have the `null_value` field, you can specify a Boolean value for the `existence` and `null_value` fields. It includes documents that have `null_value` set in the field, as shown in this listing.

#### **Listing 4.15 Specify `existence` and `null_value` fields as Boolean values**

```
% curl 'localhost:9200/get-together/_search' -d'  
{  
  "query": {  
    "filtered": {  
      "query": {  
        "match_all": {}  
      },  
      "filter": {  
        "missing": {  
          "field": "reviews",  
          "existence": false,  
          "null_value": true  
        }  
      }  
    }  
  }  
'  
  
#A  
#B  
#C
```

```
#A Again, find documents missing the reviews field
#B Match documents that have nothing in the reviews field
#C Also match documents that have the null_value in the reviews field
```

Neither the `missing` and `exists` filters are cached but can be configured to be cached if desired by specifying the `_cache` key, as we talked about in section 4.

#### 4.4.5 Transforming any query into a filter

So far, we've talked about the different types of queries and filters that Elasticsearch supports, but we've been limited to using only the filters that are already provided. Sometimes you may want to take a query such as `query_string`, which has no filter equivalent, and turn it into a filter. Elasticsearch allows you to do this with the `query` filter, which takes any query and turns it into a filter.

To transform a `query-string` query that searches for a name matching "denver clojure" to a filter, you would use a search like this:

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "match_all": {}
  },
  "filter": {
    "query" : {
      "query_string" : {
        "query" : "name:\\"denver clojure\\""
      }
    }
  }
}

#A
```

**#A Using the query filter to wrap a query that doesn't have a filter equivalent**

Using this, you can get some of the benefits of a filter (such as not having to calculate a score for that part of the query). You can also choose to cache this filter if it turns out to be used many times; the syntax for caching looks slightly different than adding the `_cache` key, as shown next.

#### Listing 4.16 Caching query filter

```
% curl 'localhost:9200/get-together/_search' -d'
{
  "query": {
    "match_all": {}
  },
  "filter": {
    "fquery": {
      "query" : {
        "query_string" : {
          "query" : "name:\\"denver clojure\\""
        }
      }
    },
    "_cache": true
}

#A
#A
#A
#A
#A
#A
#B
```

```

        }
    }
}
```

#A The “query” part is now inside the “fquery” map  
#B Tells Elasticsearch to cache this filter

The `query` part of the query has moved inside a new key named `fquery`, which is where the `_cache` key now resides. If you find yourself often using a particular query that doesn’t have a filter equivalent (like one of the `match` queries, or a `query_string` query), you may want to cache it assuming the score for that particular part of the query isn’t important.

## 4.5 Choosing the best query for the job

Now that we’ve covered some of the most popular Elasticsearch queries, let’s look at how to decide which queries to use and when. Although there is no hard-and-fast rule for which query to use for what, table 4.3 helps you determine which query to use for the general case.

Table 4.3 Which type of query to use for general use cases

Use case	Query type to use
You want to take input from a user, similar to a “Google-style” interface and search for documents with the input.	Use a <code>match</code> query.
You want to take input as a phrase and search for documents containing that phrase, perhaps with some amount of leniency (slop)	Use a <code>match phrase</code> query with an amount of slop to find phrases similar to what the user is searching for.
You want to search for a single word in a document, knowing exactly how the word should appear.	Use a <code>term</code> query because query terms aren’t analyzed.
You want to combine many different searches or types of searches, creating a single search out of them.	Use the <code>Bool</code> query to combine any number of subqueries into a single query.
You want to search for certain terms across many fields in a document.	Use the <code>multi match</code> query, which behaves similar to the <code>match</code> query, but on multiple fields.
You want to return every document from a search.	Use the <code>match all</code> query to return all documents from a search.
You want to search a field for values that are	Use a <code>range</code> query to search within documents with

between two specified values.	values between a certain range.
You want to search a field for values that start with a specified string.	Use a <code>prefix</code> query to search for terms starting with a given string.
You want to autocomplete the value of a single word based on what the user has already typed in.	Use a <code>prefix</code> query to send what the user has typed in and get back exact matches starting with the text.
You want to search for all documents that have no value for a specified field.	Use the <code>missing</code> filter to filter out documents that are missing fields.

## 4.6 Summary

- Filters can speed up queries by skipping over the scoring calculations and by caching.
- “Human language” type queries, such as the `match` and `query_string` queries, are suitable for search boxes.
- The `match` query is the go-to query for full-text search, but the `query_string` query is both more flexible and more complex because it exposes the full Lucene query syntax.
- The `match` query has multiple subtypes: `boolean`, `phrase`, and `phrase_prefix`. The main difference is that `boolean` matches individual words, whereas the `phrase` types take the order of words into account, as if they were in a phrase.
- Specialized queries such as the `prefix` and `wildcard` queries are also supported.
- To filter documents where a field value doesn’t exist, use the `missing` filter. The `exists` filter does the exact opposite; it returns only documents having the specified field value.

# 5

## Analyzing your data

### *This chapter covers*

- Analyzing your document's text with Elasticsearch
- Using the analysis API
- Tokenization
- Character Filters
- Token Filters
- Stemming
- Analyzers included with Elasticsearch

So far we've covered indexing and searching your data, but what actually happens when data is sent to Elasticsearch? What happens to the text sent in a document to Elasticsearch? How can it find specific words among sentences, even when the case changes? For example, when a user searches for the word "Bear," generally you would like a document with the sentence "I love bears and fish" to match, because the word "bear" is in there. While you could use the information you learned in the previous chapter to do a `query_string` search for "bear\*" and find the document, this can much more easily be accomplished by using analysis. Once you finish this chapter you'll have a better idea of how Elasticsearch's analysis allows you to search your document set in a more flexible manner.

### **5.1 What is analysis?**

Analysis is the process Elasticsearch performs on the body of a document before the document is sent off to be indexed. Elasticsearch goes through a number of steps for every analyzed field before the document is added to the index:

- *Character filtering*: Transform the characters using a character filter.

- *Breaking text into tokens:* Break apart the text into a set of one or more tokens.
- *Token filtering:* Transform each token using a *token filter*.
- *Token indexing:* Store those tokens into the index.

We'll talk about each step more in detail next, but first, let's see the entire process summed up in a diagram. In Figure 5.1, we'll use the text "I love Bears & Fish," which is eventually transformed into the analyzed tokens "I," "like," "bears," and "fish."

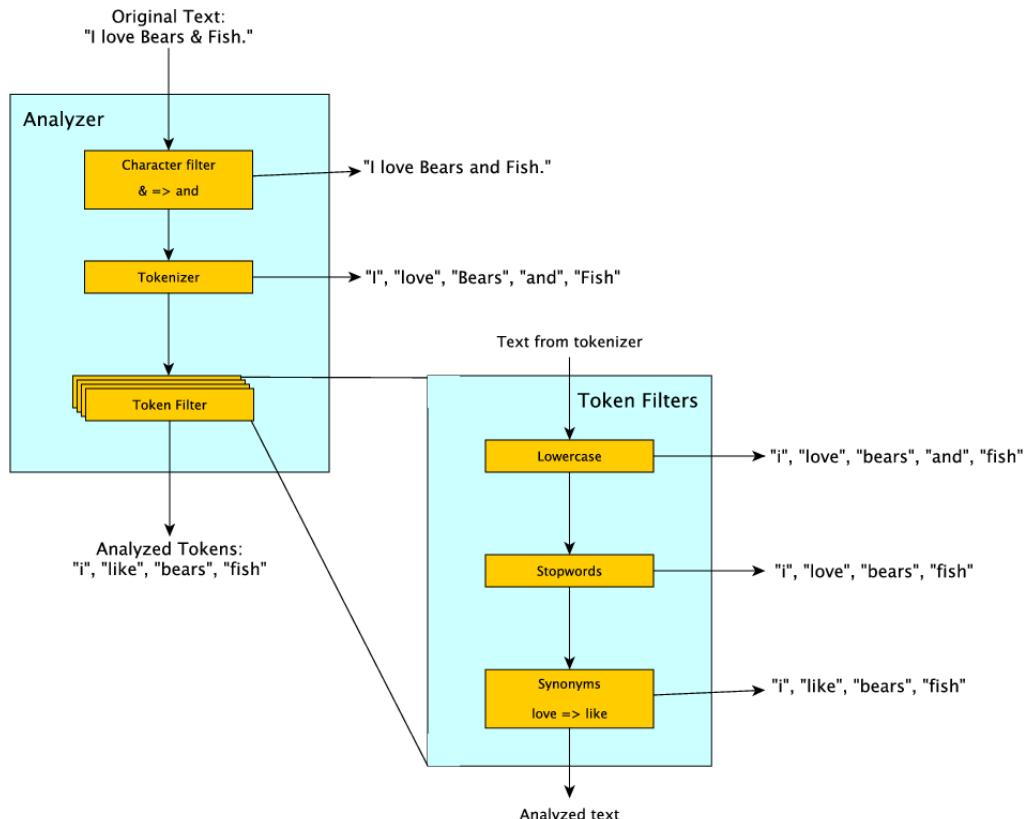


Figure 5.1 Overview of the analysis process

### CHARACTER FILTERING

As you can see in the upper left of the figure, Elasticsearch first runs the character filters; these filters are used to transform particular character sequences into other character sequences. This can be used for things like stripping HTML out of text, or converting an arbitrary number of characters into other characters (perhaps correcting the text-message

shortening of "I love u too" into "I love you too"). In figure 5.1 we use the character filter to replace "&" with the word "and,".

### **BREAKING INTO TOKENS**

After the text has had the character filters applied, it needs to be split into pieces that can be operated on. Lucene itself doesn't act on large strings of data, instead, it acts on what are known as tokens. Tokens are generated out of a piece of text, which results in any number (even zero!) of tokens. In English, for example, a common tokenization that can be used is the whitespace analyzer, which splits text into tokens, based on whitespace like spaces and newlines. In figure 5.1 this is represented by breaking the string "I love Bears and Fish." into the tokens "I," "love," "Bears," "and," and "Fish."

### **TOKEN FILTERING**

Once the block of text has been converted into tokens, Elasticsearch will then apply what are called token filters to each token. These token filters take a token as input and can modify, add, or remove more tokens as needed. One of the most useful and common examples of a token filter is the lowercase token filter, which takes in a token and simply lowerscases it, to ensure that you will be able to find a get together about "The Doors" when searching for the term "doors." The tokens can go through more than one token filter, each doing different things to the tokens to mold the data into the best format for your index.

In our example in figure 5.1, there are three token filters: the first lowercasing the tokens; the second removing the stopword "and" (we'll talk about stopwords later in this chapter); and finally substituting the word "like" for "love," using synonyms.

### **TOKEN INDEXING**

After the tokens have gone through zero-or-more token filters, they are sent to Lucene to be indexed for the document. These tokens are what make up the inverted index we discussed back in chapter 1.

Together, these different parts make up an *analyzer*, which can also be defined as zero-or-more character filters, a tokenizer, and zero-or-more token filters. There are some prebuilt analyzers we'll talk about later on in this chapter, which you can use without having to construct your own, but first we'll talk about the different components of an analyzer individually.

---

### **Analysis during a query**

Depending on what kind of query you use, this analysis can also be applied to the search text, before the search is performed against the index. In particular, queries such as the `match` and `match_phrase` queries perform analysis before searching, and queries like the `term` and `terms` query do not. It's important to keep this in mind when debugging why a particular search matches or doesn't match a document - it might be analyzed differently than what you expect!

---

Now that you have an understanding of what goes on during Elasticsearch's analysis phase, let's talk about how analyzers are specified for fields in your mapping, and how custom analyzers are specified.

## 5.2 Using analyzers for your documents

Knowing about the different types of analyzers and token filters is all fine and well, but before they can actually be used, Elasticsearch needs to know about how you want to use them. For instance, you can specify in the mapping, which individual tokenizer and token filters to use for an analyzer, and which analyzer to use for which field.

There are two ways to specify analyzers that can be used by your fields:

- when the index is created, as settings for that particular index; or
- as global analyzers in the configuration file for Elasticsearch.

Generally, to be more flexible, it's easier to specify them at the index creation time, which is also when you will want to specify your mappings. This allows you to create new indices with updated or entirely different analyzers. On the other hand, if you find yourself using the same set of analyzers across your indices, without changing them very often at all, you can also save yourself some bandwidth by putting the analyzers into the configuration file. Examine how you are using Elasticsearch and pick the option that works the best for you. You could even combine the two and put the analyzers that are used by all of your indices into the configuration file, and specify additional analyzers when you create indices, for added flexibility.

Regardless of the way you specify your custom analyzers, you will need to specify which field uses which analyzer in the mapping of your index, either by specifying the mapping when the index is created, or using the put mapping API to specify it at a later time.

### 5.2.1 Adding analyzers when an index is created

You've already seen some of the settings when an index is created, in chapter 3, notably setting the number of primary and replica shards for an index, which look something like this next listing.

#### **Listing 5.1 Setting the number of primary and replica shards**

```
% curl -XPOST 'localhost:9200/myindex' -d'
{
  "settings" : {
    "number_of_shards": 2,                                     #A
    "number_of_replicas": 1                                   #B
  },
  "mappings" : {
    ...
  }
}'
```

#A Specifying custom settings for the index, here specifying 2 primary shards

#B And specifying 1 replica here

## #C Mappings for the index

Adding a custom analyzer is done by specifying another map in the settings config, under the "index" heading. This header should specify the custom analyzer you want to use, and can also contain the custom tokenizer, token-filters and char-filters that can be used by the index. Listing 5.2 shows a custom analyzer that specifies custom parts for all the analysis steps. This is a complex example, so we've added some headings to show the different parts. Don't worry about all the code details yet, as we'll go through it later on in this chapter.

### **Listing 5.2 Adding a custom analyzer during index creation**

```
% curl -XPOST 'localhost:9200/myindex' -d '
{
  "settings" : {
    "number_of_shards": 2,                                     #A
    "number_of_replicas": 1,                                    #A
    "index": {
      "analysis": {                                         #B
        "analyzer": {                                     #C
          "myCustomAnalyzer": {                           #D
            "type": "custom",                            #E
            "tokenizer": "myCustomTokenizer",           #F
            "filter": [ "myCustomFilter1", "myCustomFilter2" ], #G
            "char_filter": [ "myCustomCharFilter" ]       #H
          }
        }
      }
    }
}
```

#### **Custom analyzer**

```
"analyzer": {                                              #D
  "myCustomAnalyzer": {                                #E
    "type": "custom",                               #F
    "tokenizer": "myCustomTokenizer",             #G
    "filter": [ "myCustomFilter1", "myCustomFilter2" ], #H
    "char_filter": [ "myCustomCharFilter" ]         #I
  }
},
```

#### **Tokenizer**

```
"tokenizer": {                                              #J
  "myCustomTokenizer": {                                #J
    "type": "letter"                                 #J
  }
}
```

#### **Custom filters**

```
},
"filter": {                                              #K
  "myCustomFilter1": {                                #K
    "type": "lowercase"                            #K
  },
  "myCustomFilter2": {                                #K
    "type": "kstem"                                #K
  }
},
```

#### **Character filter**

```
"char_filter": {                                              #L
  "myCustomCharFilter": {                                #L
  }
}
```

## Mappings

```
"mappings" : {  
    ...  
}  
}'
```

- #A Other settings for the index that we've covered before
- #B Other "index"-level settings
- #C The analysis settings for this index
- #D Specifying a custom analyzer in the "analyzer" object
- #E The custom analyzer is named "myCustomAnalyzer"
- #F It's of type "custom"
- #G It uses the "myCustomTokenizer" to tokenize text
- #H Specify two filters that text should be run through, myCustomFilter1 and myCustomFilter2
- #I Specify a custom char filter called "myCustomCharFilter" that will run before other analysis
- #J Specifying the custom tokenizer of type "letter"
- #K Two custom token filters, one for lowercasing and another using kstem
- #L A custom char filter that translates characters to other mappings
- #M Mappings for creating the index

The mappings have been left out of the code listing here, as we'll cover how to specify the analyzer for a field in section 5.2.3. In this example, a custom analyzer is created called `myCustomAnalyzer`, which uses the custom tokenizer `myCustomTokenizer`, two custom filters named `myCustomFilter1` and `myCustomFilter2`, and a custom character filter named `myCustomCharFilter` (notice a trend here?). Each of these separate analysis parts are given in their respective JSON sub-maps. Multiple analyzers can be specified with different names, and combined by custom analyzers to give you flexible analysis options when indexing and searching.

Now that you have a sense of what adding custom analyzers looks like when an index is created, let's look at the same analyzers added to the Elasticsearch configuration itself.

## **5.2.2 Adding analyzers to the Elasticsearch configuration**

In addition to specifying analyzers with settings during index creation, adding analyzers into the Elasticsearch config is another supported way of specifying custom analyzers. There are tradeoffs to this method however; if you specify the analyzers during index creation, you will always be able to make changes to the analyzers without restarting Elasticsearch. But, if you specify the analyzers in the Elasticsearch configuration, you will need to restart Elasticsearch to pick up any changes you make to the analyzers. On the flip side, you'll have less data to send when creating indices. While it's generally easier to specify them at index creation for the

larger degree of flexibility, if you plan to never change your analyzers, you can go ahead and put them into the configuration file.

Specifying analyzers in the `elasticsearch.yml` configuration file is very similar to specifying them as JSON; here are the same custom analyzers from the previous section, but specified in the configuration YAML file:

```
index:
  analysis:
    analyzer:
      myCustomAnalyzer:
        type: custom
        tokenizer: myCustomTokenizer
        filter: [myCustomFilter1, myCustomFilter2]
        char_filter: myCustomCharFilter
    tokenizer:
      myCustomTokenizer:
        type: letter
    filter:
      myCustomFilter1:
        type: lowercase
      myCustomFilter2:
        type: kstem
    char_filter:
      myCustomCharFilter:
        type: mapping
        mappings: ["ph=>f", "u => you"]
```

### 5.2.3 Specifying the analyzer for a field in the mapping

There's one piece of the puzzle left before you're off on your way, analyzing fields with custom analyzers: how to specify that a particular field in the mapping should be analyzed using one of your custom analyzers. It's quite simple to specify the analyzer for a field by setting the "analyzer" field on a mapping. For instance, if we had the mapping for a field called "description," specifying the analyzer would look like this:

```
{
  "mappings" : {
    "document" : {
      "properties" : {
        "description" : {
          "type" : "string",
          "analyzer" : "myCustomAnalyzer" #A
        }
      }
    }
  }
}
```

**#A Specifying the analyzer "myCustomAnalyzer" for the description field**

If you want a particular field to not be analyzed at all, you need to specify the "index" field with the `not_analyzed` setting. This keeps the text as a single token, without any kind of modification (no lowercasing or anything). It looks something like this:

```
{
  "mappings" : {
    "document" : {
      "properties" : {
        "name" : {
          "type" : "string",
          "index" : "not_analyzed" #A
        }
      }
    }
  }
}
```

#### #A Specifying that the name field is not to be analyzed

There is a common pattern for fields where you may want to search on both the analyzed and verbatim text of a field, which is to stick them in multi-fields.

#### USING MULTI-FIELD TYPE TO STORE DIFFERENTLY ANALYZED TEXT

Often it's quite helpful to be able to search on both the analyzed version of a field, as well as the original, non-analyzed text. This is especially useful for things like facets and aggregations, or sorting on a string field. Elasticsearch makes this simple to do by using multi-fields, which we first saw in chapter 3. Take the "name" field for example; you may want to be able to sort on the name field, but search through it using analysis. You can specify a field that does both like so:

```
% curl -XPOST 'localhost:9200/myindex' -d'
{
  "mappings": {
    "type": {
      "properties": {
        "name": {
          "type": "string",
          "analyzer": "standard", #A
          "fields": {
            "raw": {
              "index": "not_analyzed",
              "type": "string" #B
            }
          }
        }
      }
    }
  }
}'
```

#### #A The original analysis, using the standard analyzer

#### #B A raw version of the field, which is not analyzed

We've covered how to specify analyzers; now we're ready to cover a neat way to check how any arbitrary text can be analyzed: the analyze API.

### 5.3 Analyzing text with the analyze API

Using the analysis API to test the analysis process can be extremely helpful when tracking down how information is being stored in your Elasticsearch indices. This API allows you to send any text to Elasticsearch, specifying what analyzer, tokenizer or token filters to use and get back the analyzed tokens.

Here's an example of what the analyze API looks like, using the standard analyzer to analyze the text "I love Bears and Fish."

#### **Listing 5.3 Example of using the analyze API**

```
% curl -XPOST 'localhost:9200/_analyze?analyzer=standard' -d'I love Bears and Fish.'
And an example of the output:
{
  "tokens": [
    {
      "end_offset": 1,
      "position": 1,
      "start_offset": 0,
      "token": "I",
      "type": "<ALPHANUM>" #A
    },
    {
      "end_offset": 6,
      "position": 2,
      "start_offset": 2,
      "token": "love",
      "type": "<ALPHANUM>" #A
    },
    {
      "end_offset": 12,
      "position": 3,
      "start_offset": 7,
      "token": "bears",
      "type": "<ALPHANUM>" #A
    },
    {
      "end_offset": 16,
      "position": 4,
      "start_offset": 13,
      "token": "and",
      "type": "<ALPHANUM>" #A
    },
    {
      "end_offset": 21,
      "position": 5,
      "start_offset": 17,
      "token": "fish",
      "type": "<ALPHANUM>" #A
    }
  ]
}

#A The analyzed tokens: "i", "love", "bears", "and", and "fish"
```

The most important output from the analysis API is the “token” key. The output is a list of these maps, which gives you a representation of what the processed tokens (the ones that are going to actually be written to the index!) look like. For example, with our text “I like Bears and Fish.”, we get back five tokens: [“i”, “like”, “bears”, “and”, “fish”]. Notice that in this case, with the standard analyzer, each token was lowercased, and the punctuation at the end of the sentence was removed. This is a great way to test documents to see how Elasticsearch will analyze them, and has quite a few ways to customize the analysis that is performed on the text.

### **SELECTING AN ANALYZER**

If you already have an analyzer in mind, and want to see how it handles some text, you can set the `analyzer` parameter to the name of the analyzer. We’ll go over the different build-in analyzers in the next section, so keep this in mind if you want to try any of them out!

If you configured an analyzer in your `elasticsearch.yml` file, you can also reference it by name in the `analyzer` parameter. Additionally, if you’ve created an index with a custom analyzer similar to the example in 5.2.1, you can still use this analyzer by name, but instead of using the HTTP endpoint of `/_search`, you’ll need to specify the index first. An example using the index named “veggies” and an analyzer called “myanalyzer” is shown below:

```
% curl -XPOST 'localhost:9200/veggies/_analyze?analyzer=myanalyzer' -d'...'
```

### **COMBINING PARTS TO CREATE AN IMPROMPTU ANALYZER**

Sometimes you may not want to use a built-in analyzer, but instead try out a combination of tokenizers and token-filters, for instance, to see how a particular tokenizer breaks up a sentence without any other analysis. With the analysis API you can specify a tokenizer, and a list of token-filters to be used for analyzing the text. For example, if you wanted to use the whitespace tokenizer (to split the text on spaces) and then use the lowercase and reverse token filters, you could do so by using the following:

```
% curl -XPOST
      'localhost:9200/_analyze?tokenizer=whitespace&filters=lowercase,reverse' -d
      'I love Bears and Fish.'
```

And you would get back the following tokens:

```
{
  "tokens" : [ {
    "token" : "i",
    "start_offset" : 0,
    "end_offset" : 1,
    "type" : "word",
    "position" : 1
  }, {
    "token" : "evol",
    "start_offset" : 2,
    "end_offset" : 6,
    "type" : "word",
    "position" : 2
  }, {
    "token" : "sraeb",
    "start_offset" : 7,
    "end_offset" : 11,
    "type" : "word",
    "position" : 3
  } ]
```

```

    "start_offset" : 7,
    "end_offset" : 12,
    "type" : "word",
    "position" : 3
}, {
    "token" : "dna",
    "start_offset" : 13,
    "end_offset" : 16,
    "type" : "word",
    "position" : 4
}, {
    "token" : ".hsif",
    "start_offset" : 17,
    "end_offset" : 22,
    "type" : "word",
    "position" : 5
}
]
}

```

Which first has tokenized the sentence “I like Bears and Fish.” into the tokens [“I”, “like”, “Bears”, “and”, “Fish.”]; next it lowercases the tokens into [“i”, “like”, “bears”, “and”, “fish.”]; and finally, reverses each token to get [“i”, “ekil”, “sraeb”, “dna”, “.hsif”].

### **ANALYZING BASED ON A FIELD'S MAPPING**

One more helpful thing about the analysis API once you start creating mappings for an index, is that Elasticsearch allows you to analyze based on a field where the mapping has already been created. If you create a mapping with a field “description” that looks like this snippet.

```

... other mappings ...
"description": {
    "type": "string",
    "analyzer": "italian"
}

```

You can then use the analyzer associated with the field by specifying the `field` parameter with the request.

```
% curl -XPOST 'localhost:9200/veggies/_analyze?field=description' -d'Era
deliziosa'
```

And the Italian analyzer will automatically be used, since it is the analyzer associated with the `description` field. Keep in mind that in order to use this, you'll need to specify an index, because Elasticsearch needs to be able to get the mappings for a particular field from an index.

Now that we've covered how to test out different analyzers using curl, let's jump into all the different analyzers that Elasticsearch provides for you out of the box. Keep in mind that you can always create your own analyzer by combining the different parts (tokenizers and token-filters).

## **5.4 Analyzers, Tokenizers and Token Filters, oh my!**

In this section we'll discuss the built-in analyzers, tokenizers, and token filters that Elasticsearch provides. Elasticsearch provides a large number, such as lowercasing, stemming,

language-specific, synonyms and so on, so you have a lot of flexibility to combine them in different ways to get your desired tokens.

### 5.4.1 Built-in analyzers

In this section I'll give you a rundown of the analyzers that Elasticsearch comes with out of the box. Remember that an analyzer is an optional character filter, a single tokenizer, and zero-or-more token filters. Figure 5.2 is a visualization of what an analyzer looks like.

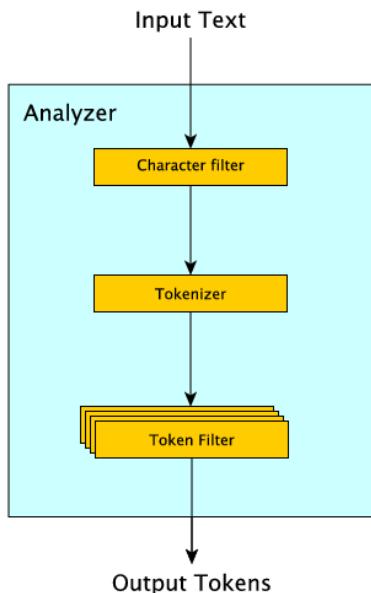


Figure 5.2 Analyzer overview

We'll be referencing tokenizers and token filters, which we'll cover in more detail in the following sections. With each analyzer, we'll include an example of some text that demonstrates what analysis using that analyzer looks like.

#### STANDARD

The *standard analyzer* is the default analyzer for text when no analyzer is specified. It combines sensible defaults for most European languages by combining the standard tokenizer, the standard token filter, and the lowercase token filter. There's not much to say about the standard analyzer; we'll talk about what exactly the standard tokenizer and standard token filter do in sections 5.4.2 and 5.4.3; just keep in mind that if you don't specify an analyzer for a field the standard analyzer is what will be used for that field.

## **SIMPLE**

The *simple* analyzer is just that - simple! It simply uses the lowercase tokenizer, which means tokens are split at non-letters and automatically lowercased. This analyzer doesn't work well for Asian languages that don't separate words with whitespace though, so use it only for European languages.

## **WHITESPACE**

The whitespace analyzer does nothing but split text into tokens around whitespace, very simple!

## **STOP**

The stop analyzer behaves just like the simple analyzer, but additionally filters out stop words from the token stream.

## **KEYWORD**

The keyword analyzer takes the entire field and generates a single token on it. Keep in mind however, that rather than using the keyword tokenizer in your mappings, it's better to set the "index" setting to "not\_analyzed."

## **PATTERN**

The pattern analyzer allows you to specify a pattern for tokens to be broken apart at. However, since the pattern would have to be specified regardless, it often makes more sense to use a custom analyzer and combine the existing pattern tokenizer with any needed token filters.

## **LANGUAGE & MULTI-LINGUAL**

Elasticsearch supports a wide variety of language-specific analyzers out of the box. There are analyzers for arabic, armenian, basque, brazilian, bulgarian, catalan, chinese, cjk, czech, danish, dutch, english, finnish, french, galician, german, greek, hindi, hungarian, indonesian, italian, norwegian, persian, portuguese, romanian, russian, spanish, swedish, turkish, and thai. If you want to analyze a language not included in this list, there may be a plugin for it as well.

## **SNOWBALL**

The snowball analyzer uses the standard tokenizer and token filter (just like the standard analyzer), with the lowercase token filter, the stop filter, as well as stemming the text using the snowball stemmer. Don't worry if you aren't sure what stemming is, as we'll discuss it in more detail near the end of this chapter.

Before these analyzers sink in, however, you need to understand the parts that make up the analyzer, so we'll discuss the tokenizers that Elasticsearch supports next.

## 5.4.2 Tokenization

### STANDARD TOKENIZER

The standard tokenizer is a grammar-based tokenization that is good for most European languages; it also handles segmenting Unicode text, although with a max token length of 255. It also removes punctuation like commas and periods.

```
% curl -XPOST 'localhost:9200/_analyze?tokenizer=standard' -d'I have, potatoes.'
{
  "tokens" : [ {
    "token" : "I",
    "start_offset" : 0,
    "end_offset" : 1,
    "type" : "<ALPHANUM>",
    "position" : 1
  }, {
    "token" : "have",
    "start_offset" : 2,
    "end_offset" : 6,
    "type" : "<ALPHANUM>",
    "position" : 2
  }, {
    "token" : "potatoes",
    "start_offset" : 8,
    "end_offset" : 16,
    "type" : "<ALPHANUM>",
    "position" : 3
  } ]
}
```

### KEYWORD

The keyword tokenizer is a very simple tokenizer that takes the entire text and provides it as a single token to the token filters. This can be useful when you only want to apply token filters without doing any kind of tokenization.

### LETTER

The letter tokenizer takes the text and divides into tokens at things that are not letters. For example, with the sentence "Hi, there." the tokens would be: Hi and there because the comma, space, and period are all non-letters.

```
% curl -XPOST 'localhost:9200/_analyze?tokenizer=letter' -d'Hi, there.'
{
  "tokens" : [ {
    "token" : "Hi",
    "start_offset" : 0,
    "end_offset" : 2,
    "type" : "word",
    "position" : 1
  }, {
    "token" : "there",
    "start_offset" : 4,
    "end_offset" : 9,
    "type" : "word",
    "position" : 2
  } ]
}
```

```
}
```

## LOWERCASE

The lowercase tokenizer combines both the regular letter tokenizer's action as well as the action of the lowercase token filter (which, as you can imagine, lowercases the entire token). The main reason to do this with a single tokenizer is that better performance is gained by doing both at once.

## WHITESPACE

The whitespace tokenizer separates tokens by whitespace. Note that this tokenizer doesn't remove any kind of punctuation, so tokenizing the text "Hi, there." results in two tokens, "Hi," and "there."

```
% curl -XPOST 'localhost:9200/_analyze?tokenizer=whitespace' -d'Hi, there.'
{
  "tokens" : [ {
    "token" : "Hi",
    "start_offset" : 0,
    "end_offset" : 3,
    "type" : "word",
    "position" : 1
  }, {
    "token" : "there.",
    "start_offset" : 4,
    "end_offset" : 10,
    "type" : "word",
    "position" : 2
  } ]
}
```

## PATTERN

The pattern tokenizer allows you to specify an arbitrary pattern where text should be split into tokens. The pattern that is specified should match the spacing characters, for example, if you wanted to split text on any two-digit number, you could create a custom analyzer that breaks tokens at wherever the text .. occurs, which would look like this:

```
% curl -XPOST 'localhost:9200/pattern' -d'{
  "settings": {
    "index": {
      "analysis": {
        "tokenizer": {
          "pattern1": {
            "type": "pattern",
            "pattern": "\\\.-\\\\."
          }
        }
      }
    }
  }
}

% curl -XPOST 'localhost:9200/pattern/_analyze?tokenizer=pattern1' -d'breaking.-
.some..text'
{
  "tokens" : [ {
```

```

    "token" : "breaking",
    "start_offset" : 0,
    "end_offset" : 8,
    "type" : "word",
    "position" : 1
  },
  {
    "token" : "some",
    "start_offset" : 11,
    "end_offset" : 15,
    "type" : "word",
    "position" : 2
  },
  {
    "token" : "text",
    "start_offset" : 18,
    "end_offset" : 22,
    "type" : "word",
    "position" : 3
  }
]
}

```

### **UAX URL EMAIL**

The standard tokenizer is pretty good at figuring out English words, but these days there is quite a bit of text that ends up having website addresses and email addresses in them. The standard analyzer breaks these apart in places where you may not intend, for example, if we take the example email address `john.smith@example.com` and analyze it with the standard tokenizer, it gets split into multiple tokens:

```

% curl -XPOST 'localhost:9200/_analyze?tokenizer=standard' -
  d'john.smith@example.com'
{
  "tokens" : [ {
    "token" : "john.smith",
    "start_offset" : 0,
    "end_offset" : 10,
    "type" : "<ALPHANUM>",
    "position" : 1
  },
  {
    "token" : "example.com",
    "start_offset" : 11,
    "end_offset" : 22,
    "type" : "<ALPHANUM>",
    "position" : 2
  }
]
}

```

Here you see it's been split into the `john.smith` part and the `example.com` part. It also splits URLs into separate parts:

```

% curl -XPOST 'localhost:9200/_analyze?tokenizer=standard' -d
  'http://example.com?q=foo'
{
  "tokens" : [ {
    "token" : "http",
    "start_offset" : 0,
    "end_offset" : 4,
    "type" : "<ALPHANUM>",
    "position" : 1
  }
]
}

```

```

}, {
  "token" : "example.com",
  "start_offset" : 7,
  "end_offset" : 18,
  "type" : "<ALPHANUM>",
  "position" : 2
}, {
  "token" : "q",
  "start_offset" : 19,
  "end_offset" : 20,
  "type" : "<ALPHANUM>",
  "position" : 3
}, {
  "token" : "foo",
  "start_offset" : 21,
  "end_offset" : 24,
  "type" : "<ALPHANUM>",
  "position" : 4
} ]
}

```

The UAX URL Email tokenizer will preserve both emails and URLs as single tokens:

```

% curl -XPOST 'localhost:9200/_analyze?tokenizer=uax_url_email' -d'
john.smith@example.com http://example.com?q=bar'
{
  "tokens" : [ {
    "token" : "john.smith@example.com",
    "start_offset" : 1,
    "end_offset" : 23,
    "type" : "<EMAIL>",
    "position" : 1
  }, {
    "token" : "http://example.com?q=bar",
    "start_offset" : 24,
    "end_offset" : 48,
    "type" : "<URL>",
    "position" : 2
  } ]
}

```

This can be extremely helpful when you want to search for exact urls or email addresses in a text field.

### PATH HIERARCHY

The path hierarchy tokenizer allows you to index filesystem paths in a way where searching for files sharing the same path will return results. For example, let's assume you have a filename you want to index that looks like "/usr/local/var/log/elasticsearch.log". Here's what the path hierarchy tokenizer tokenizes this into:

```

% curl 'localhost:9200/_analyze?tokenizer=path_hierarchy' -d
  '/usr/local/var/log/elasticsearch.log'
{
  "tokens" : [ {
    "token" : "/usr",
    "start_offset" : 0,
    "end_offset" : 4,
    "type" : "<PATH>"
  }, {
    "token" : "/local",
    "start_offset" : 5,
    "end_offset" : 9,
    "type" : "<PATH>"
  }, {
    "token" : "/var",
    "start_offset" : 10,
    "end_offset" : 14,
    "type" : "<PATH>"
  }, {
    "token" : "/log",
    "start_offset" : 15,
    "end_offset" : 19,
    "type" : "<PATH>"
  }, {
    "token" : "elasticsearch.log",
    "start_offset" : 20,
    "end_offset" : 34,
    "type" : "<PATH>"
  } ]
}

```

```

    "type" : "word",
    "position" : 1
}, {
    "token" : "/usr/local",
    "start_offset" : 0,
    "end_offset" : 10,
    "type" : "word",
    "position" : 1
}, {
    "token" : "/usr/local/var",
    "start_offset" : 0,
    "end_offset" : 14,
    "type" : "word",
    "position" : 1
}, {
    "token" : "/usr/local/var/log",
    "start_offset" : 0,
    "end_offset" : 18,
    "type" : "word",
    "position" : 1
}, {
    "token" : "/usr/local/var/log/elasticsearch.log",
    "start_offset" : 0,
    "end_offset" : 36,
    "type" : "word",
    "position" : 1
}
]
}

```

This means a user querying for a file sharing the same path hierarchy (hence the name!) as this file will find a match. Querying for "/usr/local/var/log/es.log" will still share the same tokens as "/usr/local/var/log/elasticsearch.log", so it can still be returned as a result.

Now that we've touched on the different ways of splitting a block of text into different tokens, let's talk about what we can do with each of those tokens.

### **5.4.3 Token Filters**

There are a lot of token filters included in Elasticsearch; we'll cover only the most popular ones in this section, since enumerating all of them would make this section much too verbose. Like the token filters you saw in Figure 5.1, here is an example of three token filters, the lowercase filter, the stopwords filter, and the synonym filter.

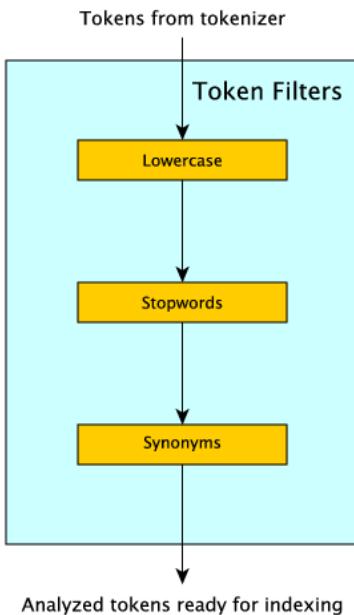


Figure 5.3 Token filters prep data for indexing

### STANDARD

Don't be fooled by thinking the standard token filter performs complex calculation; it actually does nothing at all! In the older versions of Lucene it used to remove the "'s" characters from the end of words, as well as removing some extraneous period characters, but these are handled instead by some of the other token filters and tokenizers.

### LOWERCASE

The lowercase token filter does just that: lowers any token that gets passed through it. Simple enough to understand:

```
% curl 'localhost:9200/_analyze?tokenizer=keyword&filters=lowercase' -d'HI THERE!'
{
  "tokens" : [ {
    "token" : "hi there!",
    "start_offset" : 0,
    "end_offset" : 9,
    "type" : "word",
    "position" : 1
  } ]
}
```

### LENGTH

The length token filter removes words that fall outside of a boundary for the minimum and maximum length of the token. For example, if you set the min setting to 2 and the max

setting to 8, any token shorter than two characters will be removed and any character longer than 8 characters will be removed.

### **STOP**

The stop token filter removes stop words from the token stream. For English, this means all tokens that fall into this list are entirely removed. You can also specify a list of words to be removed for this filter.

What are the stopwords? Here is the default list of stopwords for the English language:

"a", "an", "and", "are", "as", "at", "be", "but", "by", "for", "if", "in", "into", "is", "it", "no", "not", "of", "on", "or", "such", "that", "the", "their", "then", "there", "these", "they", "this", "to", "was", "will", "with"

To specify the list of stop words, you can create a custom token filter with a list of words, like this:

```
% curl -XPOST 'localhost:9200/stopwords' -d'{  
  "settings": {  
    "index": {  
      "analysis": {  
        "analyzer": {  
          "stop1": {  
            "type": "custom",  
            "tokenizer": "standard",  
            "filter": ["my-stop-filter"]  
          }  
        },  
        "filter": {  
          "my-stop-filter": {  
            "type": "stop",  
            "stopwords": ["the", "a", "an"]  
          }  
        }  
      }  
    }  
  }'
```

Or, to read the list of stopwords from a file using either a path relative to the configuration location, or an absolute path:

```
% curl -XPOST 'localhost:9200/stopwords' -d'{  
  "settings": {  
    "index": {  
      "analysis": {  
        "analyzer": {  
          "stop1": {  
            "type": "custom",  
            "tokenizer": "standard",  
            "filter": ["my-stop-filter"]  
          }  
        },  
        "filter": {  
          "my-stop-filter": {  
            "type": "stop",  
            "stopwords_path": "config/stopwords.txt"  
          }  
        }  
      }  
    }  
  }'
```

} , }

## TRUNCATE, TRIM AND LIMIT TOKEN COUNT

The next three token filters all deal with limiting the token stream in some way:

- The *truncate* token filter allows you to truncate tokens over a certain length by settings the `length` parameter in the custom configuration, by default it truncates to 10 characters.
  - The *trim* token filter removes all of the whitespace around a token, for example, the token " foo " will be transformed into the token "foo".
  - The *limit* token count token filter limits the maximum number of tokens that a particular field can contain. For example, if you create a customized token count filter with a limit of 8, only the first 8 tokens from the stream will be indexed. This is set using the `max_token_count` parameter, which defaults to 1 (only a single token will be indexed).

## REVERSE

The reverse token filter allows you to take a stream of tokens and reverse each one. This is particularly useful if you are using the edge ngram filter, or want to do leading wildcard searches. Instead of doing a leading wildcard search for `*bar`, which is very slow for Lucene, you can instead search using `rab*` on a field that has been reversed, resulting in a much faster query. Here's an example of reversing a stream of tokens.

#### **Listing 5.4 Example of the reverse token filter**

```
% curl 'localhost:9200/_analyze?tokenizer=standard&filters=reverse' -d'Reverse token filter'

{
  "tokens" : [ {
    "token" : "esreveR",                                     #A
    "start_offset" : 0,
    "end_offset" : 7,
    "type" : "<ALPHANUM>",
    "position" : 1
  }, {
    "token" : "nekot",                                      #B
    "start_offset" : 8,
    "end_offset" : 13,
    "type" : "<ALPHANUM>",
    "position" : 2
  }, {
    "token" : "retlif",                                     #C
    "start_offset" : 14,
    "end_offset" : 20,
    "type" : "<ALPHANUM>",
    "position" : 3
  } ]
}
```

```

}

#A The word "Reverse" that has been reversed
#B The word "token" that has been reversed
#C The word "filter" that has been reversed
```

You can see that each token has been reversed, but the order of the tokens has been preserved.

### UNIQUE

The unique token filter keeps only unique tokens; it keeps the metadata of the first token that matches, removing all future occurrences of it.

```
% curl 'localhost:9200/_analyze?tokenizer=standard&filters=unique' -d'foo bar foo
      bar baz'

{
  "tokens" : [ {
    "token" : "foo",
    "start_offset" : 0,
    "end_offset" : 3,
    "type" : "<ALPHANUM>",
    "position" : 1
  }, {
    "token" : "bar",
    "start_offset" : 4,
    "end_offset" : 7,
    "type" : "<ALPHANUM>",
    "position" : 2
  }, {
    "token" : "baz",
    "start_offset" : 16,
    "end_offset" : 19,
    "type" : "<ALPHANUM>",
    "position" : 3
  } ]
}
```

### ASCII FOLDING

The *ascii\_folding* token filter converts Unicode characters that aren't part of the regular ASCII character set into the ASCII equivalent, if one exists for the character. For example, we can convert the unicode "ü" into an ASCII "u" as seen here:

```
% curl 'localhost:9200/_analyze?tokenizer=standard&filters=asciifolding' -
      d'unicode'

{
  "tokens" : [ {
    "token" : "unicode",
    "start_offset" : 0,
    "end_offset" : 7,
    "type" : "<ALPHANUM>",
    "position" : 1
  } ]
}
```

## SYNONYM

The synonym token filter replaces synonyms for words in the token stream at the same offset as the original tokens, for example, let's take the text "I own that automobile" and the synonym for "automobile", "car". Without the synonym token filter we produce the following tokens:

```
% curl 'localhost:9200/_analyze?analyzer=standard' -d'I own that automobile'
{
  "tokens": [
    {
      "token": "i",
      "start_offset": 0,
      "end_offset": 1,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "own",
      "start_offset": 2,
      "end_offset": 5,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "that",
      "start_offset": 6,
      "end_offset": 10,
      "type": "<ALPHANUM>",
      "position": 3
    },
    {
      "token": "automobile",
      "start_offset": 11,
      "end_offset": 21,
      "type": "<ALPHANUM>",
      "position": 4
    }
  ]
}
```

You can define a custom analyzer that specifies a synonym for "automobile" like this:

```
% curl -XPOST 'localhost:9200/syn-test' -d'{
  "settings": {
    "index": {
      "analysis": {
        "analyzer": {
          "synonyms": {
            "type": "custom",
            "tokenizer": "standard",
            "filter": ["my-synonym-filter"]
          }
        },
        "filter": {
          "my-synonym-filter": {
            "type": "synonym",
            "expand": true,
            "synonyms": ["automobile=>car"]
          }
        }
      }
    }
}
```

```
}
```

And when we use it, you can see that the "automobile" token has been replaced by the "car" token in the results:

```
% curl 'localhost:9200/syn-test/_analyze?analyzer=synonyms' -d'I own that
      automobile'
{
  "tokens" : [
    {
      "token" : "I",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "<ALPHANUM>",
      "position" : 1
    },
    {
      "token" : "own",
      "start_offset" : 2,
      "end_offset" : 5,
      "type" : "<ALPHANUM>",
      "position" : 2
    },
    {
      "token" : "that",
      "start_offset" : 6,
      "end_offset" : 10,
      "type" : "<ALPHANUM>",
      "position" : 3
    },
    {
      "token" : "car",
      "start_offset" : 11,
      "end_offset" : 21,
      "type" : "SYNONYM",
      "position" : 4
    }
  ]
}
```

## 5.5 Ngrams, Edge Ngrams, and Shingles

Ngrams and Edge NGrams are one of the more unique ways of tokenizing text in Elasticsearch. Ngrams are a way of splitting a token into multiple sub tokens for each part of a word. Both the ngram and edge ngram filters allow you to specify what is called a `min_gram` as well as well as a `max_gram` setting. These settings control the size of the tokens that the word is being split up into. This might be kind of confusing, so let's show an example. Assuming you want to analyze the word "spaghetti" with the ngram analyzer, let's start with the simplest case, 1-grams (also known as unigrams): We'll be using the word spaghetti for these examples.

### 1-GRAMS

The 1-grams for "spaghetti" are s, p, a, g, h, e, t, t, i. The string has been split into smaller tokens according to the size of the ngram. In this case, each item is a single character since we are talking about unigrams.

### BIGRAMS

If you were to split the string into bigrams (which means a size of two). You would get the following smaller tokens: sp, pa, ag, gh, he, et, tt, ti

## TRIGRAMS

And again, if we were to use a size of three (which are called trigrams), we get the tokens spa, pag, agh, ghe, het, ett, tti

## SETTING MIN\_GRAM AND MAX\_GRAM

When using this analyzer, there are actually two different sizes you need to set: one specifies the smallest ngrams you want to generate (the `min_gram` setting), and the other specifies the largest ngrams you want to generate. Using our previous example, if we specified a `min_gram` of 2 and a `max_gram` of 3, we would get the combined tokens from our two previous examples:

```
sp, spa, pa, pag, ag, agh, ghe, he, het, et, ett, tt, tti, ti
```

If you were to set the `min_gram` setting to 1 and leave `max_gram` at 3, you get even more tokens starting with s, sp, spa, p, pa, pag, a, ... etc.

Analyzing text in this way has an interesting advantage, when you query for text, your query is going to be split into text the same way, so say you're looking for the incorrectly spelled word "spaghety". Well, one way of searching for this is to do a *fuzzy query*, which allows you to specify an edit distance for words to check matches. However, you can get a similar sort of behavior by using ngrams. Let's compare the bigrams generated for the original word ("spaghetti") with the misspelled one ("spaghety").

- Bigrams for "spaghetti": sp, pa, ag, gh, he, et, tt, ti
- Bigrams for "spaghety": sp, pa, ag, gh, he, et, ty

You can see that six of the tokens overlap; so words with spaghetti in them would still be able to be matched when the query contained spaghety. Keep in mind this means more words that you may not intend, match the original "spaghetti" word, so always make sure to test your query relevancy!

Another useful thing ngrams do is allow you to analyze text when you don't know the language beforehand, or languages that combine words in a different manner than other European languages. This also has an advantage in being able to handle multiple languages with a single analyzer, rather than having to specify different analyzers or using different fields for documents in different languages.

## EDGE NGRAMS

There is a variant to the regular ngram splitting called edge ngrams that builds up ngrams only from the front edge. In our "spaghetti" example, if we set the `min_gram` setting to 2 and the `max_gram` setting to 6, we'd get the following tokens:

```
sp, spa, spag, spagh, spaghe
```

You can see that each token is built from the edge. This can be helpful for searching for word sharing the same prefix, without actually performing a prefix query. If you need to build ngrams from the back of a word, you can put a "reverse" token filter before the edge ngrams token filter; then follow the token filter with another "reverse" token filter, so the entire stream would look like:

```

spaghetti
--reverse-->
ittehgaps
--edge ngrams-->
it, itt, itte, itteh, ittehg
--reverse again-->
ti, tti, etti, hetti, ghetti

```

### NGRAM SETTINGS

Ngrams turn out to be a great way to analyze text when you don't know what language it is, because they can analyze languages that don't have spaces between words. An example of configuring an edge ngram analyzer with min and max grams would look like this.

#### **Listing 5.5 Ngram analysis**

```

% curl -XPOST 'localhost:9200/ng' -d'{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0,
    "index": {
      "analysis": {
        "analyzer": {
          "ng1": {
            "type": "custom",
            "tokenizer": "standard",
            "filter": ["reverse", "ngf1", "reverse"]           #A
          }
        },
        "filter": {
          "ngf1": {
            "type": "edgeNgram",
            "min_gram": 2,                                     #B
            "max_gram": 6                                     #B
          }
        }
      }
    }
  }
}

% curl -XPOST 'localhost:9200/ng/_analyze?analyzer=ng1' -d'spaghetti'
{
  "tokens": [
    {
      "token": "ti",                                         #C
      "start_offset": 0,
      "end_offset": 9,
      "type": "word",
      "position": 1
    },
    {
      "token": "tti",                                        #C
      "start_offset": 0,
      "end_offset": 9,
      "type": "word",
      "position": 1
    }
  ]
}

```

```
        "type" : "word",
        "position" : 1
    }, {
        "token" : "etti",
        "start_offset" : 0,
        "end_offset" : 9,
        "type" : "word",
        "position" : 1
    }, {
        "token" : "hetti",
        "start_offset" : 0,
        "end_offset" : 9,
        "type" : "word",
        "position" : 1
    }, {
        "token" : "ghetti",
        "start_offset" : 0,
        "end_offset" : 9,
        "type" : "word",
        "position" : 1
    }
}

```

## #A Configuring an analyzer for reversing, edge ngrams, and reversing again

## #B Setting the minimum and maximum size for the edge ngram token filter

#C The analyzed tokens from the right-hand side of the word "spaghetti"

## SHINGLES

Along the same lines as ngrams and edge ngrams, we have a filter known as the shingles filter (no not the disease!). The shingles token filter is basically ngrams at the token level, instead of the character level.

Think of our favorite word, spaghetti. Using ngrams with a min and max set to 1 and 3, Elasticsearch will generate the tokens "s", "sp", "spa", "p", "pa", "pag", "a", "ag" and so on. A shingle filter does this at the token level instead, so if you had the text "foo bar baz" and used, again, a min gram of 2 and a max gram of 3, you would generate the following tokens:

foo, foo bar, foo bar baz, bar, bar baz, baz

Why is the single-token output still included? This is because by default the shingle filter includes the original tokens; so the original tokenizer produces the tokens `foo`, `bar`, `bar`, which are then passed to the shingle token filter, which generates the tokens `foo bar`, `foo bar baz`, `bar baz`. All of these tokens are combined to form the final token stream. This behavior can be disabled by setting the `output_unigrams` option to `false`.

Here is an example of a shingle token filter, note that the `min_shingle_size` option must be larger than or equal to 2.

## **Listing 5.6 Shingle token filter example**

```
% curl -XPOST 'localhost:9200/shingle' -d'{  
  "settings": {  
    "index": {  
      "analysis": {
```

```

"analyzer": {
    "shingle1": {
        "type": "custom",
        "tokenizer": "standard",
        "filter": ["shingle-filter"]
    }
},
"filter": {
    "shingle-filter": {
        "type": "shingle",
        "min_shingle_size": 2,                                     #A
        "max_shingle_size": 3,                                     #A
        "output_unigrams": false                                #B
    }
}
}
}

% curl -XPOST 'localhost:9200/shingle/_analyze?analyzer=shingle1' -d'foo bar baz'
{
  "tokens" : [ {
    "token" : "foo bar",                                         #C
    "start_offset" : 0,
    "end_offset" : 7,
    "type" : "shingle",
    "position" : 1
  }, {
    "token" : "foo bar baz",                                       #C
    "start_offset" : 0,
    "end_offset" : 11,
    "type" : "shingle",
    "position" : 1
  }, {
    "token" : "bar baz",                                         #C
    "start_offset" : 4,
    "end_offset" : 11,
    "type" : "shingle",
    "position" : 2
  } ]
}

```

#A Specifying the minimum and maximum shingle size

#B Telling the shingle token filter not to keep the original single tokens

#C The analyzed shingle tokens

## 5.6 Stemming

*Stemming* is the act of reducing a word to its base or root word. This is extremely handy when searching, because it means you are able to match things like the plural of a word, as well as words sharing the root or stem of the word (hence the name stemming!). Let's look at a concrete example. If the word is 'administrations', the root of the word is 'administr'. This allows you to match all of the other roots for this word, like 'administrator', 'administration', and 'administrate'. Stemming is a powerful way of making your searches more flexible than rigid exact matching.

### 5.6.1 Algorithmic stemming

Algorithmic stemming is stemming that is applied by using a formula or set of rules for each token in order to stem it. There are three different algorithmic stemmers that Elasticsearch currently offers the *snowball* filter, the *porter stem* filter, and the *kstem* filter. They behave in almost the same way, but have some slight differences in how aggressive they are with regard to stemming. By "aggressive" we mean that the more aggressive stemmers chop off more of the word than the less aggressive stemmers. Here's a comparison of the different algorithmic stemmers.

stemmer	administrations	administrators	Administate
<b>snowball</b>	administr	administr	Administer
<b>porter stem</b>	administr	administr	Administer
<b>kstem</b>	administration	administrator	Administate

Figure 5.4 Comparing stemming of snowball, porter stem and kstem

To try out how a stemmer stems a word, you can specify it as a token filter with the analyze API:

```
curl -XPOST 'localhost:9200/_analyze?tokenizer=standard&filters=kstem' -d
  'administrators'
```

Use either `snowball`, `porter_stem`, or `kstem` for the filter to test it out.

As an alternative to algorithmic stemming, you can stem using a dictionary, which is just a one-to-one mapping of the original word to its stem.

### 5.6.2 Stemming with dictionaries

Sometimes algorithmic stemmers can stem words in a strange way, because they don't know any of the underlying language. Because of this, there is a more accurate way to stem words that uses a dictionary of words. In Elasticsearch you can use the *hunspell* token filter, combined with a dictionary, to handle the stemming. Because of this, the quality of the stemming is directly related to the quality of the dictionaries that you use. The stemmer will only be able to stem words it has in the dictionary.

When creating a hunspell analyzer, the dictionary files should be in a directory called "hunspell", in the same directory as `elasticsearch.yml`. Inside the `hunspell` directory dictionaries for each language should be in a folder named after its associated locale. For example, to create an index with a hunspell analyzer:

```
% curl -XPOST 'localhost:9200/hspell' -d'{
  "analysis" : {
```

```

    "analyzer" : {
      "hunAnalyzer" : {
        "tokenizer" : "standard",
        "filter" : [ "lowercase", "hunFilter" ]
      }
    },
    "filter" : {
      "hunFilter" : {
        "type" : "hunspell",
        "locale" : "en_US",
        "dedup" : true
      }
    }
  }
}

```

The hunspell dictionary files should be inside `<es-config-dir>/hunspell/en_US` (replace "`<es-config-dir>`" with the location of your Elasticsearch configuration directory). The "en\_US" folder is used since this hunspell analyzer is for the English language, and corresponds to the locale setting in the previous example. You can also change where Elasticsearch looks for hunspell dictionaries by setting the `indices.analysis.hunspell.dictionary.location` setting in `elasticsearch.yml`. To test that your analyzer is working correctly, you can use the `analyze` API again:

```
% curl -XPOST 'localhost:9200/hspell/_analyze?analyzer=hunAnalyzer' -d'administrations'
```

### 5.6.3 Overriding the stemming from a token filter

Sometimes you may not want to have words be stemmed, because either the stemmer treats them incorrectly, or else you want to do exact matches on a particular word. You can accomplish this by placing a *keyword marker* token filter before the stemming filter in the chain of token filters. In this keyword marker token filter, you can specify either a list of words or a file with a list of words that should not be stemmed.

Other than preventing a word from being stemmed, it may be useful for you to manually specify a list of rules to be used for stemming words. You can achieve this with the *stemmer override* token filter, which allows you to specify rules like "cats => cat" to be applied. If the stemmer override finds a rule and applies it to a word, that word not be stemmed by any other stemmer.

Keep in mind with both of these token filters, you'll need to make sure they are placed before any other stemming filters, since they will protect the term from having stemming applied by any other token filters later on in the chain.

## 5.7 Summary

- analysis is the process of making tokens out of the text in fields of your documents. The same process is applied to your search string in queries such as the *match* query. A document matches when its tokens match tokens from the search string
- each field is assigned an analyzer through the mapping. That analyzer can be defined in

your Elasticsearch configuration or index settings. Or, it could be a default analyzer

- analyzers are processing chains made up by a tokenizer, which can be preceded by one or more char filters and succeeded by one or more token filters
- char filters are used to process strings before passing them to the tokenizer. For example, you can use the mapping char filter to change “&” to “and”
- tokenizers are breaking strings into multiple tokens. For example, the whitespace tokenizer can be used to make a token out of each word delimited by a space
- token filters are used to process tokens coming from the tokenizer. For example, you can use stemming to reduce a word to its root and make your searches work across both plural and singular versions of that word
- ngram token filters make tokens out of portions of words. For example, make a token out of every two consecutive letters. This is useful when you want your searches to work even if the search string contains typos
- edge ngrams are like ngrams, but they only work from the beginning or the end of the word. For example, you can take “event” and make “e”, “ev” and “eve”
- shingles are like ngrams at the phrase level. For example, you can generate terms out of every two consecutive words from a phrase. This is useful when you want to boost the relevance of multiple-word matches, like in the short description of a product. We'll talk more about relevancy in the next chapter

## 7

# *Exploring your data with Aggregations*

## This chapter covers

- metrics aggregations
- single and multi-bucket aggregations
- nesting aggregations
- relations among queries, filters and aggregations

So far in this book, we've concentrated on the use-case of searching: you have many documents and the user wants to find the most relevant matches to some keywords. There are more and more use-cases when users aren't interested in specific results. Instead, they want to get statistics from a set of documents. These statistics might be hot topics for news, revenue trends for different products, the number of unique visitors of your website, and much more.

Aggregations in Elasticsearch solve this problem by loading the documents matching your search, and doing all sorts of computations, such as counting the terms of a string field, or calculating the average on a numeric field. Let's look at how aggregations works, using an example from the get-together site we've worked with in previous chapters: a user entering your site may not know what groups to look for. To give the user something to start with, you could make the UI show the most popular tags for existing groups of your get-together site, as illustrated in Figure 7.1.

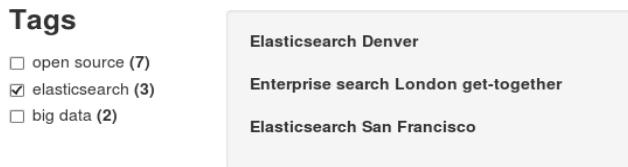


Figure 7.1 Example use-case of aggregations: top tags for get-together groups

Those tags would be stored in a separate field of your group documents. The user could then select a tag, and filter down to only documents containing that tag. This makes it easier for users to find groups relevant to their interests.

To get such a list of popular tags in Elasticsearch, you'd use aggregations, and in this specific case, the *terms aggregation*, on the `tags` field, which counts occurrences of each term in that field, and returns the most frequent terms. Many other types of aggregations are also available, and we'll discuss them later in this chapter. For example, you can use a *date histogram aggregation* to show how many events happened in each month of the last year, the *average aggregation* to show you the average number of attendees for each event, or even find out which users have similar taste for events as you do by using the *significant terms aggregation*.

### What about facets?

If you've used Lucene or Solr, or even Elasticsearch for some time, you might have heard about facets. Facets are similar to aggregations, because they also load the documents matching your query and perform computations in order to return statistics. As of version 1.2, Elasticsearch still supports facets.

The main difference between aggregations and facets is that you can't nest multiple types of facets in Elasticsearch, which limits the possibilities for exploring your data. For example, if you had a blogging site, you can use the **terms facet** to find out the "hot topics" this year, or you can use the **date histogram facet** to find out how many articles are posted each day, but you can't find the number of posts per day, separately for each topic. You would be able to do that if you could nest the date histogram facet under the terms facet.

Aggregations were born to remove this limit and allow you to get deeper insights from your documents. For example, if you store your online shop logs in Elasticsearch, you can use aggregations to find not only the best-selling products, but also best selling products in each country, the trends for each product in each country and so on.

There are types of aggregations that don't have a facet equivalent, but there are no facets that can't be done using aggregations. In practice, aggregations are the new facets in Elasticsearch, and this is the reason why we'll skip facets in this book.

In this chapter, we'll first discuss the common traits of all aggregations: how you'd run them and how they relate to the queries and filters you learned in previous chapters. Then, we'll

dive into the particularities of each type of aggregation, and in the end, we'll show you how to combine different aggregation types.

### AGGREGATION CATEGORIES: METRICS AND BUCKETS

Aggregations are divided in two main categories: metrics and bucket.

*Metrics* aggregations refer to the statistical analysis of a group of documents, resulting in metrics such as the minimum value, maximum value, standard deviation and much more. For example, you can get the average price of items from an online shop, or the number of unique users logging on to it.

*Bucket* aggregations divide matching documents into one or more containers (buckets), and then give you the number of documents in each bucket. The terms aggregation, that would give you the most popular tags in figure 1, will make a bucket of documents for each tag, and give you back the document count for each bucket.

Within a bucket aggregation, you can nest other aggregations – making the “child” aggregation run on each bucket of documents generated by the “parent.” You can see an example in figure 7.2 below.

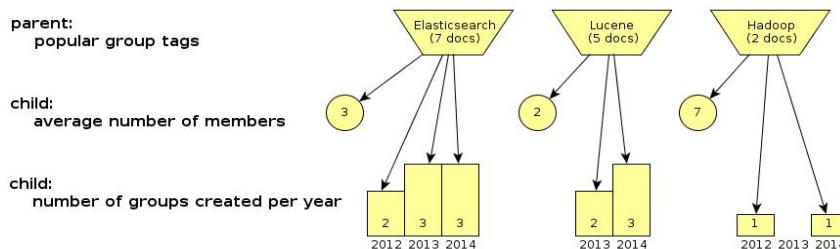


Figure 7.2 The terms bucket aggregation allows you to nest other aggregations within it

Looking at the figure from the top down, you can see that if you're using the terms aggregations to get the most popular group tags, you can also get the average number of members for groups matching each tag. You could also ask Elasticsearch to give you, per tag, the number of groups created in each month of the year.

As you may imagine, you can combine many types of aggregations in many ways. To get a better view of the available options, we'll go through metrics and bucket aggregations, then we'll discuss how you can combine them. But first, let's see what's common for all types of aggregations: how to write them and how they relate to your queries.

## 7.1 Anatomy of an aggregation

All aggregations, no matter their type, follow some rules:

- You define them in the same JSON request as your queries, and you mark them by the key aggregations or aggs. You need to give each one a name, specify the type and the options specific to that type.
- They run on the results of your query. Documents that don't match your query are not

accounted for. Unless you include them with the global aggregation, which is a bucket aggregation that will be covered later in this chapter.

- You can filter down results of your query more, without influencing aggregations. To do that, you'd use post filters. For example, when searching for a keyword in an online shop, you can build statistics on all items matching the keyword, but use post filters to only show results that are in stock

Let's take a look at the popular terms aggregation, which you've already seen in the intro to this chapter. The example use-case was getting the most popular subjects (tags) for existing groups of your get-together site. We'll use this same terms aggregation to explore the rules that all aggregations must follow.

### 7.1.1 Structure of an aggregation request

In listing 7.1, you'll run a terms aggregation that will give you the most frequent tags in the get-together groups. The structure of this terms aggregation will apply to every other aggregation.

#### **Listing 7.1 Using the terms aggregation to get top tags**

```
curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "aggregations" : {
    "top_tags" : { #A
      "terms" : { #B
        "field" : "tags.verbatim" #C
      }
    }
  }
}'

### reply
[...]
  "hits" : { #E
    "total" : 5,
    "max_score" : 1.0,
    "hits" : [ {
      [...]
      "name": "Denver Clojure",
      [...]
      "name": "Elasticsearch Denver",
      [...]
    },
    "aggregations" : { #F
      "top_tags" : { #G
        "buckets" : [ {
          "key" : "big data",
          "doc_count" : 3 #H
        },
        {
          "key" : "open source",
          "doc_count" : 3 #I
        },
        {
          "key" : "denver",
          "doc_count" : 2
        }
      ]
    }
  }
}
```

#A Aggregations key indicates that this is the aggregations part of the request  
#B Give the aggregation a name  
#C Specify the aggregation type `terms`  
#D Verbatim field is used to have “big data” as a single term, instead of “big” and “data” separately  
#E The list of results is there anyway, as if you hit the `_search` endpoint with no query  
#F Aggregation results begin here  
#G Aggregation name, as specified  
#H Each unique term is an item in the bucket  
#I For each term, you see now many times it appeared

- At the top level, there's the aggregations key, which can be shortened to `aggs`.
- On the next level, you have to give the aggregation a name. You can see that name in the reply. This is useful when you use multiple aggregations in the same request, so you can easily see the meaning of each set of results.
- Finally, you have to specify the aggregation type `terms`, and the specific option. In this case, we'll have the field name.

The aggregation request from listing 7.1 hits the `_search` endpoint, just like the queries you've seen in previous chapters. In fact, you also get back 10 group results. This is all because no query was specified, which will effectively run the `match_all` query you've seen in chapter 4. So your aggregation will run on all the group documents. Running a different query will make the aggregation run through a different set of documents.

### Field data cache for faster aggregations

When you run a regular search, it goes pretty fast because of the nature of the inverted index: you have a limited number of terms to look for; Elasticsearch will identify documents containing those terms, and return the results. Aggregations, on the other hand, require more work because it has to pull all those terms from fields you need to aggregate on, and then do the counting or other computation.

To speed up things, Elasticsearch loads those terms in memory, in the *field data* cache. The more terms it has to deal with, the more memory will be used by the field data cache. That's why you have to make sure you have enough memory, especially when you're doing aggregations on large numbers of documents, or if fields are analyzed and you have more than one term per document.

By default, the field data cache is unlimited, so running many expensive aggregations can trigger an out-of-memory error. You can change the configuration to make old items expire (`indices.fielddata.cache.expire`) and you can put a limit on the amount of memory that can be occupied by this cache (`indices.fielddata.cache.size`). Another helpful feature is the “field data circuit breaker,” which will raise an exception if an aggregation uses more field cache than a certain limit. That limit can be adjusted via `indices.fielddata.breaker.limit` in the configuration or cluster settings.

### 7.1.2 Aggregations run on query results

Computing metrics over the whole data set is just one of the possible use-cases for aggregations. Often, you want to compute metrics in the context of a query. For example, if you're searching for groups in Denver, you probably want to see the most popular tags for those groups only. As you'll see in listing 7.2, this is the default behavior for aggregations. Unlike in listing 7.1, where the implied query was `match_all`, here we query for Denver in the `location` field, and aggregations will only be about groups from Denver.

#### **Listing 7.2 Getting top tags for groups in Denver**

```
curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "query": {
    "match": {
      "location": "Denver" #A
    }
  },
  "aggregations" : {
    "top_tags" : {
      "terms" : {
        "field" : "tags.verbatim"
      }
    }
  }
}'

## reply
[...]
  "hits" : {
    "total" : 2, #B
    "max_score" : 1.44856,
    "hits" : [ {
      [...]
      "name": "Denver Clojure",
      [...]
      "name": "Elasticsearch Denver",
      [...]
    },
    "aggregations" : {
      "top_tags" : {
        "buckets" : [ {
          "key" : "denver", #C
          "doc_count" : 2 #C
        },
        {
          "key" : "big data", #C
          "doc_count" : 1 #C
        }
      ]
    }
  }
}

#A In this query we only look for groups in Denver
#B Fewer results than in listing 7.1, because we only look for Denver groups
#C Tags are only counted for Denver groups, so they look different than in listing 7.1
```

#### **FROM AND SIZE**

Recall from chapter 4 that you can use the `from` and `size` parameters of your query control the pagination of results. These parameters have no influence on aggregations, because aggregations always run on all the documents matching a query.

If you want to restrict query results more, without restricting aggregations, too, you can use post filters. We'll discuss post filters and the relationship between filters and aggregations in general next.

### 7.1.3 Filters and aggregations

In chapter 4 you saw that for most query types there is a filter equivalent. Because filters don't calculate scores and are cacheable, they're faster than their query counterparts. You've also learned that you should wrap filters in a filtered query, like this:

```
% curl localhost:9200/get-together/group/_search?pretty -d '{
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "location": "denver"
        }
      }
    }
  }
}'
```

Using the filter this way is good for the overall query performance, because the filter runs first. Then, the query – which is typically more performance-intensive – runs only on documents matching the filter. As far as aggregations are concerned, they only run on documents matching the overall filtered query, as shown in Figure 7.3.

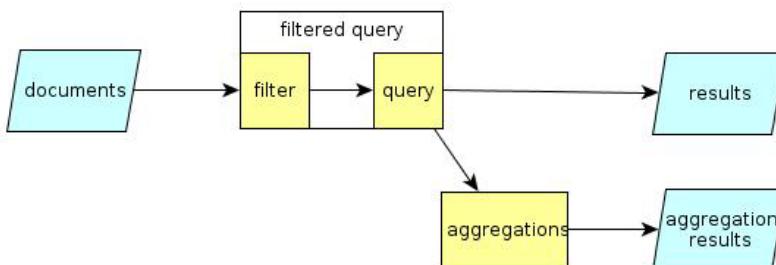


Figure 7.3 A filter wrapped in a filtered query runs first, and restricts both results and aggregations

"Nothing new so far," you might say, "the *filtered query* behaves like any other query when it comes to aggregations," and you'd be right. But there is also another way of running filters: by using a *post filter*, which will run after the query, and independent of the aggregation. The following request will give the same results as the previous filtered query:

```
% curl localhost:9200/get-together/group/_search?pretty -d '{
  "post_filter": {
    "term": {
      "location": "denver"
    }
  }
}'
```

As illustrated in figure 7.4, the post filter differs from the filter in the filtered query in two ways:

- Performance: The post filter runs after the query, making sure the query will run on all documents, and the filter run only on those matching the query. The overall request is typically slower than the filtered query equivalent, except when you have “expensive” filters, like the script filter.
- Document set processed by aggregations: If a document doesn’t match the post filter, it will still be accounted for by aggregations.

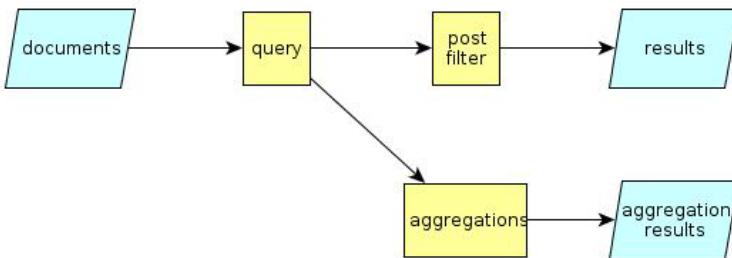


Figure 7.4 Post filter runs after the query and doesn't affect aggregations

Now that you understand the relation between queries, filters and aggregations, as well as the overall structure of an aggregation request, we can dive deeper into Aggregations Land and explore different aggregation types. We'll start with metrics aggregations, then go to bucket aggregations, then we'll discuss how to combine them to get powerful insights from your data in real-time.

## 7.2 Metrics aggregations

Metrics aggregations are all about extracting statistics from groups of documents, or, as we'll explore in section 7.4, buckets of documents coming from other aggregations.

These statistics are typically done on numeric fields, such as the minimum or average price. You can get each such statistic separately, or you can get them together, via the stats aggregation. More advanced statistics, such as the sum of squares or the standard deviation are available through the extended\_stats aggregation.

For both numeric and non-numeric fields you can get the number of unique values using the cardinality aggregation, which will be shown in section 7.2.3.

### 7.2.1 Statistics

Let's begin looking at metrics aggregations by getting some statistics on the number of attendees for each event. To do that, you'll need to index the sample dataset from the code samples that come with the book.

Once you have the documents indexed by running `populate.sh`, you can see that event documents contain an array of attendees. We can calculate the number of attendees at query

time, through a script, which we'll show in the next listing. We discussed scripting in chapter 3, when you used them for updating documents. In general, with Elasticsearch queries you can build a "script field", where you put a (typically small) piece of code that returns a value for each document. In this case, the value will be the count of elements of the attendees array.

### The flexibility of scripts comes with a price

Scripts are very flexible when it comes to querying, but you have to be aware of the caveats in terms of performance and security.

Usually, scripts slow down aggregations, because they have to be run on every document. To avoid the need of running a script, you can do the calculation at index time. In this case, you can extract the number of attendees for every event and add it to a separate field before indexing it.

In most Elasticsearch deployments, the user specifies a query string and it's up to the server-side application to construct the query out of it. But if you allow users to specify any kind of query, including scripts, someone might exploit this and run malicious code.

Because of this, if you install Elasticsearch from a package it will run as its own user instead of root. Also, running scripts is disabled by default. To enable it, you'd have to add the following line to your elasticsearch.yml configuration file:

```
script.disable_dynamic: false
```

In the following listing, we'll request statistics on the number of attendees for all events. To get the number of attendees in the script, we'll use `doc['attendees'].values.length` to get the array of attendees. Adding the `length` method to that will return their number.

### Listing 7.3 Getting stats for the number of event attendees

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "size" : 0,                                     #A
  "aggregations": {
    "attendees_stats": {
      "stats": {
        "script": "doc['''attendees'''].values.length"   #B
      }
    }
  }
}## reply
[...]
  "aggregations" : {
    "attendees_stats" : {
      "count" : 15,
      "min" : 3.0,
      "max" : 5.0,
      "avg" : 3.8666666666666667,
      "sum" : 58.0
    }
  }
}'
```

#A As we only care about aggregations, we don't ask for any result  
#B Script to generate the number of attendees. Use "field" instead of "script" to point to a real field

You can see that we got back the minimum number of attendees per event, the maximum, the sum, and the average. We also got the number of values these statistics were computed on.

### SEPARATE STATISTICS

If you only need one of those statistics, you can get it separately. For example, the average number of attendees per event will be calculated through the `avg` aggregation in this next listing.

#### **Listing 7.4 Getting the average number of event attendees**

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "attendees_avg": {
      "avg": {
        "script": "doc[''attendees''].values.length"
      }
    }
  }
}'

### reply
[...]
  "aggregations" : {
    "attendees_avg" : {
      "value" : 3.8666666666666667
    }
  }
}'
```

Similar to the `avg` aggregation, you can get the other metrics through the `min`, `max`, `sum` and `value_count` aggregations. You'd just have to replace `avg` from listing 7.4 with the needed aggregation name. The advantage of separate statistics is that Elasticsearch won't spend time computing metrics that you don't need.

### 7.2.2 Advanced statistics

In addition to statistics gathered by the `stats` aggregation, you can get the sum of squares, variance and standard deviation of your numeric field by running the `extended_stats` aggregation, shown in this next listing.

#### **Listing 7.5 Getting extended statistics on the number of attendees**

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "attendees_extended_stats": {
      "extended_stats": {
        "script": "doc[''attendees''].values.length"
      }
    }
  }
}'

### reply
  "aggregations" : {
```

```

"attendees_extended_stats" : {
  "count" : 15,
  "min" : 3.0,
  "max" : 5.0,
  "avg" : 3.8666666666666667,
  "sum" : 58.0,
  "sum_of_squares" : 230.0,
  "variance" : 0.3822222222222135,
  "std_deviation" : 0.6182412330330462
}
}

```

All these statistics are calculated by looking at all the values in the document set matching the query, so they're 100% accurate all the time. Next, we'll look at some statistics that use approximation algorithms, trading some of the accuracy for speed and less memory consumption.

### 7.2.3 Approximate statistics

Some statistics can be calculated with very good precision – though not 100% - by just looking at some of the values from your documents. This will limit both their execution time and their memory consumption.

Here, we'll look at how to get two types of such statistics from Elasticsearch: percentiles and cardinality. *Percentiles* are values below which you can find X% of the total values, where X is the given percentile. This is useful, for example, when have an online shop, you log the value of each shopping cart, and you want to see in which price range are most shopping carts. Maybe most of your users only buy an item or two and there are the upper 10% who buy a lot of items and generate most of your revenue.

*Cardinality* is the number of unique values in a field. This is useful, for example, when you want the number of unique IPs accessing your website.

#### PERCENTILES

For percentiles, let's think about the number of attendees for events once again, and determine the maximum number of attendees we'll consider "normal," and the number we'll consider "high." In listing 7.6, we'll calculate the 80<sup>th</sup> percentile and the 99<sup>th</sup>. We'll consider numbers under the 80<sup>th</sup> to be normal, and numbers under the 99<sup>th</sup> high, and we'll ignore the upper 1%, because they are "exceptionally high".

To accomplish this, we'll use the `percentiles` aggregation and we'll set the `percents` array to 80 and 99 in order to get these specific percentiles.

#### Listing 7.6 Getting the 80<sup>th</sup> and the 99<sup>th</sup> percentile from the number of attendees

```

curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "attendees_percentiles": {
      "percentiles": {
        "script": "doc[''attendees''].values.length",
        "percents": [80, 99]
      }
    }
  }
}'

```

```

        }
    }'
### reply
"aggregations" : {
  "attendees_percentiles" : {
    "values" : {
      "80.0" : 4.0,
      "99.0" : 5.0
    }
  }
}

```

For small data sets like the code samples, you have 100% accuracy, but this may not happen with large data sets in production. With the default settings, you have over 99.9% accuracy for most data sets for most percentiles. The specific percentile matters, because accuracy is at its worst for the 50<sup>th</sup> percentile, and as you go towards 0 or 100, it gets better and better.

You can trade memory for accuracy by increasing the `compression` parameter from the default 100. Memory consumption increases proportionally to the compression, which in turn controls how many values are taken into account when approximating percentiles.

## CARDINALITY

For cardinality, let's imagine you want the number of unique members of your get-together site. The following listing shows you how to do that with the `cardinality` aggregation.

### **Listing 7.7 Getting the number of unique members through the cardinality aggregation**

```

curl localhost:9200/get-together/group/_search?pretty -d '{
  "aggregations": {
    "members_cardinality": {
      "cardinality": {
        "field": "members"
      }
    }
  }
}'
### reply
"aggregations" : {
  "members_cardinality" : {
    "value" : 8
  }
}

```

Like the percentiles aggregation, the cardinality aggregation is approximate. To understand the benefit of such approximation algorithms, let's take a closer look at the alternative. Before the cardinality aggregation was introduced in version 1.1.0, the common way to get the cardinality of a field was by running the `terms` aggregation, you saw in section 7.1. Because the `terms` aggregation will get the counts of each term for top N terms – where N is the configurable `size` parameter – if you specify a `size` large enough, you could get all the unique terms back. Counting them will give you the cardinality.

Unfortunately, this approach only worked for fields with relatively low cardinality and low number of documents. Otherwise, running a terms aggregation with a huge size requires a lot of resources:

- Memory: because all the unique terms need to be loaded in memory in order to be counted.
- CPU: because those terms have to be returned in order – by default the order is on how many times each term occurs.
- Network: because from each shard, the large array of sorted unique terms has to be transferred to the node that received the client request. That node also has to merge per-shard arrays into one big array and transfer it back to the client.

This is where approximation algorithms come into play. The cardinality field works with an algorithm called HyperLogLog++ that hashes values from the field you want to examine, and uses the hashes to approximate the cardinality. It loads only some of those hashes into memory at once, so the memory usage will be constant no matter how many terms you have.

**NOTE** For more details on the HyperLogLog++ algorithm, have a look at the original paper from Google: [static.googleusercontent.com/media/research.google.com/fr/pubs/archive/40671.pdf](http://static.googleusercontent.com/media/research.google.com/fr/pubs/archive/40671.pdf)

#### MEMORY AND CARDINALITY

We said the memory usage of the cardinality aggregation is constant, but how large that constant would be? You can configure it though the `precision_threshold` parameter. The higher the threshold, the more precise the results, but more memory is consumed. If you run the cardinality aggregation on its own, it will take about `precision_threshold` times 8 bytes of memory for each shard that gets hit by the query.

The cardinality aggregation, like all other aggregations, can be nested under a bucket aggregation. When that happens, the memory usage is further multiplied by the number of buckets generated by the “parent” aggregations.

**TIP** For most cases, the default `precision_threshold` will work well, because it provides a good trade-off between memory usage and accuracy, and adjusts itself depending on the number of buckets.

Next, we'll look at the choice of multi-bucket aggregations. But before going there, Table 7.1 gives you a quick overview of each metrics aggregation and the typical use-case.

Table 7.1 Metrics aggregations and typical use-cases

Aggregation type	Example use-case
Stats	Same product sold in multiple stores. Gather statistics on the price: how many stores have it, what's the minimum, maximum and average price
individual stats (min, max, sum, avg, value_count)	Same product sold in multiple stores. Show "prices starting from" and then the minimum price.
extended_stats	Documents contain results from a personality test. Gather statistics from that group of people, such as the variance and the standard deviation.
Percentiles	Access times on your website: what are "usual" delays and how long are the longest response times.
Cardinality	Number of unique IPs accessing your service

### 7.3 Multi-bucket aggregations

As you've seen in the previous section, metrics aggregations are about taking all your documents and generating one or more numbers that describe them. Multi-bucket aggregations are about taking those documents and putting them into buckets - like the group of documents matching each tag. Then, for each bucket, you'll get one or more numbers that describe the bucket, such as counting the number of groups for each tag.

This bucket approach comes in handy when nesting multiple kinds of aggregations, and we'll discuss how you can do this in the next section. For now, let's see what kind of multi-bucket aggregations are available and where they are typically useful.

- *Terms aggregations* are all about figuring out the frequency of each term in your documents. There's the terms aggregation, which you've seen a couple of times already, that gives you back the number of times each term appears. It's useful for figuring out things like frequent posters on a blog or popular tags. There's also the significant terms aggregation, which will give you back the difference between the occurrence of a term in the whole index, and its occurrence in your query results. This is useful for suggesting terms that are significant for the search context, like "elasticsearch" would be for the context of "search engine"
- *Range aggregations* are all about figuring out how many documents fall into which numerical, date or IP address range. This is useful when analyzing data where the user has fixed expectations. For example, if someone is searching for the laptop in an online shop, you know the price ranges that are most popular

- *Histogram aggregations* – either numerical or date – are similar to range aggregations, but instead of requiring you to define each range, you have to define an interval, and Elasticsearch will build buckets based on that interval. This is useful when you don't know where the user is likely to look. For example, show a chart of how many events occur each month.

Figure 7.5 shows an overview of the major types of multi-bucket aggregations.

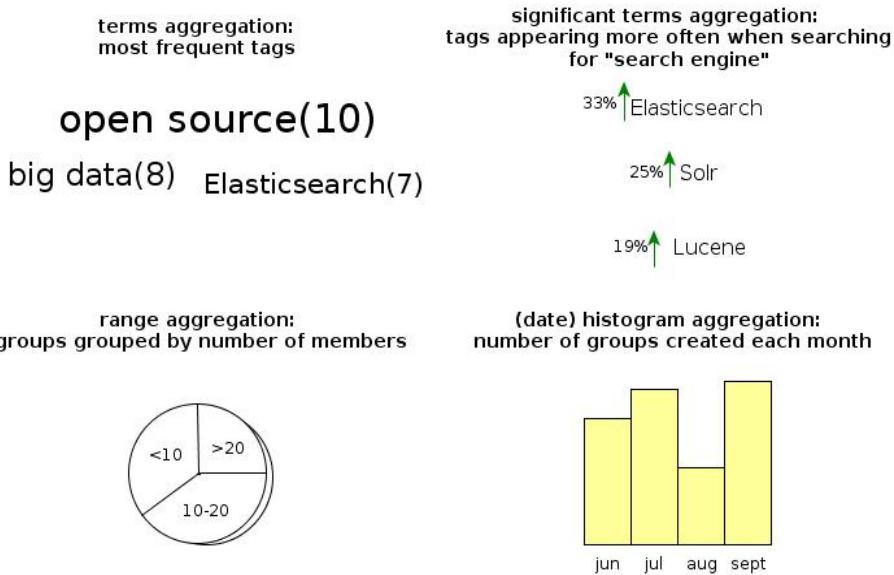


Figure 7.5 Major types of multi-bucket aggregations

Next, let's zoom into each of these multi-bucket aggregations and see how you can use them.

### 7.3.1 Terms aggregations

We first looked at terms aggregation in section 7.1 as an example of how all aggregations work. The typical use-case is to get the top frequent X, where X would be a field in your document, like the name of a user, a tag or a category. Because the terms aggregation counts every term and not every field value, you'll normally run this aggregation on a non-analyzed field, because you want big data to be counted once, and not once for big and once for data.

You could use the terms aggregation to extract the most frequent terms from an analyzed field, like the description of an event. You can use this information to generate a word cloud, like the one in figure 7.6. Just make sure you have enough memory for loading all the fields in memory, if you have many documents or the documents contain many terms.



Figure 7.6 A terms aggregation can be used to get term frequencies and generate a word cloud

By default, the order of terms is by their count, descending, which fits all the top frequent X use-cases. But you can order terms ascending, or by other criteria, like the term name itself. The following listing shows how to list the group tags ordered alphabetically by using the `order` property.

#### **Listing 7.8 Ordering tag buckets by name**

```
curl localhost:9200/get-together/group/_search?pretty -d '{
  "aggregations": {
    "tags": {
      "terms": {
        "field": "tags.verbatim",
        "order": {
          "_term": "asc"
        }
      }
    }
  }
}'

### reply
{
  "aggregations" : {
    "tags" : {
      "buckets" : [
        {
          "key" : "apache lucene",
          "doc_count" : 1
        },
        {
          "key" : "big data",
          "doc_count" : 3
        },
        {
          "key" : "closure",
          "doc_count" : 1
        }
      ]
    }
  }
}
```

If you're nesting a metric aggregation under your terms aggregation, you can order terms by the metric, too. For example, you could use the average metric aggregation under your `tags` aggregation from listing 7.7, to get the average number of group members per tag. And you can order tags by the number of members by referring your metric aggregation name, like `avg_members: desc`.

#### **WHICH TERMS TO INCLUDE IN THE REPLY**

By default, the terms aggregation will return only the top 10 terms by the order you selected. You can, however, change that number though the `size` parameter. Setting `size` to 0 will get you all the terms, but it's dangerous to use with a high-cardinality field, because returning a very large result is CPU-intensive to sort and might saturate your network.

To get back the top 10 terms – or the number of terms you configure with `size` – Elasticsearch has to get the top 10 terms for each shard and aggregate the results. The process is shown in the figure 7.7, with `size=2` for clarity.

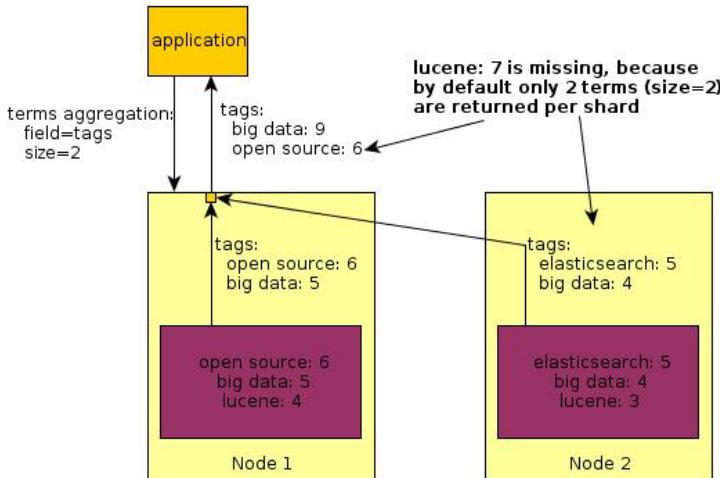


Figure 7.7 Sometimes, the overall top X is inaccurate, because only top X terms are returned per shard

This mechanism implies that you might get inaccurate counters for some terms, if those terms don't make it into the top 10 for each individual shard. This can even result in missing terms, like in the next figure where `lucene`, with a total value of 7, isn't returned in the top 2 overall tags because it didn't make the top 2 for each shard.

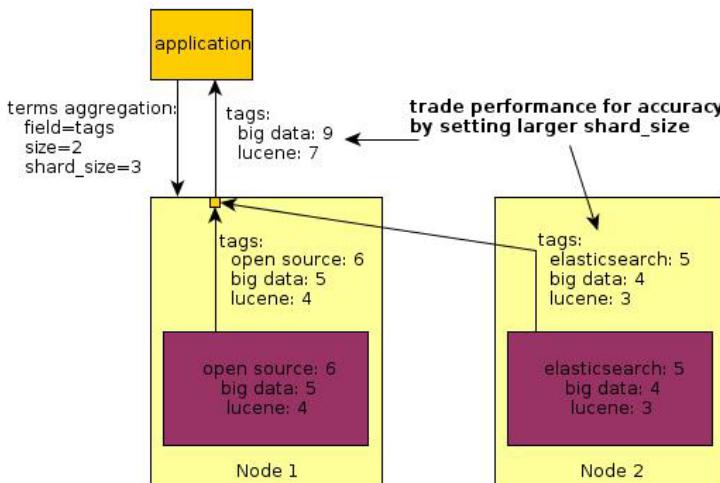


Figure 7.8 Reducing inaccuracies by increasing shard\_size

To solve this problem, you can get more than 10 results from each shard by configuring `shard_size`, while retaining the same value of `size`. You will trade some performance for this, because aggregating larger per-shard result sets is more expensive.

At the other end of the accuracy spectrum, you could consider terms with low frequency irrelevant and exclude them from the result set entirely. This is especially useful when you sort terms by something else than frequency – which makes it likely for low-frequency terms to appear – but don't want to “pollute” the results with irrelevant results like typos. To do that, you'll need to change the `min_doc_count` setting from the default value of 1.

Finally, you can include and exclude specific terms from the result. You'd do that by using the `include` and `exclude` options, and provide regular expressions as values. Using `include` alone will include only terms matching the pattern, using `exclude` alone will include terms that don't match. Using both will have `exclude` take precedence: included terms will match the `include` pattern but won't match the `exclude` pattern.

The following listing will show you how to return counters for only tags containing “search.”

### **Listing 7.9 Creating buckets only for terms containing “search”**

```
curl localhost:9200/get-together/group/_search?pretty -d '{
  "aggregations": {
    "tags": {
      "terms": {
        "field": "tags.verbatim",
        "include": ".*search.*"
      }
    }
  }
}'

### reply
{
  "aggregations" : {
    "tags" : {
      "buckets" : [ {
        "key" : "elasticsearch",
        "doc_count" : 2
      }, {
        "key" : "enterprise search",
        "doc_count" : 1
      }
    ]
  }
}
```

### **SIGNIFICANT TERMS**

The significant terms query is useful if you want to see which terms have higher frequencies than normal in your current search results. Lets take the example of get-together groups: in all the groups out there, the term `clojure` may not appear frequently enough to count. Let's assume that it appears 10 times out of 1,000,000 terms (0.0001%). If you restrict your search for Denver, let's say it appears 7 times out of 10,000 terms (0.007%). The percentage is significantly higher than before and indicates a strong Clojure community in Denver, compared to the rest. It doesn't matter that other terms, such as `programming` or `devops` have a much higher absolute frequency.

The significant terms query is much like the terms query in the sense that it's counting terms. But the resulting buckets are ordered by a score, which represents the difference in

percentage between the foreground documents (that 0.007% in the previous example) and the background documents (0.0001%). The foreground documents are those matching your query and the background documents are all the documents from the index.

In the following listing, we'll try to find out which users of the get-together site have a similar preference to Lee for events. To do that, we'll query for events where Lee attends, and use the significant terms aggregation to see which event attendees participate to those events more, compared to the overall set of events they attend to.

### **Listing 7.10 Finding attendees attending similar events to Lee**

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "query": {
    "match": {
      "attendees": "Lee"          #A
    }
  },
  "aggregations": {
    "significant_attendees": {
      "significant_terms": {
        "field": "attendees",     #B
        "min_doc_count": 2,       #C
        "exclude": "lee"          #D
      }
    }
  }
}' ## reply
"aggregations" : {
  "significant_attendees" : {
    "doc_count" : 5,             #E
    "buckets" : [ {
      "key" : "greg",           #F
      "doc_count" : 3,           #F
      "score" : 1.799999999999998, #F
      "bg_count" : 3             #F
    }, {
      "key" : "mike",            #G
      "doc_count" : 2,           #G
      "score" : 1.2000000000000002, #G
      "bg_count" : 2             #G
    }, {
      "key" : "daniel",          #H
      "doc_count" : 2,           #H
      "score" : 0.6666666666666667, #H
      "bg_count" : 3             #H
    }
  }
}
```

#A Foreground documents are events Lee attends to

#B We need attendees that appear more in these events than overall

#C Take only attendees that participated to at least 2 events

#D Exclude Lee from the analyzed terms, he has the same taste as himself

#E Total number of events Lee attends to is 5

#F Greg has similar taste: attended 3 events in total, all of them with Lee

#G Mike is after him, with 2 events in total, all of them with Lee

#H Daniel is last. He went to 3 events, but only 2 of them with Lee

As you might have guessed from the listing, the significant terms aggregation has the same size, shard\_size, min\_doc\_count, include and exclude options as the terms aggregation, that let you control the terms you get back. In addition to those, it allows you to change the background documents from all the documents in the index to only those matching a defined filter in the “background\_filter” parameter. For example, you may know that Lee only participates in technology events, so you can filter those to make sure that events irrelevant to him aren't taken into account.

Both the terms and significant terms aggregations work well for string fields. For numeric fields, range and histogram aggregations are more relevant, and we'll look at them next.

### 7.3.2 Range aggregations

The terms aggregation is most often used with strings, but it works with numeric values, too. This is useful when you have low cardinality, like when you want to give counts on how many laptops have 2 years of warranty, how many have 3 and so on.

With high-cardinality fields, such as ages or prices, you're most likely looking for ranges. For example, you may want to know how many of your users are between 18 and 39, how many between 40 and 60, and so on. You can still do that with the terms aggregation, but it's going to be tedious: in your application, you'd have to add up counters for ages 18, 19, and so on until you get to 39 to get the first bucket. And if you want to add sub-aggregations, like the ones you'll see later in this chapter, things will get even more complicated.

To solve this problem for numerical values, you have the *range aggregation*. Like the name suggests, you'd give the numerical ranges you want, and it will count the documents with values that fall in each bucket. You can use those counters to represent the data in a graphical way, for example with a pie chart, as shown below.

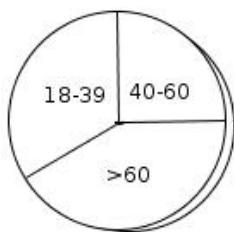


Figure 7.9 Range aggregations give you counts of documents for each range. This is good for pie charts

Recall from chapter 3 that date strings are stored as type `long` in Elasticsearch, representing the Unix time in milliseconds. To work with date ranges, you have a variant of the range aggregation called the “date range aggregation.”

#### RANGE AGGREGATION

Let's get back to our get-together site example and do a breakdown of events by their number of attendees. We'll do it with the range aggregation and give it an array of ranges. The thing to keep in mind here is that the minimum value from the range (the key from) is included in

the bucket, while the maximum value (`to`) is excluded. In the next listing, we'll have three categories:

- events with fewer than 4 members
- events with at least 4 members, but fewer than 6
- events with at least 6 members

**NOTE** Ranges don't have to be adjacent, they can be separated or they can overlap. In most cases it makes sense to cover all values, but you don't need to.

### Listing 7.11 Using a range aggregation to divide events by the number of attendees

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "attendees_breakdown": {
      "range": {
        "script": "doc[''attendees''].values.length", #A
        "ranges": [
          { "to": 4 }, #B
          { "from": 4, "to": 6 }, #B
          { "from": 6 } #B
        ]
      }
    }
  }
}## reply
"aggregations" : {
  "attendees_breakdown" : {
    "buckets" : [ {
      "key" : "-4.0",
      "to" : 4.0,
      "to_as_string" : "4.0",
      "doc_count" : 4 #C
    }, {
      "key" : "4.0-6.0",
      "from" : 4.0,
      "from_as_string" : "4.0",
      "to" : 6.0,
      "to_as_string" : "6.0",
      "doc_count" : 11
    }, {
      "key" : "6.0-*",
      "from" : 6.0,
      "from_as_string" : "6.0",
      "doc_count" : 0 #D
    }
  }
}
```

#A We use a script here to get the number, like in previous examples

#B The intervals we want to use for counting

#C For each interval, you're getting the document count

#D Even if that value is 0

You can see from Listing 7.11 that you don't have to specify both `from` and `to` for every range in the aggregation. Omitting one of these parameters will remove the respective boundary and this enables you to search for all events with less than 4 members, or with at least 6.

## DATE RANGE AGGREGATION

As you might imagine, the date range aggregation works just like the range aggregation, except you put date strings in your range definitions. And because of that, you should define the date format, so Elasticsearch will know how to translate the string you give it into the numerical UNIX time, which is how date fields are stored.

In the following listing, we'll divide events in two categories: before July 2013 and starting with July 2013. You can use a similar approach to count future events and past events, for example.

### Listing 7.12 Using a date rage aggregation to divide events by scheduled date

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "dates_breakdown": {
      "date_range": {
        "field": "date",
        "format": "YYYY.MM",          #A
        "ranges": [
          { "to": "2013.07" },        #B
          { "from": "2013.07" }       #B
        ]
      }
    }
  }
}'

### reply
"aggregations" : {
  "dates_breakdown" : {
    "buckets" : [ {
      "key" : "-2013.07",
      "to" : 1.3726368E12,
      "to_as_string" : "2013.07",
      "doc_count" : 8           #C
    },
    {
      "key" : "2013.07-*",
      "from" : 1.3726368E12,
      "from_as_string" : "2013.07",
      "doc_count" : 7
    }
  }
}
```

#A Define here a format to parse the date strings

#B Ranges are defined in date strings, too

#C For each interval, you get the document count

If the value of the `format` field looks familiar, it's because it's the same Joda Time annotation that you saw in chapter 3 when you defined date formats in the mapping. For the complete syntax, you can look at the `DateTimeFormat` documentation:

<http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html>

### 7.3.3 Histogram aggregations

For dealing with numeric ranges, you also have histogram aggregations. These are much like the range aggregations we just saw, but instead of manually defining each range, you'd define a fixed interval, and Elasticsearch will build the ranges for you. For example, if you want age

groups from people documents, you can define an interval of 10 (years) and you'd get buckets like [0-10), [10-20) and so on.

Like the range aggregation, the histogram aggregation has a variant that works with dates, called the *date histogram aggregation*. This is useful, for example, when building histogram charts of how many Emails were sent on a mailing list each day.

### HISTOGRAM AGGREGATION

Running a histogram aggregation is very similar to running a range aggregation. You just replace the `ranges` array with an `interval`, and Elasticsearch will build ranges starting with the minimum value, adding the interval until the maximum value is included. For example, in the following listing, we specify an interval of 1 and show how many events have 3 attendees, how many have 4, and how many have 5 attendees.

#### **Listing 7.13 Histogram showing the number of events for each number of attendees**

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "attendees_histogram": {
      "histogram": {
        "script": "doc[''attendees''].values.length",
        "interval": 1
      }
    }
  }
}## reply
"aggregations" : {
  "attendees_histogram" : {
    "buckets" : [ {
      "key_as_string" : "3",
      "key" : 3,
      "doc_count" : 4
    }, {
      "key_as_string" : "4",
      "key" : 4,
      "doc_count" : 9
    }, {
      "key_as_string" : "5",
      "key" : 5,
      "doc_count" : 2
    }
  }
}
```

#A Interval used for building ranges. Here, we want to see every value  
#B Keys show the “from” value of the range. “to” is key+interval  
#C Next “from” is the previous “to”

Like the terms aggregation, the histogram aggregation lets you specify a `min_doc_count` value, which is helpful if you want buckets with few documents to be ignored.

`min_doc_count` is also useful if you want to show empty buckets. By default, if there's an interval between the minimum and maximum values that has no documents, that interval will be omitted altogether. Set `min_doc_count` to 0 and those intervals will still appear with a document count of 0.

## DATE HISTOGRAM AGGREGATION

As you might expect, you'd use the date histogram aggregation like the histogram one, but you'd insert a date in the "interval" field. That date would be specified in the same Joda Time annotation as the date range aggregation, with values such as "1M" or "1.5h". For example, the following listing will give the breakdown of events happening in each month.

### **Listing 7.14 Histogram of events per month**

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "event_dates": {
      "date_histogram": {
        "field": "date",
        "interval": "1M"                                     #A
      }
    }
  }
}## reply
"aggregations" : {
  "event_dates" : {
    "buckets" : [
      {
        "key_as_string" : "2013-02-01T00:00",            #B
        "key" : 1359676800000,                            #B
        "doc_count" : 1
      },
      {
        "key_as_string" : "2013-03-01T00:00",
        "key" : 1362096000000,
        "doc_count" : 1
      },
      {
        "key_as_string" : "2013-04-01T00:00",
        "key" : 1364774400000,
        "doc_count" : 2
      }
    ]
  }
}
```

#A Interval here is specified as a date string

#B key\_as\_string is more useful here, because it's a more human-readable date format

Like the regular histogram aggregation, you can use the `min_doc_count` option to either show empty buckets or to omit buckets containing just a few documents.

You probably noticed that the date histogram aggregation has two things in common with all the other multi-bucket aggregations:

- It counts documents having certain terms.
- It creates buckets of documents falling in each category.

The buckets themselves are only useful when you nest other aggregations under the multi-bucket aggregation; this allows you to have deeper insights of your data, and we'll look at nesting aggregations in the next section. First, take time to look at Table 7.2, which gives you a quick overview of the multi-bucket aggregations and what they're typically used for.

Table 7.2 Multi-bucket aggregations and typical use-cases

Aggregation type	Example use-case
Terms	Top tags on a blogging site; hot topics this week on a news site
significant terms	Identify new technology trends by looking at what is used/downloaded a lot this month compared to overall
range and date range	Show entry-level, medium-priced and expensive laptops. Show archived events, events this week, upcoming events
histogram and date histogram	Show distributions: how much people of each age exercise. Or trends: items bought each day

## 7.4 Nesting aggregations

The real power of aggregations is the fact that you can combine them. For example, if you have a blog and you record each access to your posts, you can use the terms aggregation to show the most viewed posts. But you can also nest a cardinality aggregation under this term aggregation and show the number of unique visitors for each post, and even change the sorting in the terms aggregation to show posts with the most unique visitors.

As you may imagine, nesting aggregations open a whole new range of possibilities for exploring data. Nesting is the main reason aggregations emerged in Elasticsearch as a replacement for facets, because facets couldn't be combined.

Multi-bucket aggregations are typically the point where you start nesting. For example, the term aggregation allows you to show the top tags for get-together groups; this means you'll have a bucket of documents for each tag. You can use sub-aggregations to show more metrics for each bucket. For example, you can show how many groups are being created each month, for each tag, as illustrated in Figure 7.10.

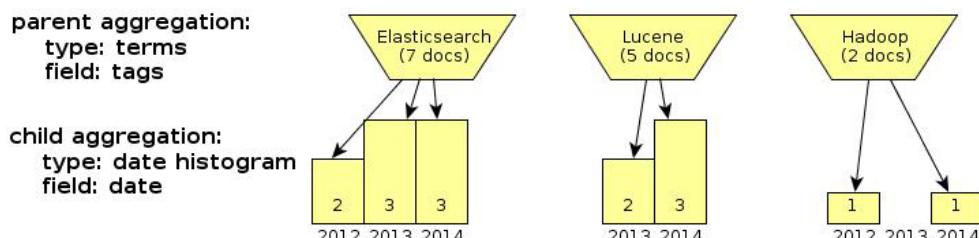


Figure 7.10 Nesting a date histogram aggregations under a terms aggregation

While multi-bucket aggregations are typically “parent” aggregations, the “children” can be a metrics aggregation, too. For example, you can show the average number of group members for each tag. There’s nothing to nest under those average number, so metrics aggregations will always be the last in a chain.

Later in this section, we’ll also discuss one particular use-case for nesting in this section: *result grouping*, which, unlike a regular search that gives you the top N results by relevance, will give you the top N results for each bucket of documents generated by the “parent” aggregation. Say you have an online shop and someone searches for “Windows.” Normally, relevance-sorted results will show many versions of the Windows operating system first. This may not be the best user experience, because at this point it’s not 100% clear whether the user is looking to buy a Windows operating system, some software built for Windows, or some hardware that works with Windows. This is where result grouping, illustrated in figure 7.11, comes in handy: you can show the top 3 results from each of the “operating systems,” “software,” and “hardware” categories, and give the user a broader range of results. The user may also want to click on the category name to narrow the search to that category only.

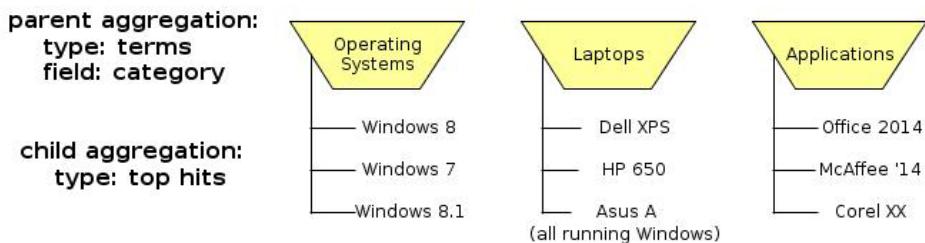


Figure 7.11 Nesting the top hits aggregation under a term aggregation to get result grouping

In Elasticsearch, you’ll be able to get result grouping by using a special aggregation called `top hits`. It retrieves the top N results, sorted by score or a criteria of your choice, for each bucket of a parent aggregation. That parent aggregation can be a terms aggregation that’s running on your category field; we’ll go over this special aggregation in the next section.

The last nesting use-case we’ll talk about is controlling the document set on which your aggregations run. For example, regardless of the query, you might want to show the top tags for get-together groups created in the last year. To do this, you’d use the `filter` aggregation, which creates a bucket of documents that match the provided filter, in which you can nest other aggregations.

#### 7.4.1 Nesting multi-bucket aggregations

To nest an aggregation within another one, you just have to use the `aggregations` or `aggs` key on the same level as the “parent” aggregation type, and then put the “child” aggregation definition as the value. For multi-bucket aggregations, this can be done indefinitely. For example, in listing 7.15, you’ll use the terms aggregation to show the top tags. For each tag, you’ll use the date histogram aggregation to show how many groups were created each

month, for each tag. Finally, for each bucket of tag-and-created-month groups, we'll use the range aggregation to show how many groups have fewer than 3 members, and how many have at least 3.

### Listing 7.15 Nesting multi-bucket aggregations three times

```
curl localhost:9200/get-together/group/_search?pretty -d '{
  "aggregations": {
    "top_tags": {
      "terms": {
        "field": "tags.verbatim"
      },
      "aggregations": {
        "groups_per_month": {
          "date_histogram": {
            "field": "created_on",
            "interval": "1M"
          },
          "aggregations": {
            "number_of_members": {
              "range": {
                "script": "doc['members'].values.length",
                "ranges": [
                  { "to": 3 },
                  { "from": 3 }
                ]
              }
            }
          }
        }
      }
    }
  }
}## reply
"aggregations" : {
  "top_tags" : {
    "buckets" : [ {
      "key" : "big data",
      "doc_count" : 3,
      "groups_per_month" : {
        "buckets" : [ {
          "key_as_string" : "2010-04-01",
          "key" : 1270080000000,
          "doc_count" : 1,
          "number_of_members" : {
            "buckets" : [ {
              "key" : "-3.0",
              "to" : 3.0,
              "to_as_string" : "3.0",
              "doc_count" : 1
            }, {
              "key" : "3.0-*",
              "from" : 3.0,
              "from_as_string" : "3.0",
              "doc_count" : 0
            } ]
          }
        }
      }
    }
  }
}
```

```

[key_as_string" : "2012-08-01", #J
[...] #K

#A Typical terms aggregation, giving top tags
#B Within it, use the "aggregation" key to define a child aggregation
#C This date histogram aggregation will run once for every top tag
#D We define a child aggregation for the date histogram, too
#E The range aggregation will run for every tag+month bucket
#F This is familiar, "big data" is the top tag, 3 documents
#G Next, we have buckets for each month where "big data" documents were created
#H One document was created in Jan 2010
#I Next, we find out that this document has less than 3 members
#J Nexts bucket of big data groups is created in August 2012
#K Analysis goes on, showing all buckets for "big data" and the rest of tags

```

As the “leaf” aggregation, you can always use a metrics aggregation. For example, if you wanted the average number of group members instead of the 0-2 and 3+ ranges that you had in the previous listing, you can use the “avg” or “stats” aggregations.

One particular type of aggregation we promised to cover in the last section is `top hits`. It will get you the top N results, sorted by the criteria you like, for each bucket of its “parent” aggregation. Next, we’ll look at how you’ll use the `top hits` aggregation to get result grouping.

#### **7.4.2 Nesting aggregations to get result grouping**

Result grouping is useful when you want to show the top results, grouped by a certain category. Like in Google when you have many results from the same site, you sometimes only see the top three or so, and then it moves on to the next site. You can always click on the site’s name to get all the results from it that match your query.

That’s what result grouping is for: it allows you to give the user a better idea of what else is in there. Say we want to show the user the most recent events, and to make results more diverse, we’ll show the most recent event for the most frequent attendees. We’ll do this in listing 7.16, by running the terms aggregation on the attendees field, and nest the `top hits` aggregation under it.

#### **Listing 7.16 Using the top hits aggregation to get result grouping**

```

curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "frequent_attendees": {
      "terms": {
        "field": "attendees", #A
        "size": 2 #A
      },
      "aggregations": {
        "recent_events": {
          "top_hits": { #B
            "sort": {
              "date": "desc" #C
            },
            "_source": { #D
              "include": [ "title" ] #D
            }
          }
        }
      }
    }
  }
}

```

```
#A This terms aggregation gives us the two users going to most events
#B The top hits aggregation will give us the actual events
#C Like with regular searches you can sort: we'll get the most recent first
#D Like with regular searches, you can select the fields to include
#E We'll use "size" to select the number of results per bucket
#F Lee is the most frequent one, with 5 events
#G Results look exactly, like the ones you get while querying
```

In order for the top hits aggregation to work, you need to have Elasticsearch 1.3 or later. At first, it may look out of place to use aggregations for getting results grouping. But now that you've learned what aggregations are all about, you can see that these concepts of buckets and nesting are very powerful and enable you to do much more than just some statistics on query results. The top hits aggregation is an example of a non-statistic outcome of aggregations.

You're not limited only to query results when you run aggregations; this is the default behavior, as you learned in section 7.1, but you can work around that if you need to. For example, let's say that you want to show the most popular blog post tags on your blog,

somewhere on a sidebar. And you want to show that sidebar no matter what the user is searching for. To achieve this, you'd need to run your terms aggregation on all blog posts, independent of your query. Here is where the "global" aggregation becomes useful: it produces a bucket with all the documents of your search context (the indices and types you're searching in) making all other aggregations nested under it to work with all these documents.

The "global" aggregation is one of the single-bucket aggregations that you can use to change the document set other aggregations run on, and that's what we'll explore next.

### **7.4.3 Using single-bucket aggregations**

As you saw in section 7.1, Elasticsearch will run your aggregations on the query results by default. If you want to change this default, you'll have to use single-bucket aggregations. Here's we'll discuss three of them:

- **Global** creates a bucket with all the documents of the indices and types you're searching on. This is useful when you want to run aggregations on all documents, no matter the query.
- **Filter** creates a bucket with all the documents matching a specified filter. This is useful when you want to further restrict the document set, for example to only run aggregations on items that are in stock.
- **Missing** creates a bucket with documents that don't have a specified field. It's useful when you have another aggregation running on a field, but you want to do some computations on documents that aren't covered by that aggregation, because the field is missing. For example, when you want to show the average price of items across multiple stores, and also want to show the number of stores not listing a price for those items.

#### **GLOBAL**

Using our get-together site from the code samples, assume you're querying for events about Elasticsearch, but you want to see the most frequent tags overall. For example, as we describe earlier, if you want to show those top tags somewhere on a sidebar, independent of what the user is searching for.

To achieve this, you need to use the global aggregation, which can alter the flow of data from query to aggregations as shown in Figure 7.12.

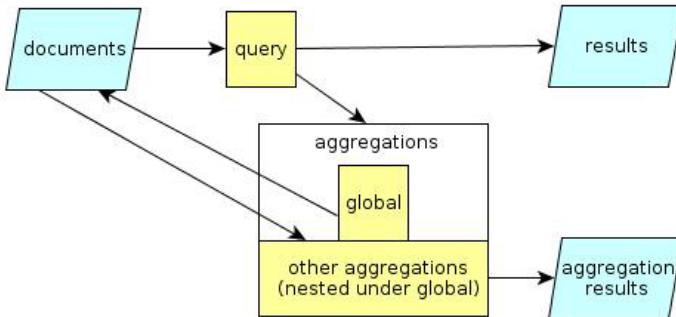


Figure 7.12 Nesting aggregations under the global aggregation makes them run on all documents

In listing 7.17, you'll nest the `terms` aggregation under the `global` aggregation to get the most frequent tags on all documents, even if the query only looks for those with `elasticsearch` in the title.

#### **Listing 7.17 Global aggregation helps show top tags overall regardless of the query**

```

curl localhost:9200/get-together/group/_search?pretty -d '{
  "query": {
    "match": {
      "name": "elasticsearch"
    }
  },
  "aggregations": {
    "all_documents": { #A
      "global": {}, #A
      "aggregations": {
        "top_tags": {
          "terms": { #B
            "field": "tags.verbatim" #B
          }
        }
      }
    }
  }
}'
### reply
[...]
  "hits" : { #C
    "total" : 2, #C
  }
[...]
  "aggregations" : {
    "all_documents" : { #D
      "doc_count" : 5, #D
      "top_tags" : {
        "buckets" : [ {
          "key" : "big data", #D
          "doc_count" : 3 #D
        }
      ]
    }
  }
[...]
  
```

#A The global aggregation is the parent

#B The terms aggregation is nested under it, to work on all data

#C The query returns two documents  
#D But aggregations run on all five  
#E The terms aggregation results are as if there was no query

When we say “all documents,” we mean all the documents from the search context defined in the search URI. In this case we’re searching in the `group` type of the `get-together` index, so all the groups will be taken into account. If we searched in the whole `get-together` index, both groups and events will be included in the aggregation.

### FILTER

Remember the post filter from section 7.1? It’s when you define a filter directly in the JSON request, instead of wrapping it in a filtered query; the post filter will restrict the results you get without affecting the aggregations.

The filter aggregation does the opposite: it will restrict the document set your aggregations run on, without affecting the results. This is illustrated in figure 7.13.

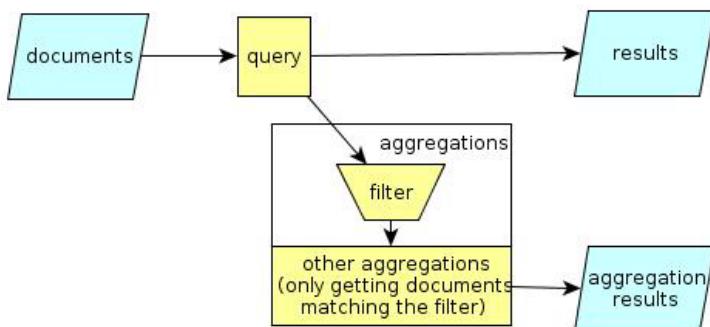


Figure 7.13 The filter aggregation restricts query results for aggregations nested under it

If you’re searching for events with “elasticsearch” in the title, you want to create a word cloud from words within the description, but you only want to account for documents recent enough. Let’s say, after July 1<sup>st</sup>, 2013.

To do that, in the following listing you’d run a query as usual, but with aggregations, you’ll first have a filter aggregation restricting the document set to those after July 1<sup>st</sup>, and under it you’ll nest the terms aggregation that generates the word could information.

### **Listing 7.18 A filter aggregation will restrict the document set coming from the query**

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  },
  "aggregations": {
    "since_july": {
      "filter": {
        "script": {
          "script": "doc['date'].value >= '2013-07-01T00:00:00Z'"
        }
      },
      "terms": {
        "field": "text"
      }
    }
  }
}'
```

```

    "range": {
      "date": {
        "gt": "2013-07-01T00:00"
      }
    },
    "aggregations": {
      "description_word_cloud": {
        "terms": {
          "field": "description"
        }
      }
    }
  }
}
## reply
[...]
  "hits" : {
    "total" : 7,           #A
  [...]
  "aggregations" : {
    "since_july" : {
      "doc_count" : 2,       #B
      "description_cloud" : {
        "buckets" : [ {
          "key" : "we",
          "doc_count" : 2
        }, {
          "key" : "with",
          "doc_count" : 2
        }
      ]
    }
  }
}
[...]

```

#A The query returns 7 results

#B The description\_cloud aggregation only runs on the two results matching the filter

## MISSING

Most of the aggregations we looked at so far make buckets of documents and get metrics from values of a field. If a document is missing that field, it will not be part of the bucket and it won't contribute to any metrics.

For example, you might have a date histogram aggregation on event dates, but some events have no date set yet. You can count them too, through the missing aggregation:

```

curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations": {
    "event_dates": {
      "date_histogram": {
        "field": "date",
        "interval": "1M"
      }
    },
    "missing_date": {
      "missing": {
        "field": "date"
      }
    }
  }
}'

```

Like with other single-bucket aggregations, the missing aggregation allows you to nest other aggregations under it. For example, you can use the max aggregation to show the maximum number of people who intend to participate to a single event that doesn't have a date set for now.

There are other important single-bucket aggregations that we didn't cover here: the nested and reverse nested aggregations, which allow you to use all the power of aggregations with nested documents.

Using nested documents is one of the ways to work with relational data in Elasticsearch. The next chapter will include all you need to know about relations among documents, including nested documents and nested aggregations.

## 7.5 Summary

- Aggregations help you get an overall view of query results, by counting terms and computing statistics from resulting documents
- Aggregations are the new facets in Elasticsearch, as there are more types of aggregations, and you can also combine them to get deeper insights of the data
- There are two main types of aggregations: bucket and metrics
- Metrics aggregations calculate statistics over a set of documents, such as the minimum, maximum or average value of a numeric field
- Some metrics aggregations are calculated with approximation algorithms, which allow them to scale a lot better than exact metrics. The percentiles and cardinality aggregations work like this.
- Bucket aggregations put documents into one or more categories, and return counters for those categories. For example, the most frequent posters in a forum. Bucket aggregations can also be "parents" of other aggregations. "Children" aggregations run one time for each bucket of the "parent". You can use this, for example, to get the average number of comments for blog posts matching each tag
- The top hits aggregation can be used as a "child" aggregation to implement result grouping
- The terms aggregation is typically used for "top frequent users/locations/items/..." kind of use-cases. Other multi-bucket aggregations are variations of the terms aggregation, such as the significant terms aggregation, which returns those words that appear more often in the query results than in the overall index
- The range and date range aggregations are useful for categorizing numeric and date fields. The histogram and date histogram aggregations are similar, but they use fixed intervals instead of manually defined ranges
- Single-bucket aggregations, such as the global, filter and missing aggregations, are used to change the document set on which other aggregations run, which default to the documents returned by the query.

# 8

## *Relations among documents*

### This chapter covers

- objects and arrays of objects
- nested mapping, queries and filters
- parent mapping, `has_parent` and `has_child` queries and filters
- denormalization techniques

Some data is inherently relational. For example, with the get-together site we've used as an example throughout the book, there are groups of people with the same interests, and events organized by those groups. So how might you search for groups that host events about a certain topic?

If your data is flat-structured, then you might as well skip this chapter and move on to scaling out, which will be discussed next. This is typically the case for logs, where you have independent field, such as timestamp, severity and message. If, on the other hand, you have related entities in your data, like blog posts and comments, users and products and so on, then by now you may wonder how should you best represent those relationships in your documents, so you can run queries and aggregations across those relationships.

With Elasticsearch you don't have joins like an SQL database. As we'll discuss in section 8.4 on denormalizing (duplicating data), that's because having query-time joins in a distributed system is typically slow, and Elasticsearch strives to be real-time and return query results in milliseconds. On the upside, there are multiple ways to define relationships in Elasticsearch. You can, for example, search for events based on their location, or search for groups based on properties of the events they host. We'll explore all the possibilities for defining relationships among documents in Elasticsearch: object types, nested documents,

parent-child relationships, and denormalizing, and explore the advantages and disadvantages of each in this chapter.

## 8.1 Options for defining relationships among documents

First, let's quickly define each of these approaches:

- *Objects type*: This allows you to have a sub-document as the value of a field in your document. For example, your *address* field of an event could be an object with its own fields: city, postal code, street name, and so on. You could even have an array of addresses if the same event happens in multiple cities.
- *Nested documents*: The problem you may have with the object type is that all the data is stored in the same document, so matches for a search can go across sub-documents. For example, *city=Paris AND street\_name=Broadway* could return an event that's hosted in New York and Paris at the same time, even though there's no Broadway street in Paris. Nested documents allow you to index the same JSON document, but will keep your addresses in separate Lucene documents, making only searches like *city>New York AND street\_name=Broadway* return the expected result.
- *Parent-child relationships between documents*: This method allows you to use completely separate Elasticsearch documents for different types of data, like events and groups, but still define a relationship between them. For example, you can have groups as "parents" of events, to indicate which event hosts which group. This will allow you to search for events hosted by groups in your area, or for groups that host events about Elasticsearch.
- *Denormalizing*: This is a general technique of duplicating data in order to represent relationships. In Elasticsearch, you're likely to employ it to represent many-to-many relationships, because other options only work on one-to-many. For example, if all groups have members, and members could belong to multiple groups. You can duplicate one side of the relationship, for example by including all the members of a group in that group's document.

Before we dive into all the details of working with each possibility, we'll overview them and their typical use-cases.

### 8.1.1 Object type

The easiest way to represent a common interest group and the corresponding events is to use the object type. This allows you to put a JSON object, or an array of JSON objects, as the value of your field, like the example below:

```
{
  "name": "Denver technology group",
  "events": [
    {
      "date": "2014-12-22",
      "title": "Introduction to Elasticsearch"
    },
  ]
}
```

```

        {
          "date": "2014-06-20",
          "title": "Introduction to Hadoop"
        }
      ]
    }
  }
}

```

If you want to search for a group with events that are about Elasticsearch, you can simply search in the events.title field.

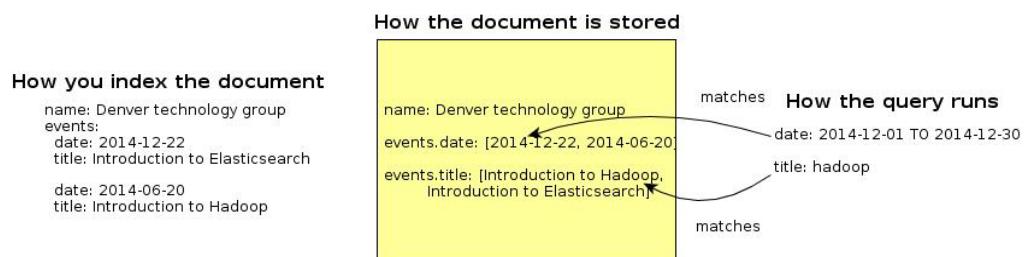
This works brilliantly for one-to-one relationships, but with one-to-many relationships you might get unexpected results. For example, let's say you want to filter groups hosting Hadoop meetings in December 2014. Your query can look like this:

```

  "bool": {
    "must": [
      {
        "term": {
          "events.title": "hadoop"
        }
      },
      {
        "range": {
          "events.date": {
            "from": "2014-12-01",
            "to": "2014-12-31"
          }
        }
      }
    ]
  }
}

```

This will match our sample document, because it has a title that matches hadoop, and a date that's in the specified range. But this is not what we want: it's the Elasticsearch event that's in December; the Hadoop one is in June. Sticking with the default object type is the fastest and easiest approach to relations, but Elasticsearch is unaware of the boundaries between documents, as illustrated in figure 8.1.



#A To the left: Elasticsearch event is in December, Hadoop event is in June

#B To the right: search for Hadoop events in December matches the document

Figure 8.1 Inner object boundaries are not accounted for when storing, leading to unexpected results

### 8.1.2 Nested type

If you need to make sure such cross-object matches don't happen, you can use the *nested type*, which will index your events in separate Lucene documents. In both cases, the group's JSON document will look exactly the same and applications will index them in the same way. The difference is in the mapping, which triggers Elasticsearch to index nested inner objects in adjacent, but separate Lucene documents, as illustrated in figure 8.2. When searching, you'll need to use nested filters and queries, which will be explored in section 8.2; those will search in all those Lucene documents.

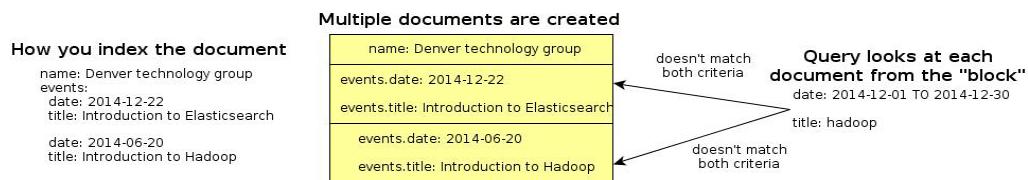


Figure 8.2 Nested type makes Elasticsearch index objects as separate Lucene documents

In some use-cases, it's not a good idea to mash all the data in the same document, like objects and nested types do. Take the case of groups and events: if a new event is organized by a group, and all that group's data is in the same document, you'll have to re-index the whole document just for that event. This can hurt performance and concurrency, depending on how big those documents get, and how often those operations are done.

### 8.1.3 Parent-child relationships

With parent-child relationships, you can use completely different Elasticsearch documents, by putting them in different types and defining their relationship in the mapping of each type. For example, you can have events in one mapping type and groups in another, and you can specify in the mapping that groups are "parents" of events. Also, when you index an event, you can point it to the group that it belongs to, like in figure 8.3. At search time, you can use `has_parent` or `has_child` queries and filters to take the other part of the relationship into account. We'll discuss them later in this chapter as well.

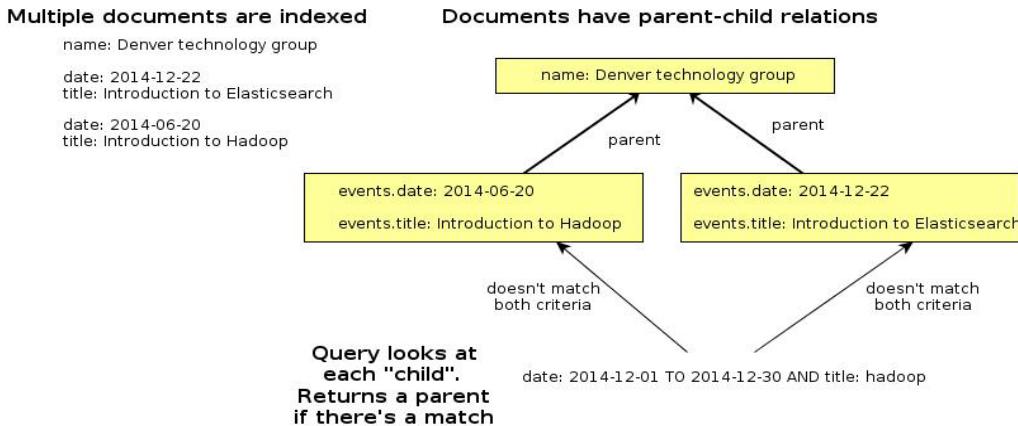


Figure 8.3 Different types of Elasticsearch documents can have parent-child relationships

#### 8.1.4 Denormalizing

For any relational work, you have objects, nested documents and parent-child. These work for one-to-one and one-to-many relationships, the “one parent with one or more children” kind. There’s also a fourth way, which is not a specific Elasticsearch feature, but a method often employed by NoSQL data-stores to overcome the lack of joins: *denormalizing*, which means a document will include data that’s related to it, even if the same data will have to be duplicated in another document.

For example, let’s take groups and their members. A group can have more members, and a user can be a member of more groups. Both have their own set of properties. To represent this relationship, you can have groups as “parents” of the members. For users who are members of multiple groups, you’d have to multiply their data: once for each group they belong to, like in the figure below:

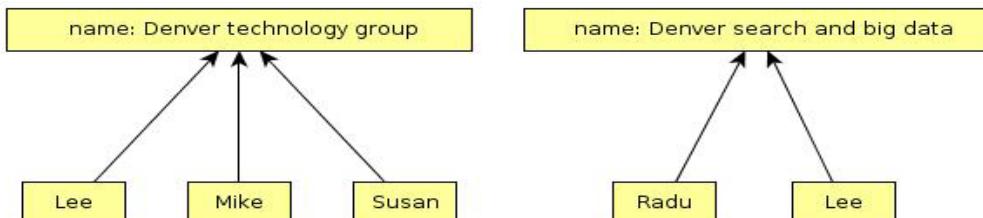


Figure 8.4 Denormalizing is the technique of multiplying data to avoid costly relations

In the rest of this chapter, we'll take a deeper look at each of these techniques: objects and arrays, nested, parent-child, and denormalizing. You'll learn how they work internally, how to define them in the mapping, how to index and how to search those documents.

## 8.2 Object type: using sub documents as field values

As you saw back in chapter 2, documents in Elasticsearch can be hierarchical. For example, in the code samples, an event of the get-together site has its location as an object with two fields: name and geolocation.

```
{
  "title": "Using Hadoop with Elasticsearch",
  "location": {
    "name": "SkillsMatter Exchange",
    "geolocation": "51.524806,-0.099095"
  }
}
```

If you're familiar with Lucene, you may ask yourself, “How can Elasticsearch documents be hierarchical, when Lucene only supports flat structures?” With objects, Elasticsearch flattens hierarchies internally, by putting each inner field with its full path as a separate field in Lucene. You can see the process in the following figure:

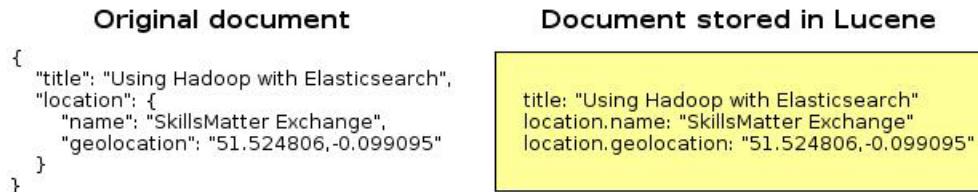


Figure 8.5 JSON hierarchical structure stored as a flat structure in Lucene

Typically, when you want to search in an event's location name, you'd refer to it as `location.name`. We'll look at that in section 8.1.2, but before we go into searching, let's define a mapping and index some documents.

### 8.2.1 Mapping and indexing objects

By default, inner object mappings are automatically detected. In listing 8.1, you will simply index a hierarchical document and see how the detected mapping looks. If those events documents look familiar to you, it's because the code samples store the location of an event in an object, too. You can go to <https://github.com/dakrone/elasticsearch-in-action> to get the code samples now if you haven't already.

#### **Listing 8.1 Inner JSON objects are mapped as the object type**

```
curl -XPUT 'localhost:9200/get-together/event-object/1' -d '{
  "title": "Introduction to objects",
  "location": {
    "name": "Elasticsearch in Action book",      #A
    "geolocation": "51.524806,-0.099095"        #A
  }
}'
```

```

        "address": "chapter 8"                      #A
    }
}
curl 'localhost:9200/get-together/event-object/_mapping?pretty'
# expected reply:
#{
#  "get-together" : {
#    "event-object" : {
#      "properties" : {
#        "location" : {
#          "properties" : {
#            "address" : {
#              "type" : "string"           #B
#            },
#            "name" : {
#              "type" : "string"         #B
#            }
#          }
#        },
#        "title" : {
#          "type" : "string"
#        }
#      }
#    }
#  }
#}

```

**#A An object within the JSON document**

**#B Object's mapping is automatically detected with its properties. Like the “root” object**

You can see that the inner object has a list of `properties` just like the root JSON object has. You would configure field types from inner objects in the same way you'd do for fields in the root object. For example, you can upgrade `location.address` to the `multi_field` type that we've discussed in chapter 3. This will allow you to index the address in different ways, like having a `not_analyzed` version for exact matches in addition to the default `analyzed` version.

**TIP** If you need to look at core types or the `multi_field` type, you can revisit chapter 3. For more details on analysis, you can go back to chapter 5.

The mapping for a single inner object will also work if you have multiple such objects in an array. For example, if you index the following document, the mapping from listing 8.1 will stay the same.

```
{
  "title": "Introduction to objects",
  "location": [
    {
      "name": "Elasticsearch in Action book",
      "address": "chapter 8"
    },
    {
      "name": "Elasticsearch Guide",
      "address": "elasticsearch/reference/current/mapping-object-type.html"
    }
}
```

```
[  
}'
```

To summarize, working with objects and arrays of objects in the mapping is very much like working with the fields and arrays you saw in chapter 3. Next, we'll look at searches, which also work like the ones you saw in chapters 4 and 6.

### 8.2.2 Searching in objects

By default, Elasticsearch will recognize and index hierarchical JSON documents with inner objects without defining anything up front. As you can see in figure 8.6 below, the same goes with searching. By default, you will have to refer to inner objects by specifying the path to the field you're looking at, such as `location.name`.

```

name: Introduction to Elasticsearch
location:
  name: Chicago
address: Philadelphia street
  
```

Figure 8.6 You can search in an object's field by specifying that field's full path

As you worked through chapters 2 and 4, you indexed documents from the code samples. You can now search through events happening in offices, as we do in listing 8.2, where you'll specify the full path of `location.name` as the field to search on.

**TIP** If you didn't index the documents from the code samples yet, you can do it now by cloning the repository at <https://github.com/dakrone/elasticsearch-in-action> and running the `populate.sh` script.

#### Listing 8.2 Searching in location.name from events indexed by the code samples

```

EVENT_PATH="localhost:9200/get-together/event"
curl "$EVENT_PATH/_search?q=location.name:office&pretty"
# relevant part of the result:
#[...]
# "title": "Hortonworks, the future of Hadoop and big data",
#[...]
# "location": {
#   "name": "SendGrid Denver office",
#   "geolocation": "39.748477,-104.998852"
#[...]
# "title": "Big Data and the cloud at Microsoft",
#[...]
# "location": {
#   "name": "Bing Boulder office",
#   "geolocation": "40.018528,-105.275806"
#[...]
  
```

## AGGREGATIONS

While searching, you would treat object fields like `location.name` in the same way as any other field. The same works with the aggregations that you saw in chapter 7. For example, the following terms aggregation get the most used words in the `location.name` field, that help you build a word cloud:

```
% curl localhost:9200/get-together/event/_search?pretty -d '{
  "aggregations" : {
    "location_cloud" : {
      "terms" : {
        "field" : "location.name"
      }
    }
  }
}'
```

## OBJECTS WORK BEST FOR ONE-TO-ONE RELATIONSHIPS

One-to-one relationships are the perfect use-case for objects: you can search in the inner object's fields as if they would be fields in the root document. That's because, in fact, they are! At the Lucene level, `location.name` really is another field in the same flat structure.

You can also have one-to-many relationships with objects, by putting them in arrays. For example, take a group with multiple members. If each member would have its own object, you'd represent them like this:

```
"members": [
  {
    "first_name": "Lee",
    "last_name": "Hinman"
  },
  {
    "first_name": "Radu",
    "last_name": "Gheorghe"
  }
]
```

You can still search for `members.first_name:lee` and it will match “Lee” as expected. However, you need to keep in mind that, in Lucene, the structure of the document looks more like this:

```
"members.first_name": ["Lee", "Radu"],
"members.last_name": ["Hinman", "Gheorghe"]
```

As a result of how it's stored, if you search for `members.first_name:lee` AND `members.last_name:gheorghe`, the document will still match, because it matches each of those two criteria. This happens even though there's no member named “Lee Gheorghe”, because Elasticsearch throws everything in the same document, and it's not aware of boundaries between objects. To have Elasticsearch understand those boundaries, you can use the `nested type`, coming up next.

## Using objects to define document relationships: pros and cons

Before moving on, here's a quick recap of why you should (or why you shouldn't) use objects. The plus points:

- Easy to use. Elasticsearch detects them by default, in most cases you don't have to define anything special upfront to index objects
- Run queries and aggregations on objects like you would do with flat documents. That's because, at the Lucene level, they are flat documents
- No joins are involved. Because everything is in the same document, using objects will give you the best performance of any of the options discussed in this chapter

The downsides:

- No boundaries between objects. If you need such functionality, you need to look at other options: nested, parent-child, denormalizing, and eventually combine them with objects if it suits your use-case
- Updating a single object will re-index the whole document

## 8.3 Nested type: connecting nested documents

Nested type is defined in the mapping much the same way as object type, which we've already discussed. Internally, nested documents are indexed as different Lucene documents. To indicate that you want to use the nested type instead of the object type, you'll have to set type to nested, as you'll see in section 8.2.1.

From an application's perspective, indexing nested documents is the same as objects, because the JSON document indexed as an Elasticsearch document looks the same. For example:

```
{
  "name": "Elasticsearch News",
  "members": [
    {
      "first_name": "Lee",
      "last_name": "Hinman"
    },
    {
      "first_name": "Radu",
      "last_name": "Gheorghe"
    }
  ]
}
```

At the Lucene level, Elasticsearch will index the root document and all the members objects in separate documents. But it will put them in a single **block**, as shown in figure 8.7.

first_name: Lee last_name: Hinman	first_name: Radu last_name: Gheorghe	name: Elasticsearch news previous 2 documents are "members"
--------------------------------------	---	--

Figure 8.7 A block of documents in Lucene storing the Elasticsearch document with nested-type objects

Documents of a block will always stay together, ensuring they get fetched and queried with the minimum number of operations.

Now that you know how nested documents work, let's see how to make Elasticsearch use them. You have to specify that you want nested at index time and at search time:

- Inner objects must have a nested mapping, to get them indexed as separate documents in the same block.
- Nested queries and filters need to be used to make use of those blocks while searching.

We'll discuss how you can do each in the next two sections.

### 8.3.1 Mapping and indexing nested documents

The nested mapping looks similar to the object mapping, except instead of the `type` being `object`, you have to make it `nested`. In the following listing you will define a mapping with a nested type field, and index the a document that with an array of nested sub-documents.

#### Listing 8.3 Mapping and indexing a nested documents

```
curl -XPUT localhost:9200/get-together/group-nested/_mapping -d '{
  "group-nested": {
    "properties": {
      "name": { "type": "string" },
      "members": {
        "type": "nested",                                     #A
        "properties": {
          "first_name": { "type": "string" },
          "last_name": { "type": "string" }
        }
      }
    }
  }
}'

curl -XPUT localhost:9200/get-together/group-nested/1 -d '{
  "name": "Elasticsearch News",                         #B
  "members": [
    {
      "first_name": "Lee",                                #C
      "last_name": "Hinman"                               #C
    },
    {
      "first_name": "Radu",                               #D
      "last_name": "Gheorghe"                            #D
    }
  ]
}'
```

#A This signals Elasticsearch to index “members” objects in separate documents of the same block  
#B This property goes in the main document  
#C and #D These objects go into their own documents, part of the same block as the “parent” document

JSON objects with the nested mapping, like the ones you just indexed in listing 8.3, allow you to search them with nested queries and filters. We'll explore those searches in just a bit, but the thing to remember now is that nested queries and filters allow you to search within the boundaries of such documents. For example, you will be able to search for groups with members with the first name “Lee” and the last name “Hinman.” Nested queries won't do cross-object matches, thus avoiding unexpected matches as “Lee” with the last name “Gheorghe.”

### **ENABLING CROSS-OBJECT MATCHES**

In some situations, you might need cross-object object matches as well. For example, if you're searching for a group that has both Lee and Radu, a query like this would work for the “regular” JSON objects we discussed in the section on the object type:

```
"query": {
  "bool": {
    "must": [
      {
        "term": {
          "members.first_name": "lee"
        }
      },
      {
        "term": {
          "members.first_name": "radu"
        }
      }
    ]
  }
}
```

This query would work, because when you have everything in the same document, both criteria will match.

With nested document, a query structured this way won't work, because `members` objects would be stored in separate Lucene documents. And there's no `members` object that will match both criteria: we have one for Lee and one for Radu; we don't have any document containing both.

In such situations, you might want to have both: objects for when you want cross-object matches, and nested documents for when you want to avoid them. Elasticsearch lets you do that through a couple of mapping options: `include_in_root` and `include_in_parent`.

### **INCLUDE\_IN\_ROOT**

Adding `include_in_root` to your nested mapping will index the inner `members` objects twice: one time as a nested document, and one time as an object within the root document, as shown in the following figure.

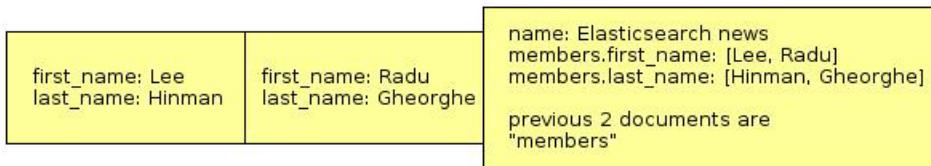


Figure 8.8 With `include_in_root`, fields of nested documents are indexed in the root document, too

The following mapping will let you use nested queries for the nested documents and the regular queries for when you need cross-object matches.

```
"members": {
    "type": "nested",
    "include_in_root": true,
    "properties": {
        "first_name": { "type": "string" },
        "last_name": { "type": "string" }
    }
}
```

#### INCLUDE\_IN\_PARENT

Elasticsearch allows you to have multiple levels of nested documents. For example, if your group can have members as its nested “children,” members can have children of their own, like the comments they posted on that group. Figure 8.9 illustrates this hierarchy.



Figure 8.9 `include_in_parent` indexes a nested document's field into the immediate parent, too

With the `include_in_root` option we just saw, you can add the fields at any level to the root document, the “grandparent” in this case. There's also `include_in_parent`, which allows you to index the fields of one nested document into the immediate parent document. For example, listing 8.4 will include the `comments` into the `members` documents.

#### Listing 8.4 Using `include_in_parent` when there are multiple nested levels

```
curl -XPUT localhost:9200/get-together/group-multinested/_mapping -d '{
    "group-multinested": {
        "properties": {
            "name": { "type": "string" },
            "members": {
                "type": "nested",
                "properties": {
                    "first_name": { "type": "string" },
                    "last_name": { "type": "string" },
                    "comment": {
                        "type": "nested",
                        "properties": {
                            "date": { "type": "string" }
                        }
                    }
                }
            }
        }
    }
}'
```

```

    "comments": {
      "#B
      "type": "nested",
      "#B
      "include_in_parenttrue,
      "#B
      "properties": {
        "#B
        "timestamp": {
          "#B
          "type": "date",
          "#B
          "format": "dateOptionalTime"
        },
        "#B
        "comment": { "type": "string" }
      }
    }
  }
}

```

#A “members” are nested documents relative to the root “group” document. No inclusion here  
#B “comments” are nested documents of the “members”. Contents are also indexed as objects for the parent “members” documents

By now you’re probably wondering how you’d query these nested structures. This is exactly what we’ll be looking at next.

### **8.3.2 Searches and aggregations on nested documents**

Like with mappings, when you run searches and aggregations on nested documents you’ll need to specify that the object you’re looking at are nested. There are nested queries, filters and aggregations that help you achieve this. Running these special queries and aggregations will trigger Elasticsearch to join the different Lucene documents within the same block and treat the resulting data as the same Elasticsearch document.

The way to search within nested documents is to use the `nested query` or `nested filter`. As you might expect after chapter 4, these are equivalent, with the traditional differences between queries and filters:

- *Queries calculate score*: thus they’re able to return results sorted by relevance.
- *Filters don’t calculate score*: making them faster and easier to cache.

**TIP** In particular, the nested filter isn’t cached by default. You can change this by setting `_cache` to `true`, like you can do in all filters.

If you want to run aggregations on nested fields, such as getting the most frequent group members, you’ll have to wrap them in a nested aggregation. If sub-aggregations have to refer to the “parent” Lucene document – like showing top group tags for each member – you can go up the hierarchy with the reverse nested aggregation.

#### **NESTED QUERY AND FILTER**

When you run a nested query or filter, you need to specify the `path` argument, to tell Elasticsearch where those nested objects are located. In addition to that, your nested query or

filter will wrap a regular query or filter respectively. In listing 8.5, you'll search for members with the first name "Lee" and the last name "Gheorghe," and see that the document indexed in listing 8.3 won't match, because you only have Lee Hinman and Radu Gheorghe, and no member called Lee Gheorghe.

### **Listing 8.5 Nested query example**

```
curl 'localhost:9200/get-together/group-nested/_search?pretty' -d '{
  "query": {
    "nested": {
      "path": "members",          #A
      "query": {                 #B
        "bool": {
          "must": [
            {
              "term": {
                "members.first_name": "lee"
              }
            },
            {
              "term": {
                "members.last_name": "gheorghe"      #C
              }
            }
          ]
        }
      }
    }
  }
}'
```

#A look for nested documents under the "members" object

#B the query would be the one that you'd normally run on objects within the same document

#C there's no member Lee Gheorghe. Change this to Hinman and it will match Lee Hinman

A nested filter would look exactly the same as the nested query you just saw. You'll just have to replace the word "query" with "filter."

### **SEARCHING IN MULTIPLE LEVELS OF NESTING**

Elasticsearch also allows you to have multiple levels of nesting. For example, back in listing 8.6 you added a mapping that nests on two levels: members and their comments. To search in the comments-nested documents, you'd have to specify members.comments as the path, like in the following listing.

### **Listing 8.6 Indexing and searching multiple levels of nested documents**

```
curl -XPUT localhost:9200/get-together/group-multinested/1 -d '{
  "name": "Elasticsearch News",
  "members": {
    "first_name": "Radu",
    "last_name": "Gheorghe",
    "comments": {           #A
      "date": "2013-12-22",
    }
  }
}'
```

```

        "comment": "hello world"
    }
}
}
curl 'localhost:9200/get-together/group-multinested/_search -d '{
  "query": {
    "nested": {
      "path": "members.comments",          #B
      "query": {
        "term": {
          "members.comments.comment": "hello"  #C
        }
      }
    }
  }
}

```

#A “comments” object is nested to the “members” object, also nested, as configured in listing 8.6

#B look in “comments”, which is under “members”

#C the query still provides still provides the full path to the field to look at

### AGGREGATING SCORES OF NESTED OBJECTS

The nested query calculates the score, but we didn't mention how. Let's say you have three members in a group: Lee Hinman, Radu Gheorghe and another guy called Lee Smith. If you have a nested query for “Lee” it will match two members. Each inner “member” document will get its own score, depending on how well it matches the criteria. But the query coming from the application is for “group” documents, so Elasticsearch will need to give back a score for the whole group document. At this point, there are four options, which can be specified with the `score_mode` option:

- `avg`: This is the default option, which will take the scores of the matching inner documents and return their average score
- `total`: This will sum up the matching inner documents' scores and return it. Useful when the number of matches counts
- `max`: The maximum inner document score is returned
- `none`: No score is kept and counted towards the total document score

If you're thinking that there are too many options around the nested type, regarding including in root or parent and the score options, have a look at table 8.1 to for a quick reference on all those options and when they're useful.

**Table 8.1 Nested type options**

<b>Option</b>	<b>Description</b>	<b>Example</b>
include_in_parent: true	Indexes the nested document in the parent document, too.	"first_name:Lee AND last_name:Hinman", for which you need the nested type, as well as "first_name:Lee AND first_name:Radu", for which you need the object type
include_in_root: true	Indexes the nested document in the root document	Same scenario as above, but you have multiple layers. For example, event>member>comment
score_mode: avg	Average score of matching nested documents count	Search for groups hosting events about "Elasticsearch"
score_mode: total	Sums up nested document scores	Search for groups hosting most events that have to do with "Elasticsearch"
score_mode: max	Maximum nested document score	Search for groups hosting top events about Elasticsearch
score_mode: none	No score counts towards the total score	Filter groups hosting events about Elasticsearch. Use the nested filter instead

### NESTED AND REVERSE NESTED AGGREGATIONS

In order to do aggregations on nested type objects, you'll have to use the nested aggregation. This is a single-bucket aggregation, where you indicate the path to the nested object containing your field. As shown in figure 8.10, the nested aggregation triggers Elasticsearch to do the necessary joins in order for other aggregations to work properly on the indicated path:

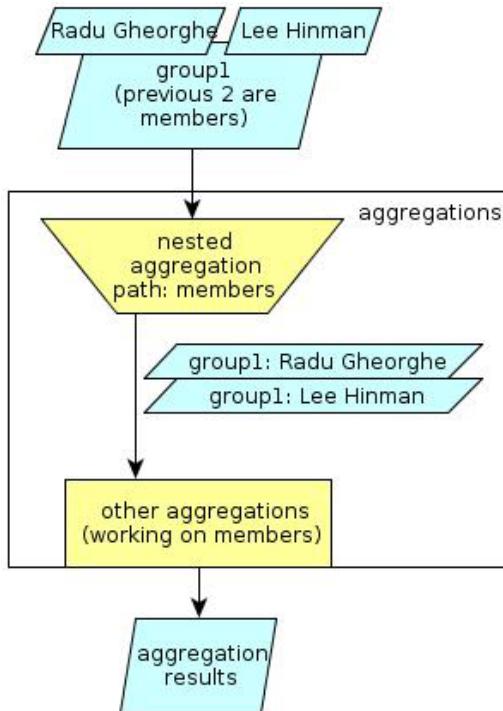


Figure 8.10 Nested aggregation doing necessary joins for other aggregations to work on the indicated path

For example, you'd normally run a `terms` aggregation on a member name field in order to get the top users by the number of groups they are part of. If that `name` field is stored within the `members` nested type object, you'll wrap that `terms` aggregation in a `nested` aggregation that has the `path` set to `members`:

```
% curl localhost:9200/get-together/group/_search?pretty -d '{
  "aggregations" : {
    "members" : {
      "nested" : {
        "path" : "members"
      },
      "aggregations" : {
        "frequent_members" : {
          "terms" : {
            "field" : "members.name"
          }
        }
      }
    }
}'
```

You can put more aggregations under the `members` nested aggregation, and Elasticsearch will know to look in the `members` type for all of them.

There are use-cases where you'd need to navigate back to the parent or root document. For example, if you want for each of the obtained frequent members to show the top group tags. To do that, you'll use the `reverse_nested` aggregation, which will tell Elasticsearch to go up the nested hierarchy:

```
"frequent_members" : {
  "terms" : {
    "field" : "members.name"
  },
  "aggregations": {
    "back_to_group": {
      "reverse_nested": {},
      "aggregations": {
        "tags_per_member": {
          "terms": {
            "field": "tags"
          }
        }
      }
    }
  }
}
```

The nested and reverse nested aggregations can effectively be used to tell Elasticsearch in which Lucene document to look for the fields of the next aggregation. This gives you the flexibility to use all the aggregation types you saw in chapter 7 for nested documents, just as you could use them for objects. The only downside for this flexibility is the performance penalty.

### **PERFORMANCE CONSIDERATIONS**

We will cover performance in more detail in chapter 10, but in general you can expect nested queries and aggregations to be slower than the object counterparts. That's because Elasticsearch needs to do some extra work to join multiple documents within a block. But because of the underlying implementation using blocks, these queries and aggregations are much faster than they would be if you had to join completely separate Elasticsearch documents.

This block implementation also has its drawbacks. Because nested documents are stuck together, updating or adding one inner document requires re-indexing the whole ensemble. Applications also work with nested documents in a single JSON.

If your nested documents become big, like they would in a get-together site if you'd have one document per group and all its events as nested, a better option might be to use separate Elasticsearch documents and define parent-child relations between them.

## Using nested type to define document relationships: pros and cons

Before moving on, here's a quick recap of why you should (or why you shouldn't) use nested documents. The plus points:

- Nested types are aware of object boundaries: no more matches for "Radu Hinman"!
- Index the whole document at once, like you would with objects, after you defined your nested mapping.
- Nested queries and aggregations join the parent and child parts and you can run any query across the union. This is a feature that no other option described in this chapter provides.
- Query-time joins are fast, because all Lucene documents making the Elasticsearch document are together in the same block in the same segment
- Can include "child" documents in "parents" to get all the functionality from objects if you need it. This functionality is transparent for your application

The downsides:

- Queries will be slower than their object equivalent. If objects provide you all the needed functionality, they are the better option because they're faster
- Updating a "child" will reindex the whole document

## 8.4 Parent-child relationships: connecting separate documents

Another option for defining relationships among data in Elasticsearch is to define a type within an index as a child of another type of the same index. This is useful when documents or relations need to be updated often. You would define the relationship in the mapping, through the `_parent` field. For example, you can see in the `mapping.json` file from the book's code samples that events are children of groups, as illustrated in figure 8.11.

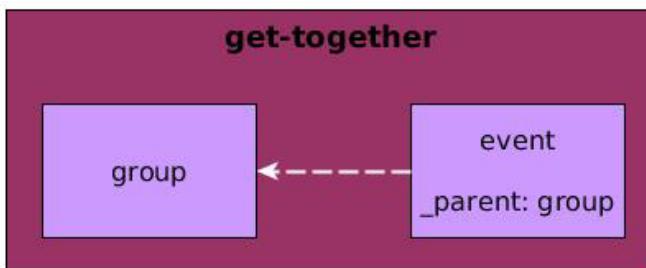


Figure 8.11 The relationship between events and groups as it's defined in the mapping

Once you have this relationship defined in the mapping, you can start indexing documents. The parents (group documents in this case) are indexed normally. For children, (events in this example) you need to specify the parent's ID in the `_parent` field. This will basically "point"

the event to its group, and allow you to search for groups including some events criteria or the other way around, like below.

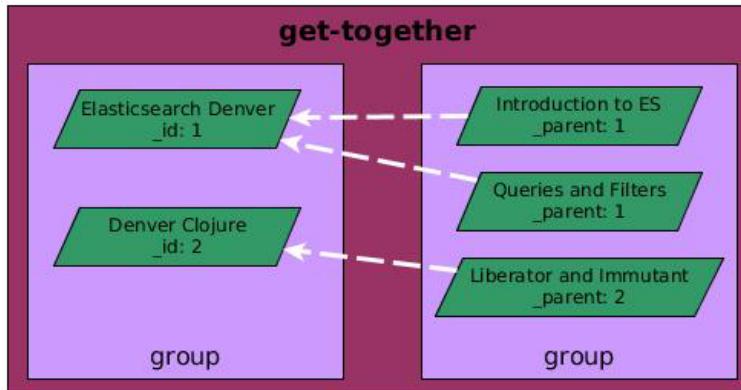


Figure 8.12 The `_parent` field of each child document is pointing to the `_id` field of its parent

When it comes to searching, you can see some disadvantages of the parent-child approach compared to nested:

- You only get the document type that you're searching for. When you search for groups, for example, even if you specify some event criteria, you only get back group documents. If you need the group's events as well, you'll need to get them with a different request.
- Searches are slower. With nested documents, the fact that all inner objects are Lucene documents in the same block pays dividends, because they can be joined easily into the root document. Parent and child documents are completely different Elasticsearch documents, and they can live in different Lucene segments of the same shard.

The parent-child approach shines when it comes to indexing, updating and deleting documents. Because parent and child documents are different Elasticsearch documents, they can be managed separately. For example, if a group has many events and you need to add a new one, you just add that new event document. Using the nested-type approach, Elasticsearch will have to re-index the group documents with the new event and all existing events, which is much slower.

A parent document can either already be indexed or not when you index its child. This is useful when you have lots of new documents and you want to index them asynchronously. For example, you can index events on your website generated by users, and index the users, too. Events may come from your logging system and users may be synchronized from a database. You don't need to worry about making sure a user exists before you can index an event that will have that user as a parent. If the user doesn't exist, the event is indexed anyway.

But how would you index parent and child documents in the first place? This is exactly what we'll explore next.

### **8.4.1 Indexing, updating and deleting child documents**

We're only worrying about child documents here because parents are indexed as any other document you've indexed so far. It's the child documents that must point to their parents, via the `_parent` field.

**NOTE** Parents of a document type can be children of another type. You can have multiple levels of such relationships, just as you can do with nested. You can even combine them. For example: a group can have its members stored as nested type, and events separately stored as their children.

When it comes to child documents, you have to define the `_parent` field in the mapping, and when indexing, specify the parent's ID in the `_parent` field. The parent's ID and type will also serve as the child's routing value.

---

### **Routing and routing values**

You may recall from chapter 2 how indexing operations get distributed to shards by default: each document you index has an ID, and that ID gets hashed. At the same time, each shard of the index has an equal slice of the total range of hashes. The document you index goes to the shard that has that document's hashed ID in its range.

The hashed ID is called the *routing value*, and the process of assigning a document to a shard is called *routing*. Because each ID is different and you hash them all, the default routing mechanism will evenly balance documents between shards.

You can also specify a custom routing value. We'll go into the details of using custom routing in chapter 9, but the basic idea is that Elasticsearch hashes that routing value, and not the document's ID to determine the shard. You'd use custom routing when you want to make sure multiple documents are in the same shard, because hashing the same routing value will always give you the same hash.

Custom routing becomes useful when you start searching, because you can provide a routing value to your query. When you do, Elasticsearch only goes to the shard that corresponds to that routing value, instead of querying all the shards. This reduces the load in your cluster a lot, and is typically used for keeping each user's documents together.

The `_parent` field provides Elasticsearch with the ID and type of the parent document, which lets it route the child documents to the same hash as the parent document. `_parent` is essentially a routing value, and you benefit from it when searching. Elasticsearch will automatically use this routing value to query only the parent's shard to get its children, or the child's shard to get its parent.

---

The common routing value makes all the children of the same parent land in the same shard as the parent itself. When searching, all the correlations that Elasticsearch has to do between a parent and its children happen on the same node. This is much faster than broadcasting all the child documents over the network in search of a parent. Another implication of routing is that when you update or delete a child document, you need to specify the `_parent` field.

Next, we'll look at how you would practically do all those things:

- Defining the `_parent` field in the mapping.
- Indexing, updating and deleting child documents by specifying the `_parent` field.

## MAPPING

Listing 8.7 shows the relevant part of the `events` mapping from the code samples. The `_parent` field has to point to the parent type, in this case `group`.

### **Listing 8.7 `_parent` mapping from the code samples**

```
% grep -A 10 ' "event" mapping.json
  "event" : {                                #A
    "_source" : {
      "enabled" : true
    },
    "_all" : {
      "enabled" : false
    },
    "_parent" : {                               #B
      "type" : "group"                         #B
    },
    "properties" : {                           #C
      "host" : {
        "type" : "string"
      }
    }
  }
#A mapping for the "event" type starts here
#B _parent points to the "group" type
#C properties (fields) of the "event" type start here
```

## INDEXING AND RETRIEVING

With the mapping in place, let's start indexing documents. Those documents have to contain the `parent` value in the URI as a parameter. For our events, that value is the document ID of the groups they belong to, like we have "2" for the Elasticsearch Denver group:

```
curl -XPOST 'localhost:9200/get-together/event/1103?parent=2' -d '{
  "host": "Radu",
  "title": "Yet another Elasticsearch intro in Denver"
}'
```

The `_parent` field is stored, so you can retrieve it later, and it's also indexed, so you can search on its value. If you look at the contents of `_parent` for a group, you will see the type you defined in the mapping, as well as the group ID you specified when indexing.

To retrieve an event document, you'd run a normal index request, and you also have to specify the `_parent` value:

```
% curl 'localhost:9200/get-together/event/1103?parent=2&pretty'
{
  "_index" : "get-together",
  "_type" : "event",
  "_id" : "1103",
  "_version" : 1,
  "found" : true, "_source" : {
    "host": "Radu",
    "title": "Yet another Elasticsearch intro in Denver"
  }
}
```

The parent value is required because you can have multiple events with the same ID, pointing to different groups. But the `_parent` and `_id` combination is unique. If you try to get the child document without specifying its parent, you'll get an error saying that a routing value is required. And the parent value is that routing value Elasticsearch is waiting for.

```
% curl 'localhost:9200/get-together/event/1103?pretty'
{
  "error" : "RoutingMissingException[routing is required for [get-together]/[event]/[1103]]",
  "status" : 400
}
```

## UPDATING

You'd update a child document through the Update API, in a similar way to what you did in chapter 3, section 3.5. The only difference here is that you have to provide the parent again. Like in the case of retrieving an event document, the parent is needed to get the routing value of the event document you're trying to change. Otherwise, you'd get the same `RoutingMissingException` you had earlier when trying to retrieve the document without specifying a parent.

The following snippet adds a description to the document we've just indexed:

```
curl -XPOST 'localhost:9200/get-together/event/1103/_update?parent=2' -d '{
  "doc": {
    "description": "Gives an overview of Elasticsearch"
  }
}'
```

## DELETE

To delete a single event document, you'd run a delete request like in chapter 3, section 3.6.1, and add the parent parameter:

```
curl -XDELETE 'localhost:9200/get-together/event/1103?parent=2'
```

Delete by query works just as before: documents that match get deleted. This API doesn't need parent values, and it doesn't take them into account either:

```
curl -XDELETE 'http://localhost:9200/get-together/event/_query?q=host:radu'
```

Speaking of queries, let's look at how you can search across parent-child relations.

### 8.4.2 Searching in parent and child documents

With parent-child relations, like you have with groups and their events, you can search for groups and add event criteria or the other way around. Let's see what the actual queries and filters are that you'll use:

- has\_child queries and filters are useful in searching for parents with criteria from their children. For example, if you need groups hosting events about Elasticsearch
- has\_parent queries and filters are useful when searching for children with criteria from their parents. For example, events that happen in Denver. Because location is a group property.

#### You get what you're searching for

All those searches will only return the "target" documents and not their parents or children. For example, when searching for groups, you'll get back group documents. You won't get those groups' events, even if your search included events criteria, like "give me the group that **has\_child** of **name:Elasticsearch**".

If you want the events as well, you'll have to do another request, and you'll see how in the rest of this section.

Nested documents work differently, because objects are joined together with the root document and a search will get you back everything.

The same rationale applies to aggregations: you can run them on the child documents alone, or on the parent documents. But, as of version 1.2, there's no way for aggregations to work across the parent-child relationship. For example, if you want to get the top group tags, and then for each tag, show a histogram of event dates, you'd have to go with objects or nested documents.

#### HAS\_CHILD QUERY AND FILTER

If you want to search in groups hosting events about Elasticsearch, you can use the has\_child query or filter. The classic difference here is that filters don't care about scoring.

A has\_child filter can wrap another filter or a query. It runs that filter or query against the specified child type and collects the matches. The matching children contain the IDs of their parents in the \_parent field. Elasticsearch collects those parent IDs and removes the duplicates – because the same parent ID can appear multiple times, once for each child – and returns the list of parent documents. The whole process is illustrated in figure 8.13.

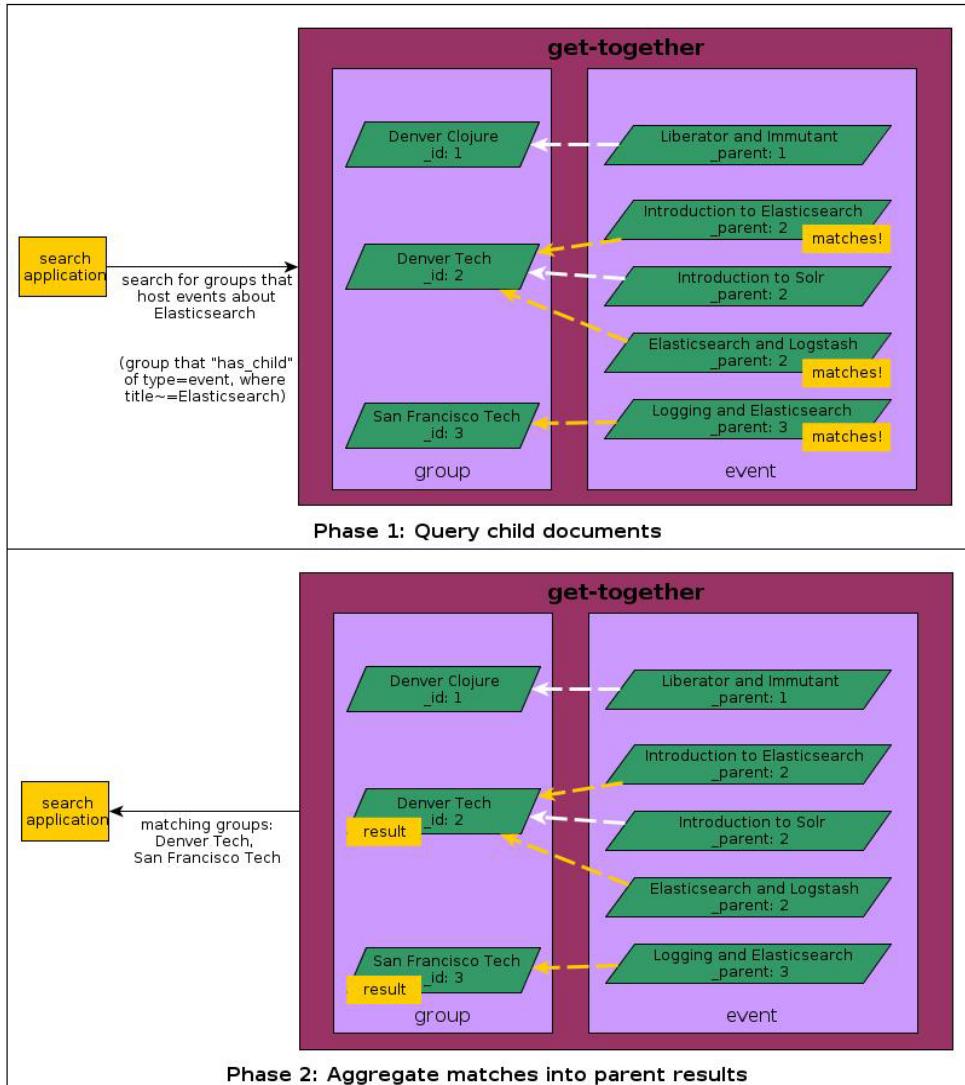


Figure 8.13 `has_child` filter first runs on children, then aggregates results into parents, which are returned

In Phase 1 of the figure:

- The application runs a `has_child` filter, requesting "group" documents with children of type "event", having "Elasticsearch" in their name
- The filter runs on the "event" type for documents matching "Elasticsearch"

- Resulting “event” documents point to their respective parents. Multiple events can point to the same group.

In Phase 2, Elasticsearch gathers all the unique group documents and returns them to the application.

The filter from figure 8.13 would look like this:

```
% curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "query": {
    "filtered": {
      "filter": {
        "has_child": {
          "type": "event",
          "filter": {
            "term": {
              "title": "elasticsearch"
            }
          }
        }
      }
    }
  }
}'
```

The `has_child` query runs in a similar way to the filter, except it can give a score to each parent by aggregating child document scores. You'd do that by setting `score_type` to `max`, `sum`, `avg` or `none`, like you can do with nested queries.

**NOTE** If the `has_child` filter can wrap a filter or a query, the `has_child` query can only wrap another query.

For example, you can set `score_type` to `max` and get the following query to return groups ordered by which one hosts the most relevant event about Elasticsearch:

```
% curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "query": {
    "has_child": {
      "type": "event",
      "score_type": "max",
      "query": {
        "term": {
          "title": "elasticsearch"
        }
      }
    }
  }
}'
```

**WARNING** In order for `has_child` queries and filters to remove parent duplicates quickly, it caches their IDs in memory. This may take a lot of JVM heap if you have lots of parent matches for your queries.

## HAS\_PARENT QUERY AND FILTER

`has_parent` is, as you might expect, the opposite of `has_child`. You'd use it when you want to search for events, but include criteria from the groups they belong to.

The `has_parent` filter can wrap a query or a filter. It runs on the "`parent_type`" that you provide, takes the parent results and returns the children pointing to their IDs from their `_parent` field.

The following listing shows how to search for events about Elasticsearch, but only if they happen in Denver.

### Listing 8.8 has\_parent query to find Elasticsearch events in Denver

```
curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "title": "elasticsearch"
          }
        },
        {
          "has_parent": {
            "type": "group",
            "query": {
              "term": {
                "location": "denver"
              }
            }
          }
        }
      ]
    }
  }
}'
```

#A The main query contains two must-have queries

#B This runs on the events, to make sure they have “elasticsearch” in their title

#C This runs on each event's group, to make sure events happen in Denver

Because a children only has a parent, there are no scores to aggregate, like it's the case with `has_child`. By default, `has_parent` has no influence on the child's score ("`score_type`": "none"). You can change "`score_type`" to "score" to make events inherit the score of their parent groups.

Like the `has_child` queries and filters, `has_parent` queries and filters have to cache the parent `_id` values to support fast lookups. That said, you can expect all those parent/child queries to be slower than the equivalent nested queries. It's the price you pay for being able to index and search all the documents independently.

You can think of nested documents as index-time joining and of parent-child relations as query-time joins. With nested, a parent and all its children are “joined” in a single Lucene

block when indexing. By contrast, the `_parent` field allows different types of documents to be correlated at query time.

Nested and parent-child structures are good for one-to-many relationships. For many-to-many relationships, you'll have to employ a technique common in the NoSQL space: denormalizing.

### **Using parent-child designation to define document relationship: pros and cons**

Before moving on, here's a quick recap of why you should (or why you shouldn't) use parent-child relationships. The plus points:

- "Children" and "parents" can be updated separately.
- Query-time join performance is good, though not as good as with nested, because all related documents are routed to the same shard.

The downsides:

- You can only get back the parent or the child documents when you query.
- Aggregations don't work across the relationship.

## **8.5 Denormalizing: using redundant data connections**

Denormalizing is about multiplying data in order to avoid expensive joins. Let's take an example we've already discussed: groups and events. It's a one-to-many relationship, because an event can be hosted by only one group, and one group can host many events.

With parent-child or nested structures, groups and events are stored in different Lucene documents, as shown in figure 8.14.

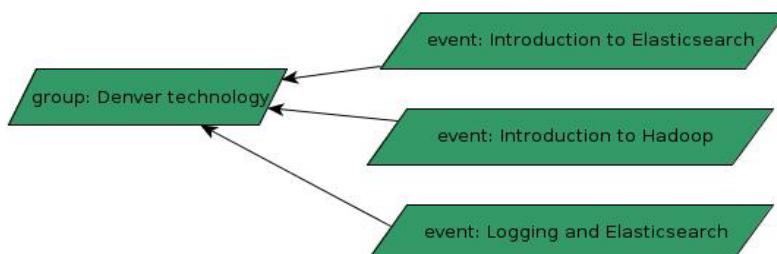


Figure 8.14 Hierarchical relationship (nested or parent-child) between different Lucene documents

This relationship can be denormalized by adding the group info to all the events, shown below.

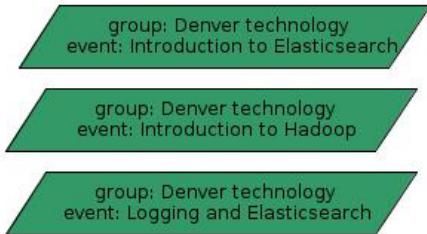


Figure 8.15 Hierarchical relationship denormalized by copying group information to each event

Next, we'll look at how and when denormalizing helps, and how you would concretely index and query denormalized data.

### 8.5.1 Use-cases for denormalizing

Let's start with the disadvantages: denormalized data takes more space and it's more difficult to manage than normalized data. In the example from figure 8.15, if you change the group's details, you have to update three documents, because those details appear three times.

On the positive side, you don't have to join different documents when you query. This is particularly important in distributed systems, because having to join documents across the network introduces big latencies, as you can see in figure 8.16 below.

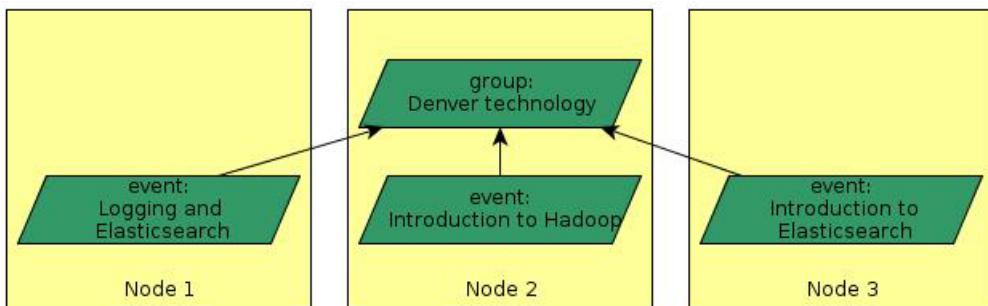


Figure 8.16 Joining documents across nodes is difficult because of network latency

Nested and parent-child documents get around this by making sure a parent and all its children are stored in the same node:

- Nested documents are indexed in Lucene blocks, which are always together in the same segment of the same shard.
- Child documents are indexed with the same routing value as their parents, making them belong to the same shard.

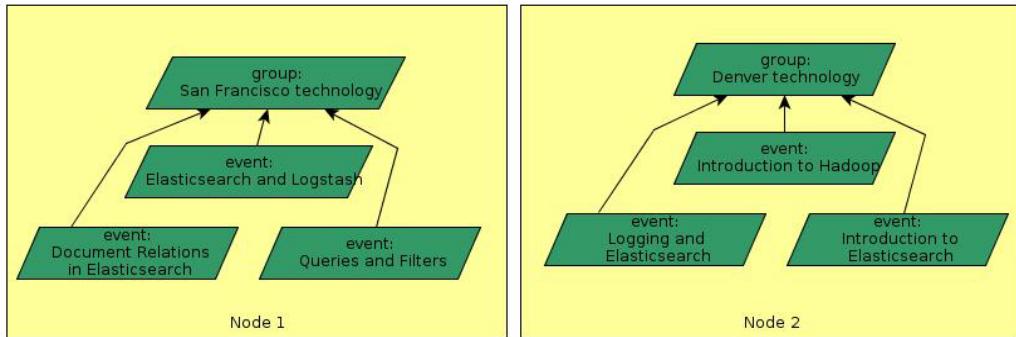


Figure 8.17 Nested and parent-child relations make sure all joins are local

### DENORMALIZING ONE-TO-MANY RELATIONS

Local joins done with nested and parent-child structures are much, much faster than remote joins could be. Still, they're more expensive than having no joins at all. This is where denormalizing can help, but it implies there's more data. So your indexing operations will cause more load because you'll index more data, and queries will run on larger indices, making them slower.

You can see that there's a trade-off when it comes to choosing between nested, parent-child and denormalizing. Typically, you'd denormalize for one-to-many relations if your data is fairly small and static and you have lots of queries. This way, disadvantages hurt less – index size acceptable, not too many indexing operations – and avoiding joins should make queries faster.

**TIP** If performance is important to you, take a look at chapter 10, which is all about indexing and searching fast.

### DENORMALIZING MANY-TO-MANY RELATIONSHIPS

Many-to-many relationships are dealt with differently than one-to-many relationships in Elasticsearch. For example, a group can contain multiple members and a person could be a member of multiple groups.

Here, denormalizing is a much better proposition because, unlike one-to-many implementations of nested and parent-child, Elasticsearch can't promise to contain many-to-many relationships in a single node. As shown in figure 8.18, a single relationship may expand to your whole dataset. This would make expensive, cross-network joins inevitable.

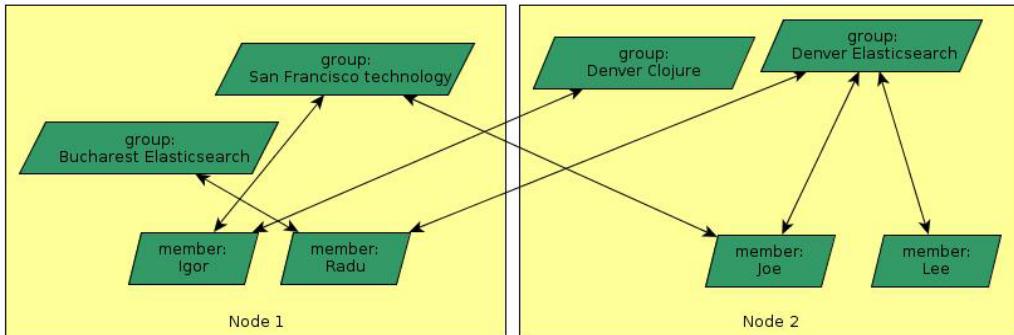


Figure 8.18 Many-to-many relationships can contain a huge amount of data, making local joins impossible

Because of how slow cross-network joins would be, as of version 1.0, denormalizing is the only way to represent many-to-many relationships in Elasticsearch. Figure 8.19 shows how the structure of figure 8.17 looks when members are denormalized as “children” of each group they belong to. We denormalize one side of the many-to-many relationship into more one-to-many relationships.

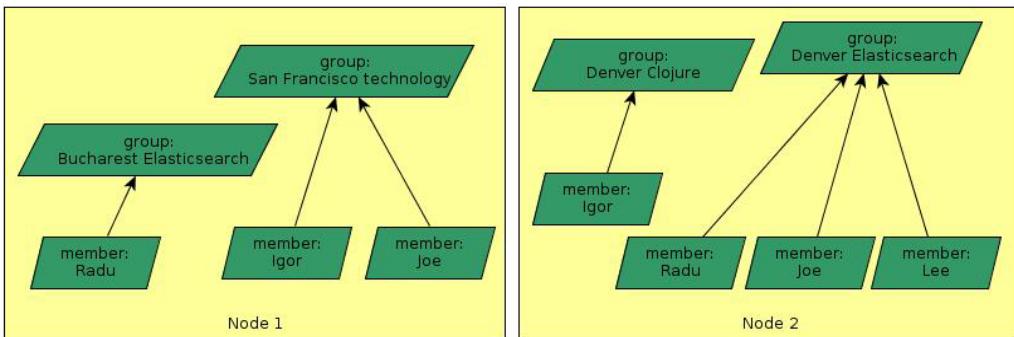


Figure 8.19 many-to-many relation denormalized into multiple one-to-many relations, allowing local joins

Next, we'll look at how you can index, update and query a structure like the one in figure 8.19.

### **8.5.2 Indexing, updating, and deleting denormalized data**

Before you start indexing, you have to decide how you want to denormalize your many-to-many into one-to-many, and there are two big decision points: which side of the relationship should you denormalize, and how you want to represent the resulting one-to-many relationship.

### WHICH SIDE WILL BE DENORMALIZED?

Will members be multiplied as children of groups or the other way around? To pick one we have to understand how data is indexed, updated, deleted and queries. The part that is denormalized – the “child” – will be more difficult to manage on all aspects:

- You index those documents multiple times: once for each of its “parents.”
- When you update, you have to update all instances of that document.
- When you delete, you have to delete all instances.
- When you query for “children” separately, you’ll get more hits with the same content, so you have to remove duplicates yourself on the application side.

Based on these assumptions, it looks like it makes more sense to make members children of groups. Member documents are smaller in size, change less often and are queried less often than groups do with their events. As a result, managing cloned member documents should be easier.

### HOW YOU WANT TO REPRESENT THE ONE-TO-MANY RELATIONSHIP?

Will you have parent-child or nested documents? You’d choose here based on how often groups and members are searched and retrieved together. Nested queries perform better and give you both the parent and the children back in the same document.

Another important aspect is how often membership changes. Parent-child structures perform better here because they can be updated separately.

For this example, let’s assume that searching and retrieving groups and members together is rare and that members often join and leave groups, so we’ll go with parent-child.

### INDEXING

Groups and their events would be indexed as before, but members have to be indexed once for every group they belong to. The following listing will first define a mapping for the new member type, and then index Mr. Hinman as a member of both the Denver Clojure and the Denver Elasticsearch group from the code samples.

#### **Listing 8.9 Indexing denormalized members**

```
curl -XPUT 'localhost:9200/get-together/member/_mapping' -d '{
  "member": {
    "_parent": { "type": "group" },                                     #A
    "properties": {
      "first_name": { "type": "string" },
      "last_name": { "type": "string" }
    }
  }
}'
curl -XPUT 'localhost:9200/get-together/member/10001?parent=1' -d '{
  "first_name": "Matthew",                                         #B
  "last_name": "Hinman"
}'
curl -XPUT 'localhost:9200/get-together/member/10001?parent=2' -d '{
  "first_name": "Matthew",                                         #B
  "last_name": "Hinman"
}'
```

```
#A First we define the mapping, specifying that the parent type for members is "group"
#B parent=1 points to the Denver Clojure group
#C parent=2 points to the Denver Elasticsearch group
```

**NOTE** Multiple indexing operations can be done in a single HTTP request, by using the Bulk API. We'll discuss the Bulk API in chapter 10, which is all about performance.

## UPDATING

Once again, groups get lucky and you'd update them just as you've seen in chapter 3, section 3.5. But if a member changes its details, because it's denormalized, you'd first have to search for all its duplicates, then you'll update each one. In the listing 8.10, we'll search for all the documents that have an \_id of "10001", and update his first name to Lee, because that's what he likes to be called.

We're searching for IDs instead of names, because IDs tend to be more reliable than other fields, such as names. You may recall from the parent-child section that when you're using the \_parent field, multiple documents within the same type within the same index can have the same \_id value. Only the \_id and \_parent combination is guaranteed to be unique. When denormalizing, you can use this "feature" and intentionally use the same \_id for the same person, once for each group they belong to. This allows you to quickly and reliably retrieve all the instances of the same person by searching for their ID.

### Listing 8.10 Updating denormalized members

```
curl 'localhost:9200/get-together/member/_search?pretty' -d '{
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "_id": "10001"                                #A
        }
      }
    }
  },
  "fields": ["_parent"]                            #B
}
'
curl -XPOST 'localhost:9200/get-together/member/10001/_update?parent=1' -d '{
  "doc": {
    "first_name": "Lee"                           #C
  }
}'
curl -XPOST 'localhost:9200/get-together/member/10001/_update?parent=2' -d '{
  "doc": {
    "first_name": "Lee"                           #C
  }
}

#A Searching for all the members with the same ID, which will return all the duplicates of this person
#B We only need the _parent field from each document, so we know how to update
#C For each of the returned documents, we update the name to "Lee"
```

**NOTE** Multiple updates can also be done in a single HTTP request over the Bulk API. As with bulk indexing, we'll discuss bulk updates in chapter 10.

### DELETING

Deleting a denormalized member requires you to identify all the copies again. Recall from the parent-child section that, in order to delete a specific document, you have to specify both the `_id` and the `_parent`; that's because the combination of the two is unique in the same index and type.

But what happens when you don't specify a `_parent` when deleting? Elasticsearch will delete all the documents with the same ID, no matter the parent. And this is exactly what we need when denormalizing.

To delete all the duplicates of Lee, you only need to send a delete request with his index, type, ID:

```
curl -XDELETE 'localhost:9200/get-together/member/10001'
```

Now that you know how to index, update and delete in denormalized members, let's look at how you can run queries on them.

#### 8.5.3 Querying denormalized data

If you need to query groups, there's nothing denormalizing-specific, because groups are not denormalized. Even if you need to criteria from their members, you'd use the `has_child` query as you did in section 8.3.2.

Members got the shortest straw with queries, too. Because they're denormalized. You can search for them, even including criteria from the groups they belong to with the `has_parent` query. But there's a problem: you'd get back identical members. In listing 8.11, we'll index another two members, and when you search, you'll get them both back.

#### Listing 8.11 Querying for denormalized data may return duplicate results

```
curl -XPUT 'localhost:9200/get-together/member/10002?parent=1' -d '{
  "first_name": "Radu",          #A
  "last_name": "Gheorghe"
}'
curl -XPUT 'localhost:9200/get-together/member/10002?parent=2' -d '{
  "first_name": "Radu",          #A
  "last_name": "Gheorghe"
}'
curl -XPOST 'localhost:9200/get-together/_refresh'
curl 'localhost:9200/get-together/member/_search?pretty' -d '{
  "query": {
    "term": {
      "first_name": "radu"        #B
    }
  }
}##relevant results
#    "hits" : [
#      "_index": "get-together",
#      "_type": "member",
```

```

#      "_id" : "10002",
#      "_score" : 2.871802, "_source" : {      #C
#"first_name": "Radu",
#"last_name": "Gheorghe"
#}
#      }, {
#      "_index" : "get-together",
#      "_type" : "member",
#      "_id" : "10002",
#      "_score" : 2.5040774, "_source" : {      #C
#"first_name": "Radu",
#"last_name": "Gheorghe"
#}
#      } ]

```

**#A Indexing a person twice, once for each group**

**#B Searching for the person by name**

**#C The same person is returned twice, once for each group**

As of version 1.2, you can only remove those duplicate members from your application. Once again, if the same person always has the same ID, you can use that ID to make this task easier: two results with the same ID are identical.

The same problem appears with aggregations: if you want to count some properties of the members, those counts will be inaccurate because the same member appears in multiple places.

The workaround that works for most searches and aggregations is to maintain a copy of all members in a separate index. Let's call it "members." Querying that index will return just that one copy of each member. The problem with this workaround is that it helps only when you query `members` alone. If you need criteria from the groups they belong to, you need to query the get-together index and remove duplicate results from your application.

---

### Using denormalization to define relationships: pros and cons

As we did with the other methods, we'll have a quick overview of the strengths and weaknesses of denormalizing. The plus points:

- the only way to work with many-to-many relationships
- no joins are involved, making querying faster, if your cluster can handle the extra data caused by duplication

The downsides:

- your application has to take care of duplicates when indexing, updating, and deleting
  - some searches and aggregations will not work as expected because data is duplicated
-

## 8.6 Summary

Lots of use cases have to deal with relational data, and in this chapter we saw how you can deal with:

- Object mapping, mostly useful for one-to-one relationships
- Nested documents, and parent-child structures, which deal with one-to-many relationships.
- Denormalizing, which is mostly helpful with many-to-many relationships.

Joining hurts performance, even when it's local, so it's typically a good idea to put as many properties as you can in a single document. Object mapping helps with this, because it allows hierarchies in your documents. Searches and aggregations work here as they do with a flat-structured document, you just have to refer to fields using their full path, like `location.name`.

When you need to avoid cross-object matches, nested and parent/child documents are available to help:

- Nested documents are basically index-time joins, putting multiple Lucene documents in a single block. To the application, the block looks like a single Elasticsearch document
- The `_parent` field allows you to point a document to another document of another type in the same index, to be its "parent". Elasticsearch will use routing to make sure a parent and all its children land in the same shard, so that it can perform a local join at query time

You can search nested and parent-child documents with the following queries and filters:

- Nested query and filter
- `has_child` query and filter
- `has_parent` query and filter

Aggregations work across the relationship only with nested documents, through the `nested` and `reverse nested` aggregation types.

Objects, nested and parent-child documents, as well as the generic technique of denormalizing, can be combined in any way, so you can get a good mix of performance and functionality.

# 10

## *Improving performance*

### ***This chapter covers:***

- Bulk, multiget and multisearch API
- Refresh, flush, merge and store
- Filter caches and tuning filters
- Tuning scripts
- Query warmers
- Balancing JVM heap size and OS caches

Elasticsearch is commonly referred to as “fast” when it comes to both indexing, searching, and extracting statistics through aggregations. “Fast” is quite a vague concept, making the “how fast?” question inevitable. As with everything, “how fast” depends on the particular use case, hardware and configuration.

In this chapter, our aim is to show you the best practices for configuring Elasticsearch, so you can make it perform well for your use case. In every situation, you need to trade something for speed, so you need to pick your battles:

- **Application complexity:** In the first part of the chapter, we'll show you how you can group multiple requests, such as index, update, delete, get, and search in a single HTTP call. This grouping is something your application needs to be aware of, but it can speed up your overall performance by a huge margin. Think 20 or 30 times better indexing because you'll have fewer network trips.
- **Indexing speed for search speed, or the other way around:** In the second section of the chapter, we'll take a deeper look at how Elasticsearch deals with Lucene segments: how refreshes, flushes, merge policies and store settings work, and how they influence index and search performance. Often, tuning for indexing performance

has a negative impact on searches and vice versa.

- **Memory:** A big chunk of Elasticsearch's speed is cause by caching. Here's we'll dive into the details of the filter cache and how to use filters to make the best use of it. We'll also look at the shard query cache and how to leave enough room for the operating system to cache your indices, while still leaving enough heap size for Elasticsearch. If running a search on cold caches gets unacceptably slow, you'll be able to keep caches warm by running queries in the background with index warmers.
- **All of the above:** Depending on the use-case, the way you analyze the text at index time and the kind of queries you use can be more complicated, slow down other operations, or use more memory. In the last part of the chapter, we'll explore the typical trade-offs you have while modeling your data and your queries: should you generate more terms when you index or look through more terms when you search? Should you take advantage of scripts or try to avoid them? How should you handle deep paging?

We'll discuss all these points and answer these questions in this chapter. By the end, you will learn how to make Elasticsearch fast for your use-case, and get a deeper understanding of how it works along the way.

For improving performance, grouping multiple operations in a single HTTP request is often the easiest and gives the most performance gain. So let's start by looking at how you can do that through the Bulk, Multiget and Multisearch APIs.

## 10.1 *Grouping requests*

The single best thing you can do for faster indexing is to send multiple documents to be indexed at once, via the bulk API. This will save network round-trips and allow for more indexing throughput. A single bulk can accept any indexing operation, for example you can create documents or overwrite them. You can also add update or delete operations to a bulk; it's not only for indexing.

If your application needs to send multiple get or search operations at once, there are bulk equivalents for them, too: the multiget and multisearch APIs. We'll explore them later, but the bulk API is more popular because in production it's "the way" to index for most use-cases. So we'll look at it first.

### 10.1.1 *Bulk indexing, updating and deleting*

So far in this book you've indexed documents one at a time. This is fine for playing around, but it implies performance penalties from at least two directions:

- Your application has to wait for a reply from Elasticsearch before it can move on.
- Elasticsearch has to process all data from the request for every indexed document.

If you need more indexing speed, Elasticsearch offers a Bulk API, which you can use to index multiple documents at once, as shown in figure 10.1.

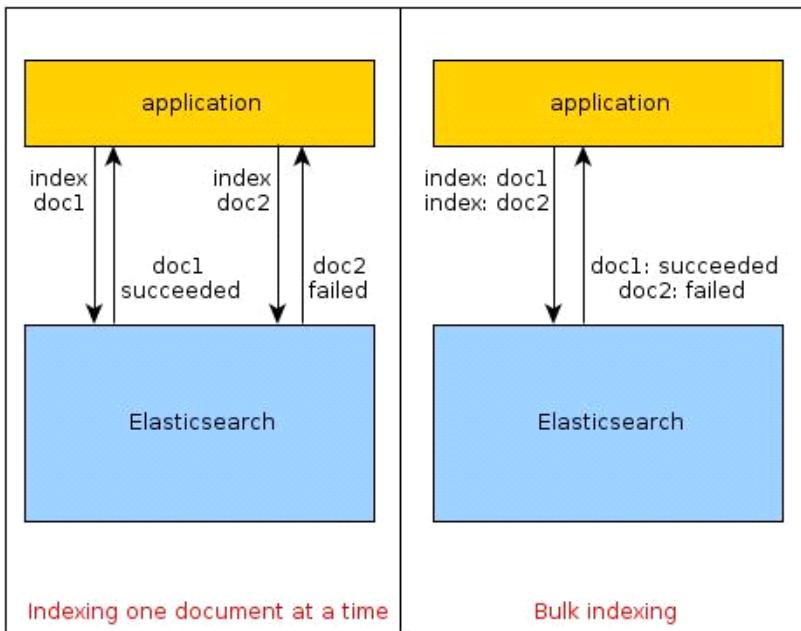


Figure 10.1 Bulk indexing allows you to send multiple documents in the same request

As the figure illustrates, you can do that using HTTP, as you've used for indexing documents so far, and you'll get a reply containing the results of all the indexing requests.

#### INDEXING IN BULKS

In listing 10.1, you'll index a bulk of two documents. To do that, you have to do an HTTP POST to the `_bulk` endpoint, with data in a specific format. The format has the following requirements:

- Each indexing request is composed by 2 JSON documents separated by a newline: one with the operation ("index" in our case) and meta-data (like index, type, and ID), and one with the document contents.
- JSON documents should be one per line. This implies that each line needs to end with a newline (\n, or the ASCII 10 character), including the last line of the whole bulk of requests.

#### **Listing 10.1 Indexing two documents in a single bulk**

```
REQUESTS_FILE=/tmp/test_bulk
echo '{"index": {"_index": "get-together", "_type": "group", "_id": "10"} }' #A
echo '{"name": "Elasticsearch Bucharest"}' #B#C
echo '{"index": {"_index": "get-together", "_type": "group", "_id": "11"} }' #B#D
echo '{"title": "Big Data Bucharest"}' > $REQUESTS_FILE #B#C
curl -XPOST localhost:9200/_bulk --data-binary @$REQUESTS_FILE #A
```

#A Using a file, and point to it via “--data-binary @file-name”, to preserve new-line characters  
#B Every JSON needs to end in a new-line (including the last one) and can't be pretty-printed  
#C First line of the requests contains operation (index) and meta-data (index,type,ID)  
#D Document content

From each of the two indexing requests, in the first line, you added the operation type and some meta-data. The main field name is the operation type: it indicates what Elasticsearch has to do with the data that follows. For now, we've used **index** for indexing, and this operation will overwrite documents with the same ID if they already exist. You can change that to **create**, to make sure documents don't get overwritten, or even **update** or **delete**, to update or delete multiple documents at once, as you can see later.

**\_index** and **\_type** indicate where to index each document. You can put the index name, or both the index and the type in the URL. This will make them the default index and type for every operation in the bulk. For example:

```
curl -XPOST localhost:9200/get-together/_bulk --data-binary @$REQUESTS_FILE
or
curl -XPOST localhost:9200/get-together/group/_bulk -data-binary @$REQUESTS_FILE
```

You can then omit the **\_index** and **\_type** fields from the request itself. If you specify them, index and type values from the request override those from the URL.

The **\_id** field indicates the ID of the document you're indexing. If you omit that, Elasticsearch will automatically generate an ID for you, which is helpful if you don't already have (or don't need) a unique ID for your documents. Logs, for example, work well with generated IDs because they don't typically have a natural unique ID and you don't need to retrieve logs by ID.

### Automatic ID generation and the “create” operation

When you omit the ID from your bulk requests, Elasticsearch generates a random 22-character ID for each document. The chances of getting the same ID for two documents are extremely low.

Since the point of automatic ID generation is to have unique IDs, Elasticsearch will automatically change your **index** operation to **create**. This helps in the unlikely case of an ID clash because, as you'll see later, you can use the bulk response to check the result of each operation. And if there is an ID clash, you'll see that the operation failed.

Automatic ID generation isn't something specific to bulk indexing. You can index a single document like that by omitting the ID, and using HTTP POST instead of PUT. For example:

```
curl -XPOST localhost:9200/cds/music/ -d {"title": "random album"}
```

Similarly, you can choose between **index** and **create** operations when you index single documents by using the **op\_type** parameter:

```
curl -XPOST localhost:9200/cds/music/1?op_type=create -d {"title": "new"}
```

If you don't need to provide IDs, and you index all documents in the same index and type, the bulk request from listing 10.1 gets quite a lot simpler, as shown in the following listing:

### **Listing 10.2 Indexing two documents in the same index and type, with automatic IDs**

```
REQUESTS_FILE=/tmp/test_bulk
echo '{"index":{}}' #A
{"title":"Elasticsearch Bucharest"}
{"index":{}} #A
{"title":"Big Data Bucharest"} > $REQUESTS_FILE
URL='localhost:9200/get-together/group' #B
curl -XPOST $URL/_bulk?pretty --data-binary @$REQUESTS_FILE
```

**#A** only specifying the operation, because index and type are provided in the URL, and IDs will be automatically generated

**#B** Specifying the index and type in the URL

The result of your bulk insert should be a JSON containing the time it took to index your bulk, and the responses for each operation. Something like this:

```
{
  "took" : 2,
  "items" : [ {
    "create" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "ijtvcHVmQd2a1qbDteQn0Q",
      "_version" : 1,
      "ok" : true
    }
  }, {
    "create" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "2QUsVotNSgKfC3_lZvzLrQ",
      "_version" : 1,
      "ok" : true
    }
  } ]
}
```

Note that, because you've used automatic ID generation, the `index` operations were changed to `create`. If one document can't be indexed for a reason, it doesn't mean the whole bulk has failed, because items from the same bulk are independent of each other. That's why you get a reply for each operation, instead of one for the whole bulk. You can use the response JSON in your application, to determine which operation succeeded and which failed.

### **UPDATING OR DELETING IN BULKS**

Within

<https://www.google.ro/search?q=mvn+skip+tests&oq=mvn+skip+&aqs=chrome.1.69i57j0l3.2493j0&sourceid=chrome&ie=UTF-8> single bulk, you can have any number of index or create operations, and also any number of update or delete operations.

a

`update` operations look similar to the index/create operations we've just discussed, except for the fact that you must specify the `ID`. Also, the document content would contain `doc` or `script`, according to the way you want to update, just like you specified `doc` or `script` in chapter 3, when you did individual updates.

`delete` operations are a bit different than the rest, because you have no document content. You just have the meta-data line, like with updates, has to contain the document's ID.

In listing 10.3, you'll have a bulk that contains all four operations: index, create, update and delete.

### Listing 10.3 Bulk with index, create, update and delete

```
echo '{"index":{}}
```

```
{"title":"Elasticsearch Bucharest"}
```

```
{"create":{}}
```

```
{"title":"Big Data in Romania"}
```

```
{"update":{"_id": "11"}}
```

```
{"doc":{"created_on" : "2014-05-06"} }
```

```
{"delete":{"_id": "10"}}' > $REQUESTS_FILE
```

```
URL='localhost:9200/get-together/group'
```

```
curl -XPOST $URL/_bulk?pretty --data-binary @$REQUESTS_FILE
```

```
# expected reply
```

```
    "took" : 37,
```

```
    "errors" : false,
```

```
    "items" : [ {
```

```
        "create" : {
```

```
            "_index" : "get-together",
```

```
            "_type" : "group",
```

```
            "_id" : "rVPtookieSxqfM6_JX-UCkg",
```

```
            "_version" : 1,
```

```
            "status" : 201
```

```
        }
```

```
    }, {
```

```
        "create" : {
```

```
            "_index" : "get-together",
```

```
            "_type" : "group",
```

```
            "_id" : "8w3GoNg5T_WEIL5jSTz_Ug",
```

```
            "_version" : 1,
```

```
            "status" : 201
```

```
        }
```

```
    }, {
```

```
        "update" : {
```

```
            "_index" : "get-together",
```

```
            "_type" : "group",
```

```
            "_id" : "11",
```

```
            "_version" : 2,
```

```
            "status" : 200
```

```
        }
```

```
    }, {
```

```
        "delete" : {
```

```
            "_index" : "get-together",
```

```
            "_type" : "group",
```

```
            "_id" : "10",
```

```
            "_version" : 2,
```

```
            "status" : 200,
```

```
            "found" : true
```

#A Update operation: specify the ID and the partial document  
#B Delete operation: no document is needed, just the ID  
#C Update and delete operations increase the version, like regular updates and deletes

If the bulk APIs can be used to group multiple index, update and delete operations together, you can do the same for search and get requests with the multisearch and multiget APIs respectively. We'll look at these next.

### 10.1.2 Multi Search and Multi Get APIs

The benefit of using multisearch and multiget is the same as with bulks: when you have to do multiple search or get requests, grouping them together saves time otherwise spent on network latency.

#### MULTI SEARCH

One use-case for sending multiple search requests at once is when you're searching in different types of documents. For example, let's assume you have a search box in your get-together site. You don't know whether a search is for groups or for events, so you're going to search for both and offer different "tabs" in the UI: one for groups, one for events. Those two searches would have completely different scoring criteria, so you'd run them in different requests, or you can group these requests together in a Multi Search.

The Multi Search API has many similarities with the bulk API:

- You hit the `_msearch` endpoint, and you may or may not specify an index and a type in the URL
- Each request has two single-line JSON strings: the first may contain the index, type, routing value or search type – the parameters you'd normally put in the URI of a single request. The second line contains the query body, which is normally the payload of a single request

Listing 10.4 shows an example Multi Search request for events and groups about Elasticsearch:

#### Listing 10.4 Multi search request for events and groups about Elasticsearch

```
echo '{"index" : "get-together", "type": "group"}      #A
{"query" : {"match" : {"name": "elasticsearch"}}}      #B
{"index" : "get-together", "type": "event"}            #C
{"query" : {"match" : {"title": "elasticsearch"}}}      #C
' > request                                         #D
curl localhost:9200/_msearch?pretty --data-binary @request #D
# reply
{
  "responses" : [ {
    "took" : 4,                                     #E
    #F
  [...]
    "hits" : [ {
      "_index" : "get-together",                      #F
      "_type" : "group",                            #F
      "_id" : "2",                                 #F
      #F
    }
  ]
}
#F
```

```

    "_score" : 1.8106999,          #F
    "_source":{                   #F
      "name": "Elasticsearch Denver", #F
    [...]                         #F
    }, {                          #G
      "took" : 7,                 #G
    [...]                         #G
    "hits" : [ {                  #G
      "_index" : "get-together",   #G
      "_type" : "event",          #G
      "_id" : "103",              #G
      "_score" : 0.9581454,       #G
      "_source":{                 #G
        "host": "Lee",            #G
        "title": "Introduction to Elasticsearch", #G
      [...]                      #G
    }
  ]
}

```

#A The header of each search contains data that can go to the URL of a single search

#B The body contains the query, like you have with single searches

#C For every other search, you have the a header and a body line

#D Like with bulk requests, it's important to preserve newline characters

#E The response is an array of individual search results

#F Reply for the first query, about events

#G All replies look like individual query replies

## MULTI GET

Multi get makes sense when some processing external to Elasticsearch requires you to fetch a set of documents without doing any search. For example, if you're storing system metrics, and the ID is a timestamp, you might need to retrieve specific metrics from specific times without doing any filtering.

To do that, you'd call the `_mget` endpoint and send a “docs” array with the index, type, and ID of the documents you want to retrieve, like in listing 10.5.

### Listing 10.5 `_mget` endpoint and “docs” array with index, type, and ID of documents

```

curl localhost:9200/_mget?pretty -d '{
  "docs" : [
    {
      "_index" : "get-together",      #A
      "_type" : "group",             #A
      "_id" : "1"                   #A
    },
    {
      "_index" : "get-together",      #A
      "_type" : "group",             #A
      "_id" : "2"                   #A
    }
  ]
}'
# reply
{
  "docs" : [ {
    "_index" : "get-together",      #B
    "_type" : "group",             #C
    "_id" : "1",                  #C
    ...
  }
]
}

```

```

    "_version" : 1,                      #C
    "found" : true,                     #C
    "_source":{                         #C
      "name": "Denver Clojure",        #C
    [...]
  }, {
    "_index" : "get-together",          #C
    "_type" : "group",                 #C
    "_id" : "2",                       #C
    "_version" : 1,                     #C
    "found" : true,                   #C
    "_source":{                         #C
      "name": "Elasticsearch Denver",  #C
    [...]
  }
}

```

**#A** The docs array identifies all documents that you want to retrieve

**#B** The reply also contains a docs array

**#C** Each element of the array is the document as you get it with single GET requests

As with most other APIs, the index and type are optional, because you can also put them in the URL of the request. When the index and type are common for all IDs, it actually is recommend to put them in the URL and put the IDs in an “ids” array, making the request from listing 10.5 much shorter:

```
% curl localhost:9200/get-together/group/_mget?pretty -d '{
  "ids" : [ "1", "2" ]
}'
```

Grouping multiple operations in the same requests with the Multi Get API might introduce a little complexity to your application, but it will make such requests faster without significant costs. The same applies to the Multi Search and Bulk APIs, and to make the best use of them, you can experiment with different request sizes and see which size works best for your documents and your hardware.

Next, we'll look at how Elasticsearch processes documents in bulk internally, in the form Lucene segments, and how you can tune these processes to speed up indexing and searching.

## 10.2 Optimizing the handling of Lucene segments

Once Elasticsearch receives documents from your application, it indexes them in memory in inverted indices called *segments*. From time to time, these segments are written to disk. Recall from chapter 3 that these segments can't be changed, only deleted, to make it easy for the operating system to cache them. Also, bigger segments are periodically created from smaller segments to consolidate the inverted indices and make searches faster.

There are lots of knobs to influence how Elasticsearch handles these segments at every step, and configuring them to fit your use-case often gives important performance gains. In this section, we'll discuss these knobs, and we'll divide them in three categories:

- **How often to refresh and flush:** *Refreshing* re-opens Elasticsearch's view on the index, making newly indexed documents available for search. *Flushing* commits

indexed data from memory to the disk. Both refresh and flush operations are expensive in terms of performance, so it's important to configure them right for your use-case.

- **Merge policies:** Lucene (and by inheritance, Elasticsearch) stores data into immutable groups of files called segments. As you index more data, more segments are created. Because a search in many segments is slow, small segments are merged in the background into bigger segments to keep their number manageable. Merging is performance-intensive, especially for the I/O subsystem. You can adjust the merge policy to influence how often merges happen and how big segments can get.
- **Store and store throttling:** Elasticsearch limits the impact of merges on your system's I/O to a certain number of bytes per second. Depending on your hardware and use-case, you can change this limit. There are also other options around how Elasticsearch uses the storage. For example, you can choose to store your indices only in memory.

We'll start with the category that typically gives you the biggest performance gain of the three: choosing how often to refresh and flush.

### 10.2.1 Refresh and flush thresholds

Recall from chapter 2 that Elasticsearch is often called "near real-time;" that is because searches are often not run on the very latest indexed data (which would be real-time), but very close.

This "near real-time" label fits, because normally Elasticsearch keeps a point-in-time view of the index opened, so multiple searches would hit the same files and re-use the same caches. During this time, newly indexed documents won't be visible to those searches, until you do a refresh.

*Refreshing*, as the name suggests, refreshes this point-in-time view of the index, so your searches can hit your newly indexed data. That's the upside. The downside is that each refresh comes with a performance penalty: some caches will be invalidated, slowing searches down, and the reopening process itself needs processing power, slowing down indexing.

#### WHEN TO REFRESH

The default behavior is to refresh every index automatically every second. You can change the interval for every index, by changing its settings. Changing settings can be done at run-time. For example, the following command will set the automatic refresh interval to 5 seconds:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.refresh_interval": "5s"
}'
```

**TIP** To confirm that your changes were applied, you can get all the index settings by running  
`curl localhost:9200/get-together/_settings?pretty`

As you increase the value of "refresh\_interval," you'll have more indexing throughput, because you'll spend fewer system resources on refreshing.

Alternatively, you can set 'refresh\_interval' to '-1' and effectively disable automatic refreshes, and rely on manual refresh. This works well for use-cases where indices only change periodically in batches, such as a retail chain where products and stocks are updated every night. Indexing throughput is important, because you want to consume those updates quickly, but data freshness isn't, because you don't get the updates in real-time anyway. So you can do nightly bulk index/update with automatic refresh disabled, and refresh manually when you're done.

To refresh manually, you hit the \_refresh endpoint of the index (or indices) you want to refresh:

```
% curl localhost:9200/get-together/_refresh
```

### WHEN TO FLUSH

If you're used to older versions of Lucene or Solr, you might be inclined to think that when a refresh happens, all data that was indexed (in memory) since the last refresh is also committed to disk.

With Elasticsearch (and Solr 4.0 or later) these process of refreshing and the process of committing in-memory segments to disk are independent. Indeed, data is indexed first in memory, but after a refresh, Elasticsearch will happily search the in-memory segments as well. The process of committing in-memory segments to the actual Lucene index you have on disk is called a *flush*, and it happens whether the segments are searchable or not.

To make sure that in-memory data isn't lost when a node goes down or a shard is relocated, Elasticsearch keeps track of the indexing operations that weren't flushed yet in a transaction log. Besides committing in-memory segments to disk, a flush also clears the transaction log, as shown below in Figure 10.2.

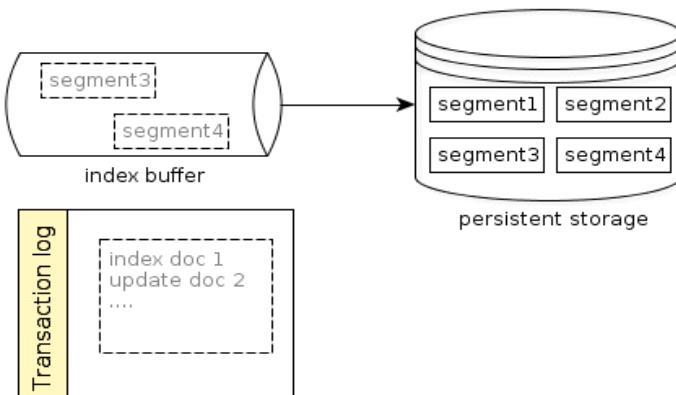


Figure 10.2 A flush moves segments from memory to disk and clears the transaction log

A flush is triggered in one of the following conditions, as shown in figure 10.3:

- the memory buffer is full
- a certain amount of time passed since the last flush
- the transaction log hit a certain size threshold

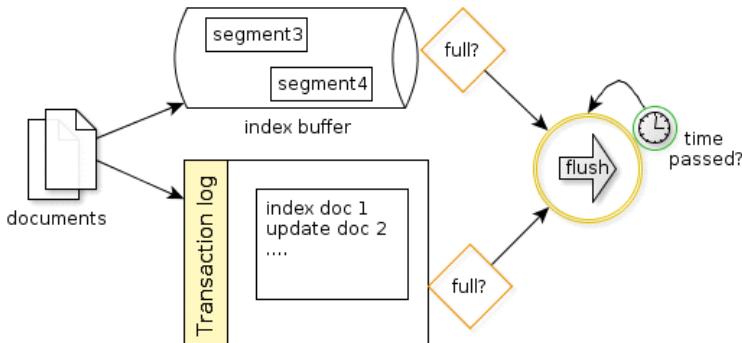


Figure 10.3 A flush is triggered when the memory buffer or transaction log is full, or at an interval

To control how often a flush happens, you have to adjust the settings that control those three conditions.

The memory buffer size is defined in the `elasticsearch.yml` configuration file, through the `"indices.memory.index_buffer_size"` setting. This controls the overall buffer for the entire node and the value can be either a percent of the overall JVM heap like "10%," or a fixed value like "100mb."

Transaction log settings are index-specific and control both the size at which a flush is triggered (via `index.translog.flush_threshold_size`) and the time since the last flush (via `index.translog.flush_translog_period`). As with most index settings, you can change them at run-time:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.translog": {
    "flush_threshold_size": "500mb",
    "flush_threshold_period": "10m"
  }
}'
```

When a flush is performed, one or more segments are created on the disk. When you run a query, Elasticsearch (through Lucene) looks in all segments and merges the results together in an overall shard result. Then, as you saw in chapter 2, per-shard results are aggregated into the overall results that go back to your application.

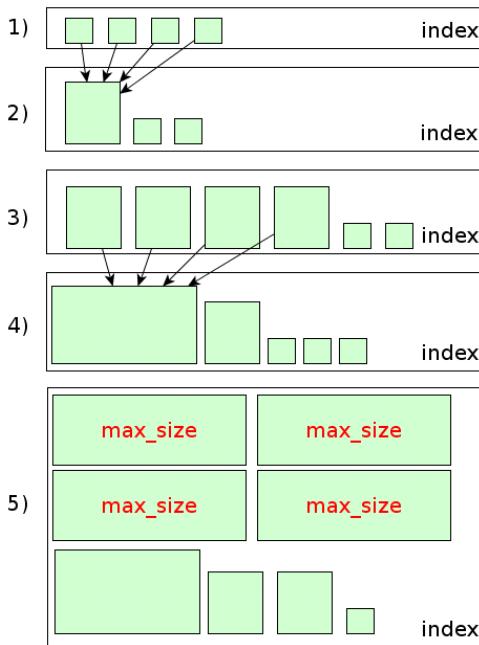
The key thing to remember here about segments is that the more segments you have to search through, the slower the search. And, to keep the number of segments at bay, Elasticsearch (again, through Lucene) merges multiple sets of smaller segments into bigger segments in the background.

## 10.2.2 Merges and merge policies

We first introduced segments in chapter 3, as immutable sets of files that Elasticsearch uses to store indexed data. Because they don't change, segments are easily cached, making searches fast. Also, changes to the data set, such as the addition of a document, won't require rebuilding the index for data stored in existing segments. This makes indexing new documents fast too, but it's not all good news. Updating a document can't change the actual document, only index a new one. This requires deleting the old document too. Deleting, in turn, can't remove a document from its segment, so it's only marked as deleted. Documents are only actually removed during segment merging.

This brings us to the two purposes of merging segments: to keep the total number of segments in check (and with it, query performance), and to remove deleted documents.

Segment merging happens in the background, according to the defined merge policy. The default merge policy is "tiered" which, as illustrated in figure 10.4 below, divides segments in tiers and, if you have more than the set maximum number of segments in a tier, a merge is triggered in that tier.



#1 Flush operations add segments in the first tier, until they are too many. Let's say 4 are too many.

#2 Small segments are merged into bigger ones. Flushing continues to add new small segments.

#3 Eventually, there will be 4 segments on the 'bigger' tier.

#4 The 4 'bigger' segments get merged into an 'even bigger' segment and the process continues.

#5 ...until a tier hits a set limit. Only smaller ones get merged, 'max' segments stay the same.

Figure 10.4 Tiered merge policy performs a merge when it finds too many segments in a tier

There are other merge policies, but in this chapter we'll focus only on the tiered merge policy, because it works best for most use-cases.

### TUNING MERGE POLICY OPTIONS

The overall purpose of merging is to trade I/O and some CPU time for search performance. Merging happens when you index, update or delete documents, so the more you merge, the more expensive these operations get. Conversely, if you want faster indexing, you'd need to merge less, and sacrifice some search performance.

In order to have more or less merging, you have a few configuration options. Here are the most important ones:

- `index.merge.policy.segments_per_tier`: The higher the value, the more segments you can have in a tier. This will translate to less merging and better indexing performance. If you have little indexing and you want better search performance, lower this value
- `index.merge.policy.max_merge_at_once`: This setting limits how many segments can be merged at once. You'd typically have it equal to the `segments_per_tier` value. You could lower the `max_merge_at_once` value to force less merging, but it's better to do that by increasing `segments_per_tier`. Make sure `max_merge_at_once` isn't higher than `segments_per_tier`, because this will cause too much merging
- `index.merge.policy.max_merged_segment`: This defines the maximum segment size – bigger segments will not be merged with other segments. You'd lower this value if you want less merging and faster indexing, because larger segments are more difficult to merge.
- `index.merge.scheduler.max_thread_count`: Merging happens in background on separate threads, and this setting controls the maximum number of threads that can be used for merging. This is the hard limit of how many merges can happen at once. You would increase this setting for an aggressive merge policy on a machine with many CPUs and fast I/O, and you would decrease it if you have slow CPU or I/O.

All those options are index-specific and, as with transaction log and refresh settings, you can change them at run-time. For example, the following snippet forces more merging by reducing `segments_per_tier` to 5 (and with it, `max_merge_at once`), lowers the maximum segment size to 1GB, and lowers the thread count to 1 to work better with slow disks:

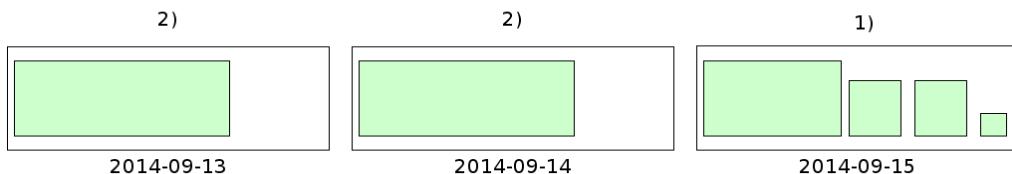
```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.merge": {
    "policy": {
      "segments_per_tier": 5,
      "max_merge_at_once": 5,
      "max_merged_segment": "1gb"
    },
    "scheduler.max_thread_count": 1
  }
}'
```

## OPTIMIZING INDICES

As with refreshing and flushing, you can trigger a merge manually. A forced merge call is also known as “optimize,” because you’d typically run it on an index that isn’t going to be changed later, to “optimize” it to a specified (low) number of segments for faster searching.

As with any aggressive merge, optimizing is I/O intensive and invalidates lots of your operating system caches. If you continue to index, update, or delete documents from that index, new segments will be created and the advantages of optimizing will no longer be there. As a result, if you want fewer segments on an index that’s constantly changing, you should tune the merge policy.

Optimizing makes sense on a static index. For example, if you index social media data, and you have one index per day, you know you’ll never change yesterday’s index until you remove it for good. It might help to optimize it to a low number of segments, as shown in figure 10.5, which will reduce its total size and speed up queries, once caches are warmed up again.



- #1 Active (today's) index: gets updated, merges work according to merge policy
- #2 Static indices: optimized to one segment for compact size and faster searches (once caches are warmed up again)

Figure 10.5 Optimizing makes sense for indices that don’t get updates

To optimize, you’d hit the `_optimize` endpoint of the index or indices you need to optimize. The `max_num_segments` option indicates how many segments you should end up with, per shard:

```
% curl localhost:9200/get-together/_optimize?max_num_segments=1
```

An optimize call can take a long time on a large index. You can send it to background by setting `wait_for_merge` to “false”.

One possible reason for an optimize (or any merge) being slow is the fact that Elasticsearch, by default, limits the amount of I/O throughput merge operations can use. This limiting is called *store throttling*, and we’ll discuss it next, along with other options for storing your data.

### 10.2.3 *Store and store throttling*

In early versions of Elasticsearch, heavy merging could slow the cluster down so much that indexing and search requests would take unacceptably long, or nodes could become unresponsive altogether. This was all due to the pressure of merging on the I/O throughput,

which would make the writing of new segments slow. Also, CPU load was higher due to I/O wait.

As a result, Elasticsearch now limits the amount of I/O throughput that merges can use through store throttling. By default, there's a node-level setting called `indices.store.throttle.max_bytes_per_sec` which defaults to 20mb, as of version 1.3.

This limit is good for stability in most use-cases, but won't work well for everyone. If you have fast machines and lots of indexing, merges won't keep up, even if there is enough CPU and I/O to perform them. In such situations, Elasticsearch makes internal indexing work only on one thread, slowing it down to allow merges to keep up. In the end, if your machines are fast, indexing might be limited by store throttling.

### **CHANGING STORE THROTTLING LIMITS**

If you have fast disks and need more I/O throughput for merging, you can raise the store throttling limit. You can also remove the limit altogether by setting `indices.store.throttle.type` to none. On the other end of the spectrum, you can apply the store throttling limit to all of Elasticsearch's disk operations, not just merging, by setting `indices.store.throttle.type` to all.

Those settings can be changed from `elasticsearch.yml` on every node, but they can also be changed at run-time through the Cluster Update Settings API. The following command would raise the throttling limit to 500MB/s, but apply it to all operations. It will also make the change persistent, to survive full cluster restarts (which is opposed to transient settings, that are lost when the cluster is restarted):

```
% curl -XPUT localhost:9200/_cluster/settings -d '{
  "persistent": {
    "indices.store.throttle": {
      "type": "all",
      "max_bytes_per_sec": "500mb"
    }
  }
}'
```

**TIP** Like with index settings, you can also get cluster settings to see if they're applied. You'd do that by running `curl localhost:9200/_cluster/settings?pretty`

### **CONFIGURING STORE**

When we talked about flushes, merges and store throttling, we said "disk" and "I/O" because that's the default: Elasticsearch will store indices in the data directory, which defaults to `/var/lib/elasticsearch/data` if you installed Elasticsearch from a RPM/DEB package, or the `data/` directory from the unpacked tar.gz or zip archive if you installed it manually. You can change the data directory from the `path.data` property of `elasticsearch.yml`.

## Multiple data directories

If you have multiple disks, you can make Elasticsearch store data on all of them by adding multiple paths to the `path.data` property. For example:

```
path.data: /mnt/sdb1,/mnt/sdc1
```

This will divide the files making up your indices in all the provided locations. This is roughly equivalent to having RAID0, because a search will typically hit files on all locations and use all the available disk throughput.

By default, Elasticsearch places files on the disk with more available space, which works well if your disks are equal. If you have a bigger disk, files will get written only there until it has the same free space as the others. You can change the strategy by setting `index.store.distributor` to `random` in the index settings when you create an index. This will write files on all disks, but disks with more space get proportionally more files.

The default store implementation stores index files in the file system, and works well for most use-cases. To access Lucene segment files, the default store implementation uses Lucene's `MMapDirectory` for files that are typically large or need to be randomly accessed, such as term dictionaries. For the other types of files, such as stored fields, Elasticsearch uses Lucene's `NIODirectory`.

### **MMapDirectory**

`MMapDirectory` takes advantage of file system caches by asking the operating system to map the needed files in virtual memory, in order to access that memory directly. To Elasticsearch, it looks as if all the files are available in memory, but that doesn't have to be the case. If your index size is larger than your available physical memory, the operating system will happily take unused files out of the caches to make room for new ones that need to be read. If Elasticsearch needs those un-cached files again, they are loaded in memory, while other unused files are taken out and so on. The virtual memory used by `MMapDirectory` works similarly to the system's virtual memory (swap), where the operating system uses the disk to page out unused memory in order to be able to serve multiple applications

### **NIODirectory**

Memory mapped files also imply an overhead, because the application has to tell the operating system to map a file, before accessing it. To reduce this overhead, Elasticsearch uses `NIODirectory` for some types of files. `NIODirectory` access files directly, but it has to copy the data it needs to read in a buffer in the JVM heap. This makes it good for small, sequentially-accessed files, while `MMapDirectory` works well for large, randomly-accessed files.

The default store implementation is best for most use-cases. You can, however, choose other implementations by changing `index.store.type` in the index settings to other values than `default`:

- `mmapfs`: This will use the MMapDirectory alone, and would work well, for example, if you have a relatively static index that fits in your physical memory.
- `niofs`: This will use NIODirectory alone, and would work well on 32-bit systems, where virtual memory address space is limited to 4GB, which will prevent you from using `mmapfs` or `default` for larger indices.
- `memory`: This will store the index in memory, and it works well for small indices that don't need persistence, like you would have in unit tests. It doesn't work well with big indices, because the operating system would still waste memory trying to cache it (yes, memory cache for memory index), and the internal buffers are small.

Store type settings need to be configured when you create the index. For example, the following command creates an in-memory index called "unit-test":

```
% curl -XPUT localhost:9200/unit-test -d '{
  "index.store.type": "memory"
}'
```

If you want to apply the same store type for all newly created indices, you can set `index.store.type` to `memory` in `elasticsearch.yml`. In chapter 11, we'll also introduce index templates, which allow you to define index settings that would apply to new indices matching specific patterns. Templates can also be changed at run-time, and we recommend using them instead of the more static `elasticsearch.yml` equivalent if you often create new indices.

### **Open files and virtual memory limits**

Lucene segments that are stored on disk can spread onto many files, and when a search runs, the operating system needs to be able to open many of them. Also, when you're using the default store type or `mmapfs`, the operating system has to map some of those stored files into memory – even though these files aren't in memory, to the application it's like they are, and the kernel takes care of loading and unloading them in the cache.

On Linux, there are configurable limits that prevent the applications from opening too many files at once, and for mapping too much memory. These limits are typically more conservative than needed for Elasticsearch deployments, so it's recommended to increase them. If you're installing Elasticsearch from a DEB or RPM package, you don't have to worry about this, because they are increased by default. You can find these variables in `/etc/default/elasticsearch` or `/etc/sysconfig/elasticsearch`:

```
MAX_OPEN_FILES=65535
MAX_MAP_COUNT=262144
```

To increase those limits manually, you have to run `ulimit -n 65535` as the user that starts Elasticsearch for the open files, and `sysctl -w vm.max_map_count=262144` as root for the virtual memory.

The default store type is typically the fastest because of the way the operating system caches files. For caching to work well, you need to have enough free memory. Also, we mentioned how merge and optimize operations invalidate caches. Managing caches for Elasticsearch to perform well deserves more explanation, so we'll discuss them next.

## 10.3 Making the best use of caches

One of Elasticsearch's strong points - if not the biggest strong point - is the fact that you can query billions of documents in milliseconds with commodity hardware. And one of the reasons this is possible is its smart caching. You might have noticed that, after indexing lots of data, the second query can be orders of magnitude faster than the first one. It's because of caching: for example, when you combine filters and queries, the filter cache plays a very important role in keeping your searches fast.

In this section, we'll discuss the filter cache, and two other types of caches: the shard query cache, useful when you run aggregations on static indices because it caches the overall result, and the operating system caches, which keep your I/O throughput high by caching indices in memory.

Finally, we'll show you how to keep all those caches warm by running queries at each refresh with index warmers. Let's start by looking at the main type of Elasticsearch-specific cache - the filter cache – and how you can run your searches to make the best use of it.

### 10.3.1 Filters and filter caches

In chapter 4 you saw that lots of queries have a filter equivalent. Let's say that you want to look for events on the get-together site that happened in the last month. To do that, you could use the range query or the equivalent range filter.

In chapter 4 we said that of the two, we recommend using the filter, because it's cacheable. In fact, the range filter is cached by default, but you can control whether a filter is cached or not through the `_cache` flag. This flag applies to all filters; for example the following snippet will filter events with "elasticsearch" in the verbatim tag, but won't cache the results:

```
% curl -XPUT localhost:9200/get-together/group/_search?pretty -d '{
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "tags.verbatim": "elasticsearch",
          "_cache": false
        }
      }
    }
  }
}'
```

**NOTE** While all filters have the `_cache` flag, it doesn't apply in 100% of cases. For the range filter, if you use "now" as one of the boundaries, the flag is ignored. For the `has_child` or `has_parent` filters, the `_cache` flag doesn't apply at all.

## FILTER CACHE

The results of a filter that is cached are stored in the filter cache. This cache is allocated at the node level, like the index buffer size you saw earlier. It defaults to 10%, but you can change it from `elasticsearch.yml` according to your needs. If you use filters a lot and cache them, it might make sense to increase the size, for example:

```
indices.cache.filter.size: 30%
```

How do you know if you need more (or less) filter cache? By monitoring your actual usage. As we'll explore in chapter 11 on administration, Elasticsearch exposes lots of metrics, including the amount of filter cache that's actually used, and the amount of cache evictions. An eviction happens when the cache gets full, and Elasticsearch drops the least recently used (LRU) entry in order to make room for the new one.

In some use-cases, filter cache entries have a short lifespan. For example, users typically filter get-together events by a particular subject, refine their queries until they find what they want, and then leave. If nobody else is searching for events on the same subject, that cache entry will stick around doing nothing until it eventually gets evicted. A full cache with many evictions would make performance suffer, because every search will consume CPU cycles to squeeze new cache entries by evicting old ones.

In such use-cases, to prevent evictions from happening exactly when queries are run, it makes sense to set a time to live (TTL) on cache entries. You can do that on a per-index basis by adjusting `index.cache.filter.expire`. For example, the following snippet will expire filter caches after 30 minutes:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.cache.filter.expire": "30m"
}'
```

Besides making sure you have enough room in your filter caches, you also need to run your filters in a way that takes advantage of these caches.

## COMBINING FILTERS

You often need to combine filters, such as when you're searching for events in a certain time range, but also with a certain number of attendees. For best performance, you'd need to make sure caches are well used when filters are combined, and that filters run in the right order.

To understand how to best combine filters, we need to revisit a concept discussed in chapter 4: bitsets. A *bitset* is a compact array of bits, and it's used by Elasticsearch to cache whether a document matches a filter or not. Most filters (such as the range and terms filter) use bitsets for caching. Other filters, such as the script filter, don't use bitsets because Elasticsearch has to iterate through all documents anyway. Table 10.1 shows which filters use bitsets and which don't.

Table 10.1

Filter type	Uses bitset
term	yes
terms	yes, but you can configure it differently, as we'll explain in a bit
exists/missing	yes
prefix	yes
regex	no
nested/has_parent/has_child	no
script	no
geo filters (see appendix A)	no

For filters that don't use bitsets, you can still set `_cache` to true in order to cache results of that exact filter. *Bitsets* are different than simply caching the results because they are:

- Compact and easy to create: so the overhead of creating the cache when the filter is first run is insignificant,
- Stored per individual filter: for example, if you use a term filter in two different queries, or within two different bool filters, the bitset of that terms can be-reused.
- Easy to combine with other bitsets: If you have two queries that use bitsets, it's easy for Elasticsearch to do a bitwise AND or OR in order to figure out which documents match the combination.

To take advantage of bitsets, you need to combine filters that use them in a bool filter, that will do that bitwise AND or OR, which is very easy for your CPU. For example, if you only want to show groups where either Lee is a member or contain the tag "elasticsearch" could look like this:

```
"filter": {
  "bool": {
    "should": [
      {
        "term": {
          "tags.verbatim": "elasticsearch"
        }
      },
      {
        "term": {
          "members": "lee"
        }
      }
    ]
  }
}
```

The alternative of combining filters is by using the `and`, `or` and `not` filters. These filters work differently because, unlike the `bool` filter, they don't use bitwise AND or OR. They just run the first filter, pass the matching documents to the next one, and so on. As a result, `and`, `or` and `not` filters are better when it comes to combining filters that don't use bitsets. For example, if you want to show groups having at least 3 members, with events organized in July 2013, the filter might look like this:

```
"filter": {
  "and": {
    "filters": [
      {
        "has_child": {
          "type": "event",
          "filter": {
            "range": {
              "date": {
                "from": "2013-07-01T00:00",
                "to": "2013-08-01T00:00"
              }
            }
          }
        }
      },
      {
        "script": {
          "script": "doc['members'].values.length > minMembers",
          "params": {
            "minMembers": 2
          },
          "lang": "groovy"
        }
      }
    ]
  }
}
```

If you're using both bitset and non-bitset filters, you can combine the bitset ones in a `bool` filter, and put that `bool` filter in an `and/or/not` filter, along with the non-bitset filters. For example, in listing 10.6, we'll look for groups with at least two members, where either Lee is one of them or the group is about Elasticsearch:

#### **Listing 10.6 Combine bitset filters in a bool filter inside an and/or/not filter**

```
curl localhost:9200/get-together/group/_search?pretty -d'{
  "query": {                      #A
    "filtered": {                  #A
      "filter": {
        "and": {                   #B
          "filters": [             #B
            {
              "bool": {             #C
                "should": [         #C
                  {
                    "term": {         #C
                      "tags.verbatim": "elasticsearch"

```

```
        },
        {
          "term": {
            "members": "lee"
          }
        }
      ],
    }
  },
  {
    "script": { #D
      "script": "doc[\"members\"].values.length > minMembers",
      "params": {
        "minMembers": 2
      },
      "lang": "groovy"
    }
  }
]
}
}
```

**#A Filtered query means if we add a query here, it will only run on documents matching the filter**

**#B The AND filter will run the bool filter first**

**#C Bool is fast when cached, because it makes use of the two bitsets of the term filters**

**#D The script filter will only work on documents matching the bool filter**

Whether you combine filter with the `bool`, `and`, `or` or `not` filters, the order in which those filters are executed is important. Cheaper filters, such as the `term` filter, should be placed before expensive filters, such as the `script` filter. This would make the expensive filter run on a smaller set of documents – those who already matched previous filters.

## RUNNING FILTERS ON FIELD DATA

So far, we've discussed how bitsets and cached results make your filters faster. Some filters use bitsets; some can cache the overall results. Some filters can also run off a different kind of in-memory structure: field data. We first discussed field data in chapter 6 as an in-memory structure that keeps a mapping of documents to terms. This mapping is the opposite of the inverted index, which maps terms to documents. Field data is typically used when sorting and during aggregations, but some filters can use it, too: the terms and the range filters.

A terms filter can have lots of terms, and a range filter with a wide range will (under the hood) match lots of numbers (and numbers are also terms). Normal execution of those filters will try to match every term separately and return the set of unique documents, as illustrated in figure 10.6:

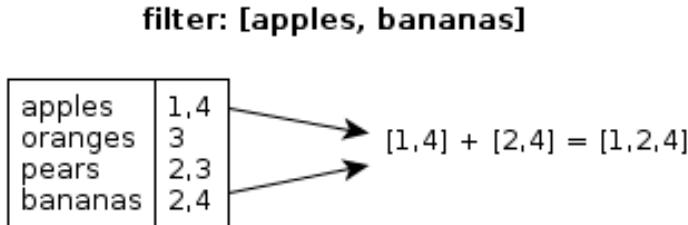


Figure 10.6 Terms filter is by default checking which documents match each term, and intersects the lists

As you can imagine, filtering on many terms can get expensive, because there would be many lists to intersect. When the number of terms is large, it can be faster to take the actual field values one by one and see if the terms match, instead of looking in the index, as illustrated in figure 10.7:

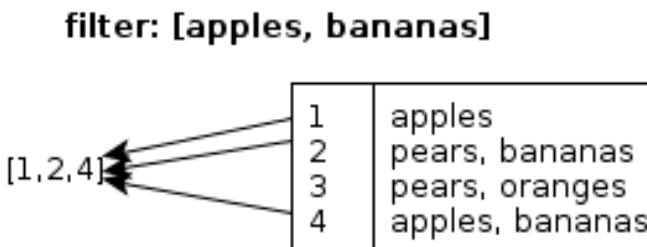


Figure 10.7 Field data execution means iterating through documents, but no list intersections

These field values would be loaded in the field data cache, by setting “execution” to “fielddata” in the terms or range filters. For example, the following range filter will get events that happened in 2013, and will be executed on field data:

```

"filter": {
  "range": {
    "date": {
      "from": "2013-01-01T00:00",
      "to": "2014-01-01T00:00"
    }
  },
  "execution": "fielddata"
}
  
```

Using field data execution is especially useful when the field data is already used by a sort operation or an aggregation. For example, running a terms aggregation on the tags field will make a subsequent terms filter for a set of tags faster, because the field data is already loaded.

### **Other execution modes for the terms filter: bool, and, or**

The terms filter has other execution modes, too. If the default execution mode (called `plain`) builds a bitset to cache the overall result, you can set it to `bool` in order to have a bitset for each term instead. This is useful when you have different terms filters, that have lots of terms in common.

Also, there are `and/or` execution modes, which perform a similar process, except the individual term filters are wrapped in an `and/or` filter instead of a `bool` filter.

Usually, the `and/or` approach is slower than `bool`, because they don't take advantage of bitsets. `and/or` might be faster if the first term filters match only a few documents, which make subsequent filter extremely fast.

To sum up, you have three options for running your filters:

- Caching them in the filter cache, which is great when filters are re-used.
- Not caching them, if they aren't re-used.
- For terms and range filters, run them on field data, which is good when you have many terms, especially if the field data for that field is already loaded.

Next, we'll look at the shard query cache, which is good for when you re-use entire search requests over static data.

#### **10.3.2      *Shard query cache***

The filter cache is purpose-built to make parts of a search, namely filters that are configured to be cached, run faster. It's also segment-specific: if some segments get removed by the merge process, other segments' caches remain intact. By contrast, the shard query cache maintains a mapping between the whole request and its results on the shard level, as illustrated in figure 10.8.

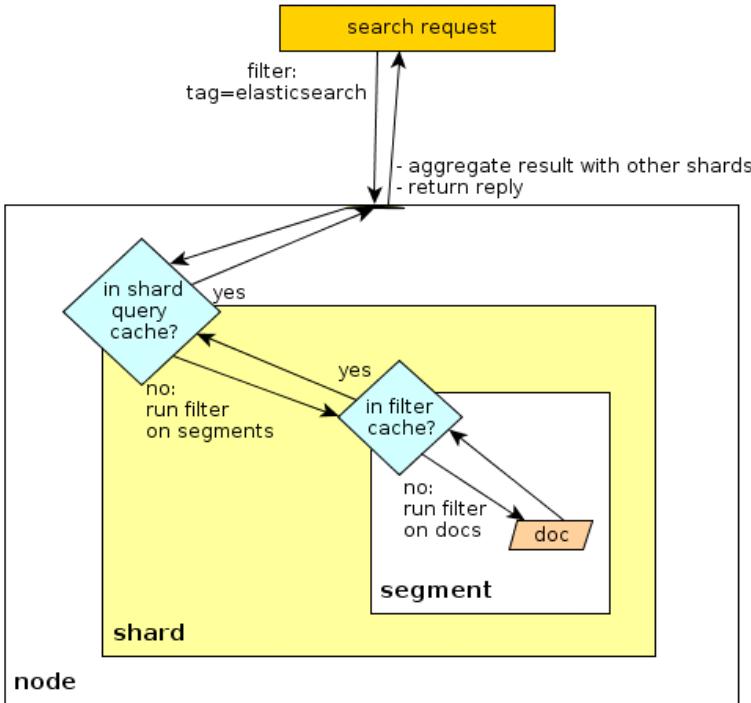


Figure 10.8 Shard query cache is more high-level than the filter cache

As of version 1.4, results cached at the shard level are limited to the total number of hits (not the hits themselves!), aggregations and suggestions. That is why, in 1.4 at least, shard query cache only works when your query has `search_type` set to `count`.

**NOTE** By setting `search_type` to `count` in the URI parameters, you tell Elasticsearch that you're not interested in the query results, just in their number. We'll look at `count` and other search types later in this section.

The shard query cache entries differ from one request to another, so they only apply to a narrow set of requests. Also, when a refresh occurs and the shard's contents changed, all shard query cache entries are invalidated – otherwise you could get outdated results from the cache.

This “narrowness” of cache entries makes the shard query cache valuable only when shards change rarely and you have many identical requests. For example, if you're indexing logs and have time-based indices, you may often run aggregations on older indices that typically remain unchanged until they are deleted. These older indices are an ideal candidate for shard query cache.

To enable the shard query cache by default on the index level, you can use the indices update settings API:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.cache.query.enable": true
}'
```

**TIP** As with all index settings, you can enable the shard query cache at index creation, but it only makes sense to do that if your new index gets queried a lot and updated rarely.

For every query, you can also enable or disable the shard query cache, overriding the index-level setting, by adding the query\_cache parameter. For example, to cache the frequent top\_tags aggregation on our get-together index, even if the default is disabled, you can run it like this:

```
% URL="localhost:9200/get-together/group/_search"
% curl "$URL?search_type=count&query_cache&pretty" -d '{
  "aggs": {
    "top_tags": {
      "terms": {
        "field": "tags.verbatim"
      }
    }
  }
}'
```

Like the filter cache, the shard query cache has a “size” configuration parameter. The limit can be changed at the node level by adjusting `indices.cache.query.size` from `elasticsearch.yml`, from the default of 1% of the JVM heap.

When sizing the JVM heap itself, you need to make sure you have enough room for both the filter and the shard query caches. Also, you need to have enough free RAM besides the JVM heap, to allow the operating system to cache indices stored on disk, otherwise you'll have a lot of disk seeks.

Next, we'll look at how you can balance the JVM heap with the OS caches, and why that matters.

### 10.3.3 *JVM heap and OS caches*

If Elasticsearch doesn't have enough heap to finish an operation, it throws an out of memory exception, which effectively makes the node crash and fall out of the cluster. This puts extra load on other nodes, as they replicate and relocate shards in order to get back to the configured state. Because nodes are typically equal, this extra load is likely to make at least another node run out of memory. Such a domino effect can bring down your entire cluster.

When JVM heap is tight, even if you don't see an out of memory error in the logs, the node may become just as unresponsive. This can happen because the lack of memory pressures the Garbage Collector to run longer and more often in order to free memory. As GC takes more

CPU time, there's less computing power on the node for serving requests or even answering pings from the master, causing the node to fall out of the cluster.

**NOTE** If GC is taking too much time (or kicks in too late or too early), but you otherwise have enough memory, have a look at appendix F for some tips on how to tune the GC and other JVM settings for your use-case. But don't take GC tuning as some magical solution, because it doesn't help if you simply don't have enough memory. The default JVM and GC settings are good for most use-cases, so before investing time in GC tuning make sure there's enough JVM heap in the first place. We'll discuss how you can monitor the JVM heap and other relevant metrics in chapter 11.

#### CAN YOU HAVE TOO LARGE OF A HEAP?

It might have been obvious that a heap that's too small is bad, but having a heap that's too large isn't great either. A heap size of more than 32GB will automatically make pointers uncompressed and waste memory. You'd typically have to go beyond 48GB to get the same usable space as with 32GB because of this. If you really need more than 32GB of heap, you'd probably be better off running two or more nodes on the same machine, each with less than 32GB of heap, and divide the data between them through sharding.

**NOTE** If you end up with multiple Elasticsearch nodes on the same physical machine, you need to make sure that two replicas of the same shard aren't allocated on the same physical machine, under different Elasticsearch nodes. Otherwise, if a physical machine goes down, you'd lose two copies of that shard. To prevent this, you can use shard allocation, as described in chapter 9.

Below 32GB, too much heap still isn't ideal. The RAM on your servers that isn't occupied by the JVM is typically used by the operating system to cache indices that are stored on the disk. This is very important especially if you have magnetic or network storage, because fetching data from the disk while running a query will delay its response. Even with fast SSDs, if you have lots of queries, you'll get the best performance if the amount of data you need to store on a node can fit in its OS caches.

So far we've shown that a heap that's too small is bad because of GC and out of memory issues, and one that's too big is bad too, because it diminishes OS caches. What is a good heap size then?

#### IDEAL HEAP SIZE; FOLLOW THE HALF RULE

Without knowing anything about the actual heap usage for your use-case, the rule of thumb is to allocate half of the node's RAM to Elasticsearch, but no more than 32GB. This "half" rule often gives a good balance between heap size and OS caches.

If you can monitor the actual heap usage (and we'll show you how to do that in chapter 11), a good heap size is just about large enough to accommodate the regular usage, plus any spikes you might expect. Memory usage spikes could happen, for example, if someone decides to run a terms aggregation with size 0 on an analyzed field with many unique terms. This will

force Elasticsearch to load all terms in memory in order to count them. If you don't know what spikes to expect, the rule of thumb is again "half": set a heap size 50% higher than your "regular" usage.

For OS caches, you depend mostly on the RAM of your servers. That said, you can design your indices in a way that works best with your operating system's caching. For example, if you're indexing application logs, you can expect that most indexing and searching will involve recent data. With time-based indices, the latest index is more likely to fit in the OS cache than the whole dataset, making most operations faster. Searches on older data will often have to hit the disk, but users are more likely to expect and tolerate slow response times on these rare searches that span on longer periods of time. In general, if you can put "hot" data in the same set of indices or shards, either by using time-based indices, user-based indices or routing, you will make better use of OS caches.

All the caches we discussed so far – filter caches, shard query caches, OS caches – are typically built when a query first runs. Loading up the caches makes that first query slower, and the slowdown increases with the amount of data and the complexity of the query. If that slowdown becomes a problem, you can warm up the caches in advance by using index warmers, as you'll see next.

### 10.3.4 *Keeping caches up with warmers*

A warmer allows you to define any kind of search request: it can contain queries, filters, sort criteria and aggregations. Once it's defined, the warmer will make Elasticsearch run the query with every refresh operation. This will slow down the refresh, but the user queries will always run on "warm" caches.

Warmers are useful when first-time queries are too slow, and it's preferable for the refresh operation to take that hit, rather than the user. If our get-together site example would have millions of events, and consistent search performance would be important, warmers would be useful: slower refreshes shouldn't concern us too much, because we expect groups and events to be searched for more often than they are modified.

To define a warmer on an existing index, you would do a PUT request to the index's URI, with `_warmer` as the type, and the chosen warmer name as an ID, like you see in listing 10.7. You can have as many warmers as you want, but keep in mind that the more warmers you have, the slower your refreshes will be. Typically, you'd use a few popular queries as your warmers. For example, in listing 10.7, we'd put two warmers: one for upcoming events and one for popular group tags:

#### **Listing 10.7 Two warmers: for upcoming events and popular group tags**

```
curl -XPUT 'localhost:9200/get-together/event/_warmer/upcoming_events' -d '{
  "sort": [
    {
      "date": { "order": "desc" }
    }
  ]
}'
# {"acknowledged": true}
curl -XPUT 'localhost:9200/get-together/group/_warmer/top_tags' -d '{'
```

```

    "aggs": {
      "top_tags": {
        "terms": {
          "field": "tags.verbatim"
        }
      }
    }
  }
# {"acknowledged": true}

```

Later on, you can get the list of warmers for an index by doing a GET request on the `_warmer` type:

```
curl localhost:9200/get-together/_warmer?pretty
```

You can also delete warmers by sending a DELETE request to the warmer's URI:

```
curl -XDELETE localhost:9200/get-together/_warmer/top_tags
```

If you're using multiple indices, it makes sense to register warmers at index creation. To do that, you'd define them under the "warmers" key, in the same way as you do with mappings and settings:

#### **Listing 10.8 Register warmer at index creation time**

```

curl -PUT 'localhost:9200/hot_index' -d '{
"warmers": {
  "date_sorting": { #A
    "types": [], #B
    "source": { #C
      "sort": [ { #D
        "date": { #D
          "order": "desc" #D
        }
      } ]
    }
  }
}'

#A Name of this warmer. You can register multiple warmers, too
#B On which types should this warmer run. Empty means all types
#C Under this key we define the warmer itself
#D This warmer sorts by date, as the name suggests

```

**TIP** If new indices are created automatically, like you might have if you're using time-based indices, you can define warmers in an index template that will be applied automatically to newly created indices. We'll talk more about index templates in chapter 11, which is all about how to administer your Elasticsearch cluster.

So far we talked about general solutions: how to keep caches warm and efficient make your searches fast, how to group requests reduce network latency, and how to configure

segment refreshing, flushing and storing in order to make your indexing and searching fast. All this also should reduce the load on your cluster.

Next, we'll talk about narrower best practices, ones that apply to specific use-cases. For example, how to make your scripts fast or how do deep paging efficiently.

## 10.4 Other performance trade-offs

In previous sections, you might have noticed that to make an operation fast, you need to pay with something. For example, if you make indexing faster by refreshing less often, you pay with searches that may not "see" recently indexed data. In this section, we'll continue looking at such trade-offs, especially those occur in more specific use-cases, by answering questions on the following topics:

- **Inexact matches:** Should you get faster searches by using ngrams and shingles at index-time? Or is it better to use fuzzy and wildcard queries?
- **Scripts:** Should you trade some flexibility by calculating as much as possible at index time? If not, how can I squeeze more performance out of them?
- **Distributed search:** Should you trade some network round-trips for more accurate scoring?
- **Deep paging:** Is it worth trading memory to get page 100 faster?

By the time this chapter ends, we'll answer all these questions and lots of others that will come up along the way. Let's start with inexact matches.

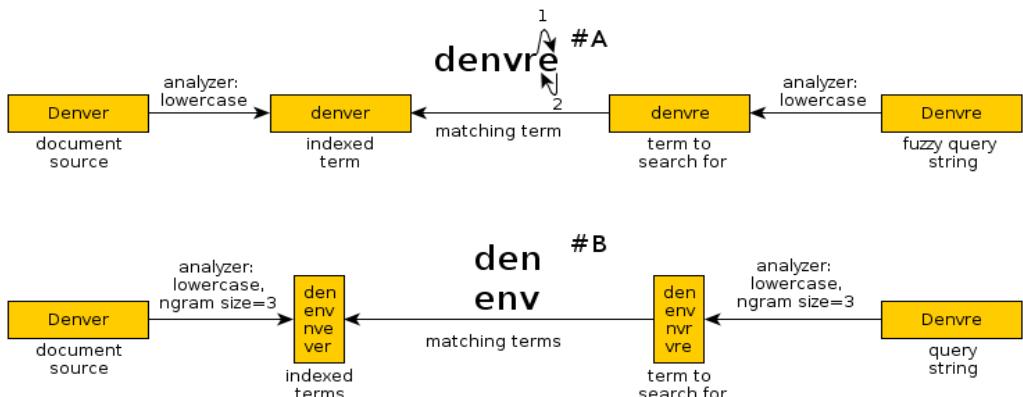
### 10.4.1 Big indices or expensive searches

Recall from chapter 4, that to get inexact matches, for example to tolerate typos, there are a number of queries that can help:

- Fuzzy query: which matches terms at a certain edit distance from the original. For example, omitting or adding an extra character would make a distance of 1
- Prefix query or filter: These match terms starting with the sequence you give
- Wildcards: which allow you to use ? And \* to substitute one or many characters. For example, "e\*search" would match "elasticsearch."

These queries offer lots of flexibility, but they are also more expensive than simple queries, such as a term query. For an exact match, Elasticsearch has to only find that one term in the term dictionary, while fuzzy, prefix and wildcard queries have to find all terms matching the given pattern.

There is also another solution for tolerating typos and other inexact matches: ngrams. Recall from chapter 5, that Ngrams generate tokens from each part of the word. So if you use them at both index and query time, you'll get similar functionality to a fuzzy query, as you can see in figure 10.9:



**#A More edits decrease the score**

**#B More terms match increase the score**

Figure 10.9 Ngrams generate more terms than you need with fuzzy queries, but they match exactly

Which approach is best for performance? As with everything in this chapter, there's a trade-off, and you need to choose where you want to pay the price:

- Fuzzy queries slow your searches down, but your index is the same as with exact matches.
- Ngrams, on the other hand, increase the size of your index. Depending on ngram and term sizes, the index size with ngrams can increase a few times. Also, if you want to change ngram settings, you have to re-index all data, so there's less flexibility. However, searches are typically faster overall with ngrams.

The ngram method is typically better when query latency is important, or when you have lots of concurrent queries to support, so you need each one to take less CPU. Ngrams cause indices to be bigger, so they need to still fit in OS caches or you need fast disks – otherwise performance will degrade because your index is too big.

The fuzzy approach, on the other hand, is better when you need indexing throughput. Or where index size is an issue or you have slow disks. Fuzzy queries also help if you need to change them often, like by adjusting the edit distance, because you can make those changes without re-indexing all data.

#### PREFIX QUERIES AND EDGE NGGRAMS

For inexact matches, you often assume that the beginning is right. For example, a search for "elastic" might be looking for "elasticsearch." Like fuzzy queries, prefix queries are more expensive than regular term queries, because there are more terms to look through.

The alternative could be to use edge ngrams, which were introduced in chapter 5. Figure 10.10 shows edge ngrams and prefix queries side by side:

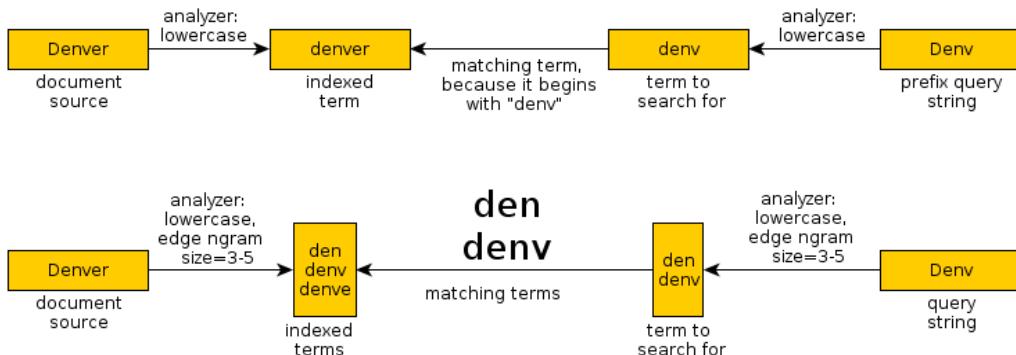


Figure 10.10 Prefix query has to match more terms, but works with a smaller index than edge ngrams

As with the fuzzy queries and ngrams, the trade-off is between flexibility and index size, which are better in the prefix approach, and query latency and CPU usage, which are better for edge ngrams.

### WILDCARDS

A wildcard query where you always put a wildcard in the beginning, such as `elastic*`, is equivalent in terms of functionality to a prefix query. In this case, you have the same alternative of using edge ngrams.

If the wildcard is in the middle, such as `e*search`, there's no real index-time equivalent. You can still use ngrams to match the provided letters e and search, but if you have no control over how wildcards are used, then the wildcard query is your only choice.

If the wildcard is always in the beginning, the wildcard query is typically more expensive than with trailing wildcards, because there's no prefix anymore to hint in which part of the term dictionary to look for matching terms. In this case, the alternative can be to use the reverse token filter in combination with edge ngrams, as you saw in chapter 5. This alternative is illustrated in figure 10.11:

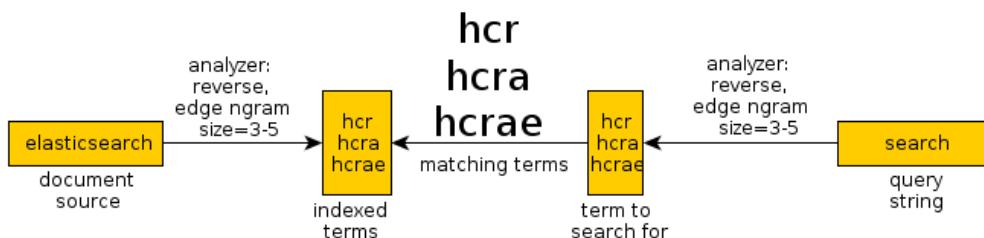


Figure 10.11 You can use the reverse and edge ngram token filters to match suffixes

## PHRASE QUERIES AND SHINGLES

When you need to account for words that are next to each other, you can use the match query with “type” set to “phrase” as you saw in chapter 4. Phrase queries are slower because they have to account not only for the terms, but for their positions in the documents.

**NOTE** Positions are enabled by default for all analyzed fields, because “index\_options” is set to “positions”. If you don’t use phrase queries, only term queries, you can disable indexing positions by setting `index_options` to “freqs”. If you don’t care about scoring at all – for example, when you index application logs and you always sort results by timestamp – you can also skip indexing frequencies by setting `index_options` to “docs”.

The index time alternative to phrase queries is to use shingles. As you saw in chapter 5, shingles are like ngrams, but for terms instead of characters. A text that was tokenized into “Introduction”, “to”, and “Elasticsearch” with a shingle size of 2 would produce the terms “Introduction to” and “to Elasticsearch”.

The resulting functionality is similar to phrase queries, and the performance implications are similar to the ngram situations we discussed earlier: shingles will increase the index size and slow down indexing, in exchange for faster queries.

The two approaches are not exactly equivalent, in the same way wildcards and ngrams aren’t equivalent. With phrase queries, for example, you can specify a `slop`, which allows for other words to appear in your phrase. For example, a slop of 2 would allow a sequence like “buy the best phone” to match a query for “buy phone”. That works because, at search time, Elasticsearch is aware of the position of each term, while shingles are effectively single terms.

The fact that shingles are single terms allow you to use them for matching compound words better. For example, many people still refer to Elasticsearch as “elastic search”, which can be a tricky match. With shingles, you can solve this by using an empty string as a separator, instead of the default white space, as shown in figure 10.12 below.

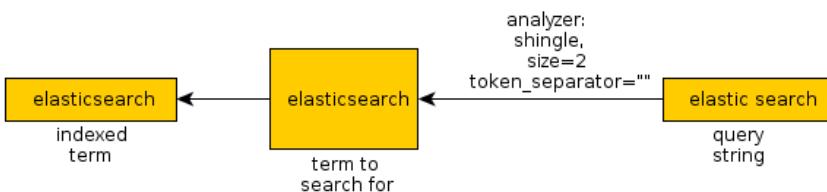


Figure 10.12 Using shingles to match compound words

Using shingles, ngrams, fuzzy and wildcard queries show you that there’s often more than one way to search your documents, but that doesn’t mean those ways are equivalent. Choosing the best one in terms of performance and flexibility depends a lot on your use-case. Next, we’ll be looking deeper at scripts, where you’ll find more of the same: multiple ways to achieve the same result, but each method comes with its own advantages and disadvantages.

### 10.4.2 Tuning scripts or not using them at all

We first introduced scripts in chapter 3, because they can be used for updates. You saw them again in chapter 6, where you used them for sorting. In chapter 7, we used scripts again, this time to build “virtual fields” at search time using “script fields.”

You get a lot of flexibility through scripting, but this flexibility has an important impact on performance. Results of a script are never cached because Elasticsearch doesn't know what's in the script. There can be something external, like a random number, that will make a document match now and not match for the next run. There's no choice for Elasticsearch other than running the same script for all documents involved.

When used, scripts are often the most time and CPU-consuming part of your searches. If you want to speed up your queries, a good starting point is to try skipping scripts altogether. If that's not possible, the general rule is to get as close to native code as you can to improve their performance.

How can you get rid of scripts or optimize them? The answer depends heavily on the exact use-case, but we'll try to cover the best practices here.

#### AVOIDING THE USE OF SCRIPTS

If you're using scripts to generate script fields, like we did in chapter 7, you can do this at index time. In our case, instead of indexing documents directly, and counting the number of group members in a script by looking at the array length, you can count the number of members in your indexing pipeline and add it to a new field. In figure 10.13, we compare the two approaches:

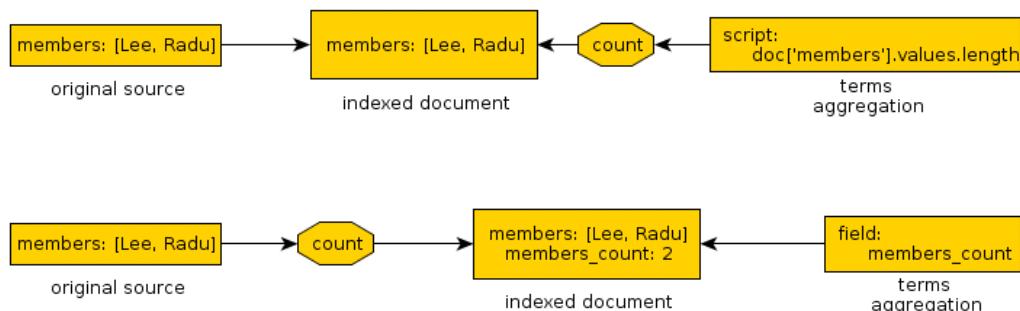


Figure 10.13 Counting members in a script or while indexing

Like with ngrams, this approach of doing the computation at index time works well if query latency is a higher priority than indexing throughput.

Besides precomputing, the general rule for performance optimization for scripting is to reuse as much of Elasticsearch's existing functionality as possible. Before using scripts, can you fulfill the requirements with the function score query that we discussed in chapter 6? The function score query offers lots of ways to manipulate the score. Let's say we want to run a

query for “elasticsearch” events, but we’ll boost the score in the following ways, based on these assumptions:

- Events happening soon are more relevant.

We’ll make events’ scores drop exponentially, the farther in the future they are, up to 60 days.

- Events with more attendees are more popular and more relevant.

We’ll increase the score linearly the more attendees an event has

If we calculate the number of event attendees at index time (let’s name the field “attendees\_count”, we can achieve both criteria without using any script:

```
"function_score": {
  "functions": [
    {
      "linear": {
        "date": {
          "origin": "2013-07-25T18:00",
          "scale": "60d"
        }
      }
    },
    {
      "field_value_factor": {
        "field": "attendees_count"
      }
    }
  ]
}
```

## NATIVE SCRIPTS

If you want the best performance from a script, writing native scripts in Java is the best way to go. Such a native script would be an Elasticsearch plugin, and you can look in appendix B for a complete guide on how to write one.

The main disadvantages with native scripts is that they have to be stored on every node. Changing a script implies updating it on all the nodes of your cluster and restarting them. This won’t be a problem if you don’t have to change your queries often.

To run a native script in your query, you set `lang` to “native” and the name of the script as the `“script”` content. For example, if you have a plugin with a script called `numberOfAttendees` that calculates the number of event attendees on the fly, you can use it in a stats aggregation like this:

```
"aggregations": {
  "attendees_stats": {
    "stats": {
      "script": "numberOfAttendees",
      "lang": "native"
    }
  }
}
```

## LUCENE EXPRESSIONS

If you have to change scripts often, or just want to be prepared to change them without restarting all your cluster, and your scripts are working with numerical fields, Lucene Expressions are likely to be the best choice.

With Lucene Expressions, you provide a JavaScript expression in the script at query time, and Elasticsearch compiles it in native code, making it as quick as a native script. The big limitation is that you only have access to indexed numeric fields.

To use Lucene Expressions, you would set “lang” to “expression” in your script. For example, you might have the number of attendees already, but you know that only half of them usually show up, so you want to calculate some stats based on that number:

```
"aggs": {
  "expected_attendees": {
    "stats": {
      "script": "doc['no_attendees'].value/2",
      "lang": "expression"
    }
  }
}
```

If you have to work with non-numeric or non-indexed fields, and you want to be able to easily change scripts, you can use Groovy – the default language for scripting since Elasticsearch 1.4. Let's see how you can optimize Groovy scripts.

## TERM STATISTICS

If you need to tune the score, you can access Lucene-level term statistics without having to calculate it in the script itself. For example, if you only want to compute the score based on the number of times that term appears in the document: unlike Elasticsearch's defaults, you don't care about the length of the field in that document, nor the number of times that term appears in other documents.

To do that, you can have a script score that only specifies the term frequency (number of times the term appears in the document), as shown in listing 10.9:

### Listing 10.9 Script score that only specifies term frequency

```
curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "function_score": {
      "filter": {
        "term": {
          "title": "elasticsearch"
        }
      },
      "functions": [
        {
          "script_score": {
            "script": "_index[\"title\"][\"elasticsearch\"].tf() + _index[\"description\"][\"elasticsearch\"].tf()", "#C
            "lang": "groovy"
          }
        }
      ]
    }
  }
}'
```

} ] }

#A First, we filter all documents with the term “elasticsearch” in the title field  
#B Then, we compute relevancy by looking at the term’s frequency in the title and description fields  
#C Term frequency is accessed via the tf() function belonging to the term, which belongs to the field

## ACCESSING FIELD DATA

If you need to work with the actual content of a document's fields in a script, one option is to use the `_source` field. For example, you would get the "organizer" field by using `source['organizer']`.

In chapter 3, you saw how you can store individual fields, instead of alongside `_source`. If an individual field is stored, you can access the stored content, too. For example, the same “organizer” field can be retrieved with `fields['organizer']`.

The problem with `_source` and `_fields` is that going to the disk in order to fetch the field content of that particular field is very expensive. Fortunately, this slowness is exactly what made field data necessary when Elasticsearch's built in sorting and aggregations need to access field content. Field data, as we explained in chapter 6, is tuned for random access, so it's best to use it in your scripts, too: it's often orders of magnitude faster than the `_source` or `_fields` equivalent, even if field data isn't already loaded for that field when the script is first run.

To access the "organizer" field via field data, you'd refer to `doc['organizer']`. For example, in listing 10.10 we'll return groups where the organizer isn't a member, so we can ask them why they don't participate to their own groups:

#### **Listing 10.10 Return groups where organizer is not a member**

```
SCRIPT="for ( organizer in doc['organizer'].values ) { \
    found = false; \
    for ( member in doc['members'].values ) { \
        if ( organizer == member ) { found = true; break } \
    }; \
    if ( ! found ) { return 1 } \
}; \
return 0"
curl localhost:9200/get-together/group/_search?pretty -d "{
  \"query\": {
    \"filtered\": {
      \"filter\": {
        \"script\": {
          \"script\": \"$SCRIPT\",
          \"lang\": \"groovy\" #D
        }
      }
    }
  }
}"
```

#A Define a script in a variable, so it doesn't have to be a one-liner. We'll run this script in a filter  
#B Check each organizer against the list of members. If we can't find an organizer, the doc matches  
#C If every organizer can be found, the doc doesn't match  
#D Wrap the defined script in a script filter

There's one caveat for using `doc['organizer']` instead of `_source['organizer']` or the `_fields` equivalent: you will access the terms, not the original field of the document. If an organizer is 'Lee', you will get 'Lee' from `_source` and 'lee' from `doc`. There are trade-offs everywhere, but we assume you got used to them at this point in the chapter.

Next, we'll take a deeper look at how distributed searches work, and how you can use search types to find a good balance between having accurate scores and low-latency searches.

#### **10.4.3            *Trading network trips for less data and better distributed scoring***

Back in chapter 2, you saw how when you hit an Elasticsearch node with a search request, that node distributes the request to all the shards that are involved, aggregates the individual shard replies into one final reply to return to the application.

Let's take this a bit deeper. Let's say that you send a request with the default size of 10, to an index with the default number of 5 shards. Does this mean that the node receiving the search request – let's call it the "coordinating node" – will fetch 10 whole documents from each shard, sort them and return only the top 10 from those 50 documents? If so, it sounds like a waste to transfer 50 potentially big documents over the network, to return only 10. What if there were 10 shards and 100 results? The overhead would explode.

How about returning only the IDs of those 50 documents, and the metadata needed for sorting, to the coordinating node? After sorting, the coordinating node can fetch only the required top 10 documents from the shards. This would reduce the network overhead for most cases, but will involve two round trips.

With Elasticsearch, both options are available by setting the "search\_type" parameter to the search. The naïve implementation of fetching all involved documents is "query\_and\_fetch," while the 2-trip method is called "query\_then\_fetch" which is also the default. A comparison of the two is shown in figure 10.14.

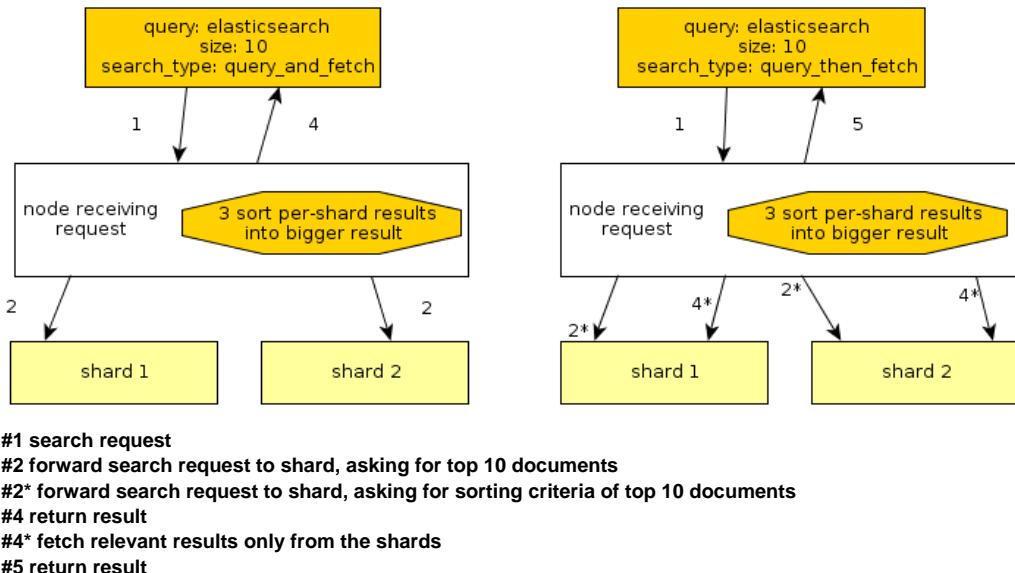


Figure 10.14 Comparison between `query_and_fetch` and `query_then_fetch`

The default `query_then_fetch` (shown on the right of the figure) gets better as you hit more shards, request more documents via the “size” parameter, and as documents get bigger, because it will transfer much less data over the network. `query_and_fetch` (on the left of the figure) becomes faster in the opposite scenario: small documents, small “size” and few shards. For example, to set our get-together searches to `query_and_fetch`, a request may look like this:

```
% URI='localhost:9200/get-together/_search'
% curl "$URI?search_type=query_and_fetch&pretty" -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  }
}'
```

**NOTE** The `query_and_fetch` search type is always used when the query only hits one shard, because the per-shard result is the same as the total result. Hitting one shard happens when the index has one primary shard or when you use custom routing.

### DISTRIBUTED SCORING

By default, scores are calculated per shard, which can lead to inaccuracies. For example, if you search for a term, one of the factors is the *document frequency* (DF), which shows how many

times the term you search for appears in all documents. Those “all documents” are by default “all documents in this shard.” If the DF of a term is significantly different between shards, scoring might not reflect reality. You can see this in figure 10.15, where doc 2 gets a higher score than doc 1, even though doc 1 has more occurrences of “elasticsearch,” because there are less documents with that term in its shard:

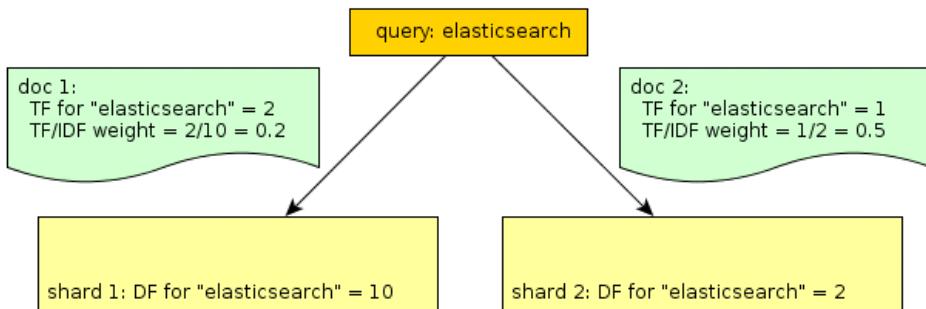


Figure 10.15 Uneven distribution of DF can lead to incorrect ranking

You can imagine that, with a high enough number of documents, DF values would naturally balance across shards, and the default behavior would work just fine. However, if score accuracy is a priority, or if DF is unbalanced for your use-case (for example, if you’re using custom routing), you need a different approach.

That approach could be to change the search type from `query_then_fetch` to `dfs_query_then_fetch`. Or from `query_and_fetch` to `dfs_query_and_fetch`. The DFS part will tell the coordinating node to make an extra call to the shards in order to gather document frequencies of the searched terms. The aggregated frequencies will be used to calculate the score, as you can see in figure 10.16, making our doc 1 and doc 2 ranked correctly:

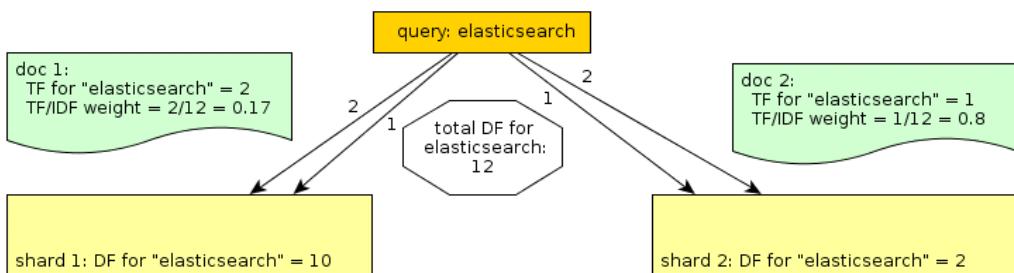


Figure 10.16 DFS search types use an extra network hop to compute global Dfs, that are used for scoring

You probably figured already that DFS queries are slower than their per-shard counterparts, so make sure that you actually get better scores before switching. If you have a low-latency

network, this overhead can be negligible. If, on the other hand, your network isn't fast enough or you have high query concurrency, you may see a significant overhead.

#### **ONLY RETURNING COUNTS**

But what if you don't care about scoring at all, and you don't need the document content, either? For example, when you only need the document count or the aggregations. In such cases, the recommended search type is "count." Count only asks the involved shards for the number of documents that match, and adds up those numbers.

#### **10.4.4 Trading memory for better deep paging**

In chapter 4, you learned that you'd use `size` and `from` to paginate the results of your query. For example, to search for 'elasticsearch' in get-together events, and get the 5<sup>th</sup> page of 100 results, you'd run a request like this:

```
% curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  },
  "from": 400,
  "size": 100
}'
```

This will effectively fetch the top 500 results, sort them, and return only the last 100. You can imagine how inefficient this gets as you go deeper with pages. For example, if you change the mapping and want to re-index all existing data into a new index, you might not have enough memory to sort through all the results in order to return the last pages.

For this kind of scenarios you can use the "scan" search type, like you'll do in listing 10.11, to go through all the get together groups. The initial reply only returns the scroll ID, which uniquely identifies this request and will remember which pages where already returned. To start fetching results, you send a request with that scroll ID. You repeat the same request to fetch the next page until you either have enough data, or there are no more hits to return – in which case the "hits" array is empty.

#### **Listing 10.11 Use scan search type**

```
curl "localhost:9200/get-together/event/_search?pretty&q=elasticsearch\
&search_type=scan\
&scroll=1m\
&size=100"          #A
# reply
{
  "_scroll_id": "c2NhbjsxOzk20jdZdkdQOTJLU1NpNGpxRWh4S0RWUVE7MTt0b3RhF9oaXRzOjc7",
  #C
[...]
  "hits": {
    "total": 7,           #D
    "max_score": 0,
    "hits": []            #D
  }
}
```

```
[...]
curl 'localhost:9200/_search/scroll?scroll=1m&pretty' -d
  'c2NhbjsxOzk20jdZdkdQOTJLU1NpNGpxRWh4S0RWUVE7MTt0b3RhbF9oaXRzOjc7'      #E
# reply
{
  "_scroll_id" : "c2NhbjswOzE7dG90YWxfaGl0cz030w==",                      #F
[...]
  "hits" : {
    "total" : 7,
    "max_score" : 0.0,
    "hits" : [ {                                         #G
      "_index" : "get-together",                         #G
    [...]
curl 'localhost:9200/_search/scroll?scroll=1m&pretty' -d
  'c2NhbjswOzE7dG90YWxfaGl0cz030w=='                  #H

#A Elasticsearch will wait 1 minute for the next request (see below)
#B The size of each “page”
#C You get back a scroll ID that you’ll use in the next request
#D You don’t get any results yet, just their number
#E Fetch the first page with the scroll ID you got previously; specify a timeout for the next request
#F You get another scroll ID, to use for the next request
#G This time you get a page of results
#H Continue getting pages by using the last scroll ID, until the “hits” array is empty again
```

As with other searches, scan searches accept a “size” parameter to control the page size. But this time, the page size is calculated per shard, so actual returned size would be “size” times the number of shards. The timeout given in the “scroll” parameter of each request is renewed each time you get a new page, that’s why you can have a different timeout with every new request.

**NOTE** It may be tempting to have big timeouts, so that you’re sure a scroll doesn’t expire while you’re processing it. The problem is, if a scroll is active and not used, it wastes resources: some JVM heap to remember the current page, and disk space taken by Lucene segments which can’t be deleted by merges until the scroll is completed or expired.

The scan search type always returns results in the order it encounters them in the index, regardless of the sort criteria. If you need both deep paging and sorting, you can add a “scroll” parameter to a regular search request. Sending a GET request to the scroll ID will get the next page of results. This time, “size” works accurately, regardless of the number of shards. You also get the first page of results with the first request, just like you get with regular searches:

```
% curl 'localhost:9200/get-together/event/_search?pretty&scroll=1m' -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  }
}'
```

From a performance perspective, adding “scroll” to a regular search is more expensive than using the “scan” search type, because there’s more information to keep in memory when results are sorted. That said, deep paging is much more efficient than by default, because Elasticsearch doesn’t have to sort all previous pages to return the current page.

Scrolling is only useful when you know in advance that you want to do deep paging, it’s not recommended for when you only need a few pages of results. Like with everything in this section and in the rest of this chapter, you pay a price for every performance improvement. In the case of scrolling, that price is to keep information about the current search in memory until the scroll expires or you have no more hits.

## 10.5 Summary

- Use the Bulk API to combine multiple index, update or delete operations in the same request
- To combine multiple get or search requests, you can use the multi get or multi search API respectively
- A flush operation commits in-memory Lucene segments to disk when the index buffer size is full, the transaction log is too large, or too much time has passed since the last flush
- A refresh makes new segments – flushed or not – available for searching. During heavy indexing, it’s best to lower the refresh rate or disable refresh altogether
- The merge policy can be tuned for more or less segments. Fewer segments make searches faster, but merges take more CPU time. More segments make indexing faster by spending less time on merging, but searches will be slower
- An optimize operation forces a merge, which works well for static indices that get lots of searches
- Store throttling may limit indexing performance if merges fall behind. Increase or remove the limits if you have fast I/O
- Combine filters that use bitsets in a bool filter and filters that don’t in and/or/not filters
- Cache counts and aggregations in the shard query cache if you have static indices
- Monitor JVM heap and leave enough headroom so you don’t experience heavy garbage collection or out of memory errors, but leave some RAM for OS caches, too
- Use index warmers if the first query is too slow, and you don’t mind slower indexing
- If you have room for bigger indices, using ngrams and shingles instead of fuzzy, wildcard or phrase queries should make your searches faster
- You can often avoid using scripts by creating new fields with needed data in your documents before indexing them
- Try to use Lucene expressions, term statistics and field data in your scripts whenever they fit
- If your scripts don’t need to change often, have a look at appendix B to learn how to write a native script in an Elasticsearch plugin
- Use DFS search types if you don’t have balance document frequencies between shards

- Use the count search type if you don't need any hits and the scan search type if you need many

# Appendix A

## *Working with Geo-Spatial Data*

Geo-spatial data is all about making your search application location-aware. For example, to search for events that are close to you, or restaurants in a certain area, or to see which park's area intersects with the area of the city center, you'd work with geo-spatial data.

We'll call events and restaurants in this context *points*, as they are essentially a point on the map. We'll put areas, such as a country, or a rectangle that you draw on a map, under the generic umbrella of *shapes*. Geo-spatial search is all about points, shapes, and various relations between them:

- *Distance between a point and another point*: If where you are is a point, and swimming pools are other points, search for the closest swimming pools. Or filter only pools that are reasonably close to you.
- *A shape containing a point*: If we select an area on the map – like the area where you work – you can filter only restaurants that are in that area.
- *A shape overlapping with another shape*: For example, if you want to search for parks in the city center.

This appendix will show you how to search and sort documents in Elasticsearch, based on their distance from a reference point on the map. You'll also learn how to search for points that fall into a rectangle, and how to search shapes that intersect with a certain area you define on the map.

### **A.1 Points and distances between them**

To search for points, you have to index them first. Elasticsearch has a *Geo Point* type especially for that. You can see an example on how to use it in the code samples, by looking at `mapping.json`

**NOTE** If you didn't use them already, the code samples for this book, along with instructions on how to use them, can be found at <https://github.com/dakrone/elasticsearch-in-action>

Each event has a location field, which is an object that includes the geolocation field as a `geo_point` type:

```
"geolocation" : { "type" : "geo_point"}
```

With the geo point type defined in your mapping, you can index points by giving the latitude and longitude as you can see in `populate.sh`:

```
"geolocation": "39.748477,-104.998852"
```

**TIP** You can also provide the latitude and longitude as properties, as an array or as a geohash. This doesn't change the way points are indexed, it's just for your convenience, in case you have a preferred way. More details can be found at <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/mapping-geo-point-type.html>

Having geo points indexed as part of your event documents (from the data set used throughout the book), enables you to add distance criteria to your searches in the following ways:

- Sort results by the distance from a given point. This would make the event closest to you appear first.
- Filter results by distance. This lets you display only events that are within 100 kilometers from you.
- Count results by distance. This allows you to create buckets of ranges. For example, get the number of events within 100km from you, and the number of events from 100km to 200km and so on.

## A.2 Add distance to your sort criteria

Using the get-together example we've used in the main chapters of the book, let's say your coordinates are "40,-105" and you need to find the event about Elasticsearch closest to you. To do that, you need to add a sort criteria called `_geo_distance`, where you specify your current location, as shown in Listing A.1.

### Listing A.1 Sorting events by distance

```
curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  }
}'
```

```

"sort" : [
  {
    "_geo_distance" : { #B
      "location.geolocation" : "40,-105", #C
      "order" : "asc", #D
      "unit" : "km" #E
    }
  }
],
}

#A The query, looking for “elasticsearch” in the title
#B The _geo_distance sort criteria containing:
#C Your current location
#D Ascending order will give closest events first
#E Each hit will have a “sort” value, representing the distance from your location in kilometers

```

#### SORTING BY DISTANCE AND OTHER CRITERIA AT THE SAME TIME BY USING SCRIPTS

A search like the one above is useful when distance is your only criteria. If you want to include other criteria in the equation, like the document's score, you can use a script. That script can generate a final score based on the initial score from your query, plus the distance from your point of interest.

Listing A.2 shows such a query. We'll use the `function_score` query, which will first run the same `match` query as listing A1, looking for events about Elasticsearch. Next, the script will take the initial score, and divide it by the distance. This way, an event will score higher, the closer it is to you. To refer to the distance from a point, you'll use the `arcDistanceInKm()` function, where you'll specify where you are. For example `doc['location.geolocation'].arcDistanceInKm(40.0, -105.0)`.

#### Listing A.2 Take distance into account when calculating the score

```

curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "function_score": {
      "query": { #A
        "match": { #A
          "title": "elasticsearch" #A
        }
      },
      "script_score": { #B
        "script": "if (doc['location.geolocation'].empty){ #C
          _score
        } else {
          _score*40000/doc['location.geolocation'].arcDistanceInKm(40.0, -105.0) #D
        }
      }
    }
  }
}

```

#A The query looking for “elasticsearch” will return a score  
#B The `script_score` will calculate the final score based on the script you run  
#C If there's no “geolocation” field, we leave the score untouched

#D Otherwise, we'll divide the score by the distance, to get higher scores for lower distances. And we'll multiply everything by 40000, to bump the score of all events with geo information past the ones without

You might be tempted to think that such scripts bring the best of both worlds: relevance from your query, and the geo-spatial dimension. While the `function_score` query is very powerful indeed, there are two aspects to be aware of.

- *Tuning the score*: How important is distance compared to relevancy? What should you do with documents that don't have geo information? These are questions that are tricky to answer.
- *Performance*: Running a script like the one in listing A2 is expensive in terms of speed, especially when you have lots of documents.

If these two aspects are bothering you, then you may want to search your events as usual, and only filter those who are within a certain distance.

### A.3 Filter based on distance

Let's say you're looking for events within a certain range from where you are, like in Figure A.1.

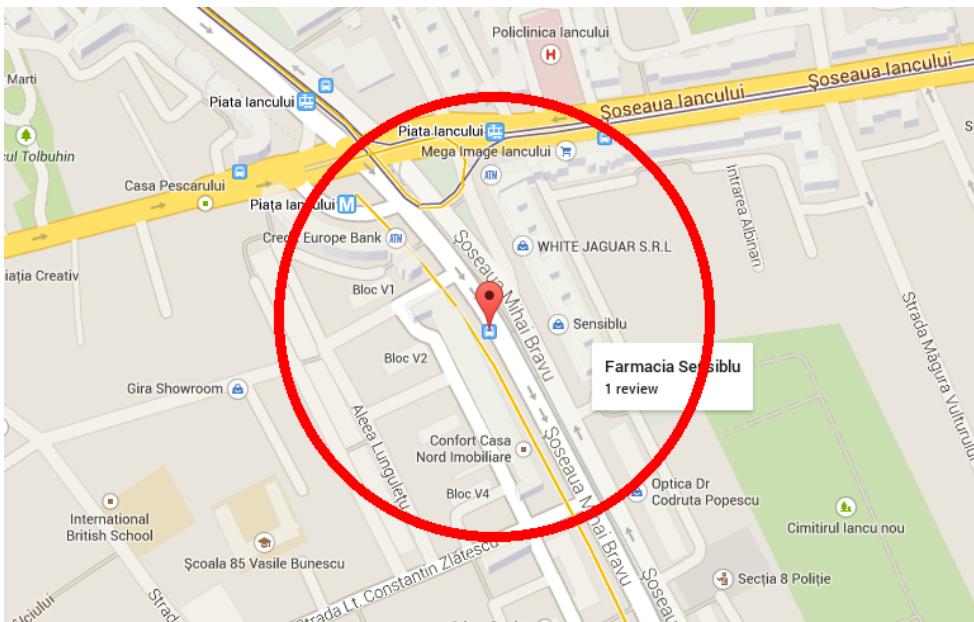


Figure . A.1 You can filter only points that fall in a certain range from a specified location

To filter such events, you would use the *Geo Distance Filter*. The parameters it needs are your reference location and the limiting distance, as shown below:

```
% curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {
          "distance": "50km",
          "location.geolocation": "40.0,-105.0"
        }
      }
    }
  }
}'
```

In this default mode, Elasticsearch will calculate the distance from 40.0,-105.0 to each event's "geolocation", and return only those who are under 50km. The way that the distance is calculated can be set via the `distance_type` parameter, which will go right next to the `distance` parameter. You have three options:

- `sloppy_arc`(default): It calculates the distance between the two points by doing a faster approximation of an arc of a circle. For most situations, this is a good option.
- `arc`: It actually calculates the arc of a circle, making it slower but more precise than `sloppy_arc`. Note that you don't get 100% precision here, either, because the Earth is not perfectly round. Still, if you need precision, this is the best option.
- `plane`: This is the fastest but less precise implementation, because it assumes the surface between the two points is plane. This option works well when you have many documents and the distance limit is fairly small.

Performance optimization doesn't end up with distance algorithms. There's another parameter to the Geo Distance Filter called `optimize_bbox`. "bbox" stands for "bounding box", which is a rectangle that you define on a map, which contains all the points and areas of interest.

Using `optimize_bbox` will first check if events match a square that contains the circle describing the distance range. Only if they match, Elasticsearch filters further by calculating the distance.

If you ask yourself if the bounding box optimization is actually worth it, then you'll be happy to know that:

- Yes: For most cases it is. Because verifying if a point belongs to a bounding box is much faster than calculating the distance and comparing it to your limit
- *It's configurable*: You can set `optimize_bbox` to `none` and check if your query times are faster or slower. The default value is `memory` and you can set it to `indexed`

Curious what the difference between `memory` and `indexed` is? We'll discuss it in the beginning of the next section. If you're not curious, and you don't want to obsess on performance improvements, sticking with the default should be good enough for most cases.

## Distance range: filter and aggregations

There is also a *Geo Distance Range Filter*. It allows you, for example, to search for events between 50 and 100 kilometers from where you are. Besides its “from” and “to” distance options, it accepts the same parameters as the Geo Distance Filter. More details about the Geo Distance Range filter can be found here: <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-geo-distance-range-filter.html>

Users will probably search for events further from their point of reference because the ones they found close weren't satisfying. For example, if the events' dates are too far. In such situations, it might come handy for the user to see in advance how many events are, say, within 50km, how many are between 50 and 100, how many between 100 and 200 and so on.

For this use-case, the *Geo Distance Range Aggregation* will come in handy. You'll specify a reference point and the distance ranges you need, and it will return how many events it found for each distance range. More information about the *Geo Distance Range Aggregation* can be found here: <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/search-aggregations-bucket-geodistance-aggregation.html>

When you index a point, one way to search for it is by calculating the distance to another point, which is what we've discussed so far. The second way to search for it is in relation to a shape, which we'll look at next.

## A.4 Does a point belong to a shape?

Shapes, especially rectangles, are easy to draw interactively on a map, as you can see in figure A.2. It's also faster to search for points in a shape than to calculate distances, because searching in a shape only requires comparing the coordinates of the point with the coordinates of the shape's corners.

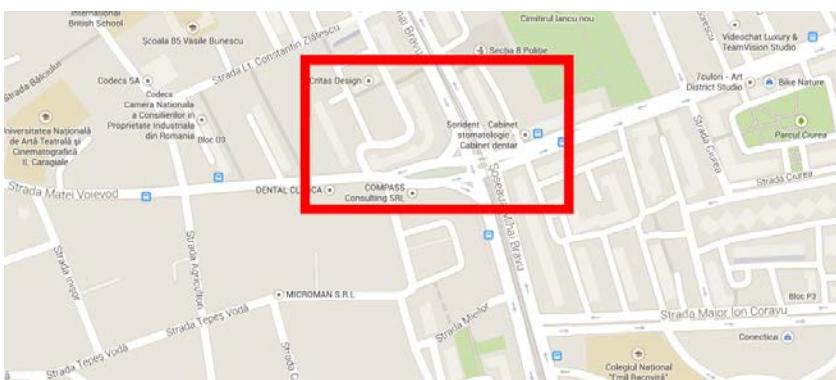


Figure A.2 You can filter points based on whether they fall within a rectangle on the map

There are three types of shapes on the map that you can match points to. Or events, if you're thinking of the get-together example we ran through the main chapters:

- *Bounding boxes (rectangles)*: These are quite fast, and give you the flexibility to draw any rectangle.
- *Polygons*: Allow you to draw a more precise shape, but it's difficult to ask a user to draw a polygon and searching is slower the more complex the polygon is.
- *Geohashes (squares defined by a hash)*: This is the least flexible, because hashes are fixed. But, as you'll see later, it's typically the fastest implementation of the three.

#### A.4.1 Bounding Box Filter

To search if a point falls within a rectangle, you'd use the Bounding Box filter. This is useful if your application allows users to click on a point on the map to define a corner of the rectangle, and then to click again to define the opposite corner. The result could be a rectangle like the one from figure A.2.

To run the Bounding Box filter, you'll specify the coordinates for the top-left and the bottom-right points that describe the rectangle:

```
% curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "location.geolocation": {
            "top_left": "40, -106",
            "bottom_right": "38, -103"
          }
        }
      }
    }
  }
}'
```

The default implementation of the Bounding Box Filter is to load the points' coordinates in memory and compare them with those provided for the bounding box. This is the equivalent of setting the `type` option under `geo_bounding_box` to `memory`.

Alternatively, you can set `type` to `indexed`, and Elasticsearch will do the same comparison using Range Filters – like the ones you learned in chapter 4. For this implementation to work, you'll need to index the point's latitude and longitude in their own fields – which isn't enabled by default.

To enable indexing latitude and longitude separately, you'll have to set `lat_lon` to `true` in your mapping, making your `geolocation` field definition look like this:

```
"geolocation" : { "type" : "geo_point", "lat_lon": true }
```

**NOTE** If you make the change above to `mapping.json` from the code samples, you'll need to run `populate.sh` again to re-index the sample dataset and have your changes take effect.

The indexed implementation is faster, but indexing latitude and longitude will make your index bigger. Also, if you have more geo points per document – like an array of points for a restaurant franchise – the indexed implementation won't work.

### Polygon filter

If you want to search for points matching a more complex shape than a rectangle, you can use the Geo Polygon filter. It allows you to enter the array of points that describe the polygon. More details about the Geo Polygon filter can be found here: <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-geo-polygon-filter.html>

### A.4.2 Geohash Cell Filter

The last point-matches-shape method you can use is by matching geohash cells. They work as suggested in figure A3: the Earth is divided into 32 rectangles/cells (divides the latitude in 4, and longitude in 8). Each cell is identified by an alpha-numeric character: its hash. Then, each rectangle – for example, “d” – can be further divided into 32 rectangles of its own, generating “d0”, “d1” and so on. The process can be repeated virtually forever, generating smaller and smaller rectangles with longer and longer hash values.



Figure A3 World divided in 32 letter-coded cells. Each cell into 32 cells and so on, making longer hashes

Because of the way geohash cells are defined, each point on the map belongs to an infinite number of such geohash cells, like "d", "d0", "d0b" and so on. Given such a cell, Elasticsearch can tell you which points match with the Geohash Cell Filter:

```
% curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "filtered": {
      "filter": {
        "geohash_cell": {
          "location.geolocation": "9xj"
        }
      }
    }
  }
}'
```

#### HOW GEOHASH FILTERING WORKS

Even though a geohash cell is a rectangle, this filter works differently than the Bounding Box filter. First, geo points have to get indexed with a geohash that describes them. For example "9xj6". Then, you also have to index all the ngrams of that hash, like "9", "9x", "9xj" and "9xj6" - which described all the "parent" cells. When you run the filter, the hash from the query is matched against the hashes indexed for that point, making a Geohash Cell filter similar in implementation to the Term filter you saw in chapter 4, which is very fast.

To enable indexing the geohash in your geo point, you have to set `geohash` to `true` in the mapping. To index that hashes' parents (ngrams), you have to set `geohash_prefix` to `true` as well.

**TIP** Because a cell will never be able to perfectly describe a point, you have to choose how precise (or big) that rectangle needs to be. The default setting for `precision` is 12 which creates hashes like `9xj64sswpkdq` with an accuracy of a few centimeters. Because you'll also be indexing all the parents, you may want to trade some precision for index size and search performance.

Understanding geohash cells is important even if you're not going to use the Geohash Cell filter. Because in Elasticsearch, geohashes are the default way of representing shapes. We'll explain how shapes use geohashes in the next section.

## A.5 Shape intersections

Elasticsearch can index documents with shapes, like polygons like the area of a park, and filter documents based on whether parks overlap other shapes, such as the city center. It does this, by default, through the geohashes that we've discussed in the previous section. The process is described in figure A.4: each shape is approximated (we'll discuss precision later) to a group of rectangles defined by geohashes. When you search, Elasticsearch will easily find out if at least one geohash of a shape overlaps a geohash of another shape.

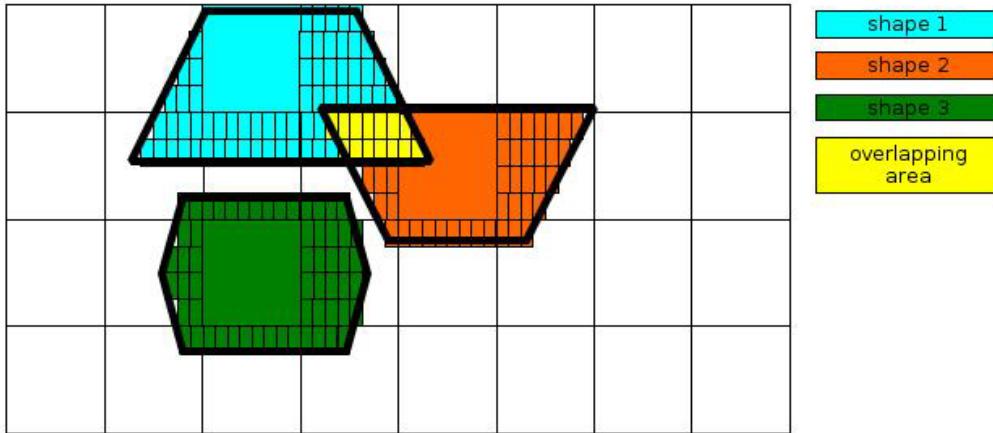


Figure A.4 Shapes represented in geohashes. Searching for shapes matching shape 1 will return shape 2

### INDEXING SHAPES

Let's say we have a shape of a park that's a polygon made out of four corners. To index it, you'd first have to define a mapping of that shape field – let's call it `area` – of type `geo_shape`. With the mapping in place, you can start indexing documents: the `area` field of each document would have to mention that the shape's `type` is `polygon`, and the array of coordinates for that polygon, as shown in the next listing.

#### **Listing A.3 Indexing a shape**

```
curl -XPUT localhost:9200/geo          #A
curl -XPUT localhost:9200/geo/park/_mapping -d '{
  "properties": {
    "area": { "type": "geo_shape" }
  }
}'                                     #B
curl -XPUT localhost:9200/geo/park/1 -d '{
  "area": {
    "type": "polygon",
    "coordinates": [
      [[45, 30], [46, 30], [45, 31], [46, 32]]           #C
    ]
  }
}'                                     #D
#A Creating a new index to index the park areas
#B Put the mapping for parks. geo-shapes will be indexed in the "area" field
#C A polygon is indexed in the "area" field
#D Coordinates for the polygon
#A This first array describes the outer boundary. Optionally, other arrays can be added to define "holes" in the polygon
```

**NOTE** Polygons are not the only shape type supported by Elasticsearch. You can have multiple polygons in a single shape (type: `multipolygon`). There's also the point and multipoint type, one or more chained lines (`linestring`), and rectangles (`envelope`).

The amount of space a shape occupies in your index depends heavily on how you index it. Because geohashes can only approximate most shapes, it's up to you to define how small those geohash rectangles can be. The smaller they are, the better the resolution/approximation, but your index size increases, because smaller geohash cells have longer strings and – more importantly – more “parent ngrams” to index as well. Depending on where you are in this trade-off, you'll specify a precision parameter in your mapping, which defaults to 50m. This means the worse-case scenario is to get an error of 50m.

#### FILTERING OVERLAPPING SHAPES

With your “park” documents indexed, let's say you have another four-cornered shape that represents your city center. To see which parks are (at least partly) in the city center, you'd use the GeoShape filter. You can provide the shape definition of your city center in the filter, like it is in the following listing.

#### Listing A.4 GeoShape filter example

```
curl localhost:9200/geo/park/_search?pretty -d '{
  "query": {
    "filtered": {
      "filter": {
        "geo_shape": {
          "area": {
            "shape": {
              "type": "polygon",
              "coordinates": [
                [[45, 30.5], [46, 30.5], [45, 31.5], [46, 32.5]] #C
              ] #C
            } #B
          } #A
        }
      }
    }
  }
}'
```

#A field to be searched on  
#B you're going to provide a shape in the query  
#C shape provided in the same way as when you index

If you followed listing A.3, you should see that the indexed shape matches. Change the query to something like `[[95, 30.5], [96, 30.5], [95, 31.5], [96, 32.5]]`, and the query won't return any hits.