Joel Abrahamsson

ElasticSearch

Quick Start

ElasticSearch Quick Start

An introduction to ElasticSearch in tutorial form.

Joel Abrahamsson

This book is for sale at http://leanpub.com/elasticsearch-quick-start

This version was published on 2015-04-06

Contents

About the book and feedback

My goal with this book is to get you as a developer or user of ElasticSearch started quickly. As such you won't find much theory or anything about configuring ElasticSearch for production use in this book. What you will find though is tutorials focusing on the basics as well as common use cases.

In its current state the book is "finished". However, it's possible that I'll add more examples in the future and/or re-write some parts based on feedback.

If you have feedback don't hesitate to drop me an e-mail at mail@joelabrahamsson.com. I'm especially interested in any questions you may feel unanswered after reading the book or things you feel could have been explained in better ways. Note though that if you have specific questions not directly related to the book, I may not be the best person to ask, and I may not have the time to help you. So, for such questions you're better off asking the ElasticSearch community[1] for help.

Finally, before we get started, if you find that you like the book, please spread the word about it. Every copy sold will motivate me to improve it and keep it up to date.

Thanks, Joel

[1] https://www.elastic.co/community/

Getting started

ElasticSearch is an open source project, under the Apache License version 2, built on top of Lucene and Java. The source code is located on GitHub at https://github.com/elastic/elasticsearch. Documentation, download links and other useful information can be found at https://www.elastic.co/products/elasticsearch.

Behind the ElasticSearch product is a company named Elastic. It's website is located at https://www.elastic.co/. Elastic is the core developers of the open source project and owns the copyright for it. Additionally the company provides training, support and a number of commercial add-ons for ElasticSearch.

In other words, ElasticSearch is free to use but there is a company that supports its development. This company also provides services and add-ons which are not free. It's entirely up to you whether you pay anything in conjunction with using ElasticSearch or not. If you don't you will still have access to the full ElasticSearch product, but if you do pay money you'll be able to get training, support and/or nice add-ons.

Apart from ElasticSearch there are a number of other projects within the same ecosystem. Two of those are LogStash and Kibana. LogStash can be used to store logs from various sources in ElasticSearch. Kibana provides functionality to visualize data stored in ElasticSearch in dashboards. Together ElasticSearch, LogStash and Kibana is referred to as the "ELK stack".

Installing ElasticSearch

ElasticSearch is a Java application built for Java 7 or higher. Therefore the first step in setting up ElasticSearch is to ensure that you have Java installed and the JAVA_HOME environment variable correctly configured.

To check that you have a compatible version of Java installed open up a terminal window and type `java -version`. The output should look something like this:

Running java -version in a console where Java 8 is installed.

```
$ java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

Once you have made sure that you have Java 7 or higher installed and the `java` executable in your path ensure that you have the JAVA_HOME environment variable configured by typing `echo $JAVA_HOME` into your terminal. The output should look something like this:

Verifying that the JAVA_HOME environment variable is set on a computer where it indeed is set.

```
$ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home
```

If you don't have Java installed and/or the JAVA_HOME environment variable set up fix that prior to proceeding. Java can be downloaded from https://java.com/en/download/. We won't go into details about setting up Java here as there is plenty of documentation online.

Next, with Java correctly set up, you're ready to download and install ElasticSearch. This can be done using various package managers such as Homebrew on OS X. However, it can also be done manually by downloading from www.elastic.co which we'll cover here.

ElasticSearch can be downloaded from https://www.elastic.co/downloads/elasticsearch. From there grab the ZIP package for ElasticSearch and unzip it to some suitable location on your computer.

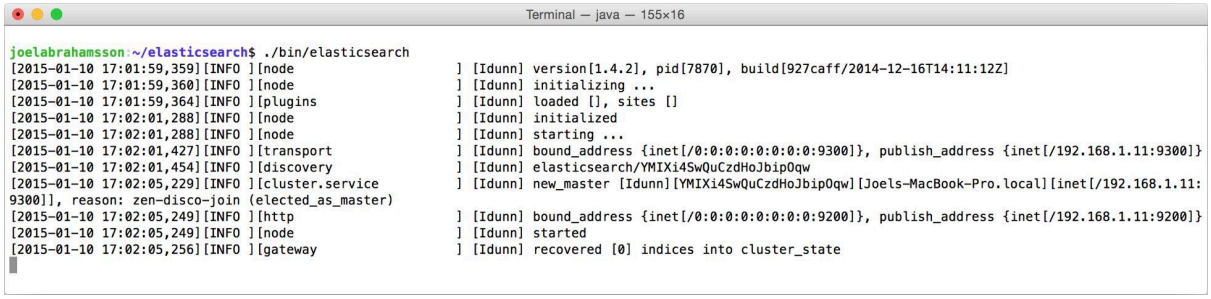
Take a look inside the unzipped folder. You should find find a few text files and some directories.

Inspecting the contents of the elasticsearch folder.

```
~/elasticsearch$ ls -p
LICENSE.txt        NOTICE.txt        README.textile        bin/                config/                li
```

The "lib" directory contains the compiled JAR files that make up ElasticSearch and the "config" directory contains configuration files. For running ElasticSearch the most interesting directory though is the "bin" directory. In there you'll find a shell script named "elasticsearch" and a Windows batch file named "elasticsearch.bat". These provide the recommended ways for starting ElasticSearch on *nix and Windows environments respectively.

If you're on Linux or OS X execute the "bin/elasticsearch" shell script to start ElasticSearch. If you're on Windows instead execute the "bin/elasticsearch.bat" batch file. The output should look something like this:



As you can see from the timestamps in the console output above it took a few seconds but ElasticSearch is now up and running. To verify this open up a browser and make a request to http://localhost:9200. The response should be in the form of JSON, looking something like this:

Example response from ElasticSearch when making a request to it's / endpoint.

```
{
  "status" : 200,
  "name" : "Tyger Tiger",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "1.5.0",
    "build_hash" : "927caff6f05403e936c20bf4529f144f0c89fd8c",
    "build_timestamp" : "2015-03-23T14:30:58Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}
```

Given that your browser can connect to http://localhost:9200 and you see a response similar to the one above ElasticSearch is running fine. The exact response isn't very interesting. ElasticSearch's "/" endpoint, which is what we've requested, responds with some basic information about the cluster, such as which version of ElasticSearch that it's running.

To shut down ElasticSearch simply press CTRL+C. To start it again execute the same command as you previously used.

Marvel and Sense

You use HTTP to communicate with ElasticSearch and as such no specific tool is required. When communicating with ElasticSearch from within an application you can use whatever HTTP libraries that are available or one of ElasticSearch's client libraries. For "manually" communicating with ElasticSearch, which you typically do when playing with it or in order to execute ad-hoc queries, you can use any HTTP client, such as cURL, a browser or any number of "REST clients".

However, there are a number of tools that can aid you beyond what generic HTTP clients provides. One such tool is Sense. Sense is a "JSON aware developer console to ElasticSearch" that offers auto completion and nice formatting of requests and responses.

These days[2] Sense is shipped as a part of Marvel, a commercial plug-in for ElasticSearch. Marvel provides management and monitoring dashboards for an ElasticSearch cluster. And, Sense.

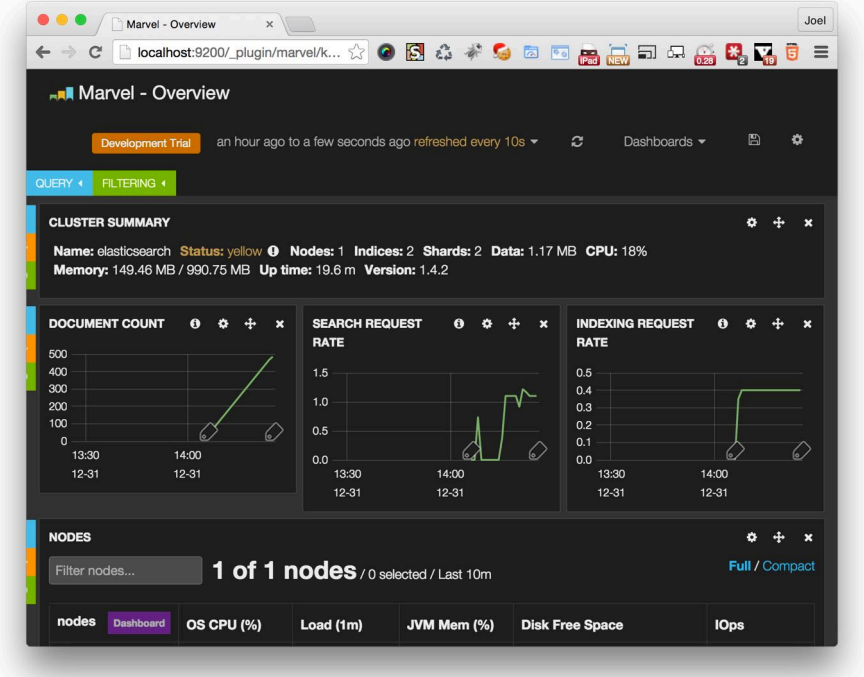
While Marvel requires a paid for license for production use it's free for development use. I recommend you to install it now as it's a good tool to get to know and as Sense will make it more convenient to play with ElasticSearch. To do so you can use the "bin/plugin" tool. From the ElasticSearch home directory run `bin/plugin -i elasticsearch/marvel/latest`. The output should look something like this:

[2] In "the early days" Sense was a Chrome plug-in but these days it's a part of Marvel. However, there has been some efforts to bring it back as a Chrome plug-in. If you're interested in that search the Chrome web store for Sense.

Installing Marvel.

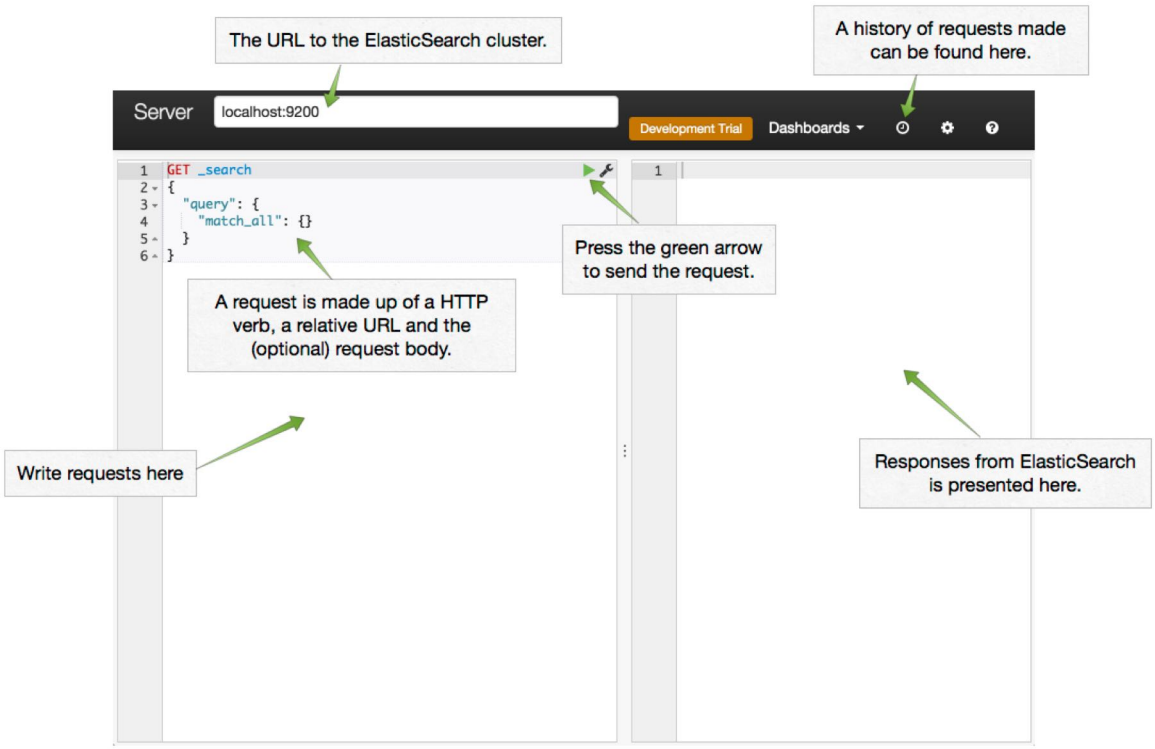
```
~/elasticsearch$ bin/plugin -i elasticsearch/marvel/latest
-> Installing elasticsearch/marvel/latest...
Trying http://download.elasticsearch.org/elasticsearch/marvel/marvel-latest.zip...
Downloading .........DONE
Installed elasticsearch/marvel/latest into /Users/joelabrahamsson/elasticsearch/plugins\
/marvel
```

Once the installation is complete restart (or start) ElasticSearch. You can now navigate to http://localhost:9200/_plugin/marvel/ where Marvel resides.



The default Marvel dashboard.

In order to access Sense use the "Dashboards" drop down menu in the top right part of Marvel and click on Sense. Alternatively you can navigate directly to Sense by directing your browser to http://localhost:9200/_plugin/marvel/sense/.



The Sense dashboard.

Curl syntax

We use HTTP requests to talk to ElasticSearch. A HTTP requests is made up of several components such as the URL to make the request to, HTTP verbs (GET, POST etc) and headers. In order to succinctly and consistently describe HTTP requests the ElasticSearch documentation uses cURL command line syntax. This is also the standard practice to describe requests made to ElasticSearch within the user community and the standard that we'll use throughout this book.

An example HTTP request using cURL syntax looks like this:

A simple search request using cURL.

```
curl -XPOST "http://localhost:9200/_search" -d'
{
  "query": {
    "match_all": {}
  }
}'
```

The above snippet, when executed in a console, runs the curl program with three arguments. The first argument, `-XPOST`, means that the request that cURL makes should use the POST HTTP verb. The second argument, `"http://localhost:9200/_search"` is the URL that the request should be made to. The final argument, `-d'{...}'` uses the `-d` flag which instructs cURL to send what follows the flag as the HTTP POST data.

Whenever you see a request formatted using cURL syntax you can either:

- Copy it and execute it in a console (given that you have cURL installed).
- Read it and translate it into whatever HTTP client that you are using.
- Paste it into the left part of Sense.

When using the last option, pasting cURL formatted requests into Sense, Sense will recognize the cURL syntax and automatically transform it to a request formatted the Sense way. Sense also offers functionality for doing the opposite. When you have a request in Sense you can click the wrench icon to bring up a dialog offering an option to "Copy as cURL".

The Copy as cURL dialog in Sense.

Hello world

Now that you have ElasticSearch up and running let's end this chapter with a quick example. We won't go into any details about what we're doing here as we'll cover that in the coming chapters. For now, just run the below HTTP requests in Sense and take them at face value. Or, if you prefer to know what you're doing, skip to the next chapter.

First, let's index a simple document:

Indexing a simple document using cURL.

```
curl -XPOST "http://localhost:9200/my-first-index/message" -d'
{
  "text": "Hello world!"
}'
```

Now, let's see if we can find it by searching for "hello":

A search request searching for the word 'hello'.

```
curl -XPOST "http://localhost:9200/_search" -d'
{
  "query": {
    "query_string": {
      "query": "hello"
    }
  }
}'
```

The response from ElasticSearch to the second HTTP request should look like the one below, containing a single hit.

Example response from ElasticSearch to the above request.

```
{
  "took": 12,
  "timed_out": false,
  "_shards": {
    "total": 12,
    "successful": 12,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0.19178301,
    "hits": [
      {
        "_index": "my-first-index",
        "_type": "message",
        "_id": "AUqiBnvdK4Rpq0ZV4-Wp",
        "_score": 0.19178301,
        "_source": {
          "text": "Hello world!"
        }
      }
    ]
  }
}
```

Curl syntax is programming language agnostic making it perfect to show HTTP interactions in a way that is both succinct and independent of any programming language. However, in the real world, except when debugging, we usually interact with ElasticSearch from our programming language of choice. Let's look at a couple of examples of how the above requests could be implemented in actual applications.

Node.JS example

For Node.JS we use the official JavaScript client which can be installed in a Node.JS application using `npm install elasticsearch`. A simple application that indexes a single document and then proceeds to search for it, printing the search results to the console, looks like this:

A simple implementation of the Hello World example in Node.JS.

```
var elasticsearch = require('elasticsearch');

var client = new elasticsearch.Client({
  host: 'localhost:9200'
});

//An object that we'll index
var theMessage = {
  text: "Hello world!"
};

//Indexing the above object
client.index({
  index: "my-first-index",
  type: "message",
  body: theMessage
}).then(function() {
  setTimeout(search, 1100);
});

//Searching and printing the results
function search() {
  client.search({
    index: "my-first-index",
    type: "message",
    body: {
      query: {
        query_string: {
          query: "hello"
        }
      }
    }
  }).then(function(response) {
    console.log("Number of hits: "
      + response.hits.total);
    response.hits.hits.forEach(function(hit) {
      console.log(hit._source.text);
    })
  });
}
```

Note that we add a waiting period between indexing and searching. 1.1 second to be exact. We do this because an indexed document won't immediately be searchable after indexing. We have to wait for the index to be refreshed which by default happens every second. It's possible to require ElasticSearch to immediately refresh the index when indexing a document but that's bad performance wise and therefore we opt to wait a little.

.NET example

In the Node.JS example we (naturally) used JavaScript and the official ElasticSearch client which more or less maps directly to ElasticSearch's HTTP/JSON API. Therefore the code for our Node.JS application looked quite similar to the original cURL based example. Now, let's look how we can interact with ElasticSearch from a strongly typed language, C#, using a client library that introduces more abstractions, NEST.

In order to implement the Hello World example in C# we start by creating a new console application to which we add the NEST ElasticSearch client using NuGet (`PM > Install-Package NEST`). Next we create a class which we'll index and search for instances of.

A C# class representing a message.

```
namespace HelloElasticSearch
{
  public class Message
  {
    public Message(string text)
    {
      Text = text;
    }

    public string Text { get; private set; }
  }
}
```

The entry point of our application which implements the Hello World example by indexing a message and then searching for it looks like this:

Basic example of indexing and searching using C# and the NEST client library.

```
using System;
using System.Threading;
using Nest;

namespace HelloElasticSearch
{
  class Program
  {
    static void Main(string[] args)
    {
      //Create a client that will talk to our ES cluster
      var node = new Uri("http://localhost:9200");
      var settings = new ConnectionSettings(
          node,
          defaultIndex: "my-first-index"
      );
      var client = new ElasticClient(settings);
```

```
      //Creating and indexing a message object
      var theMessage = new Message("Hello world!");
      client.Index(theMessage);

      //Waiting for the index to be refreshed
      Thread.Sleep(1100);

      //Search for messages and print the results
      var response = client.Search<Message>(
        body => body.Query(
          q => q.QueryString(
            qs => qs.Query("hello"))));

      Console.WriteLine("Number of hits: " + response.Total);
      foreach (var hit in response.Hits)
      {
        Console.WriteLine(hit.Source.Text);
      }
    }
  }
}
```

Basic CRUD

In order to use ElasticSearch for anything useful, such as searching, the first step is to populate an index with some data. A process known as indexing. In this chapter we'll look at how to do that as well as how to read, update and delete indexed documents. In the process we'll see that while ElasticSearch is a search engine it's also possible to use it as a general purpose data store.

Indexing

In ElasticSearch indexing corresponds to both "Create" and "Update" in CRUD - if we index a document with a given type and ID that doesn't already exists it's inserted. If a document with the same type and ID already exists it's overwritten.

What is a document? Under the covers a document in ElasticSearch is a Lucene document. However, from our perspectives as users of ElasticSearch a document is a JSON object. As such a document can can have fields in the form of JSON properties. Such properties can be values such as strings or numbers, but they can also be other JSON objects.

In order to create a document we make a PUT request to the REST API to a URL made up of the index name, type name and ID. That is: `http://localhost:9200/<index>/<type>/[<id>]` and include a JSON object as the PUT data.

Index and type are required while the id part is optional. If we don't specify an ID ElasticSearch will generate one for us. However, if we don't specify an id we should use POST instead of PUT. The index name is arbitrary. If there isn't an index with that name on the server already one will be created using default configuration.

As for the type name it too is arbitrary. It serves several purposes, including:

- Each type has its own ID space.
- Different types can have different mappings ("schema" that defines how properties/fields should be indexed).
- Although it's possible, and common, to search over multiple types, it's easy to search only for one or more specific type(s).

Indexes, types and documents

If you're new to ElasticSearch the terminology and concepts can sometimes be confusing. When I first encountered ES I had primarily worked with relational databases before. While they aren't the same I sometimes like to think of indexes in ES as databases when working with MySQL or MS SQL Server. When doing so types in ElasticSearch corresponds to tables and documents to individual rows.

Let's index something! We can put just about anything into our index as long as it can be represented as a single JSON object. For the sake of having something to work with we'll be indexing, and later searching for, movies. Here's a classic one:

Sample JSON object

```
{
    "title": "The Godfather",
    "director": "Francis Ford Coppola",
    "year": 1972
}
```

To index the above JSON object we decide on an index name ("movies"), a type name ("movie") and an ID ("1") and make a request following the pattern described above with the JSON object in the body.

A request that indexes the sample JSON object as a document of type 'movie' in an index named 'movies'.

```
curl -XPUT "http://localhost:9200/movies/movie/1" -d'
{
    "title": "The Godfather",
    "director": "Francis Ford Coppola",
    "year": 1972
}'
```

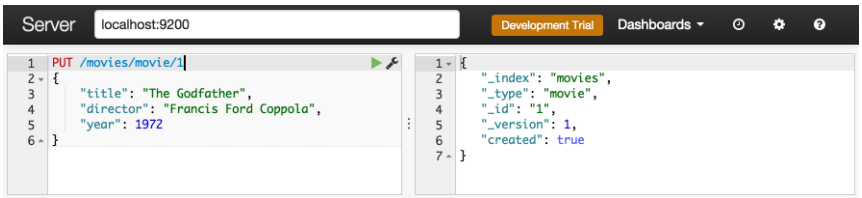
Don't we have to create the index first?

When indexing a document ElasticSearch will automatically create the index (in the example above named "movies") if it doesn't already exists. When doing so it will create the index with some default settings and mappings. We'll see later how to explicitly create indexes and how to specify settings when doing so.

Execute the above request using cURL or paste it into sense and hit the green arrow to run it. After doing so, given that ElasticSearch is running, you should see a response looking like this:

Response from ElasticSearch to the indexing request.

```
{
  "_index": "movies",
  "_type": "movie",
  "_id": "1",
  "_version": 1,
  "created": true
}
```



The request for, and result of, indexing the movie in Sense.

As you see, the response from ElasticSearch is also a JSON object. It's properties describe the result of the operation. The first three properties simply echo the information that we specified in the URL that we made the request to. While this can be convenient in some cases it may seem redundant. However, remember that the ID part of the URL is optional and if we don't specify an ID the `_id` property will be generated for us and its value may then be of great interest to us.

The fourth property, `_version`, tells us that this is the first version of this document (the document with type "movie" with ID "1") in the index. This is also confirmed by the fifth property, "created", whose value is true.

Now that we've got a movie in our index let's look at how we can update it, adding a list of genres to it. In order to do that we simply index it again using the same ID. In other words, we make the exact same indexing request as as before but with an extended JSON object containing genres.

Indexing request with the same URL as before but with an updated JSON payload.

```
curl -XPUT "http://localhost:9200/movies/movie/1" -d'
{
  "title": "The Godfather",
  "director": "Francis Ford Coppola",
  "year": 1972,
  "genres": ["Crime", "Drama"]
}'
```

This time the response from ElasticSearch looks like this:

The response after performing the updated indexing request.

```
{
  "_index": "movies",
  "_type": "movie",
  "_id": "1",
  "_version": 2,
  "created": false
}
```

Not surprisingly the first three properties are the same as before. However, the `_version` property now reflects that the document has been updated as it now has 2 a version number. The `created` property is also different, now having the value `false`. This tells us that the document already existed and therefore wasn't created from scratch.

It may seem that the `created` property is redundant. Wouldn't it be enough to inspect the `_version` property to see if its value is greater than one? In many cases that would work. However, if we were to delete the document the version number wouldn't be reset meaning that if we later indexed a document with the same ID the version number would be greater than one.

So, what's the purpose of the `_version` property then? While it can be used to track how many times a document has been modified it's primary purpose is to allow for optimistic concurrency control.

If we supply a version in indexing requests ElasticSearch will then only overwrite the document if the supplied version is the same as for the document in the index. To try this out add a `version` query string parameter to the URL of the request with "1" as value, making it look like this:

Indexing request with a 'version' query string parameter.

```
curl -XPUT "http://localhost:9200/movies/movie/1?version=1" -d'
{
  "title": "The Godfather"
}'
```

Now the response from ElasticSearch is different. This time it contains an `error` property with a message explaining that the indexing didn't happen due to a version conflict.

Response from ElasticSearch indicating a version conflict.

```
{
  "error": "VersionConflictEngineException[[movies][2] [movie][1]: version conflict, cu\
rrent [2], provided [1]]",
  "status": 409
}
```

Getting by ID

We've seen how to indexing documents, both new ones and existing ones, and have looked at how ElasticSearch responds to such requests. However, we haven't actually confirmed that the documents exists, only that ES tells us so.

So, how do we retrieve a document from an ElasticSearch index? Of course we could search for it. However that's overkill if we only want to retrieve a single document with a known ID. A simpler and faster approach is be to retrieve it by ID.

In order to do that we make a GET request to the same URL as when we indexed it, only this time the ID part of the URL is mandatory. In other words, in order to retrieve a document by ID from ElasticSearch we make a GET request to `http://localhost:9200/<index>/<type>/<id>`. Let's try it with our movie using the following request:

GET request for retrieving the movie with ID 1.

```
curl -XGET "http://localhost:9200/movies/movie/1"
```

You should see a result like this:

```
{
  "_index": "movies",
  "_type": "movie",
  "_id": "1",
  "_version": 2,
  "found": true,
  "_source": {
    "title": "The Godfather",
    "director": "Francis Ford Coppola",
    "year": 1972,
    "genres": [
      "Crime",
      "Drama"
    ]
  }
}
```

As you can see the result object contains similar meta data as we saw when indexing, such as index, type and version. Last but not least it has a property named `_source` which contains the actual document body. There's not much more to say about GET as it's pretty straightforward. Let's move on to the final CRUD operation.

Deleting documents

In order to remove a single document from the index by ID we again use the same URL as for indexing and retrieving it, only this time we change the HTTP verb to DELETE.

Request for deleting the movie with ID 1.

```
curl -XDELETE "http://localhost:9200/movies/movie/1"
```

The response object contains some of the usual suspects in terms of meta data, along with a property named "_found" indicating that the document was indeed found and that the operation was successful.

Response to the DELETE request.

```
{
  "found": true,
  "_index": "movies",
  "_type": "movie",
  "_id": "1",
  "_version": 3
}
```

If we, after executing the DELETE request, switch back to GET we can verify that the document has indeed been deleted:

Response when making the the DELETE request a second time.

```
{
  "_index": "movies",
  "_type": "movie",
  "_id": "1",
  "found": false
}
```

JSON objects in documents

In the examples in this chapter, as well as throughout most of this book, we use fairly simple JSON objects as documents in order to keep the examples short. However, it's worth pointing out that ElasticSearch supports nested JSON objects. For instance, instead of a string we could have represented the director property in our movie as a complex object, like this:

```
{
  "title": "The Godfather",
  "director": {
    "givenName": "Francis Ford",
    "surName": "Coppola"
  },
  "year": 1972
}
```

Or, like this:

```
{
  "title": "The Godfather",
  "director": {
    "givenNames": ["Francis", "Ford"],
    "surNames": ["Coppola"]
  },
  "year": 1972
}
```

Or, like this:

```
curl -XPUT "http://localhost:9200/movies/movie/1" -d'
{
  "title": "The Godfather",
  "director": {
    "givenName": "Francis Ford",
    "surName": "Coppola",
    "awards": [{
      "name": "Oscar",
      "type": "Director",
      "year": 1974,
      "movie": "The Godfather Part II"
    }]
  },
  "year": 1972
}'
```

Searching

So, we've covered the basics of working with data in an ElasticSearch index and it's time to move on to more exciting things - searching. However, considering the last thing we did was to delete the only document we had from our index we'll first need some sample data. Below is a number of indexing requests that we'll use.

Indexing request for sample data.

```
curl -XPUT "http://localhost:9200/movies/movie/1" -d'
{
  "title": "The Godfather",
  "director": "Francis Ford Coppola",
  "year": 1972,
  "genres": ["Crime", "Drama"]
}'

curl -XPUT "http://localhost:9200/movies/movie/2" -d'
{
  "title": "To Kill a Mockingbird",
  "director": "Robert Mulligan",
  "year": 1962,
  "genres": ["Crime", "Drama", "Mystery"]
}'

curl -XPUT "http://localhost:9200/movies/movie/3" -d'
{
  "title": "Lawrence of Arabia",
  "director": "David Lean",
  "year": 1962,
  "genres": ["Adventure", "Biography", "Drama"]
}'

curl -XPUT "http://localhost:9200/movies/movie/4" -d'
{
  "title": "Apocalypse Now",
  "director": "Francis Ford Coppola",
  "year": 1979,
  "genres": ["Drama", "War"]
}'

curl -XPUT "http://localhost:9200/movies/movie/5" -d'
{
  "title": "Kill Bill: Vol. 1",
  "director": "Quentin Tarantino",
```

```
  "year": 2003,
  "genres": ["Action", "Crime", "Thriller"]
}'

curl -XPUT "http://localhost:9200/movies/movie/6" -d'
{
  "title": "The Assassination of Jesse James by the Coward Robert Ford",
  "director": "Andrew Dominik",
  "year": 2007,
  "genres": ["Biography", "Crime", "Drama"]
}'
```

It's worth pointing out that ElasticSearch has and endpoint (_bulk) for indexing multiple documents with a single request. We'll cover that in a later chapter. For now we keep it simple and use six separate requests.

The _search endpoint

Now that we have some movies in our index let's see if we can find them again by searching. In order to search with ElasticSearch we use the _search endpoint, optionally with an index and type. That is, by making requests to an URL following this pattern: `<index>/<type>/_search` where index and type are both optional. In other words, in order to search for our movies we can make GET or POST requests to either of the following URLs:

- `http://localhost:9200/_search` - Search across all indexes and all types.
- `http://localhost:9200/movies/_search` - Search across all types in the movies index.
- `http://localhost:9200/movies/movie/_search` - Search explicitly for documents of type movie within the movies index.

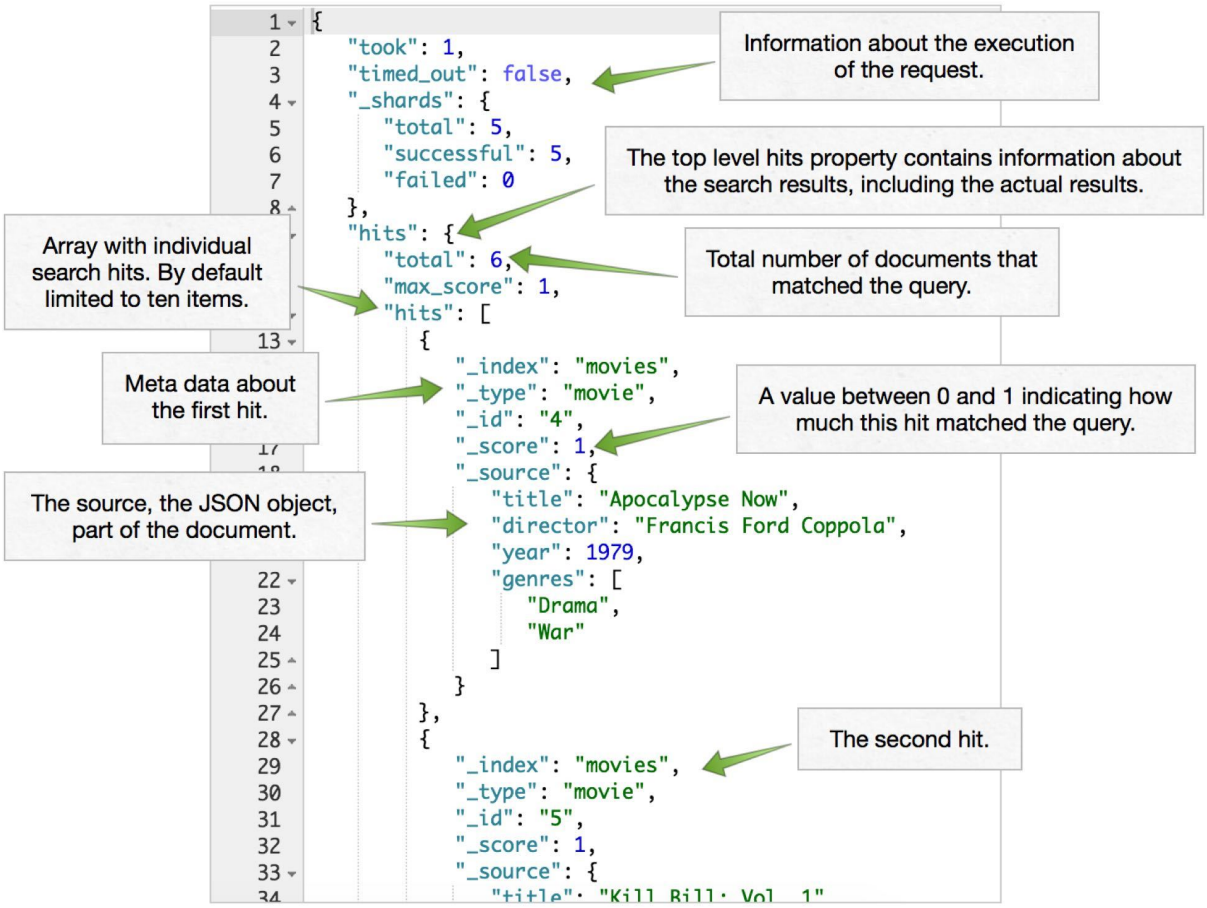
If you use the first URL, searching across all indexes, and if you have Marvel installed you'll probably get other hits from an index other than the "movies" index. This is because Marvel by default indexes various metrics to the same cluster that it's running on.

Let's give searching a try by making a GET request to the second URL above.

A search request limited to the 'movies' index but without any other criteria.

```
curl -XGET "http://localhost:9200/movies/_search"
```

The result should look something like in the image below.



The search request body and ElasticSearch's query DSL

If we simply make a GET request to the `_search` endpoint like we did above we'll get all of the movies back. In order to make a more useful search request we also need to supply a request body with a query. The request body should be a JSON object which, among other things, can contain a property named "query" in which we can use ElasticSearch's query DSL.

```
{
  "query": {
    //Query DSL here
  }
}
```

The query DSL is ElasticSearch's own domain specific language based on JSON in which queries and filters can be expressed. Think of it like ElasticSearch's equivalent of SQL for a relational database. Here's part of how ElasticSearch's own documentation explains it:

> *Think of the Query DSL as an AST of queries. Certain queries can contain other queries (like the bool query), other can contain filters (like the constant_score), and*

some can contain both a query and a filter (like the filtered). Each of those can contain any query of the list of queries or any filter from the list of filters, resulting in the ability to build quite complex (and interesting) queries.

Basic free text search

The query DSL features a long list of query types that we can use[3]. For "ordinary" free text search we'll most likely want to use one called "query string query".

A query string query is an advanced query with a lot of different options that ElasticSearch will parse and transform into a tree of simpler queries. Still, it can be very easy to use if we ignore all of its optional parameters and simply feed it a string to search for. Let's try a search for the word "Robert":

A search request, this time using POST, with a query_string query in the POST data.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "query_string": {
      "query": "Robert"
    }
  }
}'
```

Given that you have indexed six movies as listed at the start of this chapter the above request should result in two hits. The movies "The Assassination of Jesse James by the Coward Robert Ford" and "To Kill a Mockingbird". The reason for this is that a `query_string` query by default searches in a special field named "_all"[4]. This field is automatically created during indexing and is, by default, made up text extracted from each of the document's fields. As such, both documents contains the word "Robert" in their _all fields. The first from its title property and the second from its director property.

Relevance scoring

If you run the above search request the two hits in the result will both have a `_score` property. This property indicates how relevant the hit is with regards to the query and it's what ElasticSearch sorts the hits by unless we've told it otherwise.

The value of the `_score` property is always between 0 and 1 where 1 is the most relevant. Note though that this value is normalized, meaning that it's useful for comparing the difference in relevancy between two hits in the same result, but not between two different search results.

We won't delve deeper into relevance scoring in this book. However, it's an interesting topic and I recommend you to read more in the documentation[5].

[3] For a full list of queries see the documentation: http://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-queries.html.

[4] For more information about the _all field see the documentation: http://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-all-field.html.

[5] http://www.elastic.co/guide/en/elasticsearch/guide/current/controlling-relevance.html

Fine tuning query string queries

In the previous example we used a `query_string` query with no other properties than `query` in which we specified the search term. As mentioned before the query string query has a number of settings that we can specify and if we don't it will use sensible default values.

One such setting is called "fields" and that can be used to specify a list of fields to search in. Let's use that to only search in the `title` field.

The search request extended with a fields property.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "query_string": {
      "query": "Robert",
      "fields": ["title"]
    }
  }
}'
```

Run this request and this time you'll only get a single hit, The Assassination of Jesse James by the Coward Robert Ford, which is the only movie that contains the word "Ford" in the title.

The `query_string` query type also supports a wide range of other settings. For instance, take a look at the below request:

Searching for three words.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "query_string": {
      "query": "Francis Ford Coppola"
    }
  }
}'
```

The above request will result in three hits. Two of them will be the movies directed by Francis Ford Coppola. The third will be The Assassination of Jesse James by the Coward Robert Ford as that contains the word "Ford". This is because the `query` property will be parsed as "Francis OR Ford OR Coppola". Should we want to change this behavior we can do so by setting the `default_operator` of the query to "AND", like this:

Using the default_operator parameter to customize how the query parameter is parsed.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "query_string": {
      "query": "Francis Ford Coppola",
      "default_operator": "AND"
    }
  }
}'
```

The result to the above request will be limited to documents whose `_all` field contains all of the words "Francis", "Ford" and "Coppola". However, they don't have to come in that order. Should we want that, simply wrap the query in double quotes ("query": "\"Francis Ford Coppola"\").

See the documentation[6] for more parameters supported by `query_string` queries.

Highlighting

A common feature when building free text search functionality is highlighting; making matching words within the hits stand out visually to the user. In order to retrieve highlights with ElasticSearch we can add an additional property to the search request body named `highlight`. The value of this property should be a JSON object that describes which fields we want highlights from as well as, optionally, details about how the highlights should work. Below is a an example request that includes a `highlight` property.

Searching for movies with Robert in the title and requesting highlights for the title field.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "query_string": {
      "query": "Robert",
      "fields": ["title"]
    }
  },
  "highlight": {
    "fields": {
      "title": {}
    }
  }
}'
```

The response to the above request looks something like this:

[6] http://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html

The response to a search request including highlights.

```
{
  "took": 28,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0.3125,
    "hits": [
      {
        "_index": "movies",
        "_type": "movie",
        "_id": "6",
        "_score": 0.3125,
        "_source": {
          //Omitted for brevity
        },
        "highlight": {
          "title": [
            "The Assassination of Jesse James by the Coward <em>Robert</em> Ford"
          ]
        }
      }
    ]
  }
}
```

There are two interesting things to note in the above response. First of all the hit object includes a property we haven't seen before, the `highlight` property. Unsurprisingly this contains one (in this case) or more highlighted extracts from the title field. Second, within the returned highlight the word Robert has been enclosed in `em` tags.

Highlighting can be customized in a number of ways. We can choose how many fragments should be extracted from each field and how long they should be, customize the tags that enclose highlighted terms, use different highlighting implementations and even use a separate query for highlighting. To learn more about highlighting see the documentation[7].

Filtering

We've covered a couple of simple free text search queries above. Let's look at another one where we search for "crime" without explicitly specifying fields:

[7] http://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-highlighting.html

A search request for the word 'crime'.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "query_string": {
      "query": "crime"
    }
  }
}'
```

As we have four movies in our index containing the word "crime" in the _all field (from the category field) we get four hits for the above query. Now, imagine that we want to limit these hits to movies released in 1962. In order to do that we need to apply a filter requiring the "year" field to equal 1962. To add such a filter we modify our search request body so that our current top level query, the query string query, is wrapped in query of a different type, a `filtered` query:

The original query wrapped in a 'filtered' query.

```
{
  "query": {
    "filtered": {
      "query": {
        "query_string": {
          "query": "crime"
        }
      },
      "filter": {
        //Filter to apply to the query
      }
    }
  }
}
```

A filtered query is a query that has two properties, query and filter. When executed it filters the result of the query using the filter. To finalize the query we'll need to add a filter requiring the year field to have value 1962. ElasticSearch's query DSL has a wide range of filters to choose from. For this simple case where a certain field should match a specific value a `term` filter will work well.

Adding a 'term' filter to the filtered query's filter property.

```
"filter": {
  "term": { "year": 1962 }
}
```

The complete search request now looks like this:

A search request searching for the word 'crime' limited to documents with year == 1962.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "query_string": {
          "query": "crime"
        }
      },
      "filter": {
        "term": { "year": 1962 }
      }
    }
  }
}'
```

Perform the above request and you should get a single hit, To Kill a Mockingbird which has "Crime" in its genre property and whose year property has 1962 as value.

Queries and filters

The difference between queries and filters can be confusing at first. Especially if we try to relate the terminology to query languages for other data sources such as SQL. In ElasticSearch (and Lucene) a query is something that determines whether documents match a given criteria *and* how much it does so, producing a score between 0 and 1 that indicates how much it does so. For instance, if we tell ElasticSearch to index "The quick brown fox" it will by default index that as ["the", "quick", "brown", "fox"]. If we search using a query that looks for the term "brown" the indexed string will match, but the score won't be 1 as only one of the words matches.

A filter on the other hand skips scoring. It only determines whether a document matches the filter or not. Using filters we can limit the search result to documents that match a given criteria without effecting the score. Also, as ElasticSearch doesn't have to care about scoring for filters they are faster and can be cached. Therefore a general rule of thumb is to always use filters unless you need the results to be sorted by relevancy according to the query.

Filtering without a query

In the above example we limit the results of a `query_string` query using a filter. But what if all we want to do is apply a filter? For instance, if we want to find all movies from 1962. In such cases we still use the `query` property in the search request body, which expects a query. In other words, we can't just add a filter, we need to wrap it in some sort of query. One solution for doing this is to modify our current search request, replacing the query string query in the filtered query with a `match_all` query which is a query that simply matches everything. Like this:

Search request that only filters by year == 1962.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "term": { "year": 1962 }
      }
    }
  }
}'
```

Another, shorter, option is to use a constant score query:

Again filtering by year == 1962 but this time using a different type of query that can wrap a filter.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "constant_score": {
      "filter": {
        "term": { "year": 1962 }
      }
    }
  }
}'
```

Try both of these requests out and you'll see the same result for both, two movies from 1962.

Using only filters, or rather a query comprised only of filters, is very common outside of use cases related to free text search. In such cases we usually want to sort the results in other ways.

Sorting

In the previous example we searched for movies from 1962. In this case all hits have the same score. They both match the query equally. Unless told otherwise ElasticSearch will then resort to sorting the results by their IDs. In cases such as these, or when we don't want to sort by the score, we can tell ElasticSearch how to sort the results by adding a `sort` property to the request body.

The `sort` property's value should be an array with JSON objects. The first object describes the primary way the results should be sorted, the second how results with the same value for the primary way should be sorted and so on.

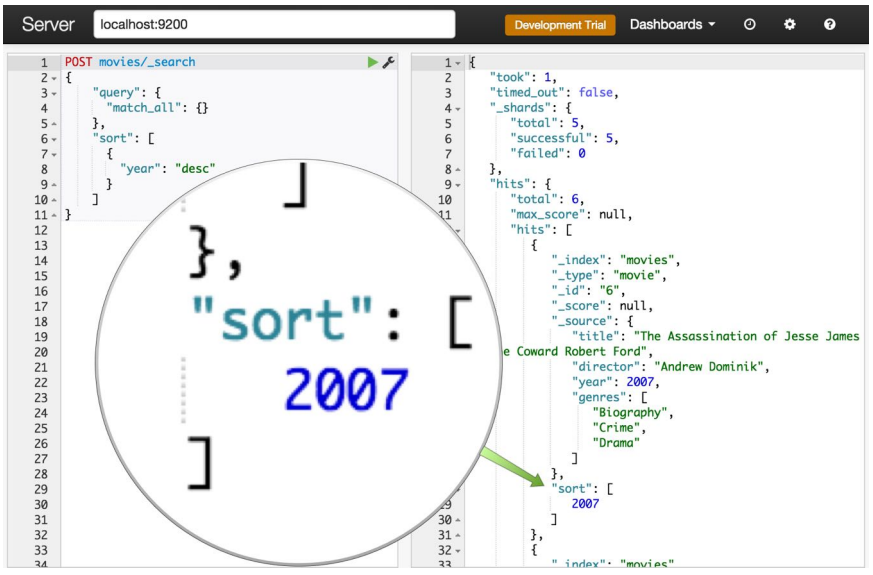
In its simplest form an object in the `sort` array is an object with a single property whose name matches the field to sort by and whose value is the order by which to sort, "asc" for ascending or "desc" for descending.

So, a request searching for all movies and sorting them by newest first can look like this:

A search request that searches for everything in the 'movies' index and sorts the result based on the 'year' property in descending order.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "year": "desc"
    }
  ]
}'
```

When there's a `sort` property in the search request body ElasticSearch won't only apply the sorting rule(s) found in it. It will also extend the hit objects with a property explaining what value(s) where used for the hit during sorting.



Search results in Sense when using a sort property in the request body.

There is more to sorting in ElasticSearch. For instance we can re-introduce sorting by score by adding the string "_score" as one of the values in the `sort` array and we can tell ElasticSearch

how it should handle missing values. When you need to know more about sorting check out the documentation[8].

Pagination

If a search request results in more than ten hits ElasticSearch will, by default, only return the first ten hits. To override that default value in order to retrieve more or fewer hits we can add a `size` parameter to the search request body. For instance, the below request finds all movies ordered by year and returns the first two:

A search request that searches for everything in the 'movies' index and sorts the result based on the 'year' property in descending order.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "year": "desc"
    }
  ],
  "size": 2
}'
```

It's also possible to exclude the N first hits, which you typically would do when building pagination functionality. To do so use the `from` parameter and inspect the `total` property in the response from ElasticSearch to know when to stop paging.

The same request as the previous one, only this time the first two hits are excluded and hit 3 and 4 is returned.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "year": "desc"
    }
  ],
  "size": 2,
  "from": 2
}'
```

[8] http://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-sort.html

Don't set the `size` parameter to some huge number or you'll get an exception back from ElasticSearch. While it's typically fine to retrieve tens, hundreds and even thousands of results with a single request you shouldn't ask for millions of results using the `size` parameter.

If you truly need to fetch a huge number of results, or even all of them, you can either iterate over them using `size + from` or, better yet, you can use the scroll API which you'll find in the documentation[9].

Retrieving only parts of documents

So far in all of the examples that we've looked at the entire JSON object that we've indexed has been included as the value of the `_source` property in each search results hit. In many cases that's not needed and only results in unused data being sent over the wire.

There are a couple of ways to control what is returned for each hit. One is by adding a `_source` parameter to the search request body. This parameter can have values of different kinds. For instance, we can give it the value `false` to not include a `_source` property at all in the hits:

A search request that only retrieves a single hit and instructs ES not to include the _source property in the hits.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "size": 1,
  "_source": false
}'
```

The response to the above request will look something like the one below. Note that the only thing that is returned for each hit is the score and the meta data.

A response without the _source property.

```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 6,
    "max_score": 1,
    "hits": [
      {
```

[9] http://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-scroll.html

```
        "_index": "movies",
        "_type": "movie",
        "_id": "4",
        "_score": 1
      }
    ]
  }
}
```

Another way to use the `_source` parameter is to give it a string with a single field name. This will result in the `_source` property in the hits to contain only that field.

A search request instructing ElasticSearch to only include the 'title' property in the _source.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "size": 1,
  "_source": "title"
}'
```

The response to the request, which includes the 'title' property in the _source.

```
"hits": [
  {
    "_index": "movies",
    "_type": "movie",
    "_id": "4",
    "_score": 1,
    "_source": {
      "title": "Apocalypse Now"
    }
  }
]
```

Should we want multiple fields we can instead use an array:

A search request instructing ElasticSearch to only include the 'title' and 'director' properties in the _source.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "size": 1,
  "_source": ["title", "director"]
}'
```

It's also possible to include, and exclude, fields whose names match one or more patterns. For examples of that see the documentation[10].

[10] http://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-source-filtering.html

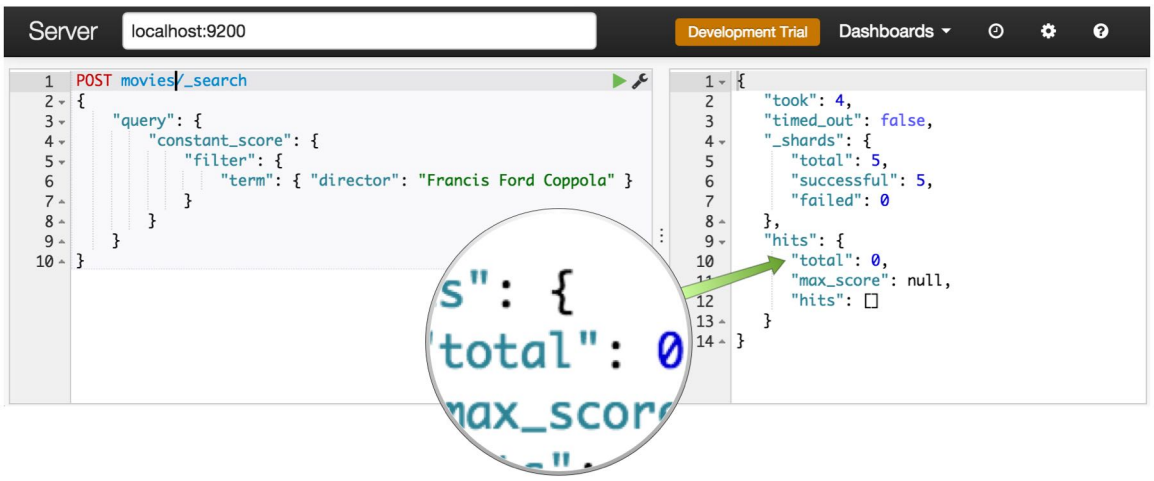
Mapping

In the previous chapter we looked at free text search using a `query_string` query and some examples of filtering with the help of some movies as sample data. Let's take a look at another search requests for movies, this time filtering by the director's name:

A search request filtering using the author's full name.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
  "query": {
    "constant_score": {
      "filter": {
        "term": { "director": "Francis Ford Coppola" }
      }
    }
  }
}'
```

As we have two movies directed by Francis Ford Coppola in our index it doesn't seem too far fetched that this request should result in two hits, right? That's not the case however.



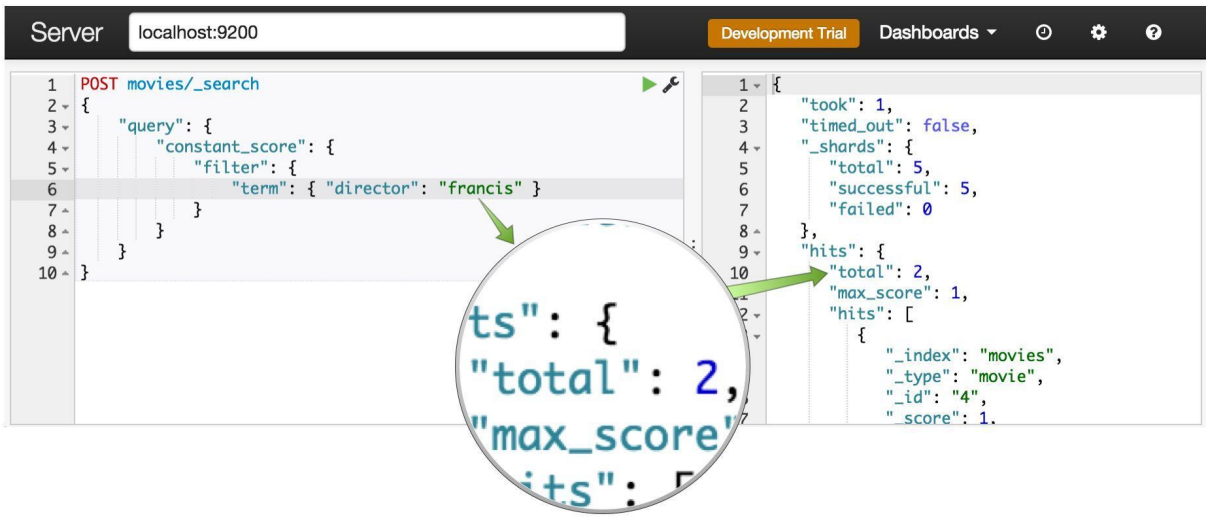
The search request and its result, without a single hit, in Sense.

What's going on here? We've obviously indexed two movies with "Francis Ford Coppola" as director and that's what we see in search results as well. Well, while ElasticSearch has a JSON object with that data that it returns to us in search results in the form of the `_source` property that's not what it has in its index.

When we index a document with ElasticSearch it (simplified) does two things: it stores the original data untouched for later retrieval in the form of `_source` and it indexes each JSON

property into one or more fields in a Lucene index. During the indexing it processes each field according to how the field is mapped. If it isn't mapped default mappings depending on the fields type (string, number etc) is used.

As we haven't supplied any mappings for our index ElasticSearch uses the default mappings for strings for the director field. This means that in the index the director fields value isn't "Francis Ford Coppola". Instead it's something more like ["francis", "ford", "coppola"]. We can verify that by modifying our filter to instead match "francis" (or "ford" or "coppola"):



The search request after modifying it so that the term filter now looks for "francis" instead of the full name and its result with two hits.

So, what to do if we want to filter by the exact name of the director? We modify how the field is mapped. There are a number of ways to add mappings to ElasticSearch, through a configuration file, as part of a HTTP request that creates and index and by calling the `_mapping` endpoint.

Using the last approach we could in theory fix the above issue by adding a mapping for the "director" field instructing ElasticSearch not to analyze (tokenize etc.) the field at all when indexing it, like this:

Explicitly mapping the director field as not_analyzed, meaning it will be indexed exactly as it is (not tokenized etc).

```
curl -XPUT "http://localhost:9200/movies/movie/_mapping" -d'
{
  "movie": {
    "properties": {
      "director": {
        "type": "string",
        "index": "not_analyzed"
      }
    }
  }
}'
```

There are however a couple of issues if we do this. First of all, it won't work as there already is a mapping for the field. Try the above request and you'll get an error message like this:

Response from ElasticSearch which doesn't allow us to change the mapping for the already mapped field.

```
{
  "error": "MergeMappingException[Merge failed with failures {[mapper [director] has di\
fferent index values, mapper [director] has different tokenize values, mapper [director\
] has different index_analyzer]}]",
  "status": 400
}
```

In many cases, such as this, it's not possible to modify existing mappings. Often the easiest work around for that is to create a new index with the desired mappings and re-index all of the data into the new index.

The second problem with adding the above mapping is that, even if we could add it, we would have limited our ability to search in the director field. That is, while a search for the exact value in the field would match we wouldn't be able to search for single words in the field any more.

Luckily, there's a simple solution to our problem. We add a mapping that upgrades the field to a "multi field". What that means is that we'll map the field multiple times for indexing. Given that one of the ways we map it match the existing mapping both by name and settings that will work fine and we won't have to create a new index. Here's a request that does that:

Upgrading the director field to a multi_field so that it will be indexed twice during indexing.

```
curl -XPUT "http://localhost:9200/movies/movie/_mapping" -d'
{
  "movie": {
    "properties": {
      "director": {
        "type": "multi_field",
        "fields": {
          "director": {
            "type": "string"
          },
          "original": {
            "type" : "string",
            "index" : "not_analyzed"
          }
        }
      }
    }
  }
}'
```

This time ElasticSearch is happy with us as we don't modify an existing mapping but only add a 'sub field'.

```
{
  "acknowledged": true
}
```

So, what did we just do? We told ElasticSearch that whenever it sees a property named `director` in a movie document that is about to be indexed in the movies index it should index it multiple times. Once into a field with the same name (`director`) and once into a field named `director.original` and the latter field should not be analyzed, maintaining the original value allowing us to filter by the exact director name.

With our new shiny mapping in place we can re-index one or both of the movies directed by Francis Ford Coppola and try the search request that filtered by director again. Only, this time we don't filter on the `director` field (which is indexed the same way as before) but instead on the `director.original` field:

A search request filtering using the author's full name, this time on the director.original field.

```
curl -XPOST "http://localhost:9200/movies/_search" -d'
{
    "query": {
        "constant_score": {
            "filter": {
                "term": { "director.original": "Francis Ford Coppola" }
            }
        }
    }
}'
```

Dynamic mapping

Upgrading the mapping for the director field to a multi field and re-indexing the documents solved the problem of being able to filter on the exact director name. However, doing this explicitly for multiple fields can be tedious and it hardly makes using ElasticSearch seem friction free. Luckily ElasticSearch has a solution for this, the concept of dynamic mapping[11].

During indexing when ElasticSearch encounters an unmapped field, a field for which we haven't provided any explicit mappings, that contains a value (not null or an empty array) it uses the name of the field and the value's type to look for a template. So far, in the examples that we've looked at, the default templates built into ElasticSearch has been used. However, we can provide templates of our own.

For instance, we can provide a template that maps all new string fields (existing ones we'll have to update ourselves, as we did with the director field) the way that we mapped the director field. Here's a request that adds a dynamic mapping for string that does just that for the movie type:

[11] http://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-dynamic-mapping.html

```
curl -XPUT "http://localhost:9200/movies/_mapping/movie" -d'
{
  "movie": {
    "dynamic_templates": [
      {
        "strings": {
          "match_mapping_type": "string",
          "path_match": "*",
          "mapping": {
            "type": "string",
            "fields": {
              "original": {
                "type": "string",
                "index": "not_analyzed"
              }
            }
          }
        }
      }
    ]
  }
}'
```

Now, whenever a new string field is added to movies we can query and filter on both `<name of field>` and `<name of field>.original`. In most cases however it's convenient to provide default templates not only for a specific type but for all types in the in the index. To do so we can explicitly create an index (using `PUT <index name>`) and provide mappings for the special `_default_` type.

Let's give that a try. First, we need to delete the existing movies index. We haven't covered how to do that yet, but it's quite easy. Simply make a `DELETE` request to a URL matching the index name. Like this:

Deleting the movies index. Note that both mappings and documents will be gone. Forever.

```
curl -XDELETE "http://localhost:9200/movies"
```

Now, to create the index again we switch the HTTP verb to `PUT`. As the request body we send a JSON object with a property named `mappings` in which we can provide mappings.

Creating the movies index again. This time with mappings for the _default_ type.

```
curl -XPUT "http://localhost:9200/movies" -d'
{
  "mappings": {
    "_default_": {
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "path_match": "*",
            "mapping": {
              "type": "string",
              "fields": {
                "original": {
                  "type": "string",
                  "index": "not_analyzed"
                }
              }
            }
          }
        }
      ]
    }
  }
}'
```

Now, whenever a new field with a string value is encountered during indexing, no matter the document type, it will be mapped using our template. Index the movies again and then inspect the mappings for the movie type (`curl -XGET "http://localhost:9200/movies/movie/_mapping"`) to verify this.

The result should look like the JSON object below. Note that the movie type has "inherited" the dynamic templates from the `_default_` type and that each string field has been mapped using our string template.

The mappings for the movie type after having indexed the movie documents.

```
{
  "movies": {
   "mappings": {
     "movie": {
      "dynamic_templates": [
        {
         "strings": {
           "mapping": {
            "type": "string",
            "fields": {
              "original": {
```

```
            "index": "not_analyzed",
            "type": "string"
           }
          }
         },
         "match_mapping_type": "string",
         "path_match": "*"
        }
       }
      ],
      "properties": {
       "director": {
        "type": "string",
        "fields": {
         "original": {
          "type": "string",
          "index": "not_analyzed"
         }
        }
       },
       "genres": {
        "type": "string",
        "fields": {
         "original": {
          "type": "string",
          "index": "not_analyzed"
         }
        }
       },
       "title": {
        "type": "string",
        "fields": {
         "original": {
          "type": "string",
          "index": "not_analyzed"
         }
        }
       },
       "year": {
        "type": "long"
       }
      }
     }
    }
}
```

More on dynamic mapping

As we've seen, dynamic mapping is a powerful feature that we can use to ensure that types and fields, even those we don't know will exist in advance, in an index will be indexed in the way we need. To further delve into dynamic mapping let's look at another example that will illustrate several useful features.

Indexing a simple 'tweet' object into an index named 'myindex'.

```
curl -XPOST "http://localhost:9200/myindex/tweet/" -d'
{
  "content": "Hello World!",
  "postDate": "2009-11-15T14:12:12"
}'
```

Given that there isn't already an indexed named "myindex" the above request will cause a number of things to happen in the ElasticSearch cluster.

1. An index named "myindex" will be created.
2. Mappings for a type named tweet will be created for the index. The mappings will contain two properties, content and postDate.
3. The JSON object in the request body will be indexed.

After having made the above request we can inspect the mappings that will have been automatically created (using `curl -XGET "http://localhost:9200/myindex/_mapping"`). The result looks like this:

The mappings that have been automatically created for the tweet type.

```
{
  "myindex": {
    "mappings": {
      "tweet": {
        "properties": {
          "content": {
            "type": "string"
          },
          "postDate": {
            "type": "date",
            "format": "dateOptionalTime"
          }
        }
      }
    }
  }
}
```

As we can see in the above response, ElasticSearch has mapped the content property as a string and the postDate property as a date. All is well. However, let's look at what happens if we delete the index and modify our indexing request to instead look like this:

Indexing another tweet object with a different value for the 'content' property.

```
curl -XPOST "http://localhost:9200/myindex/tweet/" -d'
{
  "content": "1985-12-24",
  "postDate": "2009-11-15T14:12:12"
}'
```

In the above request the content property is still a string, but the only content of the string is a date. Retrieving the mappings now gives us a different result.

The automatically generated mappings for the tweet type, again.

```
{
  "myindex": {
   "mappings": {
      "tweet": {
        "properties": {
          "content": {
            "type": "date",
            "format": "dateOptionalTime"
          },
          "postDate": {
            "type": "date",
            "format": "dateOptionalTime"
          }
        }
      }
    }
  }
}
```

ElasticSearch has now inferred that the content property also is a date. After all, JSON doesn't have a specific date type so we can hardly expect it to know that we intend a string that looks like a date to be mapped as a string. If we now try to index our original JSON object we'll get an exception in our faces:

The response from ElasticSearch when indexing a tweet with a value for the content property that can't be parsed as a date.

```
{
  "error": "MapperParsingException[failed to parse [content]]; nested: MapperParsingExc\
eption[failed to parse date field [Hello World!], tried both date format [dateOptionalT\
ime], and timestamp number with locale []]; nested: IllegalArgumentException[Invalid fo\
rmat: \"Hello World!\"]; ",
  "status": 400
}
```

We're trying to insert a string value into a field which is mapped as a date. Naturally ElasticSearch won't allow us to do that. While this scenario isn't very likely to happen, when it does it can be quite annoying and cause problems that can only be fixed by re-indexing everything into a new index. Luckily there's a number of possible solutions.

Disabling date detection

As a first step we can disable date detection for dynamic mapping. Here's how we would do that explicitly for documents of type tweet when creating the index:

Creating an index with automatic date detection disabled for the tweet type.

```
curl -XPUT "http://localhost:9200/myindex" -d'
{
  "mappings": {
    "tweet": {
      "date_detection": false
    }
  }
}'
```

Let's index the "problematic" tweet with the date in the content property again and inspect the mappings that have been dynamically created. This time we see a different result:

The automatically generated mappings for the tweet type after having disabled date detection.

```
{
  "myindex": {
    "mappings": {
      "tweet": {
        "date_detection": false,
        "properties": {
          "content": {
            "type": "string"
          },
          "postDate": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

Now both fields have been mapped as strings, which they indeed are, even though they contain values that can be parsed as dates. However, this isn't good either as we'd like the postDate field to be mapped as a date so that we can use range filters and the like on it.

Explicitly mapping date fields

We can explicitly map the postDate field as a date by re-creating the index and include a property mapping, like this:

Creating an index with automatic date detection disabled and a specific mapping for the postDate property for the tweet type.

```
curl -XPUT "http://localhost:9200/myindex" -d'
{
  "mappings": {
    "tweet": {
      "date_detection": false,
      "properties": {
        "postDate": {
          "type": "date"
        }
      }
    }
  }
}'
```

If we now index the "problematic" tweet with a date in the content field we'll get the desired mappings; the content field mapped as a string and the postDate field mapped as a date. That's nice. However, this approach can be cumbersome when dealing with many types or types that we don't know about prior to when documents of those types are indexed.

Mapping date fields using naming conventions

An alternative approach to disabling date detection and explicitly mapping specific fields as dates is instruct ElasticSearchs dynamic mapping functionality to adhere to naming conventions for dates. Take a look at the below request that (again) creates an index.

Creating an index with automatic date detection disabled for the all types (unless overridden with specific mappings) and a template for mapping fields matching a regular expression as dates.

```
curl -XPUT "http://localhost:9200/myindex" -d'
{
  "mappings": {
    "_default_": {
      "date_detection": false,
      "dynamic_templates": [
        {
          "dates": {
            "match": ".*Date|date",
            "match_pattern": "regex",
            "mapping": {
              "type": "date"
            }
```

```
          }
        }
      ]
    }
  }
}'
```

Compared to our previous requests used to creating an index with mappings this is quite different. First of all we no longer provide mappings for the tweet type. Instead we provide mappings for the `_default_` type. As we've already discussed, this is a special type whose mappings will be used as the default "template" for all other types.

As before we start by disabling date detection in the mappings. However, after that we no longer provide mappings for properties but instead provide a dynamic template named dates. Within the dates template we provide a pattern and specify that the pattern should be interpreted as a regular expression. Using this the template will be applied to all fields whose names either end with "Date" or whose names are exactly "date". For such fields the template instructs the dynamic mapping functionality to map them as dates.

Using this approach all string fields, no matter if their values can be parsed as dates or not will be mapped as string unless the field name is something like "postDate", "updateDate" or simply "date". Fields with such names will be mapped as dates instead.

While this is nice, there's one caveat. Indexing a JSON object with a property matching the naming convention for date fields but whose value can't be parsed as a date will cause an exception. Still, adhering to naming conventions for dates may be a small price to pay compared to the headaches of seemingly randomly having string fields mapped as dates simply because the first document to be indexed of a specific type happened to contain a string value that could be parsed as a date.

Stemming

Let's return to the movies and look at the below search request:

Searching for the word 'assassinations' in the title field.

```
POST movies/_search
{
    "query": {
        "query_string": {
          "fields": ["title"],
          "query": "assassinations"
        }
    }
}
```

The above request doesn't produce any hits. That's hardly surprising as there's no document in the index with the word "assassinations" in the title field. However, there is a document with

the singular form of it, “assassination”. Sometimes we want words that have similar semantic interpretations to be considered as equivalent in order to improve free text search. To do so we need to use stemming[12], reducing words to their “root” form.

In order to do so for our title field we can map it so that it’s analyzed in a way that does stemming in a given language. We can do this by mapping the title field to be analyzed with the `english` analyzer. The English analyzer is one of many language analyzers that are predefined in ElasticSearch. These optimizes for search in a given language by removing stop words (such as “and” and “or”) and by doing stemming. Below is an example request for creating the movies index with the title field mapped with the English analyzer:

Creating the movies index with the title field mapped to use the english analyzer.

```
PUT /movies
{
  "mappings": {
    "movie": {
      "properties": {
        "title": {
          "type": "string",
          "analyzer": "english"
        }
      }
    }
  }
}
```

Given that we recreate the movies index with the above request and re-index the movies we’ll now get a hit when searching for “assassinations” in the title field. Nice!

However, if we now try to search for the word “the” we won’t get any hits. That’s because the English analyzer has treated that word as a stop word, effectively not indexing it as part of the title field. Also, we have made it impossible for ourselves to search in a way that doesn’t utilize stemming.

While this may in some cases be fine other times we may need the flexibility of being able to sometimes search in the title field using the English analyzer and sometimes using the standard analyzer. Also, in the example request for creating the movies index I left out the dynamic template that mapped all strings so that they have a .original field. Finally, what if we want to be able to sometimes search using stemming and stop words in other fields?

In order to give us greater flexibility and to restore the .original field we can use dynamic mappings to map all string fields to have both the .original field and a field that is indexed using the English analyzer. Here’ how we can create the movies index with such mappings:

[12]To learn more about what stemming is see http://www.comp.lancs.ac.uk/computing/research/stemming/general/

Creating the movies index with a template for strings.

```
curl -XPUT "http://localhost:9200/movies" -d'
{
  "mappings": {
    "_default_": {
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "path_match": "*",
            "mapping": {
              "type": "string",
              "fields": {
                "original": {
                  "type": "string",
                  "index": "not_analyzed"
                },
                "english": {
                  "type": "string",
                  "analyzer": "english"
                }
              }
            }
          }
        }
      ]
    }
  }
}'
```

Using the above mappings all string fields in documents in the movies index will be indexed three times and we'll be able to search in them using `<field name>`, `<field name>.original` and `<field name>.english`. In the index the movie with ID 4, Apocalypse Now, will have the following values in these fields:

Field	Value
title	"apocalypse", "now"
title.english	"apocalyps", "now"
title.original	"Apocalypse Now"

If you want, you can use the term vectors API to verify this yourself, like this: `curl -XGET "http://localhost:9200/movies/movie/4/_termvector?fields=title*"`.

Custom analyzers

In this chapter we'll take a look at how to further customize how fields are mapped by creating, and using, custom analyzers. This is a broad and far from trivial topic and a detailed look at it is beyond the scope of this book[13]. However, we'll briefly discuss it and look at an example that I myself have found useful.

Analyzers

An analyzer is what a field's value passes through on it's way into the index and we define what analyzer, if any, should be used through mappings. In order for text based search to work as expected search terms are also passed through the same analyzer.

An analyzer is composed of:

- Zero or more char filters - These can modify a string value prior to tokenization, typically in order to remove unwanted characters.
- A single tokenizer - Breaks a single string value down to a stream of tokens.
- Zero or more token filters - These process the tokens produced by the tokenizer. A token filter can for instance lowercase the terms or remove unwanted terms, such as stop words.

There are a number of predefined analyzers in ElasticSearch such as the Standard analyzer, which is what's used for strings by default and the Language analyzers that optimizes for searching in a specific language. For a full list, see the documentation[14]. Additionally, we can also define our own analyzers when creating indexes.

An example

In the previous chapter we saw how to map a field as not analyzed, meaning that ElasticSearch will index the value as a single, unmodified, token, thereby allowing us to filter on the exact value.

Let's create a new index with dynamic mappings that does this again, mapping strings as multi field with a .original field that isn't analyzed:

[13] To learn more about analysis I recommend you to read the part about it in "The definitive guide": http://www.elastic.co/guide/en/elasticsearch/guide/current/analysis-intro.html

[14] http://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-analyzers.html

```
curl -XPUT "http://localhost:9200/pages" -d'
{
  "mappings": {
    "_default_": {
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "path_match": "*",
            "mapping": {
              "type": "string",
              "fields": {
                "original": {
                  "type": "string",
                  "index": "not_analyzed"
                }
              }
            }
          }
        }
      ]
    }
  }
}'
```

We've already seen this mapping in action, with movies, and that it's working as expected. However, there is one problem with it. Let's say we index a document with a string field that has a really long value:

```
curl -XPOST "http://localhost:9200/pages/article" -d'
{
  "title": "With a very long text",
  "text": "This is a actually a really long text..."
}'
```

It's a bit tricky to illustrate a really long text as part of a request in a book, but imagine that the text property's value in the above request is considerably longer. The response from ElasticSearch would then be:

```
{
  "error": "IllegalArgumentException[Document contains at least one immense term in fie\
ld=\"text.original\" (whose UTF8 encoding is longer than the max length 32766), all of \
which were skipped.  Please correct the analyzer to not produce such terms.  The prefix\
 of the first immense term is: '[76, 111, 114, 101, 109, 32, 105, 112, 115, 117, 109, 3\
2, 100, 111, 108, 111, 114, 32, 115, 105, 116, 32, 97, 109, 101, 116, 44, 32, 99, 111].\
..', original message: bytes can be at most 32766 in length; got 47287]; nested: MaxByt\
esLengthExceededException[bytes can be at most 32766 in length; got 47287]; ",
  "status": 500
}
```

ElasticSearch responds with an exception telling us that there's a term in the `text.original` field that exceeds 32766 bytes and that it doesn't like that. While this may seem uncooperative of ElasticSearch it's actually quite helpful. After all, having gigantic terms in our index would be problematic performance wise and neither would it be efficient in terms of disk usage.

And, when would we really be interested in filtering on the full, exact, values of really, really long strings? Probably never. However, we still need to be able to do so for shorter strings. One solution could be to skip using dynamic mapping and mapping each individual string field explicitly, only adding the .original mapping for fields that we know won't contain huge strings. Giving up the dynamic mapping and mapping all fields explicitly wouldn't be much fun though.

An alternative solution is to map the .original field as analyzed but with an analyzer that indexes shorter strings as a single token and simply ignores longer strings. To do so we need to create an analyzer that uses the `keyword` tokenizer which produces a single token, the same as the input value, and a token filter that removes tokens that are longer than a given length. Below is a request that does this.

```
curl -XPUT "http://localhost:9200/pages" -d'
{
  "settings": {
    "analysis": {
      "filter": {
        "short_original": {
          "type": "length",
          "max": 200
        }
      },
      "analyzer": {
        "original": {
          "tokenizer": "keyword",
          "filter": "short_original"
        }
      }
    }
  },
  "mappings": {
    "_default_": {
      "dynamic_templates": [
```

```
        {
          "strings": {
            "match_mapping_type": "string",
            "path_match": "*",
            "mapping": {
              "type": "string",
              "fields": {
                "original": {
                  "type": "string",
                  "index": "analyzed",
                  "analyzer": "original"
                }
              }
            }
          }
        }
      ]
    }
  }
}'
```

Using these mappings all string fields will be indexed using the Standard analyzer and also have a `<field name>.original` field. However, for a string longer than 200 characters the `.original` field will be empty.

Aggregations

We've already seen how we can search with ElasticSearch and get results in the form of hits. However, the `_search` endpoint offers more. We can also ask it for aggregations based on fields within the documents that the query. As an example, let's look at this request:

Finding all movies in the Drama genre.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "genres.original": "Drama"
        }
      }
    }
  }
}'
```

The above request helps us answer the question "What movies are there in the genre drama?". But, let's say we were interested the question "What directors have directed the movies in the drama genre?". Theoretically we could grab *all* of the results to the above request and look at their `director` fields. Of course that would result in duplicates, a single director having directed multiple drama movies. So, we'd have to group by the director name. And, if we're interested in finding out which director has directed the most dramas we'd have to sort by that.

We could of course do this through code but that would be highly inefficient. Both because we have to write that code and also because we'd have to grab all of the results (while we have just six movies in our index, imagine doing that for IMDB) from ElasticSearch. Luckily, this type of computation is easily done by asking ElasticSearch for an aggregation.

There are many types of aggregations but in this particular case where we want to group by the exact value in a field a terms aggregation is suitable. Below is the above request updated to include a terms aggregation for the director field. Note that it more specifically is for the `director.original` field as we want to aggregate based on the full director name, not the individual words.

Searching for movies in the drama genre and asking ElasticSearch for a terms aggregation for the director.original field.

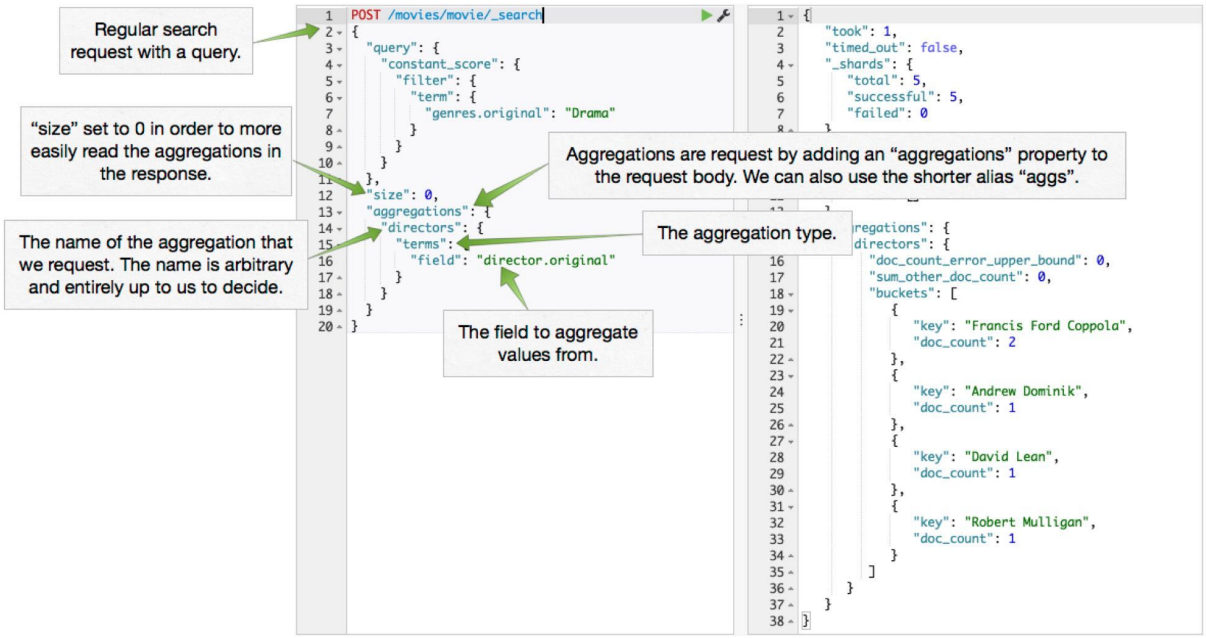
```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "genres.original": "Drama"
        }
      }
    }
  },
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original"
      }
    }
  }
}'
```

The response to the above request from ElasticSearch is this:

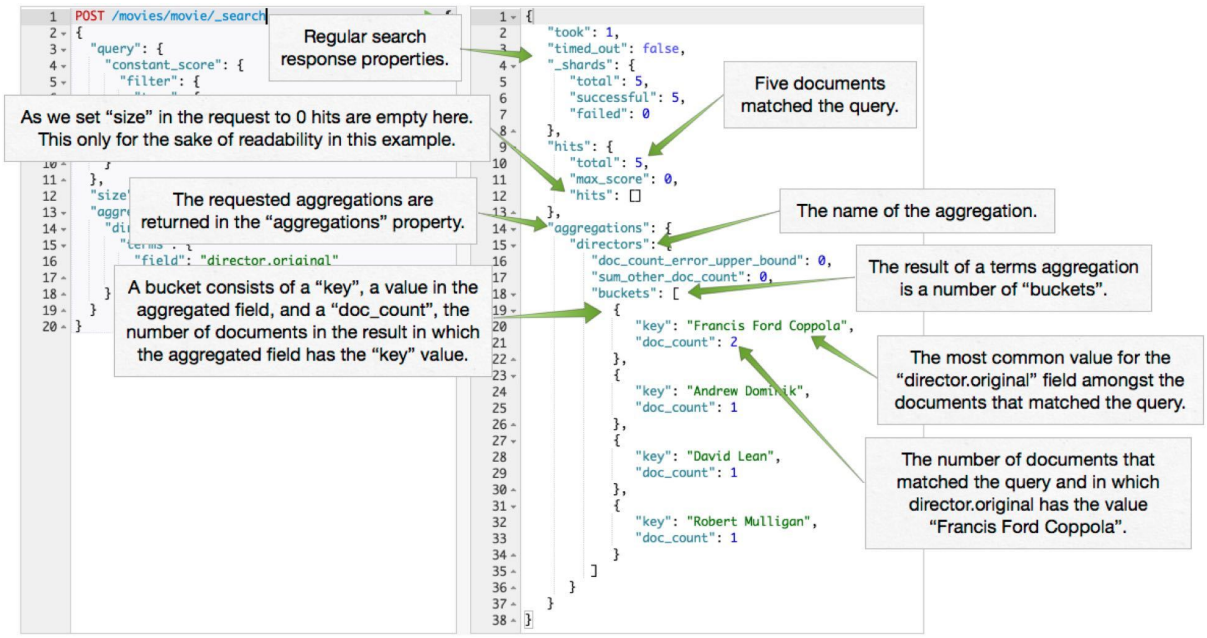
```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 5,
    "max_score": 0,
    "hits": [
      //Hits here, omitted for brevity
    ]
  },
  "aggregations": {
    "directors": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "Francis Ford Coppola",
          "doc_count": 2
```

```
        },
        {
          "key": "Andrew Dominik",
          "doc_count": 1
        },
        {
          "key": "David Lean",
          "doc_count": 1
        },
        {
          "key": "Robert Mulligan",
          "doc_count": 1
        }
      ]
    }
  }
}
```

The response now includes another top level property, `aggregations` and that contains the result of the aggregation we asked for. Let's inspect both the search request and response with a terms aggregation a bit closer.



A closer look at a search request with a terms aggregation in Sense.



A closer look at the response to a search request with a terms aggregation in Sense.

Size

As with the search results the results of a `terms` aggregation (and some other aggregation types) are also by default limited to ten items. We can override this default value by adding a `size` parameter. Here's an example:

Searching for all movies and aggregating the director.original field with size set to two.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "size": 0,
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original",
        "size": 2
      }
    }
  }
}'
```

The response from ElasticSearch to the above request.

```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 6,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "directors": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 3,
      "buckets": [
        {
          "key": "Francis Ford Coppola",
          "doc_count": 2
        },
        {
          "key": "Andrew Dominik",
          "doc_count": 1
        }
      ]
    }
  }
}
```

As we can see in the response from ElasticSearch it respects the `size` parameter in the `terms` aggregation and only returns two buckets. Also, note that the return `sum_other_doc_count` property has the value three. This tells us that while there are only two buckets returned ElasticSearch has found a total of five unique values in the `director.original` field. Two which are included in the response and three which aren't.

Contrary the top level search request body we can't give aggregations a `from` parameter. This means that it's not possible to do pagination for aggregation results.

Multiple aggregations in a single request.

In the above examples we only asked for a single aggregation in the requests. However, we're by no means limited to doing so. For instance, if we'd like to know both the number of movies grouped by author and grouped by decade we can use the below request:

A search request with two aggregations.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "size": 0,
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original",
        "size": 2
      }
    },
    "decades": {
      "histogram": {
        "field": "year",
        "interval": 10
      }
    }
  }
}'
```

In the above request we've added a second aggregation named "decades". This aggregation is of a different type than the "directors" aggregation. It's a `histogram` aggregation. Histogram aggregations can be used to group fields with numeric or date[15] values according to a specified interval. The response from ElasticSearch looks like this:

ElasticSearch's response to our request with two aggregations.

```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 6,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "directors": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 3,
```

[15]While histogram aggregations can be used for date fields it's better to use a different aggregation type for dates, the `date_histogram` type, which provides better accuracy and supports date specific parameters. See: http://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-datehistogram-aggregation.html.

```
      "buckets": [
        {
          "key": "Francis Ford Coppola",
          "doc_count": 2
        },
        {
          "key": "Andrew Dominik",
          "doc_count": 1
        }
      ]
    },
    "decades": {
      "buckets": [
        {
          "key": 1960,
          "doc_count": 2
        },
        {
          "key": 1970,
          "doc_count": 2
        },
        {
          "key": 2000,
          "doc_count": 2
        }
      ]
    }
  }
}
```

Bucket and metric aggregations

The two aggregation types that we've seen so far, `terms` and `histogram`, are both "bucket aggregations". A bucket aggregation groups by a key and calculates a count, typically the number of documents with the key.

ElasticSearch also supports another family of aggregations, "metric aggregations". Metric aggregations don't do bucketing and instead calculate one or more metrics of a set of documents.

A simple metric aggregation is the `max` aggregation which returns the maximum value found in a field. Here's an example, retrieving the highest value in the year field in our movies:

A search request with a max aggregation.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "size": 0,
  "aggregations": {
    "most_recent_year": {
      "max": {
        "field": "year"
      }
    }
  }
}'
```

The response to the above request.

```
{
   "took": 1,
   "timed_out": false,
   "_shards": {
      "total": 5,
      "successful": 5,
      "failed": 0
   },
   "hits": {
      "total": 6,
      "max_score": 0,
      "hits": []
   },
   "aggregations": {
      "most_recent_year": {
         "value": 2007
      }
   }
}
```

Nesting aggregations

Metric aggregations are simple in the sense that they only calculate one or more metrics. Bucket aggregations however offer additional functionality beyond grouping by keys and calculating how many documents match each key. We can request other aggregations inside bucket aggregations. The results of such aggregations will be scoped to the (outer) bucket. In other words, we can nest aggregations!

Let's look at an example of using nested aggregations, by combining aggregations that we've used before. In the below request we bucket movies by author using a `terms` aggregation and then we calculate the year of the most recent movie *by that author* by adding an `aggs` (alias for `aggregations`) property inside the `genre` aggregation with a `max` aggregation:

An example request using nested aggregations.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "size": 0,
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original",
        "size": 2
      },
      "aggs": {
        "most_recent_year": {
          "max": {
            "field": "year"
          }
        }
      }
    }
  }
}'
```

The response to the above request.

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 6,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "directors": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 3,
      "buckets": [
        {
          "key": "Francis Ford Coppola",
          "doc_count": 2,
          "most_recent_year": {
            "value": 1979
          }
```

```
        },
        {
          "key": "Andrew Dominik",
          "doc_count": 1,
          "most_recent_year": {
            "value": 2007
          }
        }
      ]
    }
  }
}
```

Pretty neat, huh? Also, we're not limited to nesting on one level. We can put one or more bucket aggregations inside an outer bucket aggregation. Below is an example of such a request:

A search request that buckets first by director, then by genre and finally by the largest value of the year field.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "size": 0,
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original",
        "size": 2
      },
      "aggs": {
        "genres": {
          "terms": {
            "field": "genres.original",
            "size": 2
          },
          "aggs": {
            "most_recent_year": {
              "max": {
                "field": "year"
              }
            }
          }
        }
      }
    }
  }
}'
```

Let's take a look at a part of the response to the above request:

The first bucket in the response to the above request.

```
{
   "key": "Francis Ford Coppola",
   "doc_count": 2,
   "genres": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 1,
      "buckets": [
         {
            "key": "Drama",
            "doc_count": 2,
            "most_recent_year": {
               "value": 1979
            }
         },
         {
            "key": "Crime",
            "doc_count": 1,
            "most_recent_year": {
               "value": 1972
            }
         }
      ]
   }
}
```

As shown above the response tells us that there are two movies directed by Francis Ford Coppola in our index. Both of these movies are in the drama genre and the most recent of the two is from 1979. Only one of the movies (it must be one of the two drama movies as there are only two movies in the top level bucket) is also in the crime genre, and this one is from 1972.

What are aggregations good for?

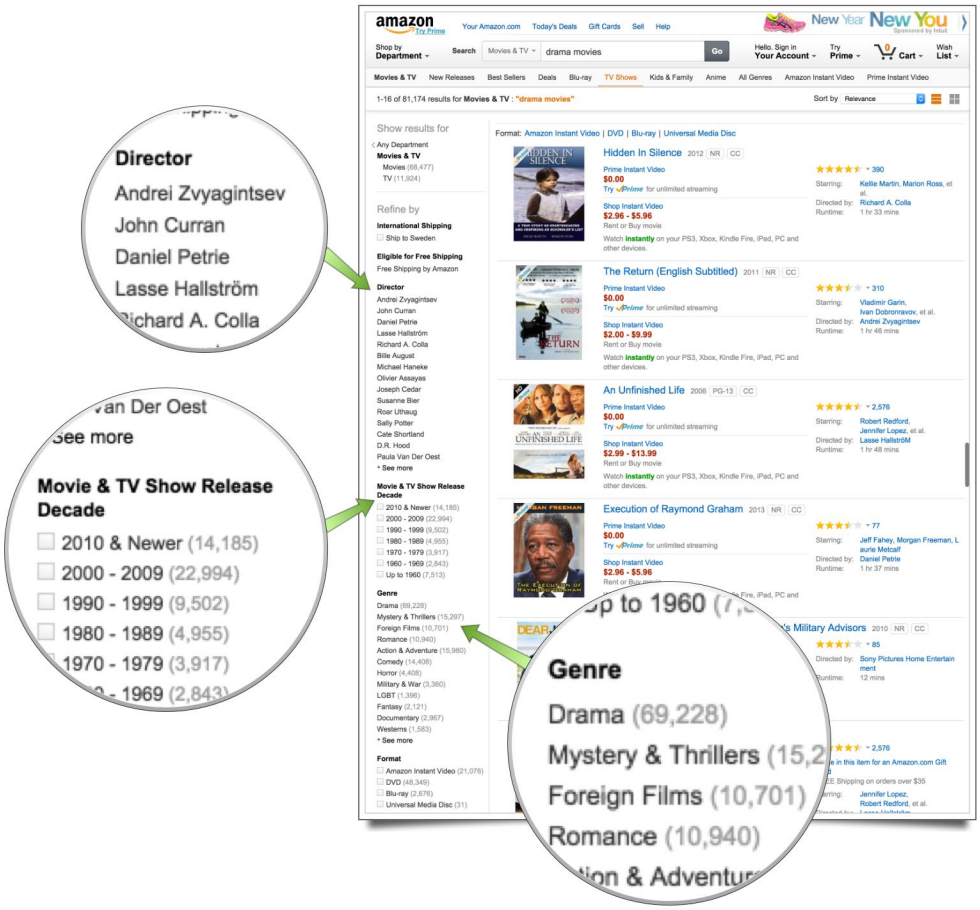
By now it should, hopefully, be clear that aggregations are generated values based on the documents that match a search request. And, as we've seen in the examples, the result of a requested aggregation is part of the response to the search request. But, what can we use aggregations for?

There are a ton of use cases for aggregations. If we use ElasticSearch to analyze logs or statistical data we can use aggregations to extract information from the data such as number of HTTP requests per URL, average call time to a call center per day of the week or number of restaurants that are open on Sundays in different geographical areas.

One especially powerful and interesting aggregation type when analyzing data is the `significant_terms` aggregation. This aggregation type allows us to find things in a foreground set (such as support tickets created this week) compared to a background set (such as all support tickets).

While outside the scope of this book I encourage you to read more about significant terms aggregations in the the documentation[16] and in this blog post[17].

Another use case for aggregations is navigation. In such cases we may use aggregations to generate a list of categories based on the content on a web site to build a menu, or we may aggregate values from many different fields from documents that match a search query to allow users to narrow their search. The below screen shot from Amazon illustrates an example of the latter:



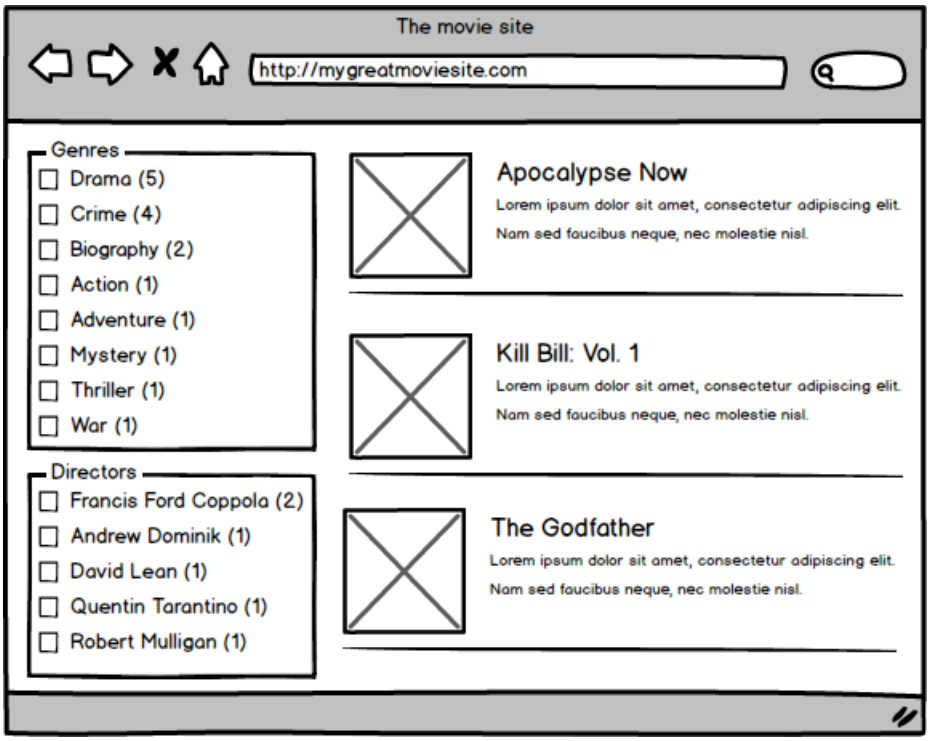
An example of how facets/aggregations are used to filter search results on Amazon.com.

Combining aggregations and filters

In the screen shot from Amazon we saw a fairly common example of how aggregations can be used for visualizing aggregated values from search results and to allow users to filter by them. If we were to do something similar for our movies it might look something like this:

[16] http://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-significantterms-aggregation.html

[17] https://www.elastic.co/blog/significant-terms-aggregation/



In order to be able to create a page such as the one above we'd use a search request such as this:

A search request for all movies and terms aggregations for directors and genres.

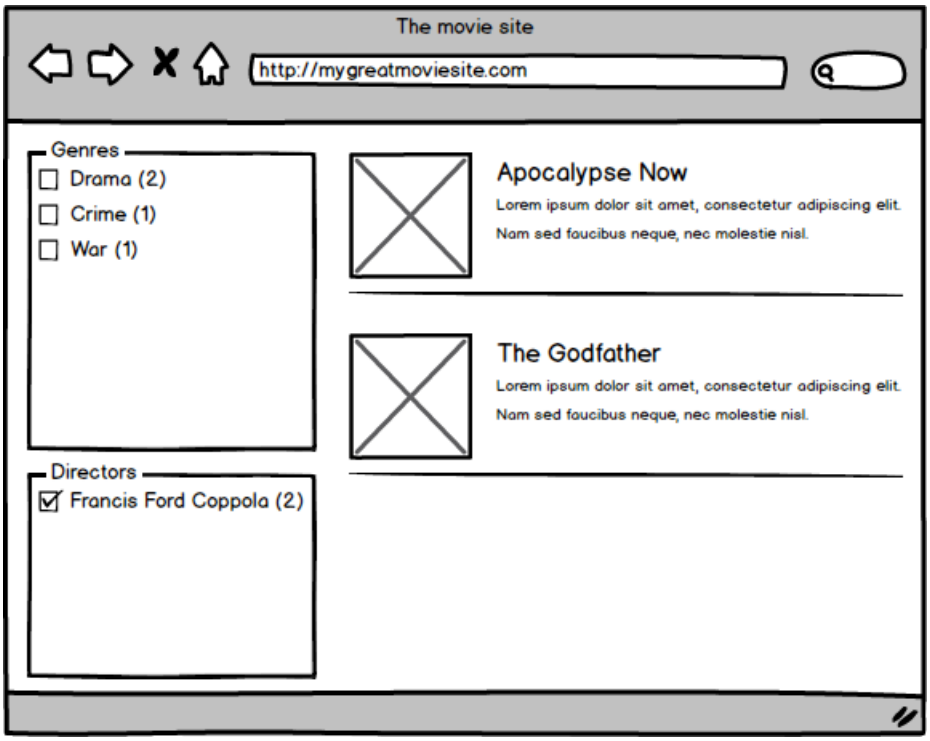
```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original"
      }
    },
    "genres": {
      "terms": {
        "field": "genres.original"
      }
    }
  }
}'
```

Now, what if a user wants to filter by a director? On the web development side of things we'd send the director name as a parameter of some sort back to the server. Once on the server we'd need to modify our request to ElasticSearch to add a filter, like this:

The same request as the previous one, only this time with filtering for movies by a specific director.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "term": {
          "director.original": "Francis Ford Coppola"
        }
      }
    }
  },
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original"
      }
    },
    "genres": {
      "terms": {
        "field": "genres.original"
      }
    }
  }
}'
```

With the filtered response from ElasticSearch we rebuild the web page based on the new response:



As the search response now only contains the two movies directed by Francis Ford Coppola only two hits will be shown. Also, as aggregations are calculated over the document set that the query generates the filters in the left part of the page has also changed. Only the genres and directors found in the movies by Francis Ford Coppola are shown.

Often this is the desired behavior, letting the aggregations reflect the result of applied queries and filters. However, sometimes it's not. For instance, what if we want to allow users to filter by multiple directors? In such cases we'd still want buckets for the other directors even though there are no documents with them in the director field that match the current query.

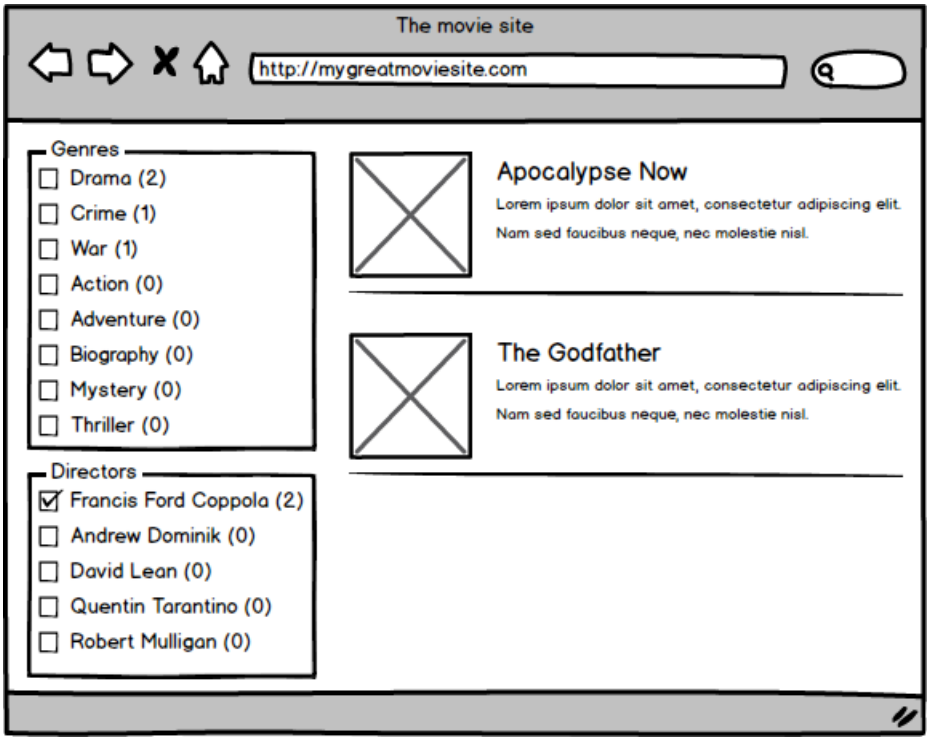
In such cases we can add a `min_doc_count` parameter to our aggregations with zero as value. Like this:

Including empty buckets by setting the min_doc_count parameter in the aggregations.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "term": {
          "director.original": "Francis Ford Coppola"
        }
      }
    }
  },
```

```
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original",
        "min_doc_count": 0
      }
    },
    "genres": {
      "terms": {
        "field": "genres.original",
        "min_doc_count": 0
      }
    }
  }
}'
```

The `min_doc_count` parameter allows us to control the minimum number of documents that must match a term in order for a bucket to be created by a terms aggregation. The default value is one. By setting it to zero buckets will be created for terms even though no document in the search results has that term. For our page this would mean that other genres and directors would still be listed:



Post filter

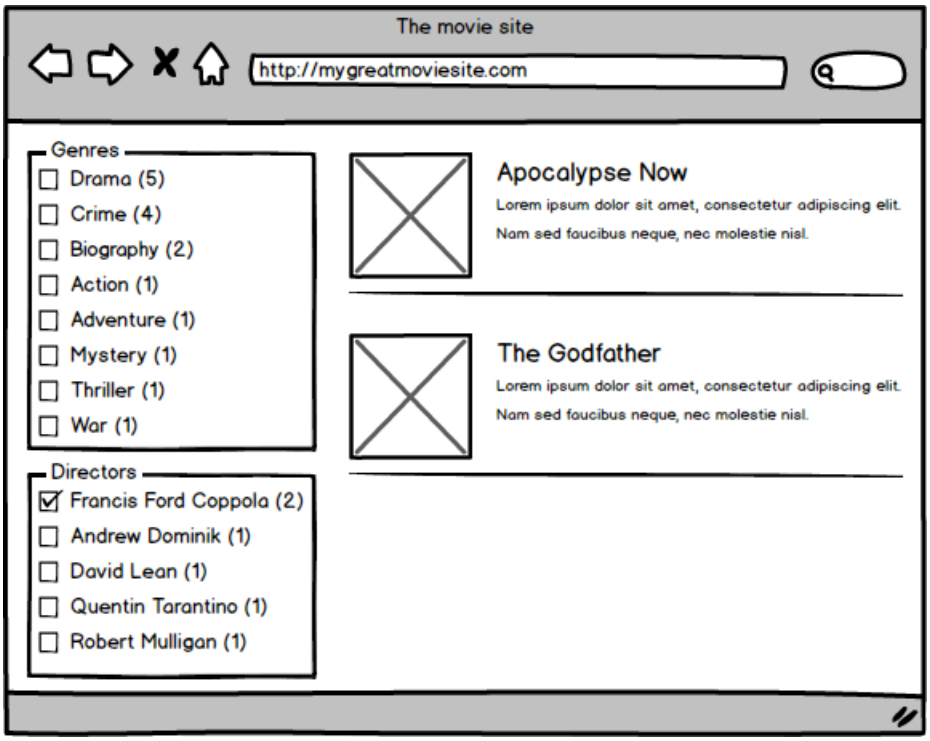
Sometimes we want to be able to filter search results without effecting the aggregations. For instance, if we wanted to show the number of movies in each genre and by each director

regardless of the director filter in the previous example, we'd have to apply the filter to the search results after aggregations has been generated. ElasticSearch supports this by allowing us to add a `post_filter` parameter to the search request body. Here's how we'd search for movies by Francis Ford Coppola but get aggregations for the director and genres fields without the filter applied:

Filtering search hits but not aggregations by using a post filter.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "post_filter": {
    "term": {
      "director.original": "Francis Ford Coppola"
    }
  },
  "aggregations": {
    "directors": {
      "terms": {
        "field": "director.original"
      }
    },
    "genres": {
      "terms": {
        "field": "genres.original"
      }
    }
  }
}'
```

If we used the response to the above request as data source for our web page the page would look like this:



Note that while we in the example requests in this chapter have used only one of the `query` and `post_filter` parameters in the same request it's possible to use both. We would probably want to do so if we were to support free text search for our movies in addition to filtering by director and genre.

Only use post_filter when needed

The `post_filter` parameter has an alias, `filter`. This is for backwards compatibility as `post_filter` used to be named `filter` in early versions of ElasticSearch. The name was changed for a reason. While it's certainly possible, and more convenient, to use `post_filter` instead of the `query` parameter when creating a request that should only filter the results it's not as good as using the `query` parameter performance wise. So, feel free to use `post_filter` even if you don't need to while debugging but only use it when you actually need it against a production cluster.

Advanced CRUD

In chapter two we saw how to create, read, update and delete a single document in an index. In this chapter we'll look at more advanced ways to do such operations, including indexing multiple documents in a single request, deleting documents that match a query and updating documents without having to send the entire document in the request.

The update API

Sometimes we may have documents in an ElasticSearch index that we want to update without having to provide the full document. In my experience this is the case when we either want to perform a minor update on a document often and saving the extra roundtrip of first having to fetch the document and then re-indexing is a good thing, or when we combine data from several data sources, fetched at different times, into common documents.

ElasticSearch has support for this through its `_update` API[18]. Using that we can either update a document by providing a script that ElasticSearch will apply to the document or we can provide a JSON object that ElasticSearch will "merge" into an existing document.

Let's look at an example. First we index a document, a product:

Indexing request for sample data.

```
curl -XPUT "http://localhost:9200/products/toys/1" -d'
{
  "name": "Rainbow Dash vinyl figure",
  "price": 12.99,
  "inStock": 42
}'
```

Now, someone comes along and buys one of these meaning we need to decrement the `inStock` field. In order to do so we can of course fetch the document with a GET request to `/products/toys/1`, modify the returned `_source` and re-index it. Or, we can use the `_update` API and use a script. The script can be written in one of many languages but unless we specify otherwise ElasticSearch expects it to be in Groovy. To read more about scripting see the documentation[19]. Here's an example request:

[18] We'll cover most of the interesting aspects of the update API in this chapter, but there's always more to learn. To do, check out the documentation: http://www.elastic.co/guide/en/elasticsearch/reference/current/docs-update.html.

[19] http://www.elastic.co/guide/en/elasticsearch/reference/current/modules-scripting.html

An update request with a script.

```
curl -XPOST "http://localhost:9200/products/toys/1/_update" -d'
{
  "script" : "ctx._source.inStock -= 1"
}'
```

As shown above the update API is used by adding `/_update` to the end of the URL to a document. The response from ElasticSearch to the above request looks similar to the response to a regular indexing request:

Response from ElasticSearch to the above request.

```
{
  "_index": "products",
  "_type": "toys",
  "_id": "1",
  "_version": 2
}
```

And if we now retrieve the document from the index we'll see that it indeed has been updated:

The result of a GET request to /products/toys/1.

```
{
  "_index": "products",
  "_type": "toys",
  "_id": "1",
  "_version": 2,
  "found": true,
  "_source": {
    "name": "Rainbow Dash vinyl figure",
    "price": 12.99,
    "inStock": 41
  }
}
```

Should we prefer to do so, and the ElasticSearch documentation certainly does, we can use variables in the script and supply values for them in a separate parameter in the request, like this:

An update request with a variable, 'count', in the script and a value assigned to it in the 'params' parameter.

```
curl -XPOST "http://localhost:9200/products/toys/1/_update" -d'
{
  "script" : "ctx._source.inStock -= count",
  "params" : {
    "count" : 1
  }
}'
```

When we use the `_update` API the document is still re-indexed. While we don't give ElasticSearch the full source of the document it will still retrieve it from the index, update it and re-index it. This includes incrementing the document's `_version`.

Upserting

The way we used the `_update` API in the previous examples assumes that the document already exists. If it doesn't ElasticSearch will give us a 404 response. Sometimes though we may want to use the `_update` API in such a way that the document will be created if it doesn't already exists. While it may not be the most realistic example, let's say that we assume that products that aren't already the index should be created when someone buys one of them and that they should then have have a default `inStock` value of ten. We could then use the following request:

An update request including an 'upsert' parameter. Note that the request is for a document with ID 2, which doesn't exist yet.

```
curl -XPOST "http://localhost:9200/products/toys/2/_update" -d'
{
  "script" : "ctx._source.inStock -= count",
  "params" : {
    "count" : 1
  },
  "upsert": {
    "inStock": 10
  }
}'
```

The response from ElasticSearch to the above request.

```
{
  "_index": "products",
  "_type": "toys",
  "_id": "2",
  "_version": 1
}
```

If we retrieve the document we'll see that it has indeed been created with value(s) from the `upsert` parameter:

The response to a GET request to /products/toys/2.

```
{
  "_index": "products",
  "_type": "toys",
  "_id": "2",
  "_version": 1,
  "found": true,
  "_source": {
    "inStock": 10
  }
}
```

Note that in the above response the newly created document's `inStock` property has the value ten. Clearly the document has been created using the `upsert` parameter but the script in the `script` parameter hasn't been used. This is the default behavior of the `_update` API. If the document exists the `script` parameter will be used and if it doesn't the `upsert` parameter will be used.

Should we want both parameters to be used for documents that don't exist, meaning that the document should first be created using the `upsert` parameter and then modified by the script we can include a `scripted_upsert` parameter with `true` as value. Here's an example of doing so:

An update request for another document that doesn't exist (ID: 3) including the scripted_upsert parameter.

```
curl -XPOST "http://localhost:9200/products/toys/3/_update" -d'
{
  "script" : "ctx._source.inStock -= count",
  "params" : {
    "count" : 1
  },
  "upsert": {
    "inStock": 10
  },
  "scripted_upsert": true
}'
```

The response to a GET request to /products/toys/3.

```
{
  "_index": "products",
  "_type": "toys",
  "_id": "3",
  "_version": 1,
  "found": true,
  "_source": {
    "inStock": 9
  }
}
```

Partial updating

Sometimes we may want to update or add new fields in existing documents. We can do that through a script, but if we the value of an updated or new field doesn't depend on existing values in the document a script is a bit of a hassle. In such cases we can instead use a `doc` parameter in the update request. The `doc` parameter's value should be a JSON object and when the request reaches ElasticSearch it will merge the properties from the object into the document. Here's an example:

An update request for our first product (ID: 1) with a 'doc' parameter.

```
curl -XPOST "http://localhost:9200/products/toys/1/_update" -d'
{
  "doc": {
    "price": 7.99,
    "onSale": true
  }
}'
```

The response to a GET request to /products/toys/1 after first having made the above update request.

```
{
  "_index": "products",
  "_type": "toys",
  "_id": "1",
  "_version": 3,
  "found": true,
  "_source": {
    "name": "Rainbow Dash vinyl figure",
    "price": 7.99,
    "inStock": 41,
    "onSale": true
  }
}
```

As we can see from the response above the `price` field has been updated and a new `onSale` field has been added to it.

Combining partial updating and upserting

Sometimes we want to build documents with data from multiple data sources. For instance, let's say that we have a system that knows about the price of our products and another that keeps track of the stock balance. Both knows the name of the product.

In order to merge the data from both systems we could create a program that fetches it from each system, merges it into a single object and then indexes a document in ElasticSearch. However, sometimes that may not be convenient, or even possible, and we'd like to be able to create the document based on data from either data source and then later merge data from the other. In such cases we can use the update API and send all properties known to each system using the `doc` parameter. However, that will, by default, only work if the document already exists.

Luckily we can combine the `doc` parameter with a `doc_as_upsert` parameter, instructing ElasticSearch to create the document if it doesn't already exist. In other words, the two systems could make requests such as the ones below, without having to care if the document exists or not and without having to worry about removing fields put there by the other system.

An imaginable request from the pricing system to upsert a product.

```
curl -XPOST "http://localhost:9200/products/toys/4/_update" -d'
{
  "doc": {
    "name": "My Little Pony Surprise Mini Figure",
    "price": 3.49
  },
  "doc_as_upsert": true
}'
```

An imaginable request from the stock balance system to upsert a product.

```
curl -XPOST "http://localhost:9200/products/toys/4/_update" -d'
{
  "doc": {
    "name": "My Little Pony Surprise Mini Figure",
    "inStock": 198
  },
  "doc_as_upsert": true
}'
```

Regardless of in what order the two above requests are made the document will be the same in the index:

The response to a GET request to /products/toys/4 after having made both of the above update requests.

```
{
  "_index": "products",
  "_type": "toys",
  "_id": "4",
  "_version": 2,
  "found": true,
  "_source": {
    "name": "My Little Pony Surprise Mini Figure",
    "price": 3.49,
    "inStock": 198
  }
}
```

The bulk API

When developing applications that interact with ElasticSearch we often work with large sets of data. While it's of course possible to do so using the ways that we've learned so far, which allows us to index, update or delete single documents, those aren't very efficient. For instance, if we were to index a million movies making a million HTTP requests, one for indexing each movie, would probably take a while. In such cases we can use ElasticsSearch's `_bulk` API through which we can perform multiple index, delete and update operations in a single request.

The endpoints for the bulk API are `/_bulk`, `/<index name>/_bulk` and `/<index name>/<type name>/_bulk`. When making requests to either of those the request body should be made up of JSON objects followed by line breaks (`\n`). The syntax for the request body is as follows:

```
{//action and meta data}\n
{//optional source}\n
...
{//Nth action and meta data}\n
{//Nth optional source}\n
```

An "action and meta data" object describes the operation that should be performed. It should contain a single top level property whose name describes what operation to perform. An example request body that indexes two movies may look like this:

```
{ "index" : { "_index" : "movies", "_type" : "movie", "_id" : "1" } }
{ "title": "The Godfather", "year": 1972 }
{ "index" : { "_index" : "movies", "_type" : "movie", "_id" : "3" } }
{ "title": "Lawrence of Arabia", "year": 1962 }
```

When sent, using POST, to either `/_bulk`, `/movies/_bulk` or `/movies/movie/_bulk` the above request body will index two movies. When all operations are for the same index or type those can be omitted from the "action and meta data" objects given that they can be found in the URL. As such the below request body could be POSTed to `/movies/movie/_bulk` and achieve the same result as the previous one.

```
{ "index" : { "_id" : "1" } }
{ "title": "The Godfather", "year": 1972 }
{ "index" : { "_id" : "3" } }
{ "title": "Lawrence of Arabia", "year": 1962 }
```

Apart from indexing the bulk API also supports deleting (using `delete` as the top level property), updating (using `update` as the top level property) and creating (using `create` as the top level property). The latter, creating, is the same as indexing with the exception that `create` operations will only be performed if the document doesn't already exist. Below is a request body, intended to be POSTed to `/movies/movie/_bulk` that utilizes all of the possible operations:

```
{ "index" : { "_id" : "1" } }
{ "title": "The Godfather", "year": 1972 }
{ "create" : { "_id" : "3" } }
{ "title": "Lawrence of Arabia", "year": 1962 }
{ "delete" : { "_id" : "2" } }
{ "update" : { "_id" : "4" } }
{ "doc": { "year": 1979 }}
```

If we were to send the above request body as a bulk request the response from ElasticSearch would look like this (given that we already have the movies in our index):

```
{
  "took" : 1,
  "errors" : true,
  "items" : [ {
    "index" : {
      "_index" : "movies",
      "_type" : "movie",
      "_id" : "1",
      "_version" : 4,
      "status" : 200
    }
  }, {
    "create" : {
      "_index" : "movies",
      "_type" : "movie",
      "_id" : "3",
      "status" : 409,
      "error" : "DocumentAlreadyExistsException[[movies][4] [movie][3]: document alread\
y exists]"
    }
  }, {
    "delete" : {
      "_index" : "movies",
      "_type" : "movie",
      "_id" : "2",
```

```
      "_version" : 2,
      "status" : 200,
      "found" : true
    }
  }, {
    "update" : {
      "_index" : "movies",
      "_type" : "movie",
      "_id" : "4",
      "_version" : 4,
      "status" : 200
    }
  } ]
}
```

Take a look at the response above. As you can see the first operation, indexing the movie with ID 1 was successful. The second operation, creating the movie with ID 3 failed as the document already exists. Both the third and the fourth operations, deleting the movie with ID 2 and updating the movie with ID 4, succeeded.

Bulk and cURL

When using cURL the `-d` flag, which we normally use to send a request body, doesn't preserve new lines. In order to make requests to the `_bulk` endpoint we must instead use the `--data-binary` flag. Below is a full cURL command for performing the bulk request that we just looked at:

Using cURL to make a bulk request.

```
curl -XPOST localhost:9200/movies/movie/_bulk?pretty=true --data-binary '
{ "index" : { "_id" : "1" } }
{ "title": "The Godfather", "year": 1972 }
{ "create" : { "_id" : "3" } }
{ "title": "Lawrence of Arabia", "year": 1962 }
{ "delete" : { "_id" : "2" } }
{ "update" : { "_id" : "4" } }
{ "doc": { "year": 1979 }}
'
```

Sense won't recognize the `--data-binary` flag when pasting, but it does support performing bulk request. In order to run the above request in Sense we can replace "–data-binary" with "-d" prior to pasting it in Sense.

Multi get

While the bulk API enables us create, update and delete multiple documents it doesn't support retrieving multiple documents at once. We can of course do that using requests to the `_search` endpoint but if the only criteria for the document is their IDs ElasticSearch offers a more efficient and convenient way; the multi get API. Below is an example multi get request:

A request that retrieves two movie documents.

```
curl -XGET "http://localhost:9200/_mget" -d'
{
  "docs": [
    {
      "_index": "movies",
      "_type": "movie",
      "_id": "1"
    },
    {
      "_index": "movies",
      "_type": "movie",
      "_id": "2"
    }
  ]
}'
```

The response from ElasticSearch looks like this:

The response from ElasticSearch to the above _mget request.

```
{
  "docs": [
    {
      "_index": "movies",
      "_type": "movie",
      "_id": "1",
      "_version": 1,
      "found": true,
      "_source": {
        "title": "The Godfather",
        "director": "Francis Ford Coppola",
        "year": 1972,
        "genres": [
          "Crime",
          "Drama"
        ]
      }
    },
    {
      "_index": "movies",
      "_type": "movie",
      "_id": "2",
      "_version": 1,
      "found": true,
      "_source": {
        // Omitted for brevity
      }
```

```
    }
  ]
}
```

If we put the index name in the URL we can omit the `_index` parameters from the body. The same goes for the type name and the `_type` parameter. And, if we only want to retrieve documents of the same type we can skip the `docs` parameter all together and instead send a list of IDs:

Shorthand form of a _mget request.

```
curl -XGET "http://localhost:9200/movies/movie/_mget" -d'
{
  "ids": [ "1", "2" ]
}'
```

The multi get API also supports source filtering, returning only parts of the documents. For more about that and the multi get API in general, see the documentation[20].

Delete by query

Sometimes we may need to delete documents that match a certain criteria from an index. If we know the IDs of the documents we can of course use the `_bulk` API, but if we don't another API comes in handy; the delete by query API.

Through this API we can delete all documents that match a query. The query is expressed using ElasticSearch's query DSL which we learned about in chapter three. Below is an example request, deleting all movies from 1962.

A delete by query request, deleting all movies with year == 1962.

```
curl -XDELETE "http://localhost:9200/movies/movie/_query" -d'
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "year": 1962
        }
      }
    }
  }
}'
```

[20] http://www.elastic.co/guide/en/elasticsearch/reference/current/docs-multi-get.html

The type in the URL is optional but the index is not. However, we can perform the operation over all indexes by using the special index name `_all` if we really want to.

While it's possible to delete *everything* in an index by using delete by query it's far more efficient to simply delete the index and re-create it instead.

Time to live

Let's say that we're indexing content from a content management system. In the system content can have a date set after which it should no longer be considered published. If we're lucky there's some event that we can intercept when content is unpublished and when that happens delete the corresponding document from our index. However, that's not always the case.

This is one of many cases where documents in ElasticSearch has an expiration date and we'd like to tell ElasticSearch, at indexing time, that a document should be removed after a certain duration. ElasticSearch supports this by allowing us to specify a "time to live" for a document when indexing it. We do that by adding a `ttl` query string parameter to the URL. The value can either be a duration in milliseconds or a duration in text, such as "1w". Below is an example, indexing a movie with time to live:

Indexing a movie with an hours (60*60*1000 milliseconds) ttl.

```
curl -XPUT "http://localhost:9200/movies/movie/4?ttl=3600000" -d'
{
  "title": "Apocalypse Now",
  "director": "Francis Ford Coppola",
  "year": 1979,
  "genres": ["Drama", "War"]
}'
```

If we were to perform the above request and return an hour later we'd expect the document to be gone from the index. That wouldn't be the case though as the time to live functionality is disabled by default and needs to be activated on a per index basis through mappings. Here's how we enable it for the movies index:

Updating the movies index's mappings to enable ttl.

```
curl -XPUT "http://localhost:9200/movies/movie/_mapping" -d'
{
  "_ttl" : { "enabled" : true }
}'
```

Apart from the `enabled` property in the above request we can also send a parameter named `default` with a default ttl value. If we don't, like in the request above, only documents where we specify ttl during indexing will have a ttl value. Anyhow, if we now, with ttl enabled in

the mappings, index the movie with ttl again it will automatically be deleted after the specified duration.

The time to live functionality works by ElasticSearch regularly searching for documents that are due to expire, in indexes with ttl enabled, and deleting them. By default this is done once every 60 seconds. It's possible to change this interval if needed. For more information about how to do that, and about ttl in general, see the documentation[21].

As the ttl functionality requires ElasticSearch to regularly perform queries it's not the most efficient way if all you want to do is limit the size of the indexes in a cluster. When, for instance, storing only the last seven days of log data it's often better to use rolling indexes, such as one index per day and delete whole indexes when the data in them is no longer needed.

[21] http://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-ttl-field.html

The nested type mapping

Documents in ElasticSearch can contain properties with arrays or other JSON objects as values. In most cases, this just works. However, some times it doesn't. Let's again index a movie, only this time we'll add an array of actors to it and let each actor be a JSON object:

Indexing a movie with a 'cast' property.

```
curl -XPUT "http://localhost:9200/movies/movie/7" -d'
{
  "title": "The Matrix",
  "cast": [
    {
      "firstName": "Keanu",
      "lastName": "Reeves"
    },
    {
      "firstName": "Laurence",
      "lastName": "Fishburne"
    }
  ]
}'
```

Now, with movie indexed, we will get a hit for it if we search for movies with an actor whose first name is "Keanu". Or, rather the lowercased version, "keanu", if we filter on a field mapped with the standard analyzer.

Searching for movies, filtering on the cast.firstName fields.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "term": {
          "cast.firstName": "keanu"
        }
      }
    }
  }
}'
```

Running the above query indeed returns The Matrix. The same is true if we try to find movies that have an actor with the first name "Keanu" and last name "Reeves":

Filtering on both the cast.firstName and cast.lastName fields.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "bool": {
          "must": [
            {
              "term": {
                "cast.firstName": "keanu"
              }
            },
            {
              "term": {
                "cast.lastName": "reeves"
              }
            }
          ]
        }
      }
    }
  }
}'
```

The above request does indeed also result in a hit for The Matrix. All is well. Or, is it? Let's see what happens if we search for movies with an actor with "Keanu" as first name and "Fishburne" as last name.

Again filtering on both cast fields, but this time with a different value for cast.lastName.

```
curl -XPOST "http://localhost:9200/movies/movie/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "bool": {
          "must": [
            {
              "term": {
                "cast.firstName": "keanu"
```

```
            }
          },
          {
            "term": {
              "cast.lastName": "fishburne"
            }
          }
        ]
      }
    }
  }
}'
```

Clearly the above request should, at first glance, not return The Matrix as there's no such actor amongst its cast. However, ElasticSearch will return The Matrix for the above query. After all, the movie does contain an actor with "Keanu" as first name and, albeit a different one, an actor with "Fishburne" as last name. Based on the above query it has no way of knowing that we want the two term filters to match the same unique object in the list of actors. And even if it did, the way the data is indexed it wouldn't be able to handle that requirement.

When ElasticSearch indexes fields from JSON objects in an array the relationship of belonging to the same object is lost. In other words, the document will have fields and values like this:

Field	Value
title	"the", "matrix"
cast.firstName	"keanu", "laurence"
cast.lastName	"reeves", "fishburn"

So, what to do? Often the simplest solution is the best. We could prepare our index for this use case by adding a property with both the first name and last name to the actors, like this:

```
{
  "firstName": "Keanu",
  "lastName": "Reeves",
  "fullName": "Keanu Reeves"
}
```

Using this approach we could simply filter on the `fullName` property to find all movies starring an actor named Keanu Reeves, and no others. However, sometimes such a simple approach doesn't cut it. Luckily ElasticSearch provides a way for us to be able to filter on multiple fields within the same objects in arrays; mapping such fields as the `nested` type. To try this out, let's create ourselves a new index with the "actors" field mapped as nested.

Creating a new index with the cast field mapped as nested.

```
curl -XPUT "http://localhost:9200/movies-2" -d'
{
   "mappings": {
      "movie": {
         "properties": {
            "cast": {
               "type": "nested"
            }
         }
      }
   }
}'
```

After indexing the same movie document into the new index we can now find movies based on multiple properties of each actor by using a nested filter. Here's how we would search for movies starring an actor named "Keanu Fishburne":

Using a nested filter in order to filter on both the cast.firstName and cast.lastName fields *within the same objects in the cast field*.

```
curl -XPOST "http://localhost:9200/movies-2/movie/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "nested": {
          "path": "cast",
          "filter": {
            "bool": {
              "must": [
                {
                  "term": {
                    "firstName": "keanu"
                  }
                },
                {
                  "term": {
                    "lastName": "fishburne"
                  }
                }
              ]
            }
          }
        }
```

```
      }
    }
  }
}'
```

As you can see we've wrapped our initial bool filter in a nested filter. The nested filter contains a path property where we specify that the filter applies to the cast property of the searched document. It also contains a filter (or a query) which will be applied to each value within the nested property.

As intended, running the abobe query doesn't return The Matrix while modifying it to instead match "reeves" as last name will make it match The Matrix. However, there's one caveat.

Including nested values in parent documents

If we go back to our very first query, filtering only on actors first names without using a nested filter, like the request below, we won't get any hits.

```
curl -XPOST "http://localhost:9200/movies-2/movie/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "term": {
          "cast.firstName": "keanu"
        }
      }
    }
  }
}'
```

This happens because movie documents no longer have cast.firstName fields. Instead each element in the cast array is, internally in ElasticSearch, indexed as a separate document. Obviously we can still search for movies based only on first names amongst the cast, by using nested filters. Like this:

Searching for movies, filtering on the cast.firstName fields, in the index with the cast field mapped as nested.

```
curl -XPOST "http://localhost:9200/movies-2/movie/_search" -d'
{
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      },
      "filter": {
        "nested": {
          "path": "cast",
          "filter": {
            "term": {
              "firstName": "keanu"
            }
          }
        }
      }
    }
  }
}'
```

The above request returns The Matrix. However, sometimes having to use nested filters or queries when all we want to do is filter on a single property is a bit tedious. To be able to utilize the power of nested filters for complex criterias while still being able to filter on values in arrays the same way as if we hadn't mapped such properties as nested we can modify our mappings so that the nested values will also be included in the parent document. This is done using the `include_in_parent` parameter, like this:

Creating a new index with the cast field mapped as nested and with include_in_parent set to true.

```
curl -XPUT "http://localhost:9200/movies-3" -d'
{
  "mappings": {
    "movie": {
      "properties": {
        "cast": {
          "type": "nested",
          "include_in_parent": true
        }
      }
    }
  }
}'
```

In an index such as the one created with the above request we'll both be able to filter on combinations of values within the same complex objects in the actors array using nested filters

and still be able to filter on single fields without using nested filters. However, we now need to carefully consider where to use, and where to not use, nested filters in our queries as a query for "keanu fishburne" will match The Matrix using a regular bool filter while it won't when wrapping it in a nested filter. In other words, when using `include_in_parent` we may get unexpected results due to queries matching documents that it shouldn't if we forget to use nested filters.