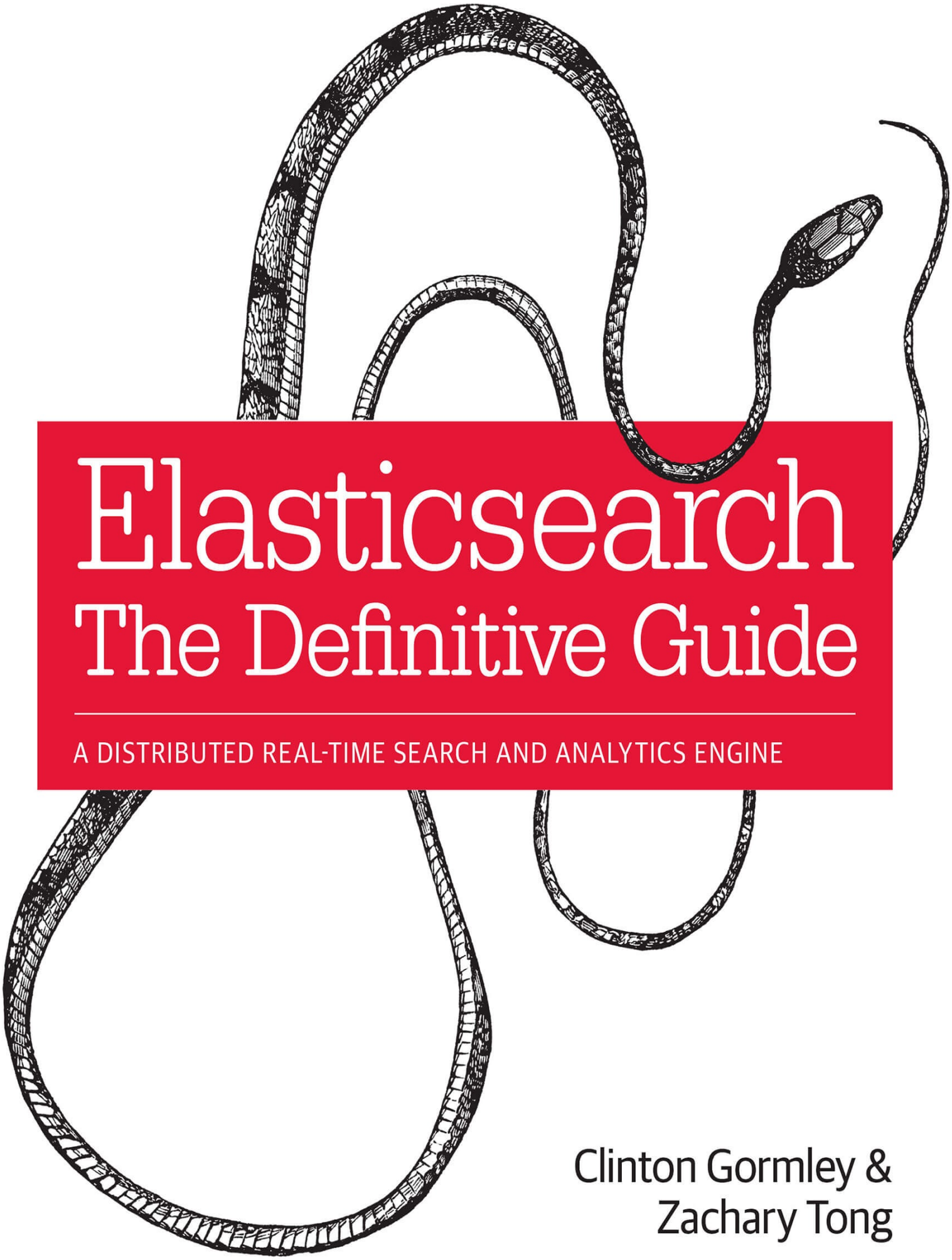
O'REILLY®



# Elasticsearch
## The Definitive Guide

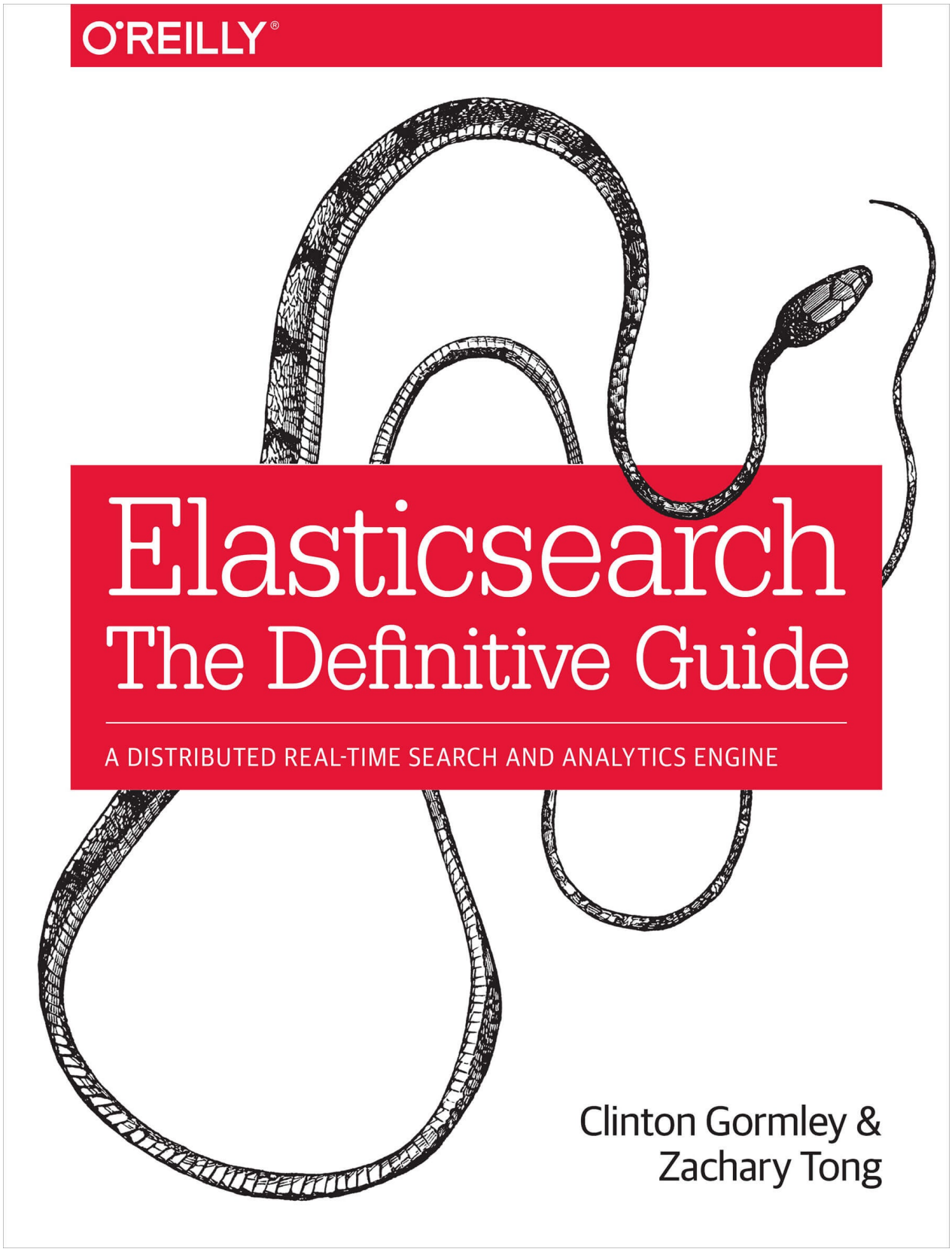
A DISTRIBUTED REAL-TIME SEARCH AND ANALYTICS ENGINE

Clinton Gormley &
Zachary Tong

# Table of Contents

# elasticsearch-definitive-guide-en

elasticsearch-definitive-guide-en Gitbook 英文版




原书地址：Elasticsearch the definitive guide

原作者： clinton gormley，zachary tong

## 项目地址：

https://github.com/allen8807/elasticsearch-definitive-guide-en

# Getting Started

*Elasticsearch* is a real-time distributed search and analytics engine. It allows you to explore your data at a speed and at a scale never before possible. It is used for full-text search, structured search, analytics, and all three in combination:

- Wikipedia uses Elasticsearch to provide full-text search with highlighted search snippets, and *search-as-you-type* and *did-you-mean* suggestions.
- *The Guardian* uses Elasticsearch to combine visitor logs with social -network data to provide real-time feedback to its editors about the public's response to new articles.
- Stack Overflow combines full-text search with geolocation queries and uses *more-like-this* to find related questions and answers.
- GitHub uses Elasticsearch to query 130 billion lines of code.

But Elasticsearch is not just for mega-corporations. It has enabled many startups like Datadog and Klout to prototype ideas and to turn them into scalable solutions. Elasticsearch can run on your laptop, or scale out to hundreds of servers and petabytes of data.

No individual part of Elasticsearch is new or revolutionary. Full-text search has been done before, as have analytics systems((("distributed databases")))((("analytics systems"))) and distributed databases. The revolution is the combination of these individually useful parts into a single, coherent, real-time application. It has a low barrier to entry for the new user, but can keep pace with you as your skills and needs grow.

If you are picking up this book, it is because you have data, and there is no point in having data unless you plan to *do something* with it.

Unfortunately, most databases are astonishingly inept at extracting actionable knowledge from your data.((("databases", "ineptness at extracting actionable data"))) Sure, they can filter by timestamp or exact values, but can they perform full-text search, handle synonyms, and score documents by relevance? Can they generate analytics and aggregations from the same data? Most important, can they do this in real time without big batch-processing jobs?

This is what sets Elasticsearch apart: Elasticsearch encourages you to explore and utilize your data, rather than letting it rot in a warehouse because it is too difficult to query.

Elasticsearch is your new best friend.

# You Know, for Search...

Elasticsearch is an open-source search engine built on top of Apache Lucene™, a full-text search-engine library. Lucene is arguably the most advanced, high-performance, and fully featured search engine library in existence today--both open source and proprietary.

But Lucene is just a library. To leverage its power, you need to work in Java and to integrate Lucene directly with your application. Worse, you will likely require a degree in information retrieval to understand how it works. Lucene is *very* complex.

Elasticsearch is also written in Java and uses Lucene internally for all of its indexing and searching, but it aims to make full-text search easy by hiding the complexities of Lucene behind a simple, coherent, RESTful API.

However, Elasticsearch is much more than just Lucene and much more than "just" full-text search. It can also be described as follows:

- A distributed real-time document store where *every field* is indexed and searchable
- A distributed search engine with real-time analytics
- Capable of scaling to hundreds of servers and petabytes of structured and unstructured data

And it packages up all this functionality into a standalone server that your application can talk to via a simple RESTful API, using a web client from your favorite programming language, or even from the command line.

It is easy to get started with Elasticsearch. It ships with sensible defaults and hides complicated search theory away from beginners. It *just works*, right out of the box. With minimal understanding, you can soon become productive.

Elasticsearch can be downloaded, used, and modified free of charge. It is available under the Apache 2 license, one of the most flexible open source licenses available.

As your knowledge grows, you can leverage more of Elasticsearch's advanced features. The entire engine is configurable and flexible. Pick and choose from the advanced features to tailor Elasticsearch to your problem domain.

**The Mists of Time**

---

Many years ago, a newly married unemployed developer called Shay Banon followed his wife to London, where she was studying to be a chef. While looking for gainful employment, he started playing with an early version of Lucene, with the intent of building his wife a recipe search engine.

Working directly with Lucene can be tricky, so Shay started work on an abstraction layer to make it easier for Java programmers to add search to their applications. He released this as his first open source project, called Compass.

Later Shay took a job working in a high-performance, distributed environment with in-memory data grids. The need for a high-performance, real-time, distributed search engine was obvious, and he decided to rewrite the Compass libraries as a standalone server called Elasticsearch.

The first public release came out in February 2010. Since then, Elasticsearch has become one of the most popular projects on GitHub with commits from over 300 contributors. A company has formed around Elasticsearch to provide commercial support and to develop new features, but Elasticsearch is, and forever will be, open source and available to all.

Shay's wife is still waiting for the recipe search...

---

# Installing Elasticsearch

The easiest way to understand what Elasticsearch can do for you is to play with it, so let's get started!

The only requirement for installing Elasticsearch is a recent version of Java. Preferably, you should install the latest version of the *www.java.com*.

You can download the latest version of Elasticsearch from *elasticsearch.org/download*.

```
curl -L -O http://download.elasticsearch.org/PATH/TO/VERSION.zip (1)
unzip elasticsearch-$VERSION.zip
cd  elasticsearch-$VERSION
```

(1) Fill in the URL for the latest version available on *elasticsearch.org/download*.

> TIP
>
> When installing Elasticsearch in production, you can use the method described previously, or the Debian or RPM packages provided on the downloads page. You can also use the officially supported Puppet module or Chef cookbook.

## Installing Marvel

Marvel is a management and monitoring tool for Elasticsearch, which is free for development use. It comes with an interactive console called Sense, which makes it easy to talk to Elasticsearch directly from your browser.

Many of the code examples in the online version of this book include a View in Sense link. When clicked, it will open up a working example of the code in the Sense console. You do not have to install Marvel, but it will make this book much more interactive by allowing you to experiment with the code samples on your local Elasticsearch cluster.

Marvel is available as a plug-in. To download and install it, run this command in the Elasticsearch directory:

```
./bin/plugin -i elasticsearch/marvel/latest
```

You probably don't want Marvel to monitor your local cluster, so you can disable data collection with this command:

```
echo 'marvel.agent.enabled: false' >> ./config/elasticsearch.yml
```

## Running Elasticsearch

Elasticsearch is now ready to run. You can start it up in the foreground with this:

```
./bin/elasticsearch
```

Add `-d` if you want to run it in the background as a daemon.

Test it out by opening another terminal window and running the following:

```
curl 'http://localhost:9200/?pretty'
```

You should see a response like this:

```
{
   "status": 200,
   "name": "Shrunken Bones",
   "version": {
      "number": "1.4.0",
      "lucene_version": "4.10"
   },
   "tagline": "You Know, for Search"
}
```

This means that your Elasticsearch *cluster* is up and running, and we can start experimenting with it.

> **NOTE**
>
> A *node* is a running instance of Elasticsearch. A *cluster* is a group of nodes with the same `cluster.name` that are working together to share data and to provide failover and scale, although a single node can form a cluster all by itself.

You should change the default `cluster.name` to something appropriate to you, like your own name, to stop your nodes from trying to join another cluster on the same network with the same name!

You can do this by editing the `elasticsearch.yml` file in the `config/` directory and then restarting Elasticsearch. When Elasticsearch is running in the foreground, you can stop it by pressing Ctrl-C; otherwise, you can shut it down with the `shutdown` API:

```
curl -XPOST 'http://localhost:9200/_shutdown'
```

## Viewing Marvel and Sense

If you installed the Marvel management and monitoring tool, you can view it in a web browser by visiting http://localhost:9200/_plugin/marvel/.

You can reach the *Sense* developer console either by clicking the "Marvel dashboards" drop-down in Marvel, or by visiting http://localhost:9200/_plugin/marvel/sense/.

# Talking to Elasticsearch

How you talk to Elasticsearch depends on whether you are using Java.

## Java API

If you are using Java, Elasticsearch comes with two built-in clients that you can use in your code:

***Node client***

- The node client joins a local cluster as a *non data node*. In other words, it doesn't hold any data itself, but it knows what data lives on which node in the cluster, and can forward requests directly to the correct node.

***Transport client***

- The lighter-weight transport client can be used to send requests to a remote cluster. It doesn't join the cluster itself, but simply forwards requests to a node in the cluster.

Both Java clients talk to the cluster over port *9300*, using the native Elasticsearch *transport* protocol. The nodes in the cluster also communicate with each other over port 9300. If this port is not open, your nodes will not be able to form a cluster.

> **TIP**
>
> The Java client must be from the same version of Elasticsearch as the nodes;otherwise, they may not be able to understand each other.

More information about the Java clients can be found in the Java API section of the Guide.

## RESTful API with JSON over HTTP

All other languages can communicate with Elasticsearch over port *9200* using a RESTful API, accessible with your favorite web client. In fact, as you have seen, you can even talk to Elasticsearch from the command line by using the `curl` command.

> NOTE
>
> Elasticsearch provides official clients for several languages--Groovy, JavaScript, .NET, PHP, Perl, Python, and Ruby--and there are numerous community-provided clients and integrations, all of which can be found in the Guide.

A request to Elasticsearch consists of the same parts as any HTTP request:

```
curl -X<VERB> '<PROTOCOL>://<HOST>/<PATH>?<QUERY_STRING>' -d '<BODY>'
```

The parts marked with `< >` above are:

| | |
|---|---|
| `VERB` | The appropriate HTTP _method_ or _verb_: `GET`, `POST`, `PUT`, `HEAD`, or `DELETE`. |

| | |
|---|---|
| `PROTOCOL` | Either `http` or `https` (if you have an `https` proxy in front of Elasticsearch.) |
| `HOST` | The hostname of any node in your Elasticsearch cluster, or +localhost+ for a node on your local machine. |
| `PORT` | The port running the Elasticsearch HTTP service, which defaults to `9200`. |
| `QUERY_STRING` | Any optional query-string parameters (for example `?pretty` will _pretty-print_ the JSON response to make it easier to read.) |
| `BODY` | A JSON-encoded request body (if the request needs one.) |

For instance, to count the number of documents in the cluster, we could use this:

```
curl -XGET 'http://localhost:9200/_count?pretty' -d '
{
    "query": {
        "match_all": {}
    }
}
'
```

Elasticsearch returns an HTTP status code like `200 OK` and (except for `HEAD` requests) a JSON-encoded response body. The preceding `curl` request would respond with a JSON body like the following:

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

We don't see the HTTP headers in the response because we didn't ask `curl` to display them. To see the headers, use the `curl` command with the `-i` switch:

```
curl -i -XGET 'localhost:9200/'
```

For the rest of the book, we will show these `curl` examples using a shorthand format that leaves out all the bits that are the same in every request, like the hostname and port, and the `curl` command itself. Instead of showing a full request like

```
curl -XGET 'localhost:9200/_count?pretty' -d '
{
    "query": {
        "match_all": {}
    }
}'
```

we will show it in this shorthand format:

```
GET /_count
{
    "query": {
        "match_all": {}
    }
}
```

In fact, this is the same format that is used by the Sense console that we installed with Marvel. If in the online version of this book, you can open and run this code example in Sense by clicking the View in Sense link above.

# Document Oriented

Objects in an application are seldom just a simple list of keys and values. More often than not, they are complex data structures that may contain dates, geo locations, other objects, or arrays of values.

Sooner or later you're going to want to store these objects in a database. Trying to do this with the rows and columns of a relational database is the equivalent of trying to squeeze your rich, expressive objects into a very big spreadsheet: you have to flatten the object to fit the table schema--usually one field per column--and then have to reconstruct it every time you retrieve it.

Elasticsearch is *document oriented*, meaning that it stores entire objects or *documents*. It not only stores them, but also *indexes* the contents of each document in order to make them searchable. In Elasticsearch, you index, search, sort, and filter documents--not rows of columnar data. This is a fundamentally different way of thinking about data and is one of the reasons Elasticsearch can perform complex full-text search.

## JSON

Elasticsearch uses JavaScript Object Notation, or *JSON*, as the serialization format for documents. JSON serialization is supported by most programming languages, and has become the standard format used by the NoSQL movement. It is simple, concise, and easy to read.

Consider this JSON document, which represents a user object:

```
{
    "email":      "john@smith.com",
    "first_name": "John",
    "last_name":  "Smith",
    "info": {
        "bio":         "Eco-warrior and defender of the weak",
        "age":         25,
        "interests": [ "dolphins", "whales" ]
    },
    "join_date": "2014/05/01"
}
```

Although the original `user` object was complex, the structure and meaning of the object has been retained in the JSON version. Converting an object to JSON for indexing in Elasticsearch is much simpler than the equivalent process for a flat table structure.

> NOTE
>
> Almost all languages have modules that will convert arbitrary data structures or objects into JSON for you, but the details are specific to each language. Look for modules that handle JSON *serialization* or *marshalling*. The official Elasticsearch clients all handle conversion to and from JSON for you automatically.

# Finding Your Feet

To give you a feel for what is possible in Elasticsearch and how easy it is to use, let's start by walking through a simple tutorial that covers basic concepts such as indexing, search, and aggregations.

We'll introduce some new terminology and basic concepts along the way, but it is OK if you don't understand everything immediately. We'll cover all the concepts introduced here in *much* greater depth throughout the rest of the book.

So, sit back and enjoy a whirlwind tour of what Elasticsearch is capable of.

## Let's Build an Employee Directory

We happen to work for *Megacorp*, and as part of HR's new *"We love our drones!"* initiative, we have been tasked with creating an employee directory. The directory is supposed to foster employer empathy and real-time, synergistic, dynamic collaboration, so it has a few business requirements:

- Enable data to contain multi value tags, numbers, and full text.
- Retrieve the full details of any employee.
- Allow structured search, such as finding employees over the age of 30.
- Allow simple full-text search and more-complex *phrase* searches.
- Return highlighted search *snippets* from the text in the matching documents.
- Enable management to build analytic dashboards over the data.

# Indexing Employee Documents

The first order of business is storing employee data. This will take the form of an *employee document*: a single document represents a single employee. The act of storing data in Elasticsearch is called *indexing*, but before we can index a document, we need to decide *where* to store it.

In Elasticsearch, a document belongs to a *type*, and those types live inside an *index*. You can draw some (rough) parallels to a traditional relational database:

```
Relational DB  ⇒ Databases ⇒ Tables ⇒ Rows      ⇒ Columns
Elasticsearch  ⇒ Indices   ⇒ Types  ⇒ Documents ⇒ Fields
```

An Elasticsearch cluster can contain multiple *indices* (databases), which in turn contain multiple *types* (tables). These types hold multiple *documents* (rows), and each document has multiple *fields* (columns).

## Index Versus Index Versus Index

---

You may already have noticed that the word *index* is overloaded with several meanings in the context of Elasticsearch. A little clarification is necessary:

***Index (noun)***

- As explained previously, an *index* is like a *database* in a traditional relational database. It is the place to store related documents. The plural of *index* is *indices* or *indexes*.

***Index (verb)***

- *To index a document* is to store a document in an *index (noun)* so that it can be retrieved and queried. It is much like the `INSERT` keyword in SQL except that, if the document already exists, the new document would replace the old.

***Inverted index***

- Relational databases add an *index*, such as a B-tree index, to specific columns in order to improve the speed of data retrieval. Elasticsearch and Lucene use a structure called an *inverted index* for exactly the same purpose.
- By default, every field in a document is *indexed* (has an inverted index) and thus is searchable. A field without an inverted index is not searchable. We discuss inverted indexes in more detail in inverted-index.

---

So for our employee directory, we are going to do the following:

- Index a *document* per employee, which contains all the details of a single employee.
- Each document will be of *type* `employee`.
- That type will live in the `megacorp` *index*.
- That index will reside within our Elasticsearch cluster.

In practice, this is easy (even though it looks like a lot of steps). We can perform all of those actions in a single command:

```
PUT /megacorp/employee/1
{
    "first_name" : "John",
    "last_name" :  "Smith",
    "age" :        25,
    "about" :      "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

Notice that the path `/megacorp/employee/1` contains three pieces of information:

**megacorp**

- The index name

**employee**

- The type name

**1**

- The ID of this particular employee

The request body--the JSON document--contains all the information about this employee. His name is John Smith, he's 25, and enjoys rock climbing.

Simple! There was no need to perform any administrative tasks first, like creating an index or specifying the type of data that each field contains. We could just index a document directly. Elasticsearch ships with defaults for everything, so all the necessary administration tasks were taken care of in the background, using default values.

Before moving on, let's add a few more employees to the directory:

```
PUT /megacorp/employee/2
{
    "first_name" :  "Jane",
    "last_name" :   "Smith",
```

```
    "age" :         32,
    "about" :       "I like to collect rock albums",
    "interests":  [ "music" ]
}

PUT /megacorp/employee/3
{
    "first_name" :  "Douglas",
    "last_name" :   "Fir",
    "age" :         35,
    "about":        "I like to build cabinets",
    "interests":  [ "forestry" ]
}
```

# Retrieving a Document

Now that we have some data stored in Elasticsearch, we can get to work on the business requirements for this application. The first requirement is the ability to retrieve individual employee data.

This is easy in Elasticsearch. We simply execute an HTTP +GET+ request and specify the *address* of the document--the index, type, and ID. Using those three pieces of information, we can return the original JSON document:

```
GET /megacorp/employee/1
```

And the response contains some metadata about the document, and John Smith's original JSON document as the `_source` field:

```
{
  "_index" :   "megacorp",
  "_type" :    "employee",
  "_id" :      "1",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "first_name" :  "John",
      "last_name" :   "Smith",
      "age" :         25,
      "about" :       "I love to go rock climbing",
      "interests":  [ "sports", "music" ]
  }
}
```

> **TIP**
>
> In the same way that we changed the HTTP verb from `PUT` to `GET` in order to retrieve the document, we could use the `DELETE` verb to delete the document, and the `HEAD` verb to check whether the document exists. To replace an existing document with an updated version, we just `PUT` it again.

# Search Lite

A `GET` is fairly simple--you get back the document that you ask for. Let's try something a little more advanced, like a simple search!

The first search we will try is the simplest search possible. We will search for all employees, with this request:

```
GET /megacorp/employee/_search
```

You can see that we're still using index `megacorp` and type `employee`, but instead of specifying a document ID, we now use the `_search` endpoint. The response includes all three of our documents in the `hits` array. By default, a search will return the top 10 results.

```
{
   "took":      6,
   "timed_out": false,
```

```
   "_shards": { ... },
   "hits": {
      "total":      3,
      "max_score":  1,
      "hits": [
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "3",
            "_score":         1,
            "_source": {
               "first_name":  "Douglas",
               "last_name":   "Fir",
               "age":         35,
               "about":       "I like to build cabinets",
               "interests": [ "forestry" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "1",
            "_score":         1,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "2",
            "_score":         1,
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

> **NOTE**
>
> The response not only tells us which documents matched, but also includes the whole document itself: all the information that we need in order to display the search results to the user.

Next, let's try searching for employees who have "Smith" in their last name. To do this, we'll use a *lightweight* search method that is easy to use from the command line. This method is often referred to as a *query-string* search, since we pass the search as a URL query-string parameter:

```
GET /megacorp/employee/_search?q=last_name:Smith
```

We use the same `_search` endpoint in the path, and we add the query itself in the `q=` parameter. The results that come back show all Smiths:

```
{
   ...
```

```
   "hits": {
      "total":      2,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

# Search with Query DSL

Query-string search is handy for ad hoc searches from the command line, but it has its limitations (see *search-lite*). Elasticsearch provides a rich, flexible, query language called the *query DSL*, which allows us to build much more complicated, robust queries.

The *domain-specific language* (DSL) is specified using a JSON request body. We can represent the previous search for all Smiths like so:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "last_name" : "Smith"
        }
    }
}
```

This will return the same results as the previous query. You can see that a number of things have changed. For one, we are no longer using *query-string* parameters, but instead a request body. This request body is built with JSON, and uses a `match` query (one of several types of queries, which we will learn about later).

# More-Complicated Searches

Let's make the search a little more complicated. We still want to find all employees with a last name of Smith, but we want only employees who are older than 30. Our query will change a little to accommodate a *filter*, which allows us to execute structured searches efficiently:

```
GET /megacorp/employee/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "range" : {
```

```
                    "age" : { "gt" : 30 } (1)
                }
            },
            "query" : {
                "match" : {
                    "last_name" : "smith" (2)
                }
            }
        }
    }
}
```

(1) This portion of the query is a `range` *filter*, which will find all ages older than 30— `gt` stands for *greater than*. (2) This portion of the query is the(("match queries"))) same `match` *query* that we used before.

Don't worry about the syntax too much for now; we will cover it in great detail later. Just recognize that we've added a *filter* that performs a range search, and reused the same `match` query as before. Now our results show only one employee who happens to be 32 and is named Jane Smith:

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

# Full-Text Search

The searches so far have been simple: single names, filtered by age. Let's try a more advanced, full-text search--a task that traditional databases would really struggle with.

We are going to search for all employees who enjoy rock climbing:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "about" : "rock climbing"
        }
    }
}
```

You can see that we use the same `match` query as before to search the `about` field for ``rock climbing.'' We get back two matching documents:

```
{
   ...
   "hits": {
```

```
      "total":      2,
      "max_score":  0.16273327,
      "hits": [
         {
            ...
            "_score":         0.16273327, (1)
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_score":         0.016878016, (1)
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

(1) The relevance scores

By default, Elasticsearch sorts matching results by their relevance score, that is, by how well each document matches the query. The first and highest-scoring result is obvious: John Smith's `about` field clearly says "rock climbing" in it.

But why did Jane Smith come back as a result? The reason her document was returned is because the word "rock" was mentioned in her `about` field. Because only "rock" was mentioned, and not "climbing," her `_score` is lower than John's.

This is a good example of how Elasticsearch can search *within* full-text fields and return the most relevant results first. This ((("relevance", "importance to Elasticsearch")))concept of *relevance* is important to Elasticsearch, and is a concept that is completely foreign to traditional relational databases, in which a record either matches or it doesn't.

# Phrase Search

Finding individual words in a field is all well and good, but sometimes you want to match exact sequences of words or *phrases*.((("phrase matching"))) For instance, we could perform a query that will match only employee records that contain both `rock'' _and_` climbing" *and* that display the words are next to each other in the phrase ``rock climbing."

To do this, we use a slight variation of the `match` query called the `match_phrase` query:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    }
}
```

This, to no surprise, returns only John Smith's document:

```
{
   ...
```

```
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         }
      ]
   }
}
```

## Highlighting Our Searches

Many applications like to *highlight* snippets of text from each search result so the user can see *why* the document matched the query. Retrieving highlighted fragments is easy in Elasticsearch.

Let's rerun our previous query, but add a new `highlight` parameter:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    },
    "highlight": {
        "fields" : {
            "about" : {}
        }
    }
}
```

When we run this query, the same hit is returned as before, but now we get a new section in the response called `highlight`. This contains a snippet of text from the `about` field with the matching words wrapped in `<em></em>` HTML tags:

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            },
            "highlight": {
               "about": [
                  "I love to go <em>rock</em> <em>climbing</em>" (1)
               ]
            }
         }
      ]
   }
}
```

(1) The highlighted fragment from the original text

You can read more about the highlighting of search snippets in the highlighting reference documentation.

# Analytics

Finally, we come to our last business requirement: allow managers to run analytics over the employee directory. Elasticsearch has functionality called *aggregations*, which allow you to generate sophisticated analytics over your data. It is similar to `GROUP BY` in SQL, but much more powerful.

For example, let's find the most popular interests enjoyed by our employees:

```
GET /megacorp/employee/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

Ignore the syntax for now and just look at the results:

```
{
   ...
   "hits": { ... },
   "aggregations": {
      "all_interests": {
         "buckets": [
            {
               "key":       "music",
               "doc_count": 2
            },
            {
               "key":       "forestry",
               "doc_count": 1
            },
            {
               "key":       "sports",
               "doc_count": 1
            }
         ]
      }
   }
}
```

We can see that two employees are interested in music, one in forestry, and one in sports. These aggregations are not precalculated; they are generated on the fly from the documents that match the current query. If we want to know the popular interests of people called Smith, we can just add the appropriate query into the mix:

```
GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
        "field": "interests"
      }
    }
  }
}
```

The `all_interests` aggregation has changed to include only documents matching our query:

```
  ...
  "all_interests": {
     "buckets": [
        {
           "key": "music",
           "doc_count": 2
        },
        {
           "key": "sports",
           "doc_count": 1
        }
     ]
  }
```

Aggregations allow hierarchical rollups too. For example, let's find the average age of employees who share a particular interest:

```
GET /megacorp/employee/_search
{
    "aggs" : {
        "all_interests" : {
            "terms" : { "field" : "interests" },
            "aggs" : {
                "avg_age" : {
                    "avg" : { "field" : "age" }
                }
            }
        }
    }
}
```

The aggregations that we get back are a bit more complicated, but still fairly easy to understand:

```
  ...
  "all_interests": {
     "buckets": [
        {
           "key": "music",
           "doc_count": 2,
           "avg_age": {
              "value": 28.5
           }
        },
        {
           "key": "forestry",
           "doc_count": 1,
           "avg_age": {
              "value": 35
           }
        },
        {
           "key": "sports",
           "doc_count": 1,
           "avg_age": {
              "value": 25
           }
        }
     ]
  }
```

The output is basically an enriched version of the first aggregation we ran. We still have a list of interests and their counts, but now each interest has an additional `avg_age`, which shows the average age for all employees having that interest.

Even if you don't understand the syntax yet, you can easily see how complex aggregations and groupings can be accomplished using this feature. The sky is the limit as to what kind of data you can extract!

# Tutorial Conclusion

Hopefully, this little tutorial was a good demonstration about what is possible in Elasticsearch. It is really just scratching the surface, and many features--such as suggestions, geolocation, percolation, fuzzy and partial matching--were omitted to keep the tutorial short. But it did highlight just how easy it is to start building advanced search functionality. No configuration was needed--just add data and start searching!

It's likely that the syntax left you confused in places, and you may have questions about how to tweak and tune various aspects. That's fine! The rest of the book dives into each of these issues in detail, giving you a solid understanding of how Elasticsearch works.

# Distributed Nature

At the beginning of this chapter, we said that Elasticsearch can scale out to hundreds (or even thousands) of servers and handle petabytes of data. While our tutorial gave examples of how to use Elasticsearch, it didn't touch on the mechanics at all. Elasticsearch is distributed by nature, and it is designed to hide the complexity that comes with being distributed.

The distributed aspect of Elasticsearch is largely transparent. Nothing in the tutorial required you to know about distributed systems, sharding, cluster discovery, or dozens of other distributed concepts. It happily ran the tutorial on a single node living inside your laptop, but if you were to run the tutorial on a cluster containing 100 nodes, everything would work in exactly the same way.

Elasticsearch tries hard to hide the complexity of distributed systems. Here are some of the operations happening automatically under the hood:

- Partitioning your documents into different containers or *shards*, which can be stored on a single node or on multiple nodes
- Balancing these shards across the nodes in your cluster to spread the indexing and search load
- Duplicating each shard to provide redundant copies of your data, to prevent data loss in case of hardware failure
- Routing requests from any node in the cluster to the nodes that hold the data you're interested in
- Seamlessly integrating new nodes as your cluster grows or redistributing shards to recover from node loss

As you read through this book, you'll encounter supplemental chapters about the distributed nature of Elasticsearch. These chapters will teach you about how the cluster scales and deals with failover (*distributed-cluster*), handles document storage (*distributed-docs*), executes distributed search (*distributed-search*), and what a shard is and how it works (*inside-a-shard*).

These chapters are not required reading--you can use Elasticsearch without understanding these internals--but they will provide insight that will make your knowledge of Elasticsearch more complete. Feel free to skim them and revisit at a later point when you need a more complete understanding.

## Next Steps

By now you should have a taste of what you can do with Elasticsearch, and how easy it is to get started. Elasticsearch tries hard to work out of the box with minimal knowledge and configuration. The best way to learn Elasticsearch is by jumping in: just start indexing and searching!

However, the more you know about Elasticsearch, the more productive you can become. The more you can tell Elasticsearch about the domain-specific elements of your application, the more you can fine-tune the output.

The rest of this book will help you move from novice to expert. Each chapter explains the essentials, but also includes expert-level tips. If you're just getting started, these tips are probably not immediately relevant to you; Elasticsearch has sensible defaults and will generally do the right thing without any interference. You can always revisit these chapters later, when you are looking to improve performance by shaving off any wasted milliseconds.

# Life Inside a Cluster

## Supplemental Chapter

As mentioned earlier, this is the first of several supplemental chapters about how Elasticsearch operates in a distributed environment. In this chapter, we explain commonly used terminology like *cluster*, *node*, and *shard*, the mechanics of how Elasticsearch scales out, and how it deals with hardware failure.

Although this chapter is not required reading--you can use Elasticsearch for a long time without worrying about shards, replication, and failover--it will help you to understand the processes at work inside Elasticsearch. Feel free to skim through the chapter and to refer to it again later.

Elasticsearch is built to be always available, and to scale with your needs. Scale can come from buying bigger servers (*vertical scale*, or *scaling up*) or from buying more servers (*horizontal scale*, or *scaling out*).

While Elasticsearch can benefit from more-powerful hardware, vertical scale has its limits. Real scalability comes from horizontal scale--the ability to add more nodes to the cluster and to spread load and reliability between them.

With most databases, scaling horizontally usually requires a major overhaul of your application to take advantage of these extra boxes. In contrast, Elasticsearch is *distributed* by nature: it knows how to manage multiple nodes to provide scale and high availability. This also means that your application doesn't need to care about it.

In this chapter, we show how you can set up your cluster, nodes, and shards to scale with your needs and to ensure that your data is safe from hardware failure.

# An Empty Cluster

If we start a single node, with no data and no indices, our cluster looks like *img-cluster*.

## A cluster with one empty node


Figure 1. A cluster with one empty node

A *node* is a running instance of Elasticsearch, while a *cluster* consists of one or more nodes with the same `cluster.name` that are working together to share their data and workload. As nodes are added to or removed from the cluster, the cluster reorganizes itself to spread the data evenly.

One node in the cluster is elected to be the *master* node, which is in charge of managing cluster-wide changes like creating or deleting an index, or adding or removing a node from the cluster. The master node does not need to be involved in document-level changes or searches, which means that having just one master node will not become a bottleneck as traffic grows. Any node can become the master. Our example cluster has only one node, so it performs the master role.

As users, we can talk to *any node in the cluster*, including the master node. Every node knows where each document lives and can forward our request directly to the nodes that hold the data we are interested in. Whichever node we talk to manages the process of gathering the response from the node or nodes holding the data and returning the final response to the client. It is all managed transparently by Elasticsearch.

# Cluster Health

Many statistics can be monitored in an Elasticsearch cluster, but the single most important one is *cluster health*, which reports a `status` of either `green`, `yellow`, or `red`:

```
GET /_cluster/health
```

On an empty cluster with no indices, this will return something like the following:

```
{
   "cluster_name":          "elasticsearch",
   "status":                "green", (1)
   "timed_out":             false,
   "number_of_nodes":       1,
   "number_of_data_nodes":  1,
   "active_primary_shards": 0,
   "active_shards":         0,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     0
}
```

(1) The `status` field is the one we're most interested in.

The `status` field provides an overall indication of how the cluster is functioning. The meanings of the three colors are provided here for reference:

***green***

- All primary and replica shards are active.

***yellow***

- All primary shards are active, but not all replica shards are active.

***red***

- Not all primary shards are active.

In the rest of this chapter, we explain what *primary* and *replica* shards are and explain the practical implications of each of the preceding colors.

# Add an Index

To add data to Elasticsearch, we need an *index*—a place to store related data. In reality, an index is just a *logical namespace* that points to one or more physical *shards*.

A *shard* is a low-level *worker unit* that holds just a slice of all the data in the index. In **inside-a-shard**, we explain in detail how a shard works, but for now it is enough to know that a shard is a single instance of Lucene, and is a complete search engine in its own right. Our documents are stored and indexed in shards, but our applications don't talk to them directly. Instead, they talk to an index.

Shards are how Elasticsearch distributes data around your cluster. Think of shards as containers for data. Documents are stored in shards, and shards are allocated to nodes in your cluster. As your cluster grows or shrinks, Elasticsearch will automatically migrate shards between nodes so that the cluster remains balanced.

A shard can be either a *primary* shard or a *replica* shard. Each document in your index belongs to a single primary shard, so the number of primary shards that you have determines the maximum amount of data that your index can hold.

> NOTE
>
> While there is no theoretical limit to the amount of data that a primary shard can hold, there is a practical limit. What constitutes the maximum shard size depends entirely on your use case: the hardware you have, the size and complexity of your documents, how you index and query your documents, and your expected response times.

A replica shard is just a copy of a primary shard. Replicas are used to provide redundant copies of your data to protect against hardware failure, and to serve read requests like searching or retrieving a document.

The number of primary shards in an index is fixed at the time that an index is created, but the number of replica shards can be changed at any time.

Let's create an index called `blogs` in our empty one-node cluster. By default, indices are assigned five primary shards, but for the purpose of this demonstration, we'll assign just three primary shards and one replica (one replica of every primary shard):

```
PUT /blogs
{
   "settings" : {
      "number_of_shards" : 3,
      "number_of_replicas" : 1
   }
}
```

Our cluster now looks like **cluster-one-node**. All three primary shards have been allocated to `Node 1`.

## A single-node cluster with an index


Figure 2.3.1. A single-node cluster with an index

If we were to check the **cluster-health** now, we would see this:

```
{
   "cluster_name":          "elasticsearch",
   "status":                "yellow", (1)
   "timed_out":             false,
   "number_of_nodes":       1,
   "number_of_data_nodes":  1,
   "active_primary_shards": 3,
   "active_shards":         3,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     3 (2)
}
```

(1) Cluster `status` is `yellow`.

(2) Our three replica shards have not been allocated to a node.

A cluster health of `yellow` means that all *primary* shards are up and running (the cluster is capable of serving any request successfully) but not all *replica* shards are active. In fact, all three of our replica shards are currently `unassigned`—they haven't been allocated to a node. It doesn't make sense to store copies of the same data on the same node. If we were to lose that node, we would lose all copies of our data.

Currently, our cluster is fully functional but at risk of data loss in case of hardware failure.

# Add Failover

Running a single node means that you have a single point of failure--there is no redundancy. Fortunately, all we need to do to protect ourselves from data loss is to start another node.

## Starting a Second Node

To test what happens when you add a second node, you can start a new node in exactly the same way as you started the first one (see running-elasticsearch), and from the same directory. Multiple nodes can share the same directory.

As long as the second node has the same `cluster.name` as the first node (see the `./config/elasticsearch.yml` file), it should automatically discover and join the cluster run by the first node. If it doesn't, check the logs to find out what went wrong. It may be that multicast is disabled on your network, or that a firewall is preventing your nodes from communicating.

If we start a second node, our cluster would look like **cluster-two-nodes**.

## A two-node cluster--all primary and replica shards are allocated

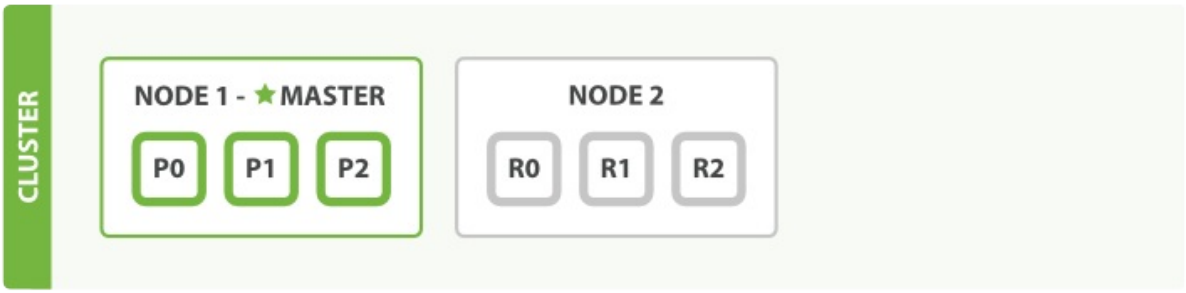



Figure 2.4.1. A two-node cluster--all primary and replica shards are allocated

The second node has joined the cluster, and three *replica shards* have been allocated to it--one for each primary shard. That means that we can lose either node, and all of our data will be intact.

Any newly indexed document will first be stored on a primary shard, and then copied in parallel to the associated replica shard(s). This ensures that our document can be retrieved from a primary shard or from any of its replicas.

The `cluster-health` now shows a status of `green`, which means that all six shards (all three primary shards and all three replica shards) are active:

```
{
   "cluster_name":          "elasticsearch",
   "status":                "green", (1)
   "timed_out":             false,
   "number_of_nodes":       2,
   "number_of_data_nodes":  2,
   "active_primary_shards": 3,
   "active_shards":         6,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     0
}
```

(1) Cluster `status` is `green`.

Our cluster is not only fully functional, but also *always available*.

# Scale Horizontally

What about scaling as the demand for our application grows? If we start a third node, our cluster reorganizes itself to look like **cluster-three-nodes**.

## A three-node cluster--shards have been reallocated to spread the load

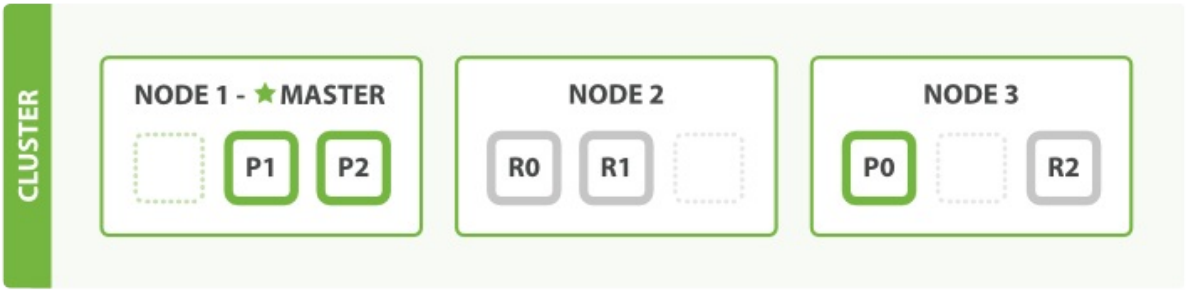



Figure 2.5.1. A three-node cluster--shards have been reallocated to spread the load

One shard each from `Node 1` and `Node 2` have moved to the new `Node 3`, and we have two shards per node, instead of three. This means that the hardware resources (CPU, RAM, I/O) of each node are being shared among fewer shards, allowing each shard to perform better.

A shard is a fully fledged search engine in its own right, and is capable of using all of the resources of a single node. With our total of six shards (three primaries and three replicas), our index is capable of scaling out to a maximum of six nodes, with one shard on each node and each shard having access to 100% of its node's resources.

# Then Scale Some More

But what if we want to scale our search to more than six nodes?

The number of primary shards is fixed at the moment an index is created. Effectively, that number defines the maximum amount of data that can be *stored* in the index. (The actual number depends on your data, your hardware and your use case.) However, read requests--searches or document retrieval--can be handled by a primary *or* a replica shard, so the more copies of data that you have, the more search throughput you can handle.

The number of replica shards can be changed dynamically on a live cluster, allowing us to scale up or down as demand requires. Let's increase the number of replicas from the default of `1` to `2`:

```
PUT /blogs/_settings
{
   "number_of_replicas" : 2
}
```

As can be seen in **cluster-three-nodes-two-replicas**, the `blogs` index now has nine shards: three primaries and six replicas. This means that we can scale out to a total of nine nodes, again with one shard per node. This would allow us to *triple* search performance compared to our original three-node cluster.

## Increasing the `number_of_replicas` to 2

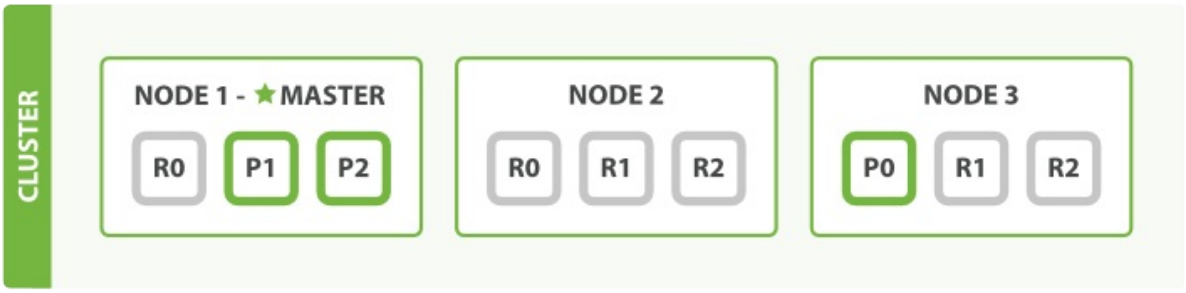



Figure 2.6.1.A three-node cluster with two replica shards

> NOTE
>
> Of course, just having more replica shards on the same number of nodes doesn't increase our performance at all because each shard has access to a smaller fraction of its node's resources. You need to add hardware to increase throughput.
>
> But these extra replicas do mean that we have more redundancy: with the node configuration above, we can now afford to lose two nodes without losing any data.

# Coping with Failure

We've said that Elasticsearch can cope when nodes fail, so let's go ahead and try it out. If we kill the first node, our cluster looks like **cluster-post-kill**.

## Cluster after killing one node

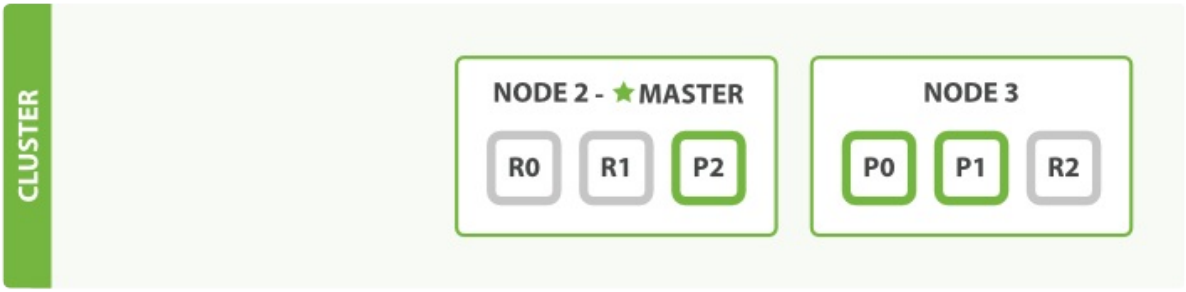



Figure 2.7.1.Cluster after killing one node

The node we killed was the master node. A cluster must have a master node in order to function correctly, so the first thing that happened was that the nodes elected a new master: `Node 2`.

Primary shards `1` and `2` were lost when we killed `Node 1`, and our index cannot function properly if it is missing primary shards. If we had checked the cluster health at this point, we would have seen status `red`: not all primary shards are active!

Fortunately, a complete copy of the two lost primary shards exists on other nodes, so the first thing that the new master node did was to promote the replicas of these shards on `Node 2` and `Node 3` to be primaries, putting us back into cluster health `yellow`. This promotion process was instantaneous, like the flick of a switch.

So why is our cluster health `yellow` and not `green`? We have all three primary shards, but we specified that we wanted two replicas of each primary, and currently only one replica is assigned. This prevents us from reaching `green`, but we're not too worried here: were we to kill `Node 2` as well, our application could *still* keep running without data loss, because `Node 3` contains a copy of every shard.

If we restart `Node 1`, the cluster would be able to allocate the missing replica shards, resulting in a state similar to the one described in **cluster-three-nodes-two-replicas**. If `Node 1` still has copies of the old shards, it will try to reuse them, copying over from the primary shard only the files that have changed in the meantime.

By now, you should have a reasonable idea of how shards allow Elasticsearch to scale horizontally and to ensure that your data is safe. Later we will examine the life cycle of a shard in more detail.

# Data In, Data Out

Whatever program we write, the intention is the same: to organize data in a way that serves our purposes. But data doesn't consist of just random bits and bytes. We build relationships between data elements in order to represent entities, or *things* that exist in the real world. A name and an email address have more meaning if we know that they belong to the same person.

In the real world, though, not all entities of the same type look the same. One person might have a home telephone number, while another person has only a cell-phone number, and another might have both. One person might have three email addresses, while another has none. A Spanish person will probably have two last names, while an English person will probably have only one.

One of the reasons that object-oriented programming languages are so popular is that objects help us represent and manipulate real-world entities with potentially complex data structures. So far, so good.

The problem comes when we need to store these entities. Traditionally, we have stored our data in columns and rows in a relational database, the equivalent of using a spreadsheet. All the flexibility gained from using objects is lost because of the inflexibility of our storage medium.

But what if we could store our objects as objects? Instead of modeling our application around the limitations of spreadsheets, we can instead focus on *using* the data. The flexibility of objects is returned to us.

An *object* is a language-specific, in-memory data structure. To send it across the network or store it, we need to be able to represent it in some standard format. JSON is a way of representing objects in human-readable text. It has become the de facto standard for exchanging data in the NoSQL world. When an object has been serialized into JSON, it is known as a *JSON document*.

Elasticsearch is a distributed *document* store. It can store and retrieve complex data structures--serialized as JSON documents--in *real time*. In other words, as soon as a document has been stored in Elasticsearch, it can be retrieved from any node in the cluster.

Of course, we don't need to only store data; we must also query it, en masse and at speed. While NoSQL solutions exist that allow us to store objects as documents, they still require us to think about how we want to query our data, and which fields require an index in order to make data retrieval fast.

In Elasticsearch, *all data in every field* is *indexed by default*. That is, every field has a dedicated inverted index for fast retrieval. And, unlike most other databases, it can use all of those inverted indices *in the same query*, to return results at breathtaking speed.

In this chapter, we present the APIs that we use to create, retrieve, update, and delete documents. For the moment, we don't care about the data inside our documents or how to query them. All we care about is how to store our documents safely in Elasticsearch and how to get them back again.

# What Is a Document?

Most entities or objects in most applications can be serialized into a JSON object, with keys and values. A *key* is the name of a field or property, and a *value* can be a string, a number, a Boolean, another object, an array of values, or some other specialized type such as a string representing a date or an object representing a geolocation:

```
{
    "name":         "John Smith",
    "age":          42,
    "confirmed":    true,
    "join_date":    "2014-06-01",
    "home": {
        "lat":      51.5,
        "lon":      0.1
    },
    "accounts": [
        {
            "type": "facebook",
            "id":   "johnsmith"
        },
        {
            "type": "twitter",
            "id":   "johnsmith"
        }
    ]
}
```

Often, we use the terms *object* and *document* interchangeably. However, there is a distinction. An object is just a JSON object--similar to what is known as a hash, hashmap, dictionary, or associative array. Objects may contain other objects. In Elasticsearch, the term *document* has a specific meaning. It refers to the top-level, or root object that is serialized into JSON and stored in Elasticsearch under a unique ID.

## Document Metadata

A document doesn't consist only of its data. It also has *metadata*—information *about* the document. The three required metadata elements are as follows:

`_index` :: Where the document lives

`_type` :: The class of object that the document represents

`_id` :: The unique identifier for the document

## _index

An *index* is like a database in a relational database; it's the place we store and index related data.

> TIP
>
> Actually, in Elasticsearch, our data is stored and indexed in *shards*, while an index is just a logical namespace that groups together one or more shards. However, this is an internal detail; our application shouldn't care about shards at all. As far as our application is concerned, our documents live in an *index*. Elasticsearch takes care of the details.

We cover how to create and manage indices ourselves in index-management, but for now we will let Elasticsearch create the index for us. All we have to do is choose an index name. This name must be lowercase, cannot begin with an

underscore, and cannot contain commas. Let's use `website` as our index name.

## _type

In applications, we use objects to represent *things* such as a user, a blog post, a comment, or an email. Each object belongs to a *class* that defines the properties or data associated with an object. Objects in the `user` class may have a name, a gender, an age, and an email address.

In a relational database, we usually store objects of the same class in the same table, because they share the same data structure. For the same reason, in Elasticsearch we use the same *type* for documents that represent the same class of *thing*, because they share the same data structure.

Every *type* has its own *mapping* or schema definition, which defines the data structure for documents of that type, much like the columns in a database table. Documents of all types can be stored in the same index, but the *mapping* for the type tells Elasticsearch how the data in each document should be indexed.

We show how to specify and manage mappings in *mapping*, but for now we will rely on Elasticsearch to detect our document's data structure automatically.

A `_type` name can be lowercase or uppercase, but shouldn't begin with an underscore or contain commas. We will use `blog` for our type name.

## _id

The *ID* is a string that, when combined with the `_index` and `_type`, uniquely identifies a document in Elasticsearch. When creating a new document, you can either provide your own `_id` or let Elasticsearch generate one for you.

## Other Metadata

There are several other metadata elements, which are presented in *mapping*. With the elements listed previously, we are already able to store a document in Elasticsearch and to retrieve it by ID--in other words, to use Elasticsearch as a document store.

[[index-doc]] === Indexing a Document

Documents are *indexed*—stored and made (("documents", "indexing")))((("indexing", "a document")))searchable--by using the `index` API. But first, we need to decide where the document lives. As we just discussed, a document's `_index`, `_type`, and `_id` uniquely identify the document. We can either provide our own `_id` value or let the `index` API generate one for us.

==== Using Our Own ID

If your document has a natural (("id", "providing for a document")))identifier (for example, a `user_account` field or some other value that identifies the document), you should provide your own `_id`, using this form of the `index` API:

[role="pagebreak-before"]

# [source,js]

PUT /{index}/{type}/{id} { "field": "value", ...

# }

For example, if our index is called `website`, our type is called `blog`, and we choose the ID `123`, then the index request looks like this:

# [source,js]

PUT /website/blog/123 { "title": "My first blog entry", "text": "Just trying this out...", "date": "2014/01/01"

# }

// SENSE: 030_Data/10_Create_doc_123.json

Elasticsearch responds as follows:

# [source,js]

{ "_index": "website", "_type": "blog", "_id": "123", "_version": 1, "created": true

# }

The response indicates that the indexing request has been successfully created and includes the `_index`, `_type`, and `_id` metadata, and a new element: `_version`.(("version number (documents)")))

Every document in Elasticsearch has a version number. Every time a change is made to a document (including deleting it), the `_version` number is incremented. In <>, we discuss how to use the `_version` number to ensure that one part of your application doesn't overwrite changes made by another part.

==== Autogenerating IDs

If our data doesn't have a natural ID, we can let Elasticsearch autogenerate one for us. (("id", "autogenerating")))The structure of the request changes: instead of using ((("HTTP methods", "POST")))((("POST method")))the `PUT` verb (`store`

`this document at this URL''), we use the `POST` verb (` store this document *under* this URL").

The URL now contains just the `_index` and the `_type`:

# [source,js]

POST /website/blog/ { "title": "My second blog entry", "text": "Still trying this out...", "date": "2014/01/01"

# }

// SENSE: 030_Data/10_Create_doc_auto_ID.json

The response is similar to what we saw before, except that the `_id` field has been generated for us:

# [source,js]

{ "_index": "website", "_type": "blog", "_id": "wM0OSFhDQXGZAWDf0-drSA", "_version": 1, "created": true

# }

Autogenerated IDs are 22 character long, URL-safe, Base64-encoded string *universally unique identifiers*, or(((("UUIDs (universally unique identifiers)"))) http://en.wikipedia.org/wiki/Uuid[UUIDs].

[[get-doc]] === Retrieving a Document

To get the document (((“documents”, “retrieving”)))out of Elasticsearch, we use the same `_index`, `_type`, and `_id`, but the HTTP verb (((“HTTP methods”, “GET”)))changes to `GET`:

# [source,sh]

# GET /website/blog/123?pretty

// SENSE: 030_Data/15_Get_document.json

The response includes the by-now-familiar metadata elements, plus (((“_source field”, sortas=“source field”)))the `_source` field, which contains the original JSON document that we sent to Elasticsearch when we indexed it:

# [source,js]

{ "_index" : "website", "_type" : "blog", "_id" : "123", "_version" : 1, "found" : true, "_source" : { "title": "My first blog entry", "text": "Just trying this out...", "date": "2014/01/01" }

# }

# [NOTE]

Adding `pretty` to the query-string parameters for any request,(((“query strings”, “adding pretty”))) as in the preceding example, causes Elasticsearch to *pretty-print* the(((“pretty-printing JSON response”))) JSON response to make it more readable. The `_source` field, however, isn't pretty-printed.

# Instead we get back exactly the same JSON string that we passed in.

The response to the +GET+ request includes `{"found": true}`. This confirms that the document was found. (((“documents”, “requesting non-existent document”)))If we were to request a document that doesn't exist, we would still get a JSON response, but `found` would be set to `false`.

Also, the HTTP response code would be `404 Not Found` instead of `200 OK`. We can see this by passing the `-i` argument to `curl`, which(((“curl command”, “-i argument”))) causes it to display the response headers:

# [source,sh]

# curl -i -XGET http://localhost:9200/website/blog/124?pretty

// SENSE: 030_Data/15_Get_document.json

The response now looks like this:

# [source,js]

HTTP/1.1 404 Not Found Content-Type: application/json; charset=UTF-8 Content-Length: 83

{ "_index" : "website", "_type" : "blog", "_id" : "124", "found" : false

# }

==== Retrieving Part of a Document

By default, a `GET` request(("documents", "retrieving part of")) will return the whole document, as stored in the `_source` field. But perhaps all you are interested in is the `title` field. Individual fields can be (("fields", "returning individual document fields")))((("_source field", sortas="source field")))requested by using the `_source` parameter. Multiple fields can be specified in a comma-separated list:

# [source,sh]

# GET /website/blog/123?_source=title,text

// SENSE: 030_Data/15_Get_document.json

The `_source` field now contains just the fields that we requested and has filtered out the `date` field:

# [source,js]

{ "_index" : "website", "_type" : "blog", "_id" : "123", "_version" : 1, "exists" : true, "_source" : { "title": "My first blog entry" , "text": "Just trying this out..." }

# }

Or if you want *just* the `_source` field without any metadata, you can use the `_source` endpoint:

# [source,sh]

# GET /website/blog/123/_source

// SENSE: 030_Data/15_Get_document.json

which returns just the following:

# [source,js]

{ "title": "My first blog entry", "text": "Just trying this out...", "date": "2014/01/01"

# }

[[doc-exists]] === Checking Whether a Document Exists

If all you want to do is to check whether a ((("documents", "checking whether a document exists")))document exists--you're not interested in the content at all--then use((("HEAD method")))((("HTTP methods", "HEAD"))) the `HEAD` method instead of the `GET` method. `HEAD` requests don't return a body, just HTTP headers:

## [source,js]

## curl -i -XHEAD http://localhost:9200/website/blog/123

Elasticsearch will return a `200 OK` status code if the document exists:

## [source,js]

HTTP/1.1 200 OK Content-Type: text/plain; charset=UTF-8

## Content-Length: 0

And a `404 Not Found` if it doesn't exist:

## [source,js]

## curl -i -XHEAD http://localhost:9200/website/blog/124

## [source,js]

HTTP/1.1 404 Not Found Content-Type: text/plain; charset=UTF-8

## Content-Length: 0

Of course, just because a document didn't exist when you checked it, doesn't mean that it won't exist a millisecond later: another process might create the document in the meantime.

[[update-doc]] === Updating a Whole Document

Documents in Elasticsearch are *immutable*; we cannot change them.((("documents", "updating whole document"))) ((("updating documents", "whole document"))) Instead, if we need to update an existing document, we *reindex* or replace it, ((("reindexing")))((("indexing", seealso="reindexing"))) which we can do using the same `index` API that we have already discussed in <>.

# [source,js]

PUT /website/blog/123 { "title": "My first blog entry", "text": "I am starting to get the hang of this...", "date": "2014/01/02"

# }

// SENSE: 030_Data/25_Reindex_doc.json

In the response, we can see that Elasticsearch has (((("version number (documents)", "incremented when document replaced")))incremented the `_version` number:

# [source,js]

{ "_index" : "website", "_type" : "blog", "_id" : "123", "_version" : 2, "created": false <1>

# }

<1> The `created` flag is((("created flag"))) set to `false` because a document with the same index, type, and ID already existed.

Internally, Elasticsearch has marked the old document as deleted and added an entirely new document.((("deleted documents"))) The old version of the document doesn't disappear immediately, although you won't be able to access it. Elasticsearch cleans up deleted documents in the background as you continue to index more data.

Later in this chapter, we introduce the `update` API, which can be used to make <>. This API *appears* to change documents in place, but actually Elasticsearch is following exactly the same process as described previously:

1. Retrieve the JSON from the old document
2. Change it
3. Delete the old document
4. Index a new document

The only difference is that the `update` API achieves this through a single client request, instead of requiring separate `get` and `index` requests.

[[create-doc]] === Creating a New Document

How can we be sure, when we index a document, that(("documents", "creating")) we are creating an entirely new document and not overwriting an existing one?

Remember that the combination of `_index`, `_type`, and `_id` uniquely identifies a document. So the easiest way to ensure that our document is new is by letting Elasticsearch autogenerate a new unique `_id`, using the `POST` version of ((("POST method")))((("HTTP methods", "POST")))the index request:

# [source,js]

POST /website/blog/

# { ... }

However, if we already have an `_id` that we want to use, then we have to tell Elasticsearch that it should accept our index request only if a document with the same `_index`, `_type`, and `_id` doesn't exist already. There are two ways of doing this, both of which amount to the same thing. Use whichever method is more convenient for you.

The first method uses the `op_type` query(((("PUT method")))((("HTTP methods", "PUT")))((("query strings", "op_type parameter")))((("op_type query string parameter")))-string parameter:

# [source,js]

PUT /website/blog/123?op_type=create

# { ... }

And the second uses the `/_create` endpoint in the URL:

# [source,js]

PUT /website/blog/123/_create

# { ... }

If the request succeeds in creating a new document, Elasticsearch will return the usual metadata and an HTTP response code of `201 Created`.

On the other hand, if a document ((("Document Already Exists Exception")))with the same `_index`, `_type`, and `_id` already exists, Elasticsearch will respond with a `409 Conflict` response code, and an error message like the following:

# [source,js]

{ "error" : "DocumentAlreadyExistsException[[website][4] [blog][123]: document already exists]", "status" : 409

# }

```
// SENSE: 030_Data/30_Create_doc.json
```

[[delete-doc]] === Deleting a Document

The syntax for deleting a document(("documents", "deleting"))) follows the same pattern that we have seen already, but ((("DELETE method", "deleting documents")))((("HTTP methods", "DELETE")))uses the `DELETE` method :

# [source,js]

# DELETE /website/blog/123

// SENSE: 030_Data/35_Delete_doc.json

If the document is found, Elasticsearch will return an HTTP response code of `200 OK` and a response body like the following. Note that the `_version` number has been incremented:

# [source,js]

{ "found" : true, "_index" : "website", "_type" : "blog", "_id" : "123", "_version" : 3

# }

If the document isn't(("version number (documents)", "incremented for document not found"))) found, we get a `404 Not Found` response code and a body like this:

# [source,js]

{ "found" : false, "_index" : "website", "_type" : "blog", "_id" : "123", "_version" : 4

# }

Even though the document doesn't exist ( `found` is `false` ), the `_version` number has still been incremented. This is part of the internal bookkeeping, which ensures that changes are applied in the correct order across multiple nodes.

NOTE: As already mentioned in <>, deleting a document doesn't immediately remove the document from disk; it just marks it as deleted. Elasticsearch will clean up deleted documents in the background as you continue to index more data.

[[version-control]] === Dealing with Conflicts

When updating a document with (("conflicts", "dealing with")))the `index` API, we read the original document, make our changes, and then reindex the *whole document* in one go. The most recent indexing request wins: whichever document was indexed last is the one stored in Elasticsearch. If somebody else had changed the document in the meantime, their changes would be lost.

Many times, this is not a problem. Perhaps our main data store is a relational database, and we just copy the data into Elasticsearch to make it searchable. Perhaps there is little chance of two people changing the same document at the same time. Or perhaps it doesn't really matter to our business if we lose changes occasionally.

But sometimes losing a change is *very important*. Imagine that we're using Elasticsearch to store the number of widgets that we have in stock in our online store. Every time that we sell a widget, we decrement the stock count in Elasticsearch.

One day, management decides to have a sale. Suddenly, we are selling several widgets every second. Imagine two web processes, running in parallel, both processing the sale of one widget each, as shown in <>.

[[img-data-lww]] .Consequence of no concurrency control image::images/elas_0301.png["Consequence of no concurrency control",width="50%",align="center"]

The change that `web_1` made to the `stock_count` has been lost because `web_2` is unaware that its copy of the `stock_count` is out-of-date. The result is that we think we have more widgets than we actually do, and we're going to disappoint customers by selling them stock that doesn't exist.

The more frequently that changes are made, or the longer the gap between reading data and updating it, the more likely it is that we will lose changes.

In the database world, two approaches are commonly used to ensure that changes are not lost when making (("pessimistic concurrency control")))((("concurrency control")))concurrent updates:

*Pessimistic concurrency control*::

Widely used by relational databases, this approach assumes that conflicting changes are likely to happen and so blocks access to a resource in order to prevent conflicts. A typical example is locking a row before reading its data, ensuring that only the thread that placed the lock is able to make changes to the data in that row.

*Optimistic concurrency control*::

Used by Elasticsearch, (("optimistic concurrency control"))) this approach assumes that conflicts are unlikely to happen and doesn't block operations from being attempted. However, if the underlying data has been modified between reading and writing, the update will fail. It is then up to the application to decide how it should resolve the conflict. For instance, it could reattempt the update, using the fresh data, or it could report the situation to the user.

[[optimistic-concurrency-control]] === Optimistic Concurrency Control

Elasticsearch is distributed. When documents(("concurrency control", "optimistic")) are created, updated, or deleted, the new version of the document has to be replicated to other nodes in the cluster. Elasticsearch is also asynchronous and concurrent, meaning that these replication requests are sent in parallel, and may arrive at their destination *out of sequence*. Elasticsearch needs a way of ensuring that an older version of a document never overwrites a newer version.

When we discussed `index`, `get`, and `delete` requests previously, we pointed out that every document has a `_version` number that is incremented whenever a document is changed. Elasticsearch uses this `_version` number to ensure that changes are applied in the correct order. If an older version of a document arrives after a new version, it can simply be ignored.

We can take advantage of the `_version` number to ensure (((("version number (documents)", "using to avoid conflicts")))that conflicting changes made by our application do not result in data loss. We do this by specifying the `version` number of the document that we wish to change. If that version is no longer current, our request fails.

Let's create a new blog post:

# [source,js]

PUT /website/blog/1/_create { "title": "My first blog entry", "text": "Just trying this out..."

# }

// SENSE: 030_Data/40_Concurrency.json

The response body tells us that this newly created document has `_version` number `1`. Now imagine that we want to edit the document: we load its data into a web form, make our changes, and then save the new version.

First we retrieve the document:

# [source,js]

# GET /website/blog/1

// SENSE: 030_Data/40_Concurrency.json

The response body includes the same `_version` number of `1`:

# [source,js]

{ "_index" : "website", "_type" : "blog", "_id" : "1", "_version" : 1, "found" : true, "_source" : { "title": "My first blog entry", "text": "Just trying this out..." }

# }

Now, when we try to save our changes by reindexing the document, we specify the `version` to which our changes should be applied:

# [source,js]

PUT /website/blog/1?version=1 <1> { "title": "My first blog entry", "text": "Starting to get the hang of this..."

# }

// SENSE: 030_Data/40_Concurrency.json

<1> We want this update to succeed only if the current `_version` of this document in our index is version `1`.

This request succeeds, and the response body tells us that the `_version` has been incremented to `2`:

# [source,js]

{ "_index": "website", "_type": "blog", "_id": "1", "_version": 2 "created": false

# }

// SENSE: 030_Data/40_Concurrency.json

However, if we were to rerun the same index request, still specifying `version=1`, Elasticsearch would respond with a `409 Conflict` HTTP response code, and a body like the following:

# [source,js]

{ "error" : "VersionConflictEngineException[[website][2] [blog][1]: version conflict, current [2], provided [1]]", "status" : 409

# }

// SENSE: 030_Data/40_Concurrency.json

This tells us that the current `_version` number of the document in Elasticsearch is `2`, but that we specified that we were updating version `1`.

What we do now depends on our application requirements. We could tell the user that somebody else has already made changes to the document, and to review the changes before trying to save them again. Alternatively, as in the case of the widget `stock_count` previously, we could retrieve the latest document and try to reapply the change.

All APIs that update or delete a document accept a `version` parameter, which allows you to apply optimistic concurrency control to just the parts of your code where it makes sense.

==== Using Versions from an External System

A common setup is to use some other database as the primary data store and Elasticsearch to make the data searchable, ((("version number (documents)", "using an external version number")))((("external version numbers"))) which means that all changes to the primary database need to be copied across to Elasticsearch as they happen. If multiple processes are responsible for this data synchronization, you may run into concurrency problems similar to those described previously.

If your main database already has version numbers--or a value such as `timestamp` that can be used as a version number--then you can reuse these same version numbers in Elasticsearch by adding `version_type=external` to the query string. ((("query strings", "version_type=external"))) Version numbers must be integers greater than zero and less than about `9.2e+18`--a positive `long` value in Java.

The way external version numbers are handled is a bit different from the internal version numbers we discussed previously. Instead of checking that the current `_version` is *the same* as the one specified in the request, Elasticsearch checks that the current `_version` is *less than* the specified version. If the request succeeds, the external version number is stored as the document's new `_version`.

External version numbers can be specified not only on index and delete requests, but also when *creating* new documents.

For instance, to create a new blog post with an external version number of `5`, we can do the following:

# [source,js]

PUT /website/blog/2?version=5&version_type=external { "title": "My first external blog entry", "text": "Starting to get the hang of this..."

# }

// SENSE: 030_Data/40_External_versions.json

In the response, we can see that the current `_version` number is `5`:

# [source,js]

{ "_index": "website", "_type": "blog", "_id": "2", "_version": 5, "created": true

# }

Now we update this document, specifying a new `version` number of `10`:

# [source,js]

PUT /website/blog/2?version=10&version_type=external { "title": "My first external blog entry", "text": "This is a piece of cake..."

# }

// SENSE: 030_Data/40_External_versions.json

The request succeeds and sets the current `_version` to `10`:

# [source,js]

{ "_index": "website", "_type": "blog", "_id": "2", "_version": 10, "created": false

# }

If you were to rerun this request, it would fail with the same conflict error we saw before, because the specified external version number is not higher than the current version in Elasticsearch.

[[partial-updates]] === Partial Updates to Documents

In <>, we said that (((("updating documents", "partial updates")))((("documents", "partial updates")))the way to update a document is to retrieve it, change it, and then reindex the whole document. This is true. However, using the `update` API, we can make partial updates like incrementing a counter in a single request.

We also said that documents are immutable: they cannot be changed, only replaced. The `update` API *must* obey the same rules. Externally, it appears as though we are partially updating a document in place. Internally, however, the `update` API simply manages the same *retrieve-change-reindex* process that we have already described. The difference is that this process happens within a shard, thus avoiding the network overhead of multiple requests. By reducing the time between the *retrieve* and *reindex* steps, we also reduce the likelihood of there being conflicting changes from other processes.

The simplest form of the `update` request accepts a partial document as the `doc` parameter, which just gets merged with the existing document. Objects are merged together, existing scalar fields are overwritten, and new fields are added. For instance, we could add a `tags` field and a `views` field to our blog post as follows:

# [source,js]

POST /website/blog/1/_update { "doc" : { "tags" : [ "testing" ], "views": 0 }

# }

// SENSE: 030_Data/45_Partial_update.json

If the request succeeds, we see a response similar to that of the `index` request:

# [source,js]

{ "_index" : "website", "_id" : "1", "_type" : "blog", "_version" : 3

# }

Retrieving the document shows the updated `_source` field:

# [source,js]

{ "_index": "website", "_type": "blog", "_id": "1", "_version": 3, "found": true, "_source": { "title": "My first blog entry", "text": "Starting to get the hang of this...", "tags": [ "testing" ], <1> "views": 0 <1> }

# }

// SENSE: 030_Data/45_Partial_update.json

<1> Our new fields have been added to the `_source`.

==== Using Scripts to Make Partial Updates

Scripts can be used in the `update` API to change the contents of the `_source` field, which (("_source field", sortas="source field"))is referred to inside an update script as `ctx._source`. For instance, we could use a script to increment the number of

`views` that our blog post has had:

# [source,js]

POST /website/blog/1/_update { "script" : "ctx._source.views+=1"

# }

// SENSE: 030_Data/45_Partial_update.json

.Scripting with Groovy

---

For those ((("documents", "partial updates", "using scripts")))((("updating documents", "partial updates", "using scripts")))moments when the API just isn't enough, Elasticsearch allows you to write your own custom logic in a script. ((("scripts", "using to make partial updates"))) Scripting is supported in many APIs including search, sorting, aggregations, and document updates. Scripts can be passed in as part of the request, retrieved from the special `.scripts` index, or loaded from disk.

The default scripting language ((("Groovy")))is a http://groovy.codehaus.org/[Groovy], a fast and expressive scripting language, similar in syntax to JavaScript. It runs in a *sandbox* to prevent malicious users from breaking out of Elasticsearch and attacking the server.

You can read more about scripting in the http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/modules-scripting.html[scripting reference documentation].

---

We can also use a script to add a new tag to the `tags` array. In this example we specify the new tag as a parameter rather than hardcoding it in the script itself. This allows Elasticsearch to reuse the script in the future, without having to compile a new script every time we want to add another tag:

# [source,js]

POST /website/blog/1/_update { "script" : "ctx._source.tags+=new_tag", "params" : { "new_tag" : "search" }

# }

// SENSE: 030_Data/45_Partial_update.json

Fetching the document shows the effect of the last two requests:

# [source,js]

{ "_index": "website", "_type": "blog", "_id": "1", "_version": 5, "found": true, "_source": { "title": "My first blog entry", "text": "Starting to get the hang of this...", "tags": ["testing", "search"], <1> "views": 1 <2> }

# }

<1> The `search` tag has been appended to the `tags` array.

<2> The `views` field has been incremented.

We can even choose to delete a document based on its contents, by setting `ctx.op` to `delete`:

# [source,js]

POST /website/blog/1/_update { "script" : "ctx.op = ctx._source.views == count ? 'delete' : 'none'", "params" : { "count": 1 }

# }

// SENSE: 030_Data/45_Partial_update.json

==== Updating a Document That May Not Yet Exist

Imagine that we need to store a(("updating documents", "that don't already exist"))) page view counter in Elasticsearch. Every time that a user views a page, we increment the counter for that page. But if it is a new page, we can't be sure that the counter already exists. If we try to update a nonexistent document, the update will fail.

In cases like these, we can use(("upsert parameter"))) the `upsert` parameter to specify the document that should be created if it doesn't already exist:

# [source,js]

POST /website/pageviews/1/_update { "script" : "ctx._source.views+=1", "upsert": { "views": 1 }

# }

// SENSE: 030_Data/45_Upsert.json

The first time we run this request, the `upsert` value is indexed as a new document, which initializes the `views` field to `1`. On subsequent runs, the document already exists, so the `script` update is applied instead, incrementing the `views` counter.

==== Updates and Conflicts

In the introduction to this section, we said(("updating documents", "conflicts and")))((("conflicts", "updates and"))) that the smaller the window between the *retrieve* and *reindex* steps, the smaller the opportunity for conflicting changes. But it doesn't eliminate the possibility completely. It is still possible that a request from another process could change the document before `update` has managed to reindex it.

To avoid losing data, the `update` API retrieves the current `_version` of the document in the *retrieve* step, and passes that to the `index` request during the *reindex* step. If another process has changed the document between retrieve and reindex, then the `_version` number won't match and the update request will fail.

For many uses of partial update, it doesn't matter that a document has been changed. For instance, if two processes are both incrementing the page-view counter, it doesn't matter in which order it happens; if a conflict occurs, the only thing we need to do is reattempt the update.

This can be done automatically by(("query strings", "retry_on_conflict parameter")))((("retry_on_conflict parameter"))) setting the `retry_on_conflict` parameter to the number of times that `update` should retry before failing; it defaults to `0`.

# [source,js]

POST /website/pageviews/1/_update?retry_on_conflict=5 <1> { "script" : "ctx._source.views+=1", "upsert": { "views": 0 }

# }

// SENSE: 030_Data/45_Upsert.json

<1> Retry this update five times before failing.

This works well for operations such as incrementing a counter, where the order of increments does not matter, but in other situations the order of changes *is* important. Like the <>, the `update` API adopts a *last-write-wins* approach by default, but it also accepts a `version` parameter that allows you to use <> to specify which version of the document you intend to update.

=== Retrieving Multiple Documents

As fast as Elasticsearch is, it can be faster still.((("documents", "retrieving multiple"))) Combining multiple requests into one avoids the network overhead of processing each request individually. If you know that you need to retrieve multiple documents from Elasticsearch, it is faster to retrieve them all in a single request by using the *multi-get*, or `mget` , API, (((("mget (multi-get) API")))instead of document by document.

The `mget` API expects a `docs` array, each (((("docs array", "in request")))element of which specifies the `_index` , `_type` , and `_id` metadata of the document you wish to retrieve. You can also specify a `_source` parameter if you just want to retrieve one or more specific fields:

# [source,js]

GET /_mget { "docs" : [ { "_index" : "website", "_type" : "blog", "_id" : 2 }, { "_index" : "website", "_type" : "pageviews", "_id" : 1, "_source": "views" } ]

# }

// SENSE: 030_Data/50_Mget.json

The response body also contains a `docs` array((("docs array", "in response body"))) that contains a response per document, in the same order as specified in the request. Each of these responses is the same response body that we would expect from an individual <>:

# [source,js]

{ "docs" : [ { "_index" : "website", "_id" : "2", "_type" : "blog", "found" : true, "_source" : { "text" : "This is a piece of cake...", "title" : "My first external blog entry" }, "_version" : 10 }, { "_index" : "website", "_id" : "1", "_type" : "pageviews", "found" : true, "_version" : 2, "_source" : { "views" : 2 } } ]

# }

// SENSE: 030_Data/50_Mget.json

If the documents you wish to retrieve are all in the same `_index` (and maybe even of the same `_type` ), you can specify a default `/_index` or a default `/_index/_type` in the URL.

You can still override these values in the individual requests:

# [source,js]

GET /website/blog/_mget { "docs" : [ { "_id" : 2 }, { "_type" : "pageviews", "_id" : 1 } ]

# }

// SENSE: 030_Data/50_Mget.json

In fact, if all the documents have the same `_index` and `_type` , you can just pass an array of `ids` instead of the full `docs` array:

# [source,js]

GET /website/blog/_mget { "ids" : [ "2", "1" ]

# }

Note that the second document that we requested doesn't exist. We specified type `blog`, but the document with ID `1` is of type `pageviews`. This nonexistence is reported in the response body:

# [source,js]

{ "docs" : [ { "_index" : "website", "_type" : "blog", "_id" : "2", "_version" : 10, "found" : true, "_source" : { "title": "My first external blog entry", "text": "This is a piece of cake..." } }, { "_index" : "website", "_type" : "blog", "_id" : "1", "found" : false <1> } ]

# }

// SENSE: 030_Data/50_Mget.json

<1> This document was not found.

The fact that the second document wasn't found didn't affect the retrieval of the first document. Each doc is retrieved and reported on individually.

# [NOTE]

The HTTP status code for the preceding request is `200`, even though one document wasn't found. In fact, it would still be `200` if *none* of the requested documents were found--because the `mget` request itself completed successfully. To determine the success or failure of

# the individual documents, you need to check (((("found flag")))the `found` flag.

[[bulk]] === Cheaper in Bulk

In the same way that `mget` allows us to retrieve multiple documents at once, the `bulk` API allows(("bulk API"))) us to make multiple `create`, `index`, `update`, or `delete` requests in a single step. This is particularly useful if you need to index a data stream such as log events, which can be queued up and indexed in batches of hundreds or thousands.

The `bulk` request body has the following, slightly unusual, format:

# [source,js]

{ action: { metadata }}\n { request body }\n { action: { metadata }}\n { request body }\n

## ...

This format is like a *stream* of valid one-line JSON documents joined together by newline ( `\n` ) characters.((("\n (newline) characters in bulk requests", sortas="n (newline)"))) Two important points to note:

- Every line must end with a newline character ( `\n` ), *including the last line*. These are used as markers to allow for efficient line separation.
- The lines cannot contain unescaped newline characters, as they would interfere with parsing. This means that the JSON must *not* be pretty-printed.

TIP: In <>, we explain why the `bulk` API uses this format.

The +action/metadata+ line specifies *what action* to do to *which document*.

The +action+ must be(("action, in bulk requests"))) one of the following:

`create` :: Create a document only if the document does not already exist. See <>.

`index` :: Create a new document or replace an existing document. See <> and <>.

`update` :: Do a partial update on a document. See <>.

`delete` :: Delete a document. See <>.

The +metadata+ should (((("metadata, document", "in bulk requests")))specify the `_index`, `_type`, and `_id` of the document to be indexed, created, updated, or deleted.

For instance, a `delete` request could look like this:

# [source,js]

# { "delete": { "_index": "website", "_type": "blog", "_id": "123" }}

The +request body+ line consists of the(("request body line, bulk requests"))) document `_source` itself--the fields and values that the document contains. It is required for `index` and `create` operations, which makes sense: you must supply the document to index.

It is also required for `update` operations and should consist of the same request body that you would pass to the `update`

API: `doc` , `upsert` , `script` , and so forth. No +request body+ line is required for a delete.

# [source,js]

{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}

# { "title": "My first blog post" }

If no `_id` is specified, an ID will be autogenerated:

# [source,js]

{ "index": { "_index": "website", "_type": "blog" }}

# { "title": "My second blog post" }

To put it all together, a complete `bulk` request (((("bulk API", "common bulk request, example")))has this form:

# [source,js]

POST /_bulk { "delete": { "_index": "website", "_type": "blog", "_id": "123" }} <1> { "create": { "_index": "website", "_type": "blog", "_id": "123" }} { "title": "My first blog post" } { "index": { "_index": "website", "_type": "blog" }} { "title": "My second blog post" } { "update": { "_index": "website", "_type": "blog", "_id": "123", "_retry_on_conflict" : 3} }

# { "doc" : {"title" : "My updated blog post"} } <2>

// SENSE: 030_Data/55_Bulk.json

<1> Notice how the `delete` action does not have a request body; it is followed immediately by another action.

<2> Remember the final newline character.

The Elasticsearch response contains the `items` array,((("items array, listing results of bulk requests")))((("bulk API", "Elasticsearch response"))) which lists the result of each request, in the same order as we requested them:

# [source,js]

{ "took": 4, "errors": false, <1> "items": [ { "delete": { "_index": "website", "_type": "blog", "_id": "123", "_version": 2, "status": 200, "found": true }}, { "create": { "_index": "website", "_type": "blog", "_id": "123", "_version": 3, "status": 201 }}, { "create": { "_index": "website", "_type": "blog", "_id": "EiwfApScQiiy7TIKFxRCTw", "_version": 1, "status": 201 }}, { "update": { "_index": "website", "_type": "blog", "_id": "123", "_version": 4, "status": 200 }} ]

# }}

// SENSE: 030_Data/55_Bulk.json

<1> All subrequests completed successfully.

Each subrequest is executed independently, so the failure of one subrequest won't affect the success of the others. If any of the requests fail, the top-level `error` flag is set to `true` and the error details will be reported under the relevant request:

# [source,js]

POST /_bulk { "create": { "_index": "website", "_type": "blog", "_id": "123" }} { "title": "Cannot create - it already exists" } { "index": { "_index": "website", "_type": "blog", "_id": "123" }}

# { "title": "But we can update it" }

// SENSE: 030_Data/55_Bulk_independent.json

In the response, we can see that it failed to `create` document `123` because it already exists, but the subsequent `index` request, also on document `123`, succeeded:

# [source,js]

{ "took": 3, "errors": true, <1> "items": [ { "create": { "_index": "website", "_type": "blog", "_id": "123", "status": 409, <2> "error": "DocumentAlreadyExistsException <3> [[website][4] [blog][123]: document already exists]" }}, { "index": { "_index": "website", "_type": "blog", "_id": "123", "_version": 5, "status": 200 <4> }} ]

# }

// SENSE: 030_Data/55_Bulk_independent.json

<1> One or more requests has failed.

<2> The HTTP status code for this request reports `409 CONFLICT`.

<3> The error message explaining why the request failed.

<4> The second request succeeded with an HTTP status code of `200 OK`.

That also means (((("bulk API", "bulk requests, not transactions")))that `bulk` requests are not atomic: they cannot be used to implement transactions. Each request is processed separately, so the success or failure of one request will not interfere with the others.

==== Don't Repeat Yourself

Perhaps you are batch-indexing logging data into the same `index`, and with the same `type`. Having to ((("metadata, document", "not repeating in bullk requests")))((("bulk API", "default /_index or _index/_type")))specify the same metadata for every document is a waste. Instead, just as for the `mget` API, the `bulk` request accepts a default `/_index` or `/_index/_type` in the URL:

# [source,js]

POST /website/_bulk { "index": { "_type": "log" }}

# { "event": "User logged in" }

// SENSE: 030_Data/55_Bulk_defaults.json

You can still override the `_index` and `_type` in the metadata line, but it will use the values in the URL as defaults:

# [source,js]

POST /website/log/_bulk { "index": {}} { "event": "User logged in" } { "index": { "_type": "blog" }}

# { "title": "Overriding the default type" }

// SENSE: 030_Data/55_Bulk_defaults.json

==== How Big Is Too Big?

The entire bulk request needs to be loaded into memory by the node that receives our request, so the bigger the request, the less memory available for other requests.((("bulk API", "optimal size of requests"))) There is an optimal size of bulk request. Above that size, performance no longer improves and may even drop off. The optimal size, however, is not a fixed number. It depends entirely on your hardware, your document size and complexity, and your indexing and search load.

Fortunately, it is easy to find this *sweet spot*: Try indexing typical documents in batches of increasing size. When performance starts to drop off, your batch size is too big. A good place to start is with batches of 1,000 to 5,000 documents or, if your documents are very large, with even smaller batches.

It is often useful to keep an eye on the physical size of your bulk requests. One thousand 1KB documents is very different from one thousand 1MB documents. A good bulk size to start playing with is around 5-15MB in size.

[[distributed-docs]] == Distributed Document Store

In the preceding chapter, we looked at all the ways to put data into your index and then retrieve it. But we glossed over many technical details surrounding how the data is distributed and fetched from the cluster. This separation is done on purpose; you don't really need to know how data is distributed to work with Elasticsearch. It just works.

In this chapter, we dive into those internal, technical details to help you understand how your data is stored in a distributed system.

.Content Warning

---

The information presented in this chapter is for your interest. You are not required to understand and remember all the detail in order to use Elasticsearch. The options that are discussed are for advanced users only.

Read the section to gain a taste for how things work, and to know where the information is in case you need to refer to it in the future, but don't be overwhelmed by the detail.

---

[[routing-value]] === Routing a Document to a Shard

When you index a document, it is stored on a single primary shard.((("shards", "routing a document to")))((("documents", "routing a document to a shard")))((("routing a document to a shard"))) How does Elasticsearch know which shard a document belongs to? When we create a new document, how does it know whether it should store that document on shard 1 or shard 2?

The process can't be random, since we may need to retrieve the document in the future. In fact, it is determined by a simple formula:

```
shard = hash(routing) % number_of_primary_shards
```

The `routing` value is an arbitrary string, which defaults to the document's `_id` but can also be set to a custom value. This `routing` string is passed through a hashing function to generate a number, which is divided by the number of primary shards in the index to return the *remainder*. The remainder will always be in the range `0` to `number_of_primary_shards - 1`, and gives us the number of the shard where a particular document lives.

This explains why the number of primary shards(((("primary shards", "fixed number of, routing and"))) can be set only when an index is created and never changed: if the number of primary shards ever changed in the future, all previous routing values would be invalid and documents would never be found.

# [NOTE]

Users sometimes think that having a fixed number of primary shards makes it difficult to scale out an index later. In reality, there are techniques that make it easy to scale out as and when you need. We talk more about these

# in <>.

All document APIs ( `get` , `index` , `delete` , `bulk` , `update` , and `mget` ) accept a `routing` parameter (((("routing parameter")))that can be used to customize the document-to- shard mapping. A custom routing value could be used to ensure that all related documents--for instance, all the documents belonging to the same user--are stored on the same shard. We discuss in detail why you may want to do this in <>.

=== How Primary and Replica Shards Interact

For explanation purposes, let's(("shards", "interaction of primary and replica shards")))(("primary shards", "interaction with replica shards")))(("replica shards", "interaction with primary shards"))) imagine that we have a cluster consisting of three nodes. It contains one index called `blogs` that has two primary shards. Each primary shard has two replicas. Copies of the same shard are never allocated to the same node, so our cluster looks something like <>.

[[img-distrib]] .A cluster with three nodes and one index image::images/elas_0401.png["A cluster with three nodes and one index"]

We can send our requests to any node in the cluster.(("nodes", "sending requests to")) Every node is fully capable of serving any request. Every node knows the location of every document in the cluster and so can forward requests directly to the required node. In the following examples, we will send all of our requests to `Node 1`, which we will refer to as the *requesting node*.

TIP: When sending requests, it is good practice to round-robin through all the nodes in the cluster, in order to spread the load.

[[distrib-write]] === Creating, Indexing, and Deleting a Document

Create, index, and delete(((("documents", "creating, indexing, and deleting"))) requests are *write* operations,(((("write operations"))) which must be successfully completed on the primary shard before they can be copied to any associated replica shards, as shown in <>.

[[img-distrib-write]] .Creating, indexing, or deleting a single document image::images/elas_0402.png["Creating, indexing or deleting a single document"]

Here is the sequence (((("primary shards", "creating, indexing, and deleting a document")))(((("replica shards", "creating, indexing, and deleting a document")))of steps necessary to successfully create, index, or delete a document on both the primary and any replica shards:

1. The client sends a create, index, or delete request to `Node 1`.
2. The node uses the document's `_id` to determine that the document belongs to shard `0`. It forwards the request to `Node 3`, where the primary copy of shard `0` is currently allocated.
3. `Node 3` executes the request on the primary shard. If it is successful, it forwards the request in parallel to the replica shards on `Node 1` and `Node 2`. Once all of the replica shards report success, `Node 3` reports success to the requesting node, which reports success to the client.

By the time the client receives a successful response, the document change has been executed on the primary shard and on all replica shards. Your change is safe.

There are a number of optional request parameters that allow you to influence this process, possibly increasing performance at the cost of data security. These options are seldom used because Elasticsearch is already fast, but they are explained here for the sake of completeness:

`replication`::

+

The default value for (((("replication request parameter", "sync and async values")))replication is `sync`. (((("sync value, replication parameter")))This causes the primary shard to wait for successful responses from the replica shards before returning.

If you set `replication` to `async`,((("async value, replication parameter"))) it will return success to the client as soon as the request has been executed on the primary shard. It will still forward the request to the replicas, but you will not know whether the replicas succeeded.

This option is mentioned specifically to advise against using it. The default `sync` replication allows Elasticsearch to exert back pressure on whatever system is feeding it with data. With `async` replication, it is possible to overload Elasticsearch by sending too many requests without waiting for their completion.

--

`consistency`::

+

By default, the primary shard((("consistency request parameter")))((("quorum"))) requires a *quorum*, or majority, of shard copies (where a shard copy can be a primary or a replica shard) to be available before even attempting a write operation.

This is to prevent writing data to the ``wrong side'' of a network partition. A quorum is defined as follows:

```
int( (primary + number_of_replicas) / 2 ) + 1
```

The allowed values for `consistency` are `one` (just the primary shard), `all` (the primary and all replicas), or the default `quorum`, or majority, of shard copies.

Note that the `number_of_replicas` is the number of replicas *specified* in the index settings, not the number of replicas that are currently active. If you have specified that an index should have three replicas, a quorum would be as follows:

```
int( (primary + 3 replicas) / 2 ) + 1 = 3
```

But if you start only two nodes, there will be insufficient active shard copies to satisfy the quorum, and you will be unable to index or delete any documents.

--

`timeout`::

What happens if insufficient shard copies are available? Elasticsearch waits, in the hope that more shards will appear. By default, it will wait up to 1 minute. If you need to, you can use the `timeout` parameter((("timeout parameter"))) to make it abort sooner: `100` is 100 milliseconds, and `30s` is 30 seconds.

# [NOTE]

A new index has `1` replica by default, which means that two active shard copies *should* be required in order to satisfy the need for a `quorum`. However, these default settings would prevent us from doing anything useful with a single-node cluster. To avoid this problem, the requirement for

# a quorum is enforced only when `number_of_replicas` is greater than `1`.

[[distrib-read]] === Retrieving a Document

A document can be retrieved from a ((("documents", "retrieving")))primary shard or from any of its replicas, as shown in <>.

[[img-distrib-read]] .Retrieving a single document image::images/elas_0403.png["Retrieving a single document"]

Here is the sequence of steps to retrieve a document from either a primary or replica shard:

1. The client sends a get request to `Node 1`.
2. The node uses the document's `_id` to determine that the document belongs to shard `0`. Copies of shard `0` exist on all three nodes. On this occasion, it forwards the request to `Node 2`.
3. `Node 2` returns the document to `Node 1`, which returns the document to the client.

For read requests, the requesting node will choose a different shard copy on every request in order to balance the load; it round-robins through all shard copies.

It is possible that, while a document is being indexed, the document will already be present on the primary shard but not yet copied to the replica shards. In this case, a replica might report that the document doesn't exist, while the primary would have returned the document successfully. Once the indexing request has returned success to the user, the document will be available on the primary and all replica shards.

=== Partial Updates to a Document

The `update` API , as shown in <>, combines the read and(((("updating documents", "partial updates")))((("documents", "partial updates"))) write patterns explained previously.

[[img-distrib-update]] .Partial updates to a document image::images/elas_0404.png["Partial updates to a document"]

Here is the sequence of steps used to perform a partial update on a document:

1. The client sends an update request to `Node 1`.
2. It forwards the request to `Node 3`, where the primary shard is allocated.
3. `Node 3` retrieves the document from the primary shard, changes the JSON in the `_source` field, and tries to reindex the document on the primary shard. If the document has already been changed by another process, it retries step 3 up to `retry_on_conflict` times, before giving up.
4. If `Node 3` has managed to update the document successfully, it forwards the new version of the document in parallel to the replica shards on `Node 1` and `Node 2` to be reindexed. Once all replica shards report success, `Node 3` reports success to the requesting node, which reports success to the client.

The `update` API also accepts the `routing`, `replication`, `consistency`, and `timeout` parameters that are explained in <>.

.Document-Based Replication

---

When a primary shard forwards changes to its replica shards,(((("primary shards", "forwarding changes to replica shards"))) it doesn't forward the update request. Instead it forwards the new version of the full document. Remember that these changes are forwarded to the replica shards asynchronously, and there is no guarantee that they will arrive in the same order that they were sent. If Elasticsearch forwarded just the change, it is possible that changes would be applied in the wrong order, resulting in a corrupt document.

---

[[distrib-multi-doc]] === Multidocument Patterns

The patterns for the `mget` and `bulk` APIs(("mget (multi-get) API", "retrieving multiple documents, process of"))(("documents", "retrieving multiple with mget")) are similar to those for individual documents. The difference is that the requesting node knows in which shard each document lives. It breaks up the multidocument request into a multidocument request *per shard*, and forwards these in parallel to each participating node.

Once it receives answers from each node, it collates their responses into a single response, which it returns to the client, as shown in <>.

[[img-distrib-mget]] .Retrieving multiple documents with `mget` image::images/elas_0405.png["Retrieving multiple documents with mget"]

Here is the sequence of steps necessary to retrieve multiple documents with a single `mget` request:

1. The client sends an `mget` request to `Node 1`.
2. `Node 1` builds a multi-get request per shard, and forwards these requests in parallel to the nodes hosting each required primary or replica shard. Once all replies have been received, `Node 1` builds the response and returns it to the client.

A `routing` parameter can (("routing parameter"))be set for each document in the `docs` array.

The bulk API, as depicted in <>, allows the execution of multiple create, index, delete, and update requests within a single bulk request.

[[img-distrib-bulk]] .Multiple document changes with `bulk` image::images/elas_0406.png["Multiple document changes with bulk"]

The sequence of steps(("bulk API", "multiple document changes with"))(("documents", "multiple changes with bulk")) followed by the `bulk` API are as follows:

1. The client sends a `bulk` request to `Node 1`.
2. `Node 1` builds a bulk request per shard, and forwards these requests in parallel to the nodes hosting each involved primary shard.
3. The primary shard executes each action serially, one after another. As each action succeeds, the primary forwards the new document (or deletion) to its replica shards in parallel, and then moves on to the next action. Once all replica shards report success for all actions, the node reports success to the requesting node, which collates the responses and returns them to the client.

The `bulk` API also accepts(("replication request parameter", "in bulk requests"))(("consistency request parameter", "in bulk requests")) the `replication` and `consistency` parameters at the top level for the whole `bulk` request, and the `routing` parameter in the metadata for each request.

[[bulk-format]] [role="pagebreak-before"] ==== Why the Funny Format?

When we learned about bulk requests (((("bulk API", "format of requests")))earlier in <>, you may have asked yourself, `Why does the bulk API require the funny format with the newline characters, instead of just sending the requests wrapped in a JSON array, like the mget` API?"

To answer this, we need to explain a little background: Each document referenced in a bulk request may belong to a different primary shard, each of which may be allocated to any of the nodes in the cluster. This means that ((("action, in bulk requests")))every *action* inside a `bulk` request needs to be forwarded to the correct shard on the correct node.

If the individual requests were wrapped up in a JSON array, that would mean that we would need to do the following:

- Parse the JSON into an array (including the document data, which can be very large)
- Look at each request to determine which shard it should go to
- Create an array of requests for each shard
- Serialize these arrays into the internal transport format
- Send the requests to each shard

It would work, but would need a lot of RAM to hold copies of essentially the same data, and would create many more data structures that the Java Virtual Machine (JVM) would have to spend time garbage collecting.

Instead, Elasticsearch reaches up into the networking buffer, where the raw request has been received, and reads the data directly. It uses the newline characters to identify and parse just the small +action/metadata+ lines in order to decide which shard should handle each request.

These raw requests are forwarded directly to the correct shard. There is no redundant copying of data, no wasted data structures. The entire request process is handled in the smallest amount of memory possible.

[[search]] == Searching--The Basic Tools

So far, we have learned how to use Elasticsearch as a simple NoSQL-style distributed document store. We can ((("searching")))throw JSON documents at Elasticsearch and retrieve each one by ID. But the real power of Elasticsearch lies in its ability to make sense out of chaos -- to turn Big Data into Big Information.

This is the reason that we use structured JSON documents, rather than amorphous blobs of data. Elasticsearch not only *stores* the document, but also *indexes* the content of the document in order to make it searchable.

*Every field in a document is indexed and can be queried*. ((("indexing"))) And it's not just that. During a single query, Elasticsearch can use *all* of these indices, to return results at breath-taking speed. That's something that you could never consider doing with a traditional database.

A *search* can be any of the following:

- A structured query on concrete fields(((("fields", "searching on")))(((("searching", "types of searches"))) like `gender` or `age`, sorted by a field like `join_date`, similar to the type of query that you could construct in SQL
- A full-text query, which finds all documents matching the search keywords, and returns them sorted by *relevance*
- A combination of the two

While many searches will just work out of(((("full text search"))) the box, to use Elasticsearch to its full potential, you need to understand three subjects:

*Mapping*::
How the data in each field is interpreted

*Analysis*::
How full text is processed to make it searchable

*Query DSL*::
The flexible, powerful query language used by Elasticsearch

Each of these is a big subject in its own right, and we explain them in detail in <>. The chapters in this section introduce the basic concepts of all three--just enough to help you to get an overall understanding of how search works.

We will start by explaining the `search` API in its simplest form.

.Test Data

---

The documents that we will use for test purposes in this chapter can be found in this gist: https://gist.github.com/clintongormley/8579281.

You can copy the commands and paste them into your shell in order to follow along with this chapter.

Alternatively, if you're in the online version of this book, you can link:sense_widget.html?snippets/050_Search/Test_data.json[click here to open in Sense].

---

[[empty-search]] === The Empty Search

The most basic form of the(("searching", "empty search"))((("empty search"))) search API is the *empty search*, which doesn't specify any query but simply returns all documents in all indices in the cluster:

# [source,js]

# GET /_search

// SENSE: 050_Search/05_Empty_search.json

The response (edited for brevity) looks something like this:

# [source,js]

{ "hits" : { "total" : 14, "hits" : [ { "_index": "us", "_type": "tweet", "_id": "7", "_score": 1, "_source": { "date": "2014-09-17", "name": "John Smith", "tweet": "The Query DSL is really powerful and flexible", "user_id": 2 } }, ... 9 RESULTS REMOVED ... ], "max_score" : 1 }, "took" : 4, "_shards" : { "failed" : 0, "successful" : 10, "total" : 10 }, "timed_out" : false

# }

==== hits

The most important section of the response is `hits`, which(("searching", "empty search", "hits"))((("hits"))) contains the `total` number of documents that matched our query, and a `hits` array containing the first 10 of those matching documents--the results.

Each result in the `hits` array contains the `_index`, `_type`, and `_id` of the document, plus the `_source` field. This means that the whole document is immediately available to us directly from the search results. This is unlike other search engines, which return just the document ID, requiring you to fetch the document itself in a separate step.

Each element also ((("score", "for empty search")))((("relevance scores")))has a `_score`. This is the *relevance score*, which is a measure of how well the document matches the query. By default, results are returned with the most relevant documents first; that is, in descending order of `_score`. In this case, we didn't specify any query, so all documents are equally relevant, hence the neutral `_score` of `1` for all results.

The `max_score` value is the highest `_score` of any document that matches our query.(("max_score value"))

==== took

The `took` value(("took value (empty search)")) tells us how many milliseconds the entire search request took to execute.

==== shards

The `_shards` element(("shards", "number involved in an empty search")) tells us the `total` number of shards that were involved in the query and,(("failed shards (in a search)"))((("successful shards (in a search)"))) of them, how many were `successful` and how many `failed`. We wouldn't normally expect shards to fail, but it can happen. If we were to suffer a major disaster in which we lost both the primary and the replica copy of the same shard, there would be no copies of that shard available to respond to search requests. In this case, Elasticsearch would report the shard as `failed`, but continue to return results from the remaining shards.

==== timeout

The `timed_out` value tells(("timed_out value in search results"))) us whether the query timed out. By default, search requests do not time out.(("timeout parameter", "specifying in a request"))) If low response times are more important to you than complete results, you can specify a `timeout` as `10` or `10ms` (10 milliseconds), or `1s` (1 second):

# [source,js]

# GET /_search?timeout=10ms

Elasticsearch will return any results that it has managed to gather from each shard before the requests timed out.

# [WARNING]

It should be noted that this `timeout` does not(("timeout parameter", "not halting query execution"))) halt the execution of the query; it merely tells the coordinating node to return the results collected *so far* and to close the connection. In the background, other shards may still be processing the query even though results have been sent.

Use the time-out because it is important to your SLA, not because you want to abort the execution of long-running queries.

==========

[[multi-index-multi-type]] === Multi-index, Multitype

Did you notice that the results from the preceding <> contained documents (("searching", "multi-index, multi-type search")))of different types— `user` and `tweet` —from two different indices— `us` and `gb`?

By not limiting our search to a particular index or type, we have searched across *all* documents in the cluster. Elasticsearch forwarded the search request in parallel to a primary or replica of every shard in the cluster, gathered the results to select the overall top 10, and returned them to us.

Usually, however, you will((("types", "specifying in search requests")))((("indices", "specifying in search requests"))) want to search within one or more specific indices, and probably one or more specific types. We can do this by specifying the index and type in the URL, as follows:

`/_search` ::
Search all types in all indices

`/gb/_search` :: Search all types in the `gb` index

`/gb,us/_search` ::
Search all types in the `gb` and `us` indices

`/g*,u*/_search` ::
Search all types in any indices beginning with `g` or beginning with `u`

`/gb/user/_search` ::
Search type `user` in the `gb` index

`/gb,us/user,tweet/_search` :: Search types `user` and `tweet` in the `gb` and `us` indices

`/_all/user,tweet/_search` ::
Search types `user` and `tweet` in all indices

When you search within a single index, Elasticsearch forwards the search request to a primary or replica of every shard in that index, and then gathers the results from each shard. Searching within multiple indices works in exactly the same way--there are just more shards involved.

# [TIP]

Searching one index that has five primary shards is *exactly equivalent* to searching five indices that have one primary shard each.

==========

Later, you will see how this simple fact makes it easy to scale flexibly as your requirements change.

[[pagination]] === Pagination

Our preceding <> told us that 14 documents in the(("pagination"))) cluster match our (empty) query. But there were only 10 documents in the `hits` array. How can we see the other documents?

In the same way as SQL uses the `LIMIT` keyword to return a single `` `page'' of results, Elasticsearch accepts ((("from parameter")))((("size parameter")))the `` from `and` size` parameters:

`size` :: Indicates the number of results that should be returned, defaults to `10`

`from` :: Indicates the number of initial results that should be skipped, defaults to `0`

If you wanted to show five results per page, then pages 1 to 3 could be requested as follows:

# [source,js]

GET /_search?size=5 GET /_search?size=5&from=5

# GET /_search?size=5&from=10

// SENSE: 050_Search/15_Pagination.json

Beware of paging too deep or requesting too many results at once. Results are sorted before being returned. But remember that a search request usually spans multiple shards. Each shard generates its own sorted results, which then need to be sorted centrally to ensure that the overall order is correct.

.Deep Paging in Distributed Systems

---

To understand why ((("deep paging, problems with")))deep paging is problematic, let's imagine that we are searching within a single index with five primary shards. When we request the first page of results (results 1 to 10), each shard produces its own top 10 results and returns them to the *requesting node*, which then sorts all 50 results in order to select the overall top 10.

Now imagine that we ask for page 1,000--results 10,001 to 10,010. Everything works in the same way except that each shard has to produce its top 10,010 results. The requesting node then sorts through all 50,050 results and discards 50,040 of them!

You can see that, in a distributed system, the cost of sorting results grows exponentially the deeper we page. There is a good reason that web search engines don't return more than 1,000 results for any query.

---

TIP: In <> we explain how you *can* retrieve large numbers of documents efficiently.

[[search-lite]] === Search *Lite*

There are two forms of the `search` API: a ``lite'' *query-string* version that expects all its(("searching", "query string searches")))(("query strings", "searching with"))) parameters to be passed in the query string, and the full *request body* version that expects a JSON request body and uses a rich search language called the query DSL.

The query-string search is useful for running ad hoc queries from the command line. For instance, this query finds all documents of type `tweet` that contain the word `elasticsearch` in the `tweet` field:

# [source,js]

# GET /_all/tweet/_search?q=tweet:elasticsearch

// SENSE: 050_Search/20_Query_string.json

The next query looks for `john` in the `name` field and `mary` in the `tweet` field. The actual query is just

```
+name:john +tweet:mary
```

but the *percent encoding* needed for query-string parameters makes it appear more cryptic than it really is:

# [source,js]

# GET /_search?q=%2Bname%3Ajohn+%2Btweet%3Amary

// SENSE: 050_Search/20_Query_string.json

The `+` prefix indicates conditions that *must* be satisfied for our query to match. Similarly a `-` prefix would indicate conditions that *must not* match. All conditions without a `+` or `-` are optional--the more that match, the more relevant the document.

[[all-field-intro]] ==== The _all Field

This simple search returns all documents that contain the word `mary`:

# [source,js]

# GET /_search?q=mary

// SENSE: 050_Search/20_All_field.json

In the previous examples, we searched for words in the `tweet` or `name` fields. However, the results from this query mention `mary` in three fields:

- A user whose name is Mary
- Six tweets by Mary
- One tweet directed at @mary

How has Elasticsearch managed to find results in three different fields?

When you index a document, Elasticsearch takes the string values of all of its fields and concatenates them into one big string, which it indexes as the special `_all` field.(("_all field", sortas="all field"))) For example, when we index this document:

# [source,js]

{ "tweet": "However did I manage before Elasticsearch?", "date": "2014-09-14", "name": "Mary Jones", "user_id": 1

# }

it's as if we had added an extra field called `_all` with this value:

# [source,js]

# "However did I manage before Elasticsearch? 2014-09-14 Mary Jones 1"

The query-string search uses the `_all` field unless another field name has been specified.

TIP: The `_all` field is a useful feature while you are getting started with a new application. Later, you will find that you have more control over your search results if you query specific fields instead of the `_all` field. When the `_all` field is no longer useful to you, you can disable it, as explained in <>.

[[query-string-query]] [role="pagebreak-before"] ==== More Complicated Queries

The next query searches for tweets, using the following criteria:

- The `name` field contains `mary` or `john`
- The `date` is greater than `2014-09-10`
- The +_all+ field contains either of the words `aggregations` or `geo`

# [source,js]

# +name:(mary john) +date:>2014-09-10 +(aggregations geo)

// SENSE: 050_Search/20_All_field.json

As a properly encoded query string, this looks like the slightly less readable result:

# [source,js]

# ?q=%2Bname%3A(mary+john)+%2Bdate%3A%3E2014-09-10+%2B(aggregations+geo)

As you can see from the preceding examples, this *lite* query-string search is surprisingly powerful.(("query strings",

"syntax, reference for"))) Its query syntax, which is explained in detail in the http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html#query-string-syntax[Query String Syntax] reference docs, allows us to express quite complex queries succinctly. This makes it great for throwaway queries from the command line or during development.

However, you can also see that its terseness can make it cryptic and difficult to debug. And it's fragile--a slight syntax error in the query string, such as a misplaced `-`, `:`, `/`, or `"`, and it will return an error instead of results.

Finally, the query-string search allows any user to run potentially slow, heavy queries on any field in your index, possibly exposing private information or even bringing your cluster to its knees!

# [TIP]

For these reasons, we don't recommend exposing query-string searches directly to your users, unless they are power users who can be trusted with your data and

# with your cluster.

Instead, in production we usually rely on the full-featured *request body* search API, which does all of this, plus a lot more. Before we get there, though, we first need to take a look at how our data is indexed in Elasticsearch.

[[mapping-analysis]] == Mapping and Analysis

While playing around with the data in our index, we notice something odd. Something seems to be broken: we have 12 tweets in our indices, and only one of them contains the date `2014-09-15`, but have a look at the `total` hits for the following queries:

# [source,js]

GET /_search?q=2014 # 12 results GET /_search?q=2014-09-15 # 12 results ! GET /_search?q=date:2014-09-15 # 1 result

# GET /_search?q=date:2014 # 0 results !

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

Why does querying the <> for the full date return all tweets, and querying the `date` field for just the year return no results? Why do our results differ when searching within the `_all` field or the `date` field?

Presumably, it is because the way our data has been indexed in the `_all` field is different from how it has been indexed in the `date` field. So let's take a look at how Elasticsearch has interpreted our document structure, by requesting((("mapping (types)"))) the *mapping* (or schema definition) for the `tweet` type in the `gb` index:

# [source,js]

# GET /gb/_mapping/tweet

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

This gives us the following:

# [source,js]

{ "gb": { "mappings": { "tweet": { "properties": { "date": { "type": "date", "format": "dateOptionalTime" }, "name": { "type": "string" }, "tweet": { "type": "string" }, "user_id": { "type": "long" } } } } }

# }

Elasticsearch has dynamically generated a mapping for us, based on what it could guess about our field types. The response shows us that the `date` field has been recognized as a field of type `date`. (((_all field", sortas="all field")))The `_all` field isn't mentioned because it is a default field, but we know that the `_all` field is of type `string`.(((string fields")))

So fields of type `date` and fields of type `string` are(((indexing", "differences in, for different core types"))) indexed differently, and can thus be searched differently. That's not entirely surprising. You might expect that each of the (((data types", "core, different indexing of")))core data types--strings, numbers, Booleans, and dates--might be indexed slightly differently. And this is true: there are slight differences.

But by far the biggest difference is between fields(((exact values", "fields representing")))(((full text", "fields representing"))) that represent *exact values* (which can include `string` fields) and fields that represent *full text*. This distinction is really important--it's the thing that separates a search engine from all other databases.

=== Exact Values Versus Full Text

Data in Elasticsearch can be broadly divided into two types: exact values and full text.

*Exact values* are exactly what they sound like.((("exact values"))) Examples are a date or a user ID, but can also include exact strings such as a username or an email address. The exact value `Foo` is not the same as the exact value `foo`. The exact value `2014` is not the same as the exact value `2014-09-15`.

*Full text*, on the other hand, refers (((("full text")))to textual data--usually written in some human language -- like the text of a tweet or the body of an email.

# [NOTE]

Full text is often referred to as *unstructured data*, which is a misnomer--natural language is highly structured. The problem is that the rules of natural languages are complex, which makes them difficult for computers to parse correctly. For instance, consider this sentence:

```
May is fun but June bores me.
```

# Does it refer to months or to people?

Exact values are easy to query. The decision is binary; a value either matches the query, or it doesn't. This kind of query is easy to express with SQL:

# [source,js]

WHERE name = "John Smith" AND user_id = 2

# AND date > "2014-09-15"

Querying full-text data is much more subtle. We are not just asking, `Does this document match the query'' but` How *well* does this document match the query?" In other words, how *relevant* is this document to the given query?

We seldom want to match the whole full-text field exactly. Instead, we want to search *within* text fields. Not only that, but we expect search to understand our *intent*:

- A search for `UK` should also return documents mentioning the `United Kingdom`.
- A search for `jump` should also match `jumped`, `jumps`, `jumping`, and perhaps even `leap`.
- `johnny walker` should match `Johnnie Walker`, and `johnnie depp` should match `Johnny Depp`.
- `fox news hunting` should return stories about hunting on Fox News, while `fox hunting news` should return news stories about fox hunting.

To facilitate these types of queries on full-text fields, Elasticsearch first *analyzes* the text, and then uses the results to build an *inverted index*. We will discuss the inverted index and the analysis process in the next two sections.

[[inverted-index]] === Inverted Index

Elasticsearch uses a structure called(((("inverted index", id="ix*invertidx", range="startofrange"))) an _inverted index_, which is designed to allow very fast full-text searches. An inverted index consists of a list of all the unique words that appear in any document, and for each word, a list of the documents in which it appears.

For example, let's say we have two documents, each with a `content` field containing the following:

1. The quick brown fox jumped over the lazy dog
2. Quick brown foxes leap over lazy dogs in summer

To create an inverted index, we first split the `content` field of each document into separate ((("tokenization")))((("terms")))((("tokens")))words (which we call *terms*, or *tokens*), create a sorted list of all the unique terms, and then list in which document each term appears. The result looks something like this:

```
Term      Doc_1  Doc_2
------------------------
Quick   |       |  X
The     |   X   |
brown   |   X   |  X
dog     |   X   |
dogs    |       |  X
fox     |   X   |
foxes   |       |  X
in      |       |  X
jumped  |   X   |
lazy    |   X   |  X
leap    |       |  X
over    |   X   |  X
quick   |   X   |
summer  |       |  X
the     |   X   |
------------------------
```

Now, if we want to search for `quick brown`, we just need to find the documents in which each term appears:

```
Term      Doc_1  Doc_2
------------------------
brown   |   X   |  X
quick   |   X   |
------------------------
Total   |   2   |  1
```

Both documents match, but the first document has more matches than the second. If we apply a naive *similarity algorithm* that((("similarity algorithms"))) just counts the number of matching terms, then we can say that the first document is a better match--is *more relevant* to our query--than the second document.

But there are a few problems with our current inverted index:

- `Quick` and `quick` appear as separate terms, while the user probably thinks of them as the same word.
- `fox` and `foxes` are pretty similar, as are `dog` and `dogs`; They share the same root word.
- `jumped` and `leap`, while not from the same root word, are similar in meaning. They are synonyms.

With the preceding index, a search for `+Quick +fox` wouldn't match any documents. (Remember, a preceding `+` means that the word must be present.) Both the term `Quick` and the term `fox` have to be in the same document in order to satisfy the query, but the first doc contains `quick fox` and the second doc contains `Quick foxes`.

Our user could reasonably expect both documents to match the query. We can do better.

If we normalize the terms into a standard (((("normalization")))format, then we can find documents that contain terms that are not exactly the same as the user requested, but are similar enough to still be relevant. For instance:

- `Quick` can be lowercased to become `quick`.
- `foxes` can be *stemmed*--reduced to its root form--to become `fox`. Similarly, `dogs` could be stemmed to `dog`.
- `jumped` and `leap` are synonyms and can be indexed as just the single term `jump`.

Now the index looks like this:

```
Term      Doc_1  Doc_2
-------------------------
brown   |   X   |  X
dog     |   X   |  X
fox     |   X   |  X
in      |       |  X
jump    |   X   |  X
lazy    |   X   |  X
over    |   X   |  X
quick   |   X   |  X
summer  |       |  X
the     |   X   |  X
------------------------
```

But we're not there yet. Our search for `+Quick +fox` would *still* fail, because we no longer have the exact term `Quick` in our index. However, if we apply the same normalization rules that we used on the `content` field to our query string, it would become a query for `+quick +fox`, which would match both documents!

NOTE: This is very important. You can find only terms that exist in your index, so *both the indexed text and the query string must be normalized into the same form*.

This process of tokenization and normalization is called *analysis*, which we discuss in the next section.((("inverted index", range="endofrange", startref="ix_invertidx")))

[[analysis-intro]] [role="pagebreak-before"] === Analysis and Analyzers

*Analysis* is a (("analysis", "defined")))process that consists of the following:

- First, tokenizing a block of text into individual *terms* suitable for use in an inverted index,
- Then normalizing these terms into a standard form to improve their ``searchability,'' or *recall*

This job is ((("analyzers")))performed by analyzers. An *analyzer* is really just a wrapper that combines three functions into a(("character filters"))) single package:

Character filters::

```
First, the string is passed through any _character filters_ in turn. Their
job is to tidy up the string before tokenization. A character filter could
be used to strip out HTML, or to convert `&` characters to the word
`and`.
```

Tokenizer::

Next, the string is tokenized into individual terms by a *tokenizer*. A simple tokenizer might split the text into terms whenever it encounters whitespace or punctuation.

Token filters::

Last, each term is passed through any *token filters* in turn, which can change terms (for example, lowercasing `Quick`), remove terms (for example, stopwords such as `a`, `and`, `the`) or add terms (for example, synonyms like `jump` and `leap`).

Elasticsearch provides many character filters, ((("token filters")))(("tokenizers")))tokenizers, and token filters out of the box. These can be combined to create custom analyzers suitable for different purposes. We discuss these in detail in <>.

==== Built-in Analyzers

However, Elasticsearch also ships with prepackaged analyzers that you can use directly.(("analyzers", "built-in"))) We list the most important ones next and, to demonstrate the difference in behavior, we show what terms each would produce from this string:

```
"Set the shape to semi-transparent by calling set_trans(5)"
```

Standard analyzer::

The standard analyzer (("standard analyzer")))is the default analyzer that Elasticsearch uses. It is the best general choice for analyzing text that may be in any language. It splits the text on *word boundaries*, as(("word boundaries"))) defined by the http://www.unicode.org/reports/tr29/[Unicode Consortium], and removes most punctuation. Finally, it lowercases all terms. It would produce + set, the, shape, to, semi, transparent, by, calling, set_trans, 5

Simple analyzer::

The simple analyzer splits (("simple analyzer")))the text on anything that isn't a letter, and lowercases the terms. It would produce + set, the, shape, to, semi, transparent, by, calling, set, trans

Whitespace analyzer::

The whitespace analyzer splits (("whitespace analyzer")))the text on whitespace. It doesn't lowercase. It would produce + Set, the, shape, to, semi-transparent, by, calling, set_trans(5)

Language analyzers::

Language-specific analyzers (((("language analyzers")))are available for http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-lang-analyzer.html[many languages]. They are able to take the peculiarities of the specified language into account. For instance, the `english` analyzer comes with a set of English ((("stopwords")))stopwords (common words like `and` or `the` that don't have much impact on relevance), which it removes. This analyzer also is able to *stem* English ((("stemming words")))words because it understands the rules of English grammar. + The `english` analyzer would produce the following: + set, shape, semi, transpar, call, set_tran, 5 + Note how `transparent`, `calling`, and `set_trans` have been stemmed to their root form.

==== When Analyzers Are Used

When we *index* a document, its full-text fields are analyzed into terms that are used to create the inverted index. ((("indexing", "analyzers, use on full text fields"))) However, when we *search* on a full-text field, we need to pass the query string through the *same analysis process*, to ensure that we are searching for terms in the same form as those that exist in the index.

Full-text queries, which we discuss later, understand how each field is defined, and so they can do((("full text", "querying fields representing"))) the right thing:

- When you query a *full-text* field, the query will apply the same analyzer to the query string to produce the correct list of terms to search for.

- When you query an *exact-value* field, the query will not analyze the query string, ((("exact values", "querying fields representing")))but instead search for the exact value that you have specified.

Now you can understand why the queries that we demonstrated at the <> return what they do:

- The `date` field contains an exact value: the single term `2014-09-15`.
- The `_all` field is a full-text field, so the analysis process has converted the date into the three terms: `2014`, `09`, and `15`.

When we query the `_all` field for `2014`, it matches all 12 tweets, because all of them contain the term `2014`:

# [source,sh]

# GET /_search?q=2014 # 12 results

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

When we query the `_all` field for `2014-09-15`, it first analyzes the query string to produce a query that matches *any* of the terms `2014`, `09`, or `15`. This also matches all 12 tweets, because all of them contain the term `2014`:

# [source,sh]

# GET /_search?q=2014-09-15 # 12 results !

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

When we query the `date` field for `2014-09-15`, it looks for that *exact* date, and finds one tweet only:

## [source,sh]

## GET /_search?q=date:2014-09-15 # 1 result

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

When we query the `date` field for `2014`, it finds no documents because none contain that exact date:

## [source,sh]

## GET /_search?q=date:2014 # 0 results !

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

[[analyze-api]] ==== Testing Analyzers

Especially when you are new (("analyzers", "testing")))to Elasticsearch, it is sometimes difficult to understand what is actually being tokenized and stored into your index. To better understand what is going on, you can use the `analyze` API to see how text is analyzed. Specify which analyzer to use in the query-string parameters, and the text to analyze in the body:

## [source,js]

GET /_analyze?analyzer=standard

## Text to analyze

// SENSE: 052_Mapping_Analysis/40_Analyze.json

Each element in the result represents a single term:

## [source,js]

{ "tokens": [ { "token": "text", "start_offset": 0, "end_offset": 4, "type": "", "position": 1 }, { "token": "to", "start_offset": 5, "end_offset": 7, "type": "", "position": 2 }, { "token": "analyze", "start_offset": 8, "end_offset": 15, "type": "", "position": 3 } ]

## }

The `token` is the actual term that will be stored in the index. The `position` indicates the order in which the terms appeared in the original text. The `start_offset` and `end_offset` indicate the character positions that the original word occupied in the original string.

TIP: The `type` values like `<ALPHANUM>` vary (("types", "type values returned by analyzers")))per analyzer and can be ignored. The only place that they are used in Elasticsearch is in the http://www.elasticsearch.org/guide/en/elasticsearch/guide/current/analysis-intro.html#analyze-api[`keep_types` token filter].

The `analyze` API is a useful tool for understanding what is happening inside Elasticsearch indices, and we will talk more about it as we progress.

==== Specifying Analyzers

When Elasticsearch detects a new string field(((("analyzers", "specifying"))) in your documents, it automatically configures it as a full-text `string` field and analyzes it with the `standard` analyzer.((("standard analyzer")))

You don't always want this. Perhaps you want to apply a different analyzer that suits the language your data is in. And sometimes you want a string field to be just a string field--to index the exact value that you pass in, without any analysis, such as a string user ID or an internal status field or tag.

To achieve this, we have to configure these fields manually by specifying the mapping.

[[mapping-intro]] === Mapping

In order to be able to treat date fields as dates, numeric fields as numbers, and string fields as full-text or exact-value strings, Elasticsearch needs to know what type of data each field contains. (((("mapping (types)"))) This information is contained in the mapping.

As explained in <>, each document in an index (((("types", "mapping for")))has a *type*. Every type has its own *mapping*, or *schema definition*.((("schema definition, types"))) A mapping defines the fields within a type, the datatype for each field, and how the field should be handled by Elasticsearch. A mapping is also used to configure metadata associated with the type.

We discuss mappings in detail in <>. In this section, we're going to look at just enough to get you started.

[[core-fields]] ==== Core Simple Field Types

Elasticsearch supports the ((("fields", "core simple types")))(((("types", "core simple field types")))following simple field types:

[horizontal]

- String: `string`
- Whole number: `byte`, `short`, `integer`, `long`
- Floating-point: `float`, `double`
- Boolean: `boolean`
- Date: `date`

When you index a document that contains a new field--one previously not seen--Elasticsearch ((("types", "mapping for", "dynamic mapping of new types")))((("JSON", "datatypes", "simple core types")))((("dynamic mapping")))((("boolean type")))((("long type")))((("double type")))((("date type")))((("strings", "sring type")))will use <> to try to guess the field type from the basic datatypes available in JSON, using the following rules:

[horizontal] *JSON type* :: *Field type*

Boolean: `true` or `false` :: `boolean`

Whole number: `123` :: `long`

Floating point: `123.45` :: `double`

String, valid date: `2014-09-15` :: `date`

String: `foo bar` :: `string`

NOTE: This means that if you index a number in quotes ( `"123"` ), it will be mapped as type `string`, not type `long`. However, if the field is already mapped as type `long`, then Elasticsearch will try to convert the string into a long, and throw an exception if it can't.

==== Viewing the Mapping

We can view the mapping that Elasticsearch has((("mapping (types)", "viewing"))) for one or more types in one or more indices by using the `/_mapping` endpoint. At the <>, we already retrieved the mapping for type `tweet` in index `gb`:

# [source,js]

# GET /gb/_mapping/tweet

This shows us the mapping for the (((("properties")))fields (called *properties*) that Elasticsearch generated dynamically from the documents that we indexed:

# [source,js]

{ "gb": { "mappings": { "tweet": { "properties": { "date": { "type": "date", "format": "dateOptionalTime" }, "name": { "type": "string" }, "tweet": { "type": "string" }, "user_id": { "type": "long" } } } } }

# }

# [TIP]

Incorrect mappings, such as ((("mapping (types)", "incorrect mapping")))having an `age` field mapped as type `string` instead of `integer`, can produce confusing results to your queries.

# Instead of assuming that your mapping is correct, check it!

[[custom-field-mappings]] ==== Customizing Field Mappings

While the basic field datatypes are ((("mapping (types)", "customizing field mappings")))((("fields", "customizing field mappings")))sufficient for many cases, you will often need to customize the mapping (((“string fields”, “customized mappings”)))for individual fields, especially string fields. Custom mappings allow you to do the following:

- Distinguish between full-text string fields and exact value string fields
- Use language-specific analyzers
- Optimize a field for partial matching
- Specify custom date formats
- And much more

The most important attribute of a field is the `type`. For fields other than `string` fields, you will seldom need to map anything other than `type`:

# [source,js]

{ "number_of_clicks": { "type": "integer" }

# }

Fields of type `string` are, by default, considered to contain full text. That is, their value will be passed through(((“analyzers”, “string values passed through”))) an analyzer before being indexed, and a full-text query on the field will pass the query string through an analyzer before searching.

The two most important mapping(((“string fields”, “mapping attributes, index and analyzer”))) attributes for `string` fields are `index` and `analyzer`.

===== index

The `index` attribute controls(("index attribute, strings"))) how the string will be indexed. It can contain one of three values:

`analyzed` :: First analyze the string and then index it. In other words, index this field as full text.

`not_analyzed` ::
Index this field, so it is searchable, but index the value exactly as specified. Do not analyze it.

`no` ::
Don't index this field at all. This field will not be searchable.

The default value of `index` for a `string` field is `analyzed`. If we want to map the field as an exact value, we need to set it to `not_analyzed` :

# [source,js]

{ "tag": { "type": "string", "index": "not_analyzed" }

# }

# [NOTE]

The other simple types (such as `long` , `double` , `date` etc) also accept the `index` parameter, but the only relevant values are `no` and `not_analyzed` ,

# as their values are never analyzed.

===== analyzer

For `analyzed` string fields, use (("analyzer attribute, string fields")))the `analyzer` attribute to specify which analyzer to apply both at search time and at index time. By default, Elasticsearch uses the `standard` analyzer,(("standard analyzer", "specifying another analyzer for strings"))) but you can change this by specifying one of the built-in analyzers, such((("english analyzer"))) as `whitespace` , `simple` , or `english` :

# [source,js]

{ "tweet": { "type": "string", "analyzer": "english" }

# }

In <>, we show you how to define and use custom analyzers as well.

[[updating-a-mapping]] ==== Updating a Mapping

You can specify the mapping for a type when you first ((("types", "mapping for", "updating")))(("mapping (types)", "updating")))create an index. Alternatively, you can add the mapping for a new type (or update the mapping for an existing type) later, using the `/_mapping` endpoint.

# [NOTE]

Although you can *add* to an existing mapping, you can't *change* it. If a field already exists in the mapping, the data from that

# field probably has already been indexed. If you were to change the field mapping, the already indexed data would be wrong and would not be properly searchable.

We can update a mapping to add a new field, but we can't change an existing field from `analyzed` to `not_analyzed`.

To demonstrate both ways of specifying mappings, let's first delete the `gb` index:

# [source,sh]

# DELETE /gb

// SENSE: 052_Mapping_Analysis/45_Mapping.json

Then create a new index, specifying that the `tweet` field should use the `english` analyzer:

# [source,js]

PUT /gb <1> { "mappings": { "tweet" : { "properties" : { "tweet" : { "type" : "string", "analyzer": "english" }, "date" : { "type" : "date" }, "name" : { "type" : "string" }, "user_id" : { "type" : "long" } } } }

# }

// SENSE: 052_Mapping_Analysis/45_Mapping.json

<1> This creates the index with the `mappings` specified in the body.

Later on, we decide to add a new `not_analyzed` text field called `tag` to the `tweet` mapping, using the `_mapping` endpoint:

# [source,js]

PUT /gb/_mapping/tweet { "properties" : { "tag" : { "type" : "string", "index": "not_analyzed" } }

# }

// SENSE: 052_Mapping_Analysis/45_Mapping.json

Note that we didn't need to list all of the existing fields again, as we can't change them anyway. Our new field has been merged into the existing mapping.

==== Testing the Mapping

You can use the `analyze` API to(("mapping (types)", "testing"))) test the mapping for string fields by name. Compare the output of these two requests:

# [source,js]

GET /gb/_analyze?field=tweet Black-cats <1>

GET /gb/_analyze?field=tag

# Black-cats <1>

// SENSE: 052_Mapping_Analysis/45_Mapping.json

<1> The text we want to analyze is passed in the body.

The `tweet` field produces the two terms `black` and `cat`, while the `tag` field produces the single term `Black-cats`. In other words, our mapping is working correctly.

[[complex-core-fields]] === Complex Core Field Types

Besides the simple scalar datatypes that we have mentioned,((("data types", "complex core field types")))((("JSON", "datatypes", "complex"))) JSON also has `null` values, arrays, and objects, all of which are supported by Elasticsearch.

==== Multivalue Fields

It is quite possible that we want our `tag` field (((("fields", "multi-value")))to contain more than one tag. Instead of a single string, we could index an array of tags:

# [source,js]

# { "tag": [ "search", "nosql" ]}

There is no special mapping required for arrays.((("arrays"))) Any field can contain zero, one, or more values, in the same way as a full-text field is analyzed to produce multiple terms.

By implication, this means that *all the values of an array must be of the same datatype*. You can't mix dates with strings. If you create a new field by indexing an array, Elasticsearch will use the datatype of the first value in the array to determine the `type` of the new field.

# [NOTE]

When you get a document back from Elasticsearch, any arrays will be in the same order as when you indexed the document. The `_source` field that you get back contains exactly the same JSON document that you indexed.

However, arrays are *indexed*—made searchable--as multivalue fields, which are unordered. ((("indexing", "of arrays")))((("arrays", "indexed as multi-value fields"))) At search time, you can't refer to `the first element'' or` the last element." Rather, think of an array as a *bag of values*.

# ((("arrays", "empty")))

==== Empty Fields

Arrays can, of course, be empty. ((("fields", "empty")))This is the equivalent of having zero values. In fact, there is no way of storing a `null` value in Lucene, so a field with a `null` value is also considered to be an empty field.((("null values", "empty fields as")))

These four fields would all be considered to be empty, and would not be indexed:

# [source,js]

"null_value": null, "empty_array": [],

# "array_with_null_value": [ null ]

[[inner-objects]] ==== Multilevel Objects

The last native JSON datatype that we need to ((("objects")))discuss is the *object* -- known in other languages as a hash,

hashmap, dictionary or associative array.

*Inner objects* are often used((("objects", "inner objects")))((("inner objects"))) to embed one entity or object inside another. For instance, instead of having fields called `user_name` and `user_id` inside our `tweet` document, we could write it as follows:

# [source,js]

{ "tweet": "Elasticsearch is very flexible", "user": { "id": "@johnsmith", "gender": "male", "age": 26, "name": { "full": "John Smith", "first": "John", "last": "Smith" } }

# }

==== Mapping for Inner Objects

Elasticsearch will detect new object fields((("mapping (types)", "inner objects")))((("inner objects", "mapping for"))) dynamically and map them as type `object`, with each inner field listed under `properties`:

# [source,js]

{ "gb": { "tweet": { <1> "properties": { "tweet": { "type": "string" }, "user": { <2> "type": "object", "properties": { "id": { "type": "string" }, "gender": { "type": "string" }, "age": { "type": "long" }, "name": { <2> "type": "object", "properties": { "full": { "type": "string" }, "first": { "type": "string" }, "last": { "type": "string" } } } } } } } } }

# }

<1> Root object

<2> Inner objects

The mapping for the `user` and `name` fields has a similar structure to the mapping for the `tweet` type itself. In fact, the `type` mapping is just a special type of `object` mapping, which we refer to as the *root object*. ((("root object")))It is just the same as any other object, except that it has some special top-level fields for document metadata, such as `_source`, and the `_all` field.

==== How Inner Objects are Indexed

Lucene doesn't understand inner objects.((("indexing", "of inner objects")))((("inner objects", "indexing of"))) A Lucene document consists of a flat list of key-value pairs. In order for Elasticsearch to index inner objects usefully, it converts our document into something like this:

# [source,js]

{ "tweet": [elasticsearch, flexible, very], "user.id": [@johnsmith], "user.gender": [male], "user.age": [26], "user.name.full": [john, smith], "user.name.first": [john], "user.name.last": [smith]

# }

*Inner fields* can be referred to by((("inner fields"))) name (for example, `first`). To distinguish between two fields that have

the same name, we can use the full *path* (for example, `user.name.first`) or even the `type` name plus the path (`tweet.user.name.first`).

NOTE: In the preceding simple flattened document, there is no field called `user` and no field called `user.name`. Lucene indexes only scalar or simple values, not complex data structures.

[[object-arrays]] ==== Arrays of Inner Objects

Finally, consider how an array containing((("arrays", "of inner objects")))((("inner objects", "arrays of"))) inner objects would be indexed. Let's say we have a `followers` array that looks like this:

# [source,js]

{ "followers": [ { "age": 35, "name": "Mary White"}, { "age": 26, "name": "Alex Jones"}, { "age": 19, "name": "Lisa Smith"} ]

# }

This document will be flattened as we described previously, but the result will look like this:

# [source,js]

{ "followers.age": [19, 26, 35], "followers.name": [alex, jones, lisa, smith, mary, white]

# }

The correlation between `{age: 35}` and `{name: Mary White}` has been lost as each multivalue field is just a bag of values, not an ordered array. This is sufficient for us to ask, "Is there a follower who is 26 years old?"

But we can't get an accurate answer to this: "Is there a follower who is 26 years old *and who is called Alex Jones*?"

Correlated inner objects, which are able to answer queries like these, are called *nested* objects, and we cover them later, in <>.

[[full-body-search]] == Full-Body Search

Search *lite*—a <>—is useful for ad hoc queries from the command line. ((("searching", "request body search", id="ix*reqbodysearch")))To harness the full power of search, however, you should use the _request body* `search` API, ((("request body search"))) so called because most parameters are passed in the HTTP request body instead of in the query string.

Request body search--henceforth known as *search*—not only handles the query itself, but also allows you to return highlighted snippets from your results, aggregate analytics across all results or subsets of results, and return *did-you-mean* suggestions, which will help guide your users to the best results quickly.

=== Empty Search

Let's start with the simplest form of ((("request body search", "empty search")))(("empty search")))the `search` API, the empty search, which returns all documents in all indices:

# [source,js]

GET /_search

# {} <1>

// SENSE: 054_Query_DSL/60_Empty_query.json

<1> This is an empty request body.

Just as with a query-string search, you can search on one, many, or `_all` indices, and one, many, or all types:

# [source,js]

GET /index_2014*/type1,type2/_search

# {}

And you can use the `from` and `size` parameters(("pagination")))(("size parameter")))(("from parameter"))) for pagination:

# [source,js]

GET /_search { "from": 30, "size": 10

# }

[[get_vs_post]] .A GET Request with a Body?

---

The HTTP libraries of certain languages (notably JavaScript) don't allow `GET` requests to have a request body. ((("searching", "using GET and POST HTTP methods for search requests")))(("HTTP methods", "GET and POST, use for search requests")))(("GET method", "no body for GET requests"))) In fact, some users are suprised that `GET` requests are ever allowed to have a body.

The truth is that http://tools.ietf.org/html/rfc7231#page-24[RFC 7231]—the RFC that deals with HTTP semantics and content--does not define what should happen to a `GET` request with a body! As a result, some HTTP servers allow it, and some--especially caching proxies--don't.

The authors of Elasticsearch prefer using `GET` for a search request because they feel that it describes the action--retrieving information--better than the `POST` verb. However, because `GET` with a request body is not universally supported, the `search` API also(("POST method", "use for search requests"))) accepts `POST` requests:

# [source,js]

POST /_search { "from": 30, "size": 10

# }

The same rule applies to any other `GET` API that requires a request body.

---

We present aggregations in depth in <>, but for now, we're going to focus just on the query.

Instead of the cryptic query-string approach, a request body search allows us to write queries by using the *query domain-specific language*, or query DSL. (("searching", "request body search", startref ="ix_reqbodysearch")))

[[query-dsl-intro]] === Query DSL

The query DSL is a flexible, expressive search (((("Query DSL")))language that Elasticsearch uses to expose most of the power of Lucene through a simple JSON interface. It is what you should be using to write your queries in production. It makes your queries more flexible, more precise, easier to read, and easier to debug.

To use the Query DSL, pass a query((("query parameter"))) in the `query` parameter:

# [source,js]

GET /_search { "query": YOUR_QUERY_HERE

# }

The *empty search*— `{}` —is ((("empty search", "equivalent to match_all query clause")))functionally equivalent to using the `match_all` query clause, which,((("match_all query clause"))) as the name suggests, matches all documents:

# [source,js]

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

# }

// SENSE: 054_Query_DSL/60_Empty_query.json

==== Structure of a Query Clause

A query clause typically((("Query DSL", "structure of a query clause"))) has this structure:

# [source,js]

{ QUERY_NAME: { ARGUMENT: VALUE, ARGUMENT: VALUE,... }

# }

If it references one particular field, it has this structure:

# [source,js]

{ QUERY_NAME: { FIELD_NAME: { ARGUMENT: VALUE, ARGUMENT: VALUE,... } }

# }

For instance, you can use a `match` query clause((("match query"))) to find tweets that mention `elasticsearch` in the `tweet` field:

# [source,js]

{ "match": { "tweet": "elasticsearch" }

# }

The full search request would look like this:

# [source,js]

GET /_search { "query": { "match": { "tweet": "elasticsearch" } }

# }

// SENSE: 054_Query_DSL/60_Match_query.json

==== Combining Multiple Clauses

*Query clauses* are simple building blocks(((("Query DSL", "combining multiple clauses"))) that can be combined with each other to create complex queries. Clauses can be as follows:

- *Leaf clauses* (like the `match` clause) that(((("leaf clauses"))) are used to compare a field (or fields) to a query string.
- *Compound* clauses that are used ((("compound query clauses")))to combine other query clauses. For instance, a `bool` clause(((("bool clause"))) allows you to combine other clauses that either `must` match, `must_not` match, or `should` match if possible:

# [source,js]

{ "bool": { "must": { "match": { "tweet": "elasticsearch" }}, "must_not": { "match": { "name": "mary" }}, "should": { "match": { "tweet": "full text" }} }

# }

// SENSE: 054_Query_DSL/60_Bool_query.json

It is important to note that a compound clause can combine *any* other query clauses, including other compound clauses. This means that compound clauses can be nested within each other, allowing the expression of very complex logic.

As an example, the following query looks for emails that contain `business opportunity` and should either be starred, or be both in the Inbox and not marked as spam:

# [source,js]

{ "bool": { "must": { "match": { "email": "business opportunity" }}, "should": [ { "match": { "starred": true }}, { "bool": { "must": { "folder": "inbox" }}, "must_not": { "spam": true }} } ], "minimum_should_match": 1 }

# }

Don't worry about the details of this example yet; we will explain in full later. The important thing to take away is that a

compound query clause can combine multiple clauses--both leaf clauses and other compound clauses--into a single query.

=== Queries and Filters

Although we refer to the query DSL, in reality there are two DSLs: the query DSL and the filter DSL.(((("DSL (Domain Specific Language)", "Query and Filter DSL")))(("Filter DSL"))) Query clauses and filter clauses are similar in nature, but have slightly different purposes.

A *filter* asks a yes|no question of(((("filters", "queries versus")))(("exact values", "filters with yes|no questions for fields containing"))) every document and is used for fields that contain exact values:

- Is the `created` date in the range `2013` - `2014`?
- Does the `status` field contain the term `published`?
- Is the `lat_lon` field within `10km` of a specified point?

A *query* is similar to a filter, but also asks(("queries", "filters versus")) the question: How *well* does this document match?

A typical use for a query is to find documents

- Best matching the words `full text search`
- Containing the word `run`, but maybe also matching `runs`, `running`, `jog`, or `sprint`
- Containing the words `quick`, `brown`, and `fox`—the closer together they are, the more relevant the document
- Tagged with `lucene`, `search`, or `java`—the more tags, the more relevant the document

A query calculates how *relevant* each document(("relevance", "calculation by queries")) is to the query, and assigns it a relevance `_score`, which is later used to sort matching documents by relevance. This concept of relevance is well suited to full-text search, where there is seldom a completely ``correct'' answer.

==== Performance Differences

The output from most filter clauses--a simple(("filters", "performance, queries versus")) list of the documents that match the filter--is quick to calculate and easy to cache in memory, using only 1 bit per document. These cached filters can be reused efficiently for subsequent requests.

Queries have to not only find(("queries", "performance, filters versus")) matching documents, but also calculate how relevant each document is, which typically makes queries heavier than filters. Also, query results are not cachable.

Thanks to the inverted index, a simple query that matches just a few documents may perform as well or better than a cached filter that spans millions of documents. In general, however, a cached filter will outperform a query, and will do so consistently.

The goal of filters is to *reduce the number of documents that have to be examined by the query*.

==== When to Use Which

As a general rule, use(("filters", "when to use"))(("queries", "when to use")) query clauses for *full-text* search or for any condition that should affect the *relevance score*, and use filter clauses for everything else.

=== Most Important Queries and Filters

While Elasticsearch comes with many queries and filters, you will use just a few frequently. We discuss them in much greater detail in <> but next we give you a quick introduction to the most important queries and filters.

==== term Filter

The `term` filter is used to filter by(((("filters", "important")))(((("term filter")))) exact values, be they numbers, dates, Booleans, or `not_analyzed` exact-value string fields:

# [source,js]

{ "term": { "age": 26 }} { "term": { "date": "2014-09-01" }} { "term": { "public": true }}

# { "term": { "tag": "full_text" }}

// SENSE: 054_Query_DSL/70_Term_filter.json

==== terms Filter

The `terms` filter is((("terms filter"))) the same as the `term` filter, but allows you to specify multiple values to match. If the field contains any of the specified values, the document matches:

# [source,js]

# { "terms": { "tag": [ "search", "full_text", "nosql" ] }}

// SENSE: 054_Query_DSL/70_Terms_filter.json

==== range Filter

The `range` filter allows you to find(((("range filters"))) numbers or dates that fall into a specified range:

# [source,js]

{ "range": { "age": { "gte": 20, "lt": 30 } }

# }

// SENSE: 054_Query_DSL/70_Range_filter.json

The operators that it accepts are as follows:

`gt` ::
Greater than

`gte` ::
Greater than or equal to

`lt` ::

Less than

`lte` ::
Less than or equal to

==== exists and missing Filters

The `exists` and `missing` filters are (("exists filter")))((("missing filter")))used to find documents in which the specified field either has one or more values ( `exists` ) or doesn't have any values ( `missing` ). It is similar in nature to `IS_NULL` ( `missing` ) and `NOT IS_NULL` ( `exists` )in SQL:

# [source,js]

{ "exists": { "field": "title" }

# }

// SENSE: 054_Query_DSL/70_Exists_filter.json

These filters are frequently used to apply a condition only if a field is present, and to apply a different condition if it is missing.

==== bool Filter

The `bool` filter is used (("bool filter")))((("must clause", "in bool filters")))((("must_not clause", "in bool filters")))((("should clause", "in bool filters")))to combine multiple filter clauses using Boolean logic. ((("bool filter", "must, must_not, and should clauses"))) It accepts three parameters:

`must` :: These clauses *must* match, like `and` .

`must_not` :: These clauses *must not* match, like `not` .

`should` :: At least one of these clauses must match, like `or` .

Each of these parameters can accept a single filter clause or an array of filter clauses:

# [source,js]

{ "bool": { "must": { "term": { "folder": "inbox" }}, "must_not": { "term": { "tag": "spam" }}, "should": [ { "term": { "starred": true }}, { "term": { "unread": true }} ] }

# }

// SENSE: 054_Query_DSL/70_Bool_filter.json

==== match_all Query

The `match_all` query simply(("match_all query")))((("queries", "important"))) matches all documents. It is the default query that is used if no query has been specified:

# [source,js]

# { "match_all": {}}

// SENSE: 054_Query_DSL/70_Match_all_query.json

This query is frequently used in combination with a filter--for instance, to retrieve all emails in the inbox folder. All documents are considered to be equally relevant, so they all receive a neutral `_score` of `1`.

==== match Query

The `match` query should be the standard(("match query"))) query that you reach for whenever you want to query for a full-text or exact value in almost any field.

If you run a `match` query against a full-text field, it will analyze the query string by using the correct analyzer for that field before executing the search:

# [source,js]

# { "match": { "tweet": "About Search" }}

// SENSE: 054_Query_DSL/70_Match_query.json

If you use it on a field containing an exact value, ((("exact values", "searching for, match queries and")))such as a number, a date, a Boolean, or a `not_analyzed` string field, then it will search for that exact value:

# [source,js]

{ "match": { "age": 26 }} { "match": { "date": "2014-09-01" }} { "match": { "public": true }}

# { "match": { "tag": "full_text" }}

// SENSE: 054_Query_DSL/70_Match_query.json

TIP: For exact-value searches, you probably want to use a filter instead of a query, as a filter will be cached.

Unlike the query-string search that we showed in <>, the `match` query does not use a query syntax like `+user_id:2 +tweet:search`. It just looks for the words that are specified. This means that it is safe to expose to your users via a search field; you control what fields they can query, and it is not prone to throwing syntax errors.

==== multi_match Query

The `multi_match` query allows(("multi_match queries"))) to run the same `match` query on multiple fields:

# [source,js]

{ "multi_match": { "query": "full text search", "fields": [ "title", "body" ] }

# }

// SENSE: 054_Query_DSL/70_Multi_match_query.json

==== bool Query

The `bool` query, like the `bool` filter,((("bool query"))) is used to combine multiple query clauses. However, there are some differences. Remember that while filters give binary yes/no answers, queries calculate a relevance score instead. The `bool` query combines the `_score` from each `must` or `should` clause that matches.((("bool query", "must, must_not, and should clauses")))((("should clause", "in bool queries")))((("must_not clause", "in bool queries")))((("must clause", "in bool queries"))) This query accepts the following parameters:

`must` ::
Clauses that *must* match for the document to be included.

`must_not` ::
Clauses that *must not* match for the document to be included.

`should` ::
If these clauses match, they increase the `_score`; otherwise, they have no effect. They are simply used to refine the relevance score for each document.

The following query finds documents whose `title` field matches the query string `how to make millions` and that are not marked as `spam`. If any documents are `starred` or are from 2014 onward, they will rank higher than they would have otherwise. Documents that match *both* conditions will rank even higher:

# [source,js]

{ "bool": { "must": { "match": { "title": "how to make millions" }}, "must_not": { "match": { "tag": "spam" }}, "should": [ { "match": { "tag": "starred" }}, { "range": { "date": { "gte": "2014-01-01" }}} ] }

# }

// SENSE: 054_Query_DSL/70_Bool_query.json

TIP: If there are no `must` clauses, at least one `should` clause has to match. However, if there is at least one `must` clause, no `should` clauses are required to match.

=== Combining Queries with Filters

Queries can be used in *query context*, and filters can be used in *filter context*. ((("filters", "combining with queries"))) ((("queries", "combining with filters"))) Throughout the Elasticsearch API, you will see parameters with `query` or `filter` in the name. These expect a single argument containing either a single query or filter clause respectively. In other words, they establish the outer *context* as query context or filter context.

Compound query clauses can wrap other query clauses, and compound filter clauses can wrap other filter clauses. However, it is often useful to apply a filter to a query or, less frequently, to use a full-text query as a filter.

To do this, there are dedicated query clauses that wrap filter clauses, and vice versa, thus allowing us to switch from one context to another. It is important to choose the correct combination of query and filter clauses to achieve your goal in the most efficient way.

[[filtered-query]] ==== Filtering a Query

Let's say we have(((("queries", "combining with filters", "filtering a query")))((("filters", "combining with queries", "filtering a query"))) this query:

# [source,js]

# { "match": { "email": "business opportunity" }}

We want to combine it with the following `term` filter, which will match only documents that are in our inbox:

# [source,js]

# { "term": { "folder": "inbox" }}

The `search` API accepts only a single `query` parameter, so we need to wrap the query and the filter in another query, called the `filtered` query:

# [source,js]

{ "filtered": { "query": { "match": { "email": "business opportunity" }}, "filter": { "term": { "folder": "inbox" }} }

# }

We can now pass this query to the `query` parameter of the `search` API:

# [source,js]

GET /_search { "query": { "filtered": { "query": { "match": { "email": "business opportunity" }}, "filter": { "term": { "folder": "inbox" }} } }

# }

// SENSE: 054_Query_DSL/75_Filtered_query.json

[role="pagebreak-before"] ==== Just a Filter

While in query context, if ((("filters", "combining with queries", "using just a filter in query context")))((("queries", "combining with filters", "using just a filter in query context")))you need to use a filter without a query (for instance, to match all emails in the inbox), you can just omit the query:

# [source,js]

GET /_search { "query": { "filtered": { "filter": { "term": { "folder": "inbox" }} } }

# }

// SENSE: 054_Query_DSL/75_Filtered_query.json

If a query is not specified it defaults to using the `match_all` query, so the preceding query is equivalent to the following:

# [source,js]

GET /_search { "query": { "filtered": { "query": { "match_all": {}}, "filter": { "term": { "folder": "inbox" }} } }

# }

==== A Query as a Filter

Occasionally, you will want to use a query while you are in filter context. This can be achieved with the `query` filter, which ((("filters", "combining with queries", "query as a filter")))((("queries", "combining with filters", "query filter")))just wraps a query. The following example shows one way we could exclude emails that look like spam:

# [source,js]

GET /_search { "query": { "filtered": { "filter": { "bool": { "must": { "term": { "folder": "inbox" }}, "must_not": { "query": { <1> "match": { "email": "urgent business proposal" } } } } } } }

# }

// SENSE: 054_Query_DSL/75_Filtered_query.json

<1> Note the `query` filter, which is allowing us to use the `match` *query* inside a `bool` *filter*.

NOTE: You seldom need to use a query as a filter, but we have included it for completeness' sake. The only time you may need it is when you need to use full-text matching while in filter context.

=== Validating Queries

Queries can become quite complex and, especially(((("validate query API")))(((("queries", "validating"))) when combined with different analyzers and field mappings, can become a bit difficult to follow. The `validate-query` API can be used to check whether a query is valid.

# [source,js]

GET /gb/tweet/_validate/query { "query": { "tweet" : { "match" : "really powerful" } }

# }

// SENSE: 054_Query_DSL/80_Validate_query.json

The response to the preceding `validate` request tells us that the query is invalid:

# [source,js]

{ "valid" : false, "_shards" : { "total" : 1, "successful" : 1, "failed" : 0 }

# }

==== Understanding Errors

To find out ((("validate query API", "understqnding errors")))why it is invalid, add the `explain` parameter(((("explain parameter"))) to the query string:

# [source,js]

GET /gb/tweet/_validate/query?explain <1> { "query": { "tweet" : { "match" : "really powerful" } }

# }

// SENSE: 054_Query_DSL/80_Validate_query.json

<1> The `explain` flag provides more information about why a query is invalid.

Apparently, we've mixed up the type of query ( `match` ) with the name of the field ( `tweet` ):

# [source,js]

{ "valid" : false, "_shards" : { ... }, "explanations" : [ { "index" : "gb", "valid" : false, "error" : "org.elasticsearch.index.query.QueryParsingException: [gb] No query registered for [tweet]" } ]

# }

==== Understanding Queries

Using the `explain` parameter has the added advantage of returning a human-readable description of the (valid) query, which can be useful for understanding exactly how your query has been interpreted by Elasticsearch:

# [source,js]

GET /_validate/query?explain { "query": { "match" : { "tweet" : "really powerful" } }

# }

// SENSE: 054_Query_DSL/80_Understanding_queries.json

An `explanation` is returned for each index (((("indices", "explanation for each index queried")))that we query, because each index can have different mappings and analyzers:

# [source,js]

{ "valid" : true, "_shards" : { ... }, "explanations" : [ { "index" : "us", "valid" : true, "explanation" : "tweet:really tweet:powerful" }, { "index" : "gb", "valid" : true, "explanation" : "tweet:realli tweet:power" } ]

# }

From the `explanation`, you can see how the `match` query for the query string `really powerful` has been rewritten as two single-term queries against the `tweet` field, one for each term.

Also, for the `us` index, the two terms are `really` and `powerful`, while for the `gb` index, the terms are `realli` and `power`. The reason for this is that we changed the `tweet` field in the `gb` index to use the `english` analyzer.

[[sorting]] == Sorting and Relevance

By default, results are returned sorted by *relevance*—with the most relevant docs first.((("sorting", "by relevance"))) ((("relevance", "sorting results by"))) Later in this chapter, we explain what we mean by *relevance* and how it is calculated, but let's start by looking at the `sort` parameter and how to use it.

=== Sorting

In order to sort by relevance, we need to represent relevance as a value. In Elasticsearch, the *relevance score* is represented by the floating-point number returned in the search results as the `_score`, ((("relevance scores", "returned in search results score")))((("score", "relevance score of search results")))so the default sort order is `_score` descending.

Sometimes, though, you don't have a meaningful relevance score. For instance, the following query just returns all tweets whose `user_id` field has the value `1`:

# [source,js]

GET /_search { "query" : { "filtered" : { "filter" : { "term" : { "user_id" : 1 } } } }

# }

Filters have no bearing on `_score`, and the((("score", seealso="relevance; relevance scores")))((("match_all query", "score as neutral 1")))((("filters", "score and"))) missing-but-implied `match_all` query just sets the `_score` to a neutral value of `1` for all documents. In other words, all documents are considered to be equally relevant.

==== Sorting by Field Values

In this case, it probably makes sense to sort tweets by recency, with the most recent tweets first.((("sorting", "by field values")))((("fields", "sorting search results by field values")))((("sort parameter"))) We can do this with the `sort` parameter:

# [source,js]

GET /_search { "query" : { "filtered" : { "filter" : { "term" : { "user_id" : 1 }} } }, "sort": { "date": { "order": "desc" }}

# }

// SENSE: 056_Sorting/85_Sort_by_date.json

You will notice two differences in the results:

# [source,js]

"hits" : { "total" : 6, "max_score" : null, <1> "hits" : [ { "_index" : "us", "_type" : "tweet", "_id" : "14", "_score" : null, <1> "_source" : { "date": "2014-09-24", ... }, "sort" : [ 1411516800000 ] <2> }, ...

# }

<1> The `_score` is not calculated, because it is not being used for sorting.

<2> The value of the `date` field, expressed as milliseconds since the epoch, is returned in the `sort` values.

The first is that we have (((("date field, sorting search results by")))a new element in each result called `sort`, which contains the value(s) that was used for sorting. In this case, we sorted on `date`, which internally is((("milliseconds-since-the-epoch (date)"))) indexed as *milliseconds since the epoch*. The long number `1411516800000` is equivalent to the date string `2014-09-24 00:00:00 UTC`.

The second is that the `_score` and `max_score` are both `null`. ((("score", "not calculating"))) Calculating the `_score` can be quite expensive, and usually its only purpose is for sorting; we're not sorting by relevance, so it doesn't make sense to keep track of the `_score`. If you want the `_score` to be calculated regardless, you can set((("track_scores parameter"))) the `track_scores` parameter to `true`.

# [TIP]

As a shortcut, you can ((("sorting", "specifying just the field name to sort on")))specify just the name of the field to sort on:

# [source,js]

```
"sort": "number_of_children"
```

---

Fields will be sorted in ((("sorting", "default ordering")))ascending order by default, and

# the `_score` value in descending order.

==== Multilevel Sorting

Perhaps we want to combine the `_score` from a((("sorting", "multilevel")))((("multilevel sorting"))) query with the `date`, and show all matching results sorted first by date, then by relevance:

# [source,js]

GET /_search { "query" : { "filtered" : { "query": { "match": { "tweet": "manage text search" }}, "filter" : { "term" : { "user_id" : 2 }} } }, "sort": [ { "date": { "order": "desc" }}, { "_score": { "order": "desc" }} ]

# }

// SENSE: 056_Sorting/85_Multilevel_sort.json

Order is important. Results are sorted by the first criterion first. Only results whose first `sort` value is identical will then be sorted by the second criterion, and so on.

Multilevel sorting doesn't have to involve the `_score`. You could sort by using several different fields,((("fields", "sorting by multiple fields"))) on geo-distance or on a custom value calculated in a script.

# [NOTE]

Query-string search((("sorting", "in query string searches")))((("sort parameter", "using in query strings")))((("query strings", "sorting search results for"))) also supports custom sorting, using the `sort` parameter in the query string:

# [source,js]

# GET /_search?sort=date:desc&sort=_score&q=search

====

==== Sorting on Multivalue Fields

When sorting on fields with more than one value,((("sorting", "on multivalue fields")))((("fields", "multivalue", "sorting on"))) remember that the values do not have any intrinsic order; a multivalue field is just a bag of values. Which one do you choose to sort on?

For numbers and dates, you can reduce a multivalue field to a single value by using the `min`, `max`, `avg`, or `sum` *sort modes*. (((("sum sort mode")))((("avg sort mode")))((("max sort mode")))((("min sort mode")))((("sort modes")))((("dates field, sorting on earliest value")))For instance, you could sort on the earliest date in each `dates` field by using the following:

[role="pagebreak-before"]

# [source,js]

"sort": { "dates": { "order": "asc", "mode": "min" }

# }

[[multi-fields]] === String Sorting and Multifields

Analyzed string fields are also multivalue fields,(("strings", "sorting on string fields")))((("analyzed fields", "string fields")))((("sorting", "string sorting and multifields"))) but sorting on them seldom gives you the results you want. If you analyze a string like `fine old art`, it results in three terms. We probably want to sort alphabetically on the first term, then the second term, and so forth, but Elasticsearch doesn't have this information at its disposal at sort time.

You could use the `min` and `max` sort modes (it uses `min` by default), but that will result in sorting on either `art` or `old`, neither of which was the intent.

In order to sort on a string field, that field should contain one term only: the whole `not_analyzed` string.(("not_analyzed string fields", "sorting on"))) But of course we still need the field to be `analyzed` in order to be able to query it as full text.

The naive approach to indexing the same string in two ways would be to include two separate fields in the document: one that is `analyzed` for searching, and one that is `not_analyzed` for sorting.

But storing the same string twice in the `_source` field is waste of space. What we really want to do is to pass in a *single field* but to *index it in two different ways*. All of the *core* field types (strings, numbers, Booleans, dates) accept a `fields` parameter ((("mapping (types)", "transforming simple mapping to multifield mapping")))((("types", "core simple field types", "accepting fields parameter")))((("fields parameter")))((("multifield mapping")))that allows you to transform a simple mapping like

# [source,js]

"tweet": { "type": "string", "analyzer": "english"

# }

into a *multifield* mapping like this:

# [source,js]

"tweet": { <1> "type": "string", "analyzer": "english", "fields": { "raw": { <2> "type": "string", "index": "not_analyzed" } }

# }

// SENSE: 056_Sorting/88_Multifield.json

<1> The main `tweet` field is just the same as before: an `analyzed` full-text field.

<2> The new `tweet.raw` subfield is `not_analyzed`.

Now, or at least as soon as we have reindexed our data, we can use the `tweet` field for search and the `tweet.raw` field for sorting:

# [source,js]

GET /_search { "query": { "match": { "tweet": "elasticsearch" } }, "sort": "tweet.raw"

# }

// SENSE: 056_Sorting/88_Multifield.json

WARNING: Sorting on a full-text `analyzed` field can use a lot of memory. See <> for more information.

[[relevance-intro]] === What Is Relevance?

We've mentioned that, by default, results are returned in descending order of relevance.(("relevance", "defined"))) But what is relevance? How is it calculated?

The relevance score of each document is represented by a positive floating-point number called the `_score`.(("score", "calculation of"))) The higher the `_score`, the more relevant the document.

A query clause generates a `_score` for each document. How that score is calculated depends on the type of query clause. (("fuzzy queries", "calculation of relevence score"))) Different query clauses are used for different purposes: a `fuzzy` query might determine the `_score` by calculating how similar the spelling of the found word is to the original search term; a `terms` query would incorporate the percentage of terms that were found. However, what we usually mean by *relevance* is the algorithm that we use to calculate how similar the contents of a full-text field are to a full-text query string.

The standard *similarity algorithm* used in Elasticsearch is((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm")))((("similarity algorithms", "Term Frequency/Inverse Document Frequency (TF/IDF)"))) known as *term frequency/inverse document frequency*, or *TF/IDF*, which takes the following factors into(("inverse document frequency"))) account:

Term frequency::

How often does the term appear in the field? The more often, the more relevant. A field containing five mentions of the same term is more likely to be relevant than a field containing just one mention.

Inverse document frequency::

How often does each term appear in the index? The more often, the *less* relevant. Terms that appear in many documents have a lower *weight* than more-uncommon terms.

Field-length norm::

How long is the field? The longer it is, the less likely it is that words in the field will be relevant. A term appearing in a short `title` field carries more weight than the same term appearing in a long `content` field.

Individual (("field-length norm")))queries may combine the TF/IDF score with other factors such as the term proximity in phrase queries, or term similarity in fuzzy queries.

Relevance is not just about full-text search, though. It can equally be applied to yes/no clauses, where the more clauses that match, the higher the `_score`.

When multiple query clauses are combined using a compound query(("compound query clauses", "relevance score for results"))) like the `bool` query, the `_score` from each of these query clauses is combined to calculate the overall `_score` for the document.

TIP: We have a whole chapter dedicated to relevance calculations and how to bend them to your will: <>.

[[explain]] ==== Understanding the Score

When debugging a complex query,(("score", "calculation of")))((("relevance scores", "understanding"))) it can be difficult to understand exactly how a `_score` has been calculated. Elasticsearch has the option of producing an *explanation* with every search result, by setting the `explain` parameter(("explain parameter"))) to `true`.

# [source,js]

GET /_search?explain <1> { "query" : { "match" : { "tweet" : "honeymoon" }}

# }

// SENSE: 056_Sorting/90_Explain.json

<1> The `explain` parameter adds an explanation of how the `_score` was calculated to every result.

# [NOTE]

Adding `explain` produces a lot((("explain parameter", "for relevance score calculation"))) of output for every hit, which can look overwhelming, but it is worth taking the time to understand what it all means. Don't worry if it doesn't all make sense now; you can refer to this section

# when you need it. We'll work through the output for one `hit` bit by bit.

First, we have the metadata that is returned on normal search requests:

# [source,js]

{ "_index" : "us", "_type" : "tweet", "_id" : "12", "_score" : 0.076713204,

```
"_source" :     { ... trimmed ... },
```

It adds information about the shard and the node that the document came from, which is useful to know because term and document frequencies are calculated per shard, rather than per index:

# [source,js]

```
"_shard" :      1,
"_node" :       "mzIVYCsqSWCG_M_ZffSs9Q",
```

Then it provides the `_explanation`. Each ((("explanation of relevance score calculation")))((("description", "of relevance score calculations")))entry contains a `description` that tells you what type of calculation is being performed, a `value` that gives you the result of the calculation, and the `details` of any subcalculations that were required:

# [source,js]

"_explanation": { <1> "description": "weight(tweet:honeymoon in 0) [PerFieldSimilarity], result of:", "value": 0.076713204, "details": [ { "description": "fieldWeight in 0, product of:", "value": 0.076713204, "details": [ { <2> "description": "tf(freq=1.0), with freq of:", "value": 1, "details": [ { "description": "termFreq=1.0", "value": 1 } ] }, { <3> "description": "idf(docFreq=1, maxDocs=1)", "value": 0.30685282 }, { <4> "description": "fieldNorm(doc=0)", "value": 0.25, } ] } ]

# }

<1> Summary of the score calculation for `honeymoon`

<2> Term frequency

<3> Inverse document frequency

<4> Field-length norm

WARNING: Producing the `explain` output is expensive.((("explain parameter", "overhead of using"))) It is a debugging tool only. Don't leave it turned on in production.

The first part is the summary of the calculation. It tells us that it has calculated the *weight*—the ((("weight", "calculation of")))((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm", "weight calculation for a term")))TF/IDF--of the term `honeymoon` in the field `tweet`, for document `0`. (This is an internal document ID and, for our purposes, can be ignored.)

It then provides details((("field-length norm")))((("inverse document frequency"))) of how the weight was calculated:

Term frequency::

How many times did the term `honeymoon` appear in the `tweet` field in this document?

Inverse document frequency::

How many times did the term `honeymoon` appear in the `tweet` field of all documents in the index?

Field-length norm::

How long is the `tweet` field in this document? The longer the field, the smaller this number.

Explanations for more-complicated queries can appear to be very complex, but really they just contain more of the same calculations that appear in the preceding example. This information can be invaluable for debugging why search results appear in the order that they do.

# [TIP]

The output from `explain` can be difficult to read in JSON, but it is easier

# when it is formatted as YAML.((("explain parameter", "formatting output in YAML")))((("YAML, formatting explain output in"))) Just add `format=yaml` to the query string.

[[explain-api]] ==== Understanding Why a Document Matched

While the `explain` option adds an explanation for every result, you can use the `explain` API to understand why one particular document matched or, more important, why it *didn't* match.((("relevance", "understanding why a document matched")))((("explain API, understanding why a document matched")))

The path for the request is `/index/type/id/_explain`, as in the following:

# [source,js]

GET /us/tweet/12/_explain { "query" : { "filtered" : { "filter" : { "term" : { "user_id" : 2 }}, "query" : { "match" : { "tweet" : "honeymoon" }} } }

# }

// SENSE: 056_Sorting/90_Explain_API.json

Along with the full explanation(("description", "of why a document didn't match"))) that we saw previously, we also now have a `description` element, which tells us this:

# [source,js]

# "failure to match filter: cache(user_id:[2 TO 2])"

In other words, our `user_id` filter clause is preventing the document from matching.

[[fielddata-intro]] === Fielddata

Our final topic in this chapter is about an internal aspect of Elasticsearch. While we don't demonstrate any new techniques here, fielddata is an important topic that we will refer to repeatedly, and is something that you should be aware of. (((“fielddata”)))

When you sort on a field, Elasticsearch needs access to the value of that field for every document that matches the query. (((“inverted index”, “sorting and”))) The inverted index, which performs very well when searching, is not the ideal structure for sorting on field values:

- When searching, we need to be able to map a term to a list of documents.
- When sorting, we need to map a document to its terms. In other words, we need to ``uninvert'' the inverted index.

To make sorting efficient, Elasticsearch loads all the values for the field that you want to sort on into memory. This is referred to as *fielddata*.

WARNING: Elasticsearch doesn't just load the values for the documents that matched a particular query. It loads the values from *every document in your index*, regardless of the document `type`.

The reason that Elasticsearch loads all values into memory is that uninverting the index from disk is slow. Even though you may need the values for only a few docs for the current request, you will probably need access to the values for other docs on the next request, so it makes sense to load all the values into memory at once, and to keep them there.

Fielddata is used in several places in Elasticsearch:

- Sorting on a field
- Aggregations on a field
- Certain filters (for example, geolocation filters)
- Scripts that refer to fields

Clearly, this can consume a lot of memory, especially for high-cardinality string fields--string fields that have many unique values--like the body of an email. Fortunately, insufficient memory is a problem that can be solved by horizontal scaling, by adding more nodes to your cluster.

For now, all you need to know is what fielddata is, and to be aware that it can be memory hungry. Later, we will show you how to determine the amount of memory that fielddata is using, how to limit the amount of memory that is available to it, and how to preload fielddata to improve the user experience.

# Distributed Search Execution

Before moving on, we are going to take a detour and talk about how search is executed in a distributed environment.It is a bit more complicated than the basic *create-read-update-delete* (CRUD) requests that we discussed in [distributed-docs].

> ## Content Warning
>
> The information presented in this chapter is for your interest. You are not required to understand and remember all the detail in order to use Elasticsearch.
>
> Read this chapter to gain a taste for how things work, and to know where the information is in case you need to refer to it in the future, but don't be overwhelmed by the detail.

A CRUD operation deals with a single document that has a unique combination of `_index`, `_type`, and `routing-value` (which defaults to the document's `_id`). This means that we know exactly which shard in the cluster holds that document.

Search requires a more complicated execution model because we don't know which documents will match the query: they could be on any shard in the cluster. A search request has to consult a copy of every shard in the index or indices we're interested in to see if they have any matching documents.

But finding all matching documents is only half the story. Results from multiple shards must be combined into a single sorted list before the `search` API can return a ``page'' of results. For this reason, search is executed in a two-phase process called *query then fetch*.

# Query Phase

During the initial *query phase*, the query is broadcast to a shard copy (a primary or replica shard) of every shard in the index. Each shard executes the search locally and builds a *priority queue* of matching documents.

## Priority Queue

A *priority queue* is just a sorted list that holds the *top-n* matching documents. The size of the priority queue depends on the pagination parameters `from` and `size`. For example, the following search request would require a priority queue big enough to hold 100 documents:

```
GET /_search
{
    "from": 90,
    "size": 10
}
```

The query phase process is depicted in Query phase of distributed search.

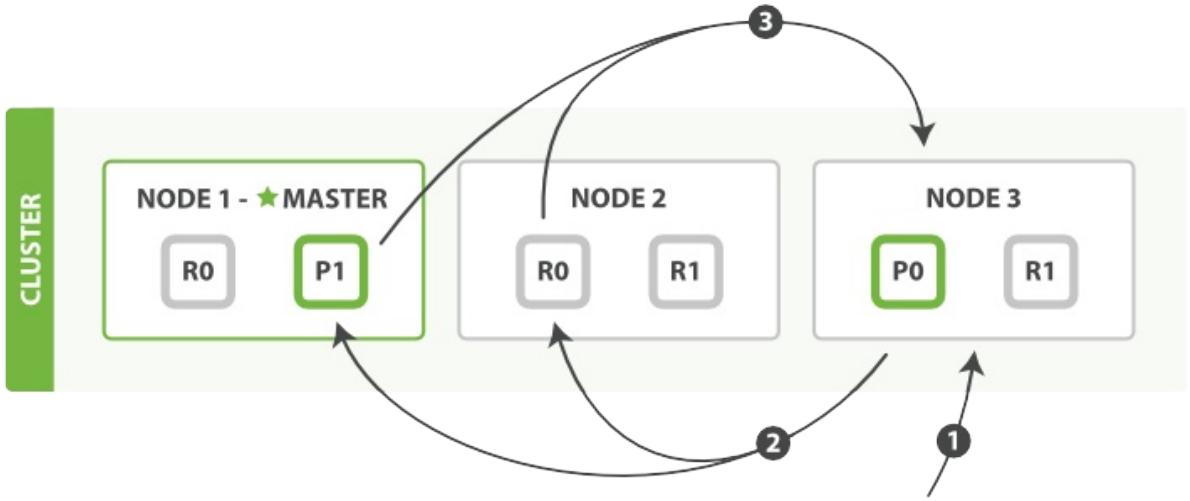



Figure 1. Query phase of distributed search

The query phase consists of the following three steps:

1. The client sends a `search` request to `Node 3`, which creates an empty priority queue of size `from + size`.
2. `Node 3` forwards the search request to a primary or replica copy of every shard in the index. Each shard executes the query locally and adds the results into a local sorted priority queue of size `from + size`.
3. Each shard returns the doc IDs and sort values of all the docs in its priority queue to the coordinating node, `Node 3`, which merges these values into its own priority queue to produce a globally sorted list of results.

When a search request is sent to a node, that node becomes the coordinating node. It is the job of this node to broadcast the search request to all involved shards, and to gather their responses into a globally sorted result set that it can return to the client.

The first step is to broadcast the request to a shard copy of every node in the index. Just like document `GET` requests,

search requests can be handled by a primary shard or by any of its replicas. This is how more replicas (when combined with more hardware) can increase search throughput. A coordinating node will round-robin through all shard copies on subsequent requests in order to spread the load.

Each shard executes the query locally and builds a sorted priority queue of length `from + size`—in other words, enough results to satisfy the global search request all by itself. It returns a lightweight list of results to the coordinating node, which contains just the doc IDs and any values required for sorting, such as the `_score`.

The coordinating node merges these shard-level results into its own sorted priority queue, which represents the globally sorted result set. Here the query phase ends.

> **NOTE**
>
> An index can consist of one or more primary shards,((("indices", "multi-index search"))) so a search request against a single index needs to be able to combine the results from multiple shards. A search against *multiple* or *all* indices works in exactly the same way--there are just more shards involved.

# Fetch Phase

The query phase identifies which documents satisfythe search request, but we still need to retrieve the documents themselves. This is the job of the fetch phase, shown in Fetch phase of distributed search.

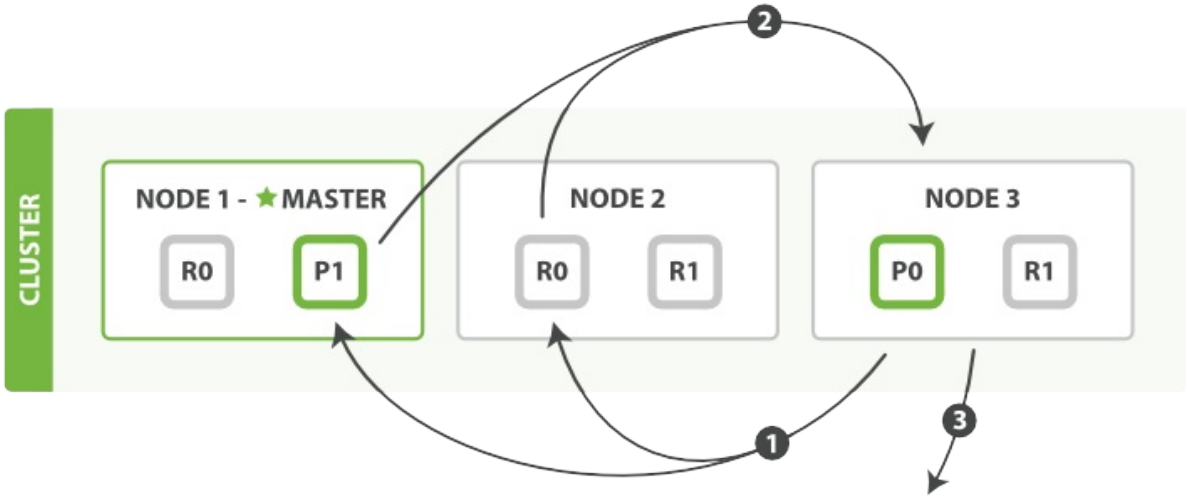



Figure 1. Fetch phase of distributed search

The distributed phase consists of the following steps:

1. The coordinating node identifies which documents need to be fetched and issues a multi `GET` request to the relevant shards.

2. Each shard loads the documents and *enriches* them, if required, and then returns the documents to the coordinating node.

3. Once all documents have been fetched, the coordinating node returns the results to the client.

The coordinating node first decides which documents *actually* need to be fetched. For instance, if our query specified `{ "from": 90, "size": 10 }`, the first 90 results would be discarded and only the next 10 results would need to be retrieved. These documents may come from one, some, or all of the shards involved in the original search request.

The coordinating node builds a `multi-get request` for each shard that holds a pertinent document and sends the request to the same shard copy that handled the query phase.

The shard loads the document bodies--the `_source` field--and, if requested, enriches the results with metadata and search snippet highlighting. Once the coordinating node receives all results, it assembles them into a single response that it returns to the client.

**Deep Pagination**

---

The query-then-fetch process supports pagination with the `from` and `size` parameters, but *within limits*. Remember that each shard must build a priority queue of length `from + size`, all of which need to be passed back to the coordinating node. And the coordinating node needs to sort through `number_of_shards * (from + size)` documents in order to find the correct `size` documents.

Depending on the size of your documents, the number of shards, and the hardware you are using, paging 10,000 to 50,000

results (1,000 to 5,000 pages) deep should be perfectly doable. But with big-enough `from` values, the sorting process can become very heavy indeed, using vast amounts of CPU, memory, and bandwidth. For this reason, we strongly advise against deep paging.

In practice, "deep pagers" are seldom human anyway. A human will stop paging after two or three pages and will change the search criteria. The culprits are usually bots or web spiders that tirelessly keep fetching page after page until your servers crumble at the knees.

If you *do* need to fetch large numbers of docs from your cluster, you can do so efficiently by disabling sorting with the `scan` search type, which we discuss later in this chapter.

---

# Search Options

A few optional query-string parameters can influence the search process.

## preference

The `preference` parameter allows you to control which shards or nodes are used to handle the search request. It accepts values such as `_primary`, `_primary_first`, `_local`, `_only_node:xyz`, `_prefer_node:xyz`, and `_shards:2,3`, which are explained in detail on the search `preference` documentation page.

However, the most generally useful value is some arbitrary string, to avoid the *bouncing results* problem.

**Bouncing Results**

---

Imagine that you are sorting your results by a `timestamp` field, and two documents have the same timestamp. Because search requests are round-robined between all available shard copies, these two documents may be returned in one order when the request is served by the primary, and in another order when served by the replica.

This is known as the *bouncing results* problem: every time the user refreshes the page, the results appear in a different order. The problem can be avoided by always using the same shards for the same user, which can be done by setting the `preference` parameter to an arbitrary string like the user's session ID.

---

## timeout

By default, the coordinating node waits to receive a response from all shards. If one node is having trouble, it could slow down the response to all search requests.

The `timeout` parameter tells the coordinating node how long it should wait before giving up and just returning the results that it already has. It can be better to return some results than none at all.

The response to a search request will indicate whether the search timed out and how many shards responded successfully:

```
...
"timed_out":     true, [1]
"_shards": {
   "total":      5,
   "successful": 4,
   "failed":     1     [2]
},
...
```

[1] The search request timed out.

[2] One shard out of five failed to respond in time.

If all copies of a shard fail for other reasons--perhaps because of a hardware failure--this will also be reflected in the `_shards` section of the response.

## routing

In routing-value, we explained how a custom `routing` parameter could be provided at index time to ensure that all related documents, such as the documents belonging to a single user, are stored on a single shard. At search time, instead of

searching on all the shards of an index, you can specify one or more `routing` values to limit the search to just those shards:

```
GET /_search?routing=user_1,user2
```

This technique comes in handy when designing very large search systems, and we discuss it in detail in scale.

## search_type

While `query_then_fetch` is the default search type, other search types can be specified for particular purposes, for example:

```
GET /_search?search_type=count
```

***count***

The `count` search type has only a `query` phase. It can be used when you don't need search results, just a document count or aggregations on documents matching the query.

***query_and_fetch***

The `query_and_fetch` search type combines the query and fetch phases into a single step. This is an internal optimization that is used when a search request targets a single shard only, such as when a `routing` value has been specified. While you can choose to use this search type manually, it is almost never useful to do so.

***`dfs_query_then_fetch`* and *`dfs_query_and_fetch`***

The `dfs` search types(("dfs search types"))) have a prequery phase that fetches the term frequencies from all involved shards in order to calculate global term frequencies. We discuss this further in <>.

`scan`::

The `scan` search type is(("scan search type"))) used in conjunction with the `scroll` API (("scroll API")))to retrieve large numbers of results efficiently. It does this by disabling sorting. We discuss *scan-and-scroll* in the next section.

[[scan-scroll]] === scan and scroll

The `scan` search type and the `scroll` API(("scroll API", "scan and scroll")) are used together to retrieve large numbers of documents from Elasticsearch efficiently, without paying the penalty of deep pagination.

`scroll` ::

## +

A *scrolled search* allows us to(("scrolled search")) do an initial search and to keep pulling batches of results from Elasticsearch until there are no more results left. It's a bit like a *cursor* in (("cursors")))a traditional database.

A scrolled search takes a snapshot in time. It doesn't see any changes that are made to the index after the initial search request has been made. It does this by keeping the old data files around, so that it can preserve its ``view'' on what the index looked like at the time it started.

--

`scan` ::

The costly part of deep pagination is the global sorting of results, but if we disable sorting, then we can return all documents quite cheaply. To do this, we use the `scan` search type.(("scan search type"))) Scan instructs Elasticsearch to do no sorting, but to just return the next batch of results from every shard that still has results to return.

To use *scan-and-scroll*, we execute a search(("scan-and-scroll"))) request setting `search_type` to(("search_type", "scan and scroll"))) `scan`, and passing a `scroll` parameter telling Elasticsearch how long it should keep the scroll open:

# [source,js]

GET /old_index/_search?search_type=scan&scroll=1m <1> { "query": { "match_all": {}}, "size": 1000

# }

<1> Keep the scroll open for 1 minute.

The response to this request doesn't include any hits, but does include a `_scroll_id`, which is a long Base-64 encoded(("scroll_id"))) string. Now we can pass the `_scroll_id` to the `_search/scroll` endpoint to retrieve the first batch of results:

# [source,js]

GET /_search/scroll?scroll=1m <1>
c2Nhbjs1OzExODpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlBOzExOTpRNV9aY1VyUVM4U0 <2>
NMd2pjWlJ3YWlBOzExNjpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlBOzExNzpRNV9aY1Vy
UVM4U0NMd2pjWlJ3YWlBOzEyMDpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlBOzE7dG90YW

# xfaGl0czoxOw==

<1> Keep the scroll open for another minute.

<2> The `_scroll_id` can be passed in the body, in the URL, or as a query parameter.

Note that we again specify `?scroll=1m`. The scroll expiry time is refreshed every time we run a scroll request, so it needs to give us only enough time to process the current batch of results, not all of the documents that match the query.

The response to this scroll request includes the first batch of results. Although we specified a `size` of 1,000, we get back many more documents.((("size parameter", "in scanning"))) When scanning, the `size` is applied to each shard, so you will get back a maximum of `size * number_of_primary_shards` documents in each batch.

NOTE: The scroll request also returns a *new* `_scroll_id`. Every time we make the next scroll request, we must pass the `_scroll_id` returned by the *previous* scroll request.

When no more hits are returned, we have processed all matching documents.

TIP: Some of the http://www.elasticsearch.org/guide[official Elasticsearch clients] provide *scan-and-scroll* helpers that provide an easy wrapper around this functionality.((("clients", "providing scan-and-scroll helpers")))

[[index-management]] == Index Management

We have seen how Elasticsearch makes it easy to start developing a new application without requiring any advance planning or setup. However, it doesn't take long before you start wanting to fine-tune the indexing and search process to better suit your particular use case. Almost all of these customizations relate to the index, and the types that it contains. In this chapter, we introduce the APIs for managing indices and type mappings, and the most important settings.

include::070_Index_Mgmt/05_Create_Delete.asciidoc[]

include::070_Index_Mgmt/10_Settings.asciidoc[]

include::070_Index_Mgmt/15_Configure_Analyzer.asciidoc[]

include::070_Index_Mgmt/20_Custom_Analyzers.asciidoc[]

include::070_Index_Mgmt/25_Mappings.asciidoc[]

include::070_Index_Mgmt/30_Root_Object.asciidoc[]

include::070_Index_Mgmt/35_Dynamic_Mapping.asciidoc[]

include::070_Index_Mgmt/40_Custom_Dynamic_Mapping.asciidoc[]

include::070_Index_Mgmt/45_Default_Mapping.asciidoc[]

include::070_Index_Mgmt/50_Reindexing.asciidoc[]

include::070_Index_Mgmt/55_Aliases.asciidoc[]

=== Creating an Index

Until now, we have created a new index(((("indices", "creating"))) by simply indexing a document into it. The index is created with the default settings, and new fields are added to the type mapping by using dynamic mapping. Now we need more control over the process: we want to ensure that the index has been created with the appropriate number of primary shards, and that analyzers and mappings are set up *before* we index any data.

To do this, we have to create the index manually, passing in any settings or type mappings in the request body, as follows:

# [source,js]

PUT /my_index { "settings": { ... any settings ... }, "mappings": { "type_one": { ... any mappings ... }, "type_two": { ... any mappings ... }, ... }

# }

In fact, if you want to, you ((("indices", "preventing automatic creation of")))can prevent the automatic creation of indices by adding the following setting to the `config/elasticsearch.yml` file on each node:

# [source,js]

# action.auto_create_index: false

# [NOTE]

Later, we discuss how you can use <> to preconfigure automatically created indices. This is particularly useful when indexing log data: you log into an index whose name includes the date and, as midnight rolls over, a new properly configured index automatically springs into

# existence.

=== Deleting an Index

To delete an index, use ((("HTTP methods", "DELETE")))((("DELETE method", "deleting indices")))((("indices", "deleting")))the following request:

# [source,js]

# DELETE /my_index

You can delete multiple indices with this:

# [source,js]

DELETE /index_one,index_two

## DELETE /index_*

You can even delete *all* indices with this:

## [source,js]

## DELETE /_all

=== Index Settings

There are many many knobs(("index settings")) that you can twiddle to customize index behavior, which you can read about in the http://www.elasticsearch.org/guide/en/elasticsearch/guide/current/_index_settings.html#_index_settings[Index Modules reference documentation], but...

TIP: Elasticsearch comes with good defaults. Don't twiddle these knobs until you understand what they do and why you should change them.

Two of the most important(("shards", "number_of_shards index setting"))((("number_of_shards setting")))((("index settings", "number_of_shards"))) settings are as follows:

`number_of_shards` ::

```
The number of primary shards that an index should have,
which defaults to `5`.  This setting cannot be changed
after index creation.
```

`number_of_replicas` ::

```
The number of replica shards (copies) that each primary shard
should have, which defaults to `1`.  This setting can be changed
at any time on a live index.
```

For instance, we could create a small index--just(((("index settings", "number_of_replicas")))((("replica shards", "number_of_replicas index setting"))) one primary shard--and no replica shards with the following request:

# [source,js]

PUT /my_temp_index { "settings": { "number_of_shards" : 1, "number_of_replicas" : 0 }

# }

// SENSE: 070_Index_Mgmt/10_Settings.json

Later, we can change the number of replica shards dynamically using the `update-index-settings` API as(("update-index-settings API"))) follows:

# [source,js]

PUT /my_temp_index/_settings { "number_of_replicas": 1

# }

// SENSE: 070_Index_Mgmt/10_Settings.json

[[configuring-analyzers]] === Configuring Analyzers

The third important index setting is the `analysis` section,((("index settings", "analysis"))) which is used to configure existing analyzers or to create new custom analyzers specific to your index.

In <>, we introduced some of the built-in (((("analyzers", "built-in")))analyzers, which are used to convert full-text strings into an inverted index, suitable for searching.

The `standard` analyzer, which is the default analyzer used for full-text fields,((("standard analyzer", "components of"))) is a good choice for most Western languages.((("tokenization", "in standard analyzer")))((("standard token filter")))((("stop token filter")))((("standard tokenizer")))((("lowercase token filter"))) It consists of the following:

- The `standard` tokenizer, which splits the input text on word boundaries
- The `standard` token filter, which is intended to tidy up the tokens emitted by the tokenizer (but currently does nothing)
- The `lowercase` token filter, which converts all tokens into lowercase
- The `stop` token filter, which removes stopwords--common words that have little impact on search relevance, such as `a`, `the`, `and`, `is`.

By default, the stopwords filter is disabled. You can enable it by creating a custom analyzer based on the `standard` analyzer and setting the `stopwords` parameter.((("stopwords parameter"))) Either provide a list of stopwords or tell it to use a predefined stopwords list from a particular language.

In the following example, we create a new analyzer called the `es_std` analyzer, which uses the predefined list of (((("Spanish", "analyzer using Spanish stopwords")))Spanish stopwords:

# [source,js]

PUT /spanish*docs { "settings": { "analysis": { "analyzer": { "es_std": { "type": "standard", "stopwords": "_spanish*" } } } }

# }

// SENSE: 070_Index_Mgmt/15_Configure_Analyzer.json

The `es_std` analyzer is not global--it exists only in the `spanish_docs` index where we have defined it. To test it with the `analyze` API, we must specify the index name:

# [source,js]

GET /spanish_docs/_analyze?analyzer=es_std

# El veloz zorro marrón

// SENSE: 070_Index_Mgmt/15_Configure_Analyzer.json

The abbreviated results show that the Spanish stopword `El` has been removed correctly:

# [source,js]

{ "tokens" : [ { "token" : "veloz", "position" : 2 }, { "token" : "zorro", "position" : 3 }, { "token" : "marrón", "position" : 4 } ]

```
}
```

[[custom-analyzers]] === Custom Analyzers

While Elasticsearch comes with a number of analyzers available out of the box, the real power comes from the ability to create your own custom analyzers by combining character filters, tokenizers, and token filters in a configuration that suits your particular data.

In <>, we said that an *analyzer* is a wrapper that combines three functions into a single package,(("analyzers", "character filters, tokenizers, and token filters in"))) which are executed in sequence:

Character filters::

+

---

Character filters(("character filters"))) are used to `tidy up'' a string before it is tokenized. For instance, if our text is in HTML format, it will contain HTML tags like

`or`

`that we don't want to be indexed. We can use the http://bit.ly/1B6f4Ay[`html_strip`character filter] to remove all HTML tags and to convert HTML entities like`Á`into the corresponding Unicode character`Á`.

# An analyzer may have zero or more character filters.

Tokenizers::

+

---

An analyzer *must* have a single tokenizer.(("tokenizers", "in analyzers"))) The tokenizer breaks up the string into individual terms or tokens. The http://bit.ly/1E3Fd1b[`standard` tokenizer], which is used(("standard tokenizer"))) in the `standard` analyzer, breaks up a string into individual terms on word boundaries, and removes most punctuation, but other tokenizers exist that have different behavior.

For instance, the http://bit.ly/1ICd585[`keyword` tokenizer] outputs exactly(("keyword tokenizer"))) the same string as it received, without any tokenization. The http://bit.ly/1xt3t7d[`whitespace` tokenizer] splits text(("whitespace tokenizer"))) on whitespace only. The http://bit.ly/1ICdozA[`pattern` tokenizer] can

# be used to split text on a (("pattern tokenizer")))matching regular expression.

Token filters::

+

---

After tokenization, the resulting *token stream* is passed through any specified token filters,(("token filters"))) in the order in which they are specified.

Token filters may change, add, or remove tokens. We have already mentioned the http://bit.ly/1DIeXvZ[`lowercase`] and http://bit.ly/1INX4tN[`stop` token filters], but there are many more available in Elasticsearch. http://bit.ly/1AUfpDN[Stemming token filters] `stem'' words to ((("stemming token filters")))their root form. The http://bit.ly/1ylU7Q7[`ascii_folding`filter] removes diacritics,((("ascii_folding filter"))) converting a term

`like` "très" `into` "tres" `. The http://bit.ly/1CbkmYe[` ngram `] and http://bit.ly/1DIf6j5[` edge_ngram` token filters] can produce(((("edge_engram token filter")))((("ngram and edge_ngram token filters")))

# tokens suitable for partial matching or autocomplete.

In <>, we discuss examples of where and how to use these tokenizers and filters. But first, we need to explain how to create a custom analyzer.

==== Creating a Custom Analyzer

In the same way as(((("index settings", "analysis", "creating custom analyzers")))((("analyzers", "custom", "creating"))) we configured the `es_std` analyzer previously, we can configure character filters, tokenizers, and token filters in their respective sections under `analysis`:

# [source,js]

PUT /my_index { "settings": { "analysis": { "char_filter": { ... custom character filters ... }, "tokenizer": { ... custom tokenizers ... }, "filter": { ... custom token filters ... }, "analyzer": { ... custom analyzers ... } } }

# }

As an example, let's set up a custom analyzer that will do the following:

1. Strip out HTML by using the `html_strip` character filter.
2. Replace `&` characters with `" and "`, using a custom `mapping` character filter: + [source,js]

---

"char_filter": { "&_to_and": { "type": "mapping", "mappings": [ "&=> and "] }

# }

1. Tokenize words, using the `standard` tokenizer.
2. Lowercase terms, using the `lowercase` token filter.
3. Remove a custom list of stopwords, using a custom `stop` token filter: + [source,js]

---

"filter": { "my_stopwords": { "type": "stop", "stopwords": [ "the", "a" ] }

# }

Our analyzer definition combines the predefined tokenizer and filters with the custom filters that we have configured previously:

# [source,js]

"analyzer": { "my_analyzer": { "type": "custom", "char_filter": [ "html_strip", "&_to_and" ], "tokenizer": "standard", "filter": [

"lowercase", "my_stopwords" ] }

# }

To put it all together, the whole `create-index` request(("create-index request")) looks like this:

# [source,js]

PUT /my_index { "settings": { "analysis": { "char_filter": { "&_to_and": { "type": "mapping", "mappings": [ "&=> and "] }}, "filter": { "my_stopwords": { "type": "stop", "stopwords": [ "the", "a" ] }}, "analyzer": { "my_analyzer": { "type": "custom", "char_filter": [ "html_strip", "&_to_and" ], "tokenizer": "standard", "filter": [ "lowercase", "my_stopwords" ] }}

# }}}

// SENSE: 070_Index_Mgmt/20_Custom_analyzer.json

After creating the index, use the `analyze` API to(("analyzers", "testing using analyze API")) test the new analyzer:

# [source,js]

GET /my_index/_analyze?analyzer=my_analyzer

# The quick & brown fox

// SENSE: 070_Index_Mgmt/20_Custom_analyzer.json

The following abbreviated results show that our analyzer is working correctly:

# [source,js]

{ "tokens" : [ { "token" : "quick", "position" : 2 }, { "token" : "and", "position" : 3 }, { "token" : "brown", "position" : 4 }, { "token" : "fox", "position" : 5 } ]

# }

The analyzer is not much use unless we tell (("analyzers", "custom", "telling Elasticsearch where to use")))((("mapping (types)", "applying custom analyzer to a string field")))Elasticsearch where to use it. We can apply it to a `string` field with a mapping such as the following:

# [source,js]

PUT /my_index/_mapping/my_type { "properties": { "title": { "type": "string", "analyzer": "my_analyzer" } }

# }

// SENSE: 070_Index_Mgmt/20_Custom_analyzer.json

[[mapping]] === Types and Mappings

A *type* in Elasticsearch represents a class of similar documents.(("types", "defined"))) A type consists of a *name*—such as `user` or `blogpost`—and a *mapping*. The mapping, ((("mapping (types)")))like a database schema, describes the fields or *properties* that documents of that type may have, ((("fields", "datatypes")))the datatype of each field--such as `string`, `integer`, or `date`—and how those fields should be indexed and stored by Lucene.

In <>, we said that a type is like a table in a relational database. While this is a useful way to think about types initially, it is worth explaining in more detail exactly what a type is and how they are implemented on top of Lucene.

==== How Lucene Sees Documents

A document in Lucene consists of a simple list of field-value pairs.((("documents", "in Lucene"))) A field must have at least one value, but any field can contain multiple values. Similarly, a single string value may be converted into multiple values by the analysis process. Lucene doesn't care if the values are strings or numbers or dates--all values are just treated as *opaque bytes*.

When we index a document in Lucene, the values for each field are added to the inverted index for the associated field. Optionally, the original values may also be *stored* unchanged so that they can be retrieved later.

==== How Types Are Implemented

Elasticsearch types are ((("types", "implementation in Elasticsearch")))implemented on top of this simple foundation. An index may have several types, each with its own mapping, and documents of any of these types may be stored in the same index.

Because Lucene has no concept of document types, the type name of each document is stored with the document in a metadata field called `_type`.((("type field"))) When we search for documents of a particular type, Elasticsearch simply uses a filter on the `_type` field to restrict results to documents of that type.

Lucene also has no concept of mappings.((("mapping (types)"))) Mappings are the layer that Elasticsearch uses to *map* complex JSON documents into the simple flat documents that Lucene expects to receive.

For instance, the mapping for the `name` field in the `user` type may declare that the field is a `string` field, and that its value should be analyzed by the `whitespace` analyzer before being indexed into the inverted index called `name`:

# [source,js]

"name": { "type": "string", "analyzer": "whitespace"

# }

==== Avoiding Type Gotchas

The fact that documents of different types can be added to the same index introduces some unexpected((("types", "gotchas, avoiding"))) complications.

Imagine that we have two types in our index: `blog_en` for blog posts in English, and `blog_es` for blog posts in Spanish. Both types have a `title` field, but one type uses the `english` analyzer and the other type uses the `spanish` analyzer.

The problem is illustrated by the following query:

# [source,js]

GET /_search { "query": { "match": { "title": "The quick brown fox" } }

# }

We are searching in the `title` field in both types. The query string needs to be analyzed, but which analyzer does it use: `spanish` or `english`? It will use the analyzer for the first `title` field that it finds, which will be correct for some docs and incorrect for the others.

We can avoid this problem either by naming the fields differently--for example, `title_en` and `title_es`—or by explicitly including the type name in the field name and querying each field separately:

# [source,js]

GET /_search { "query": { "multi_match": { <1> "query": "The quick brown fox", "fields": [ "blog_en.title", "blog_es.title" ] } }

# }

<1> The `multi_match` query runs a `match` query on multiple fields and combines the results.

Our new query uses the `english` analyzer for the field `blog_en.title` and the `spanish` analyzer for the field `blog_es.title`, and combines the results from both fields into an overall relevance score.

This solution can help when both fields have the same datatype, but consider what would happen if you indexed these two documents into the same index:

- Type: user

# [source,js]

# { "login": "john_smith" }

[role="pagebreak-before"]

- Type: event

# [source,js]

# { "login": "2014-06-01" }

Lucene doesn't care that one field contains a string and the other field contains a date. It will happily index the byte values from both fields.

However, if we now try to *sort* on the `event.login` field, Elasticsearch needs to load the values in the `login` field into memory. As we said in <>, it loads the values for *all documents* in the index regardless of their type.

It will try to load these values either as a string or as a date, depending on which `login` field it sees first. This will either produce unexpected results or fail outright.

TIP: To ensure that you don't run into these conflicts, it is advisable to ensure that fields with the *same name* are mapped in the *same way* in every type in an index.

[[root-object]] === The Root Object

The uppermost level of a mapping is known (((("mapping (types)", "root object")))((("root object")))as the *root object*. It may contain the following:

- A *properties* section, which lists the mapping for each field that a document may contain
- Various metadata fields, all of which start with an underscore, such as `_type`, `_id`, and `_source`
- Settings, which control how the dynamic detection of new fields is handled, such as `analyzer`, `dynamic_date_formats`, and `dynamic_templates`
- Other settings, which can be applied both to the root object and to fields of type `object`, such as `enabled`, `dynamic`, and `include_in_all`

==== Properties

We have already discussed the three most important settings for document fields or (((("root object", "properties")))(((("properties", "important settings")))properties in <> and <>:

`type` :: The datatype that the field contains, such as `string` or `date`

`index` ::
Whether a field should be searchable as full text ( `analyzed` ), searchable as an exact value ( `not_analyzed` ), or not searchable at all ( `no` )

`analyzer` ::
Which `analyzer` to use for a full-text field, both at index time and at search time

We will discuss other field types such as `ip`, `geo_point`, and `geo_shape` in the appropriate sections later in the book.

include::31_Metadata_source.asciidoc[]

include::32_Metadata_all.asciidoc[]

include::33_Metadata_ID.asciidoc[]

[[source-field]] ==== Metadata: _source Field

By default, Elasticsearch (((("metadata, document", "_source field")))((("_source field", sortas="source field")))stores the JSON string representing the document body in the `_source` field. Like all stored fields, the `_source` field is compressed before being written to disk.

This is almost always desired functionality because it means the following:

- The full document is available directly from the search results--no need for a separate round-trip to fetch the document from another data store.
- Partial `update` requests will not function without the `_source` field.
- When your mapping changes and you need to reindex your data, you can do so directly from Elasticsearch instead of having to retrieve all of your documents from another (usually slower) data store.
- Individual fields can be extracted from the `_source` field and returned in `get` or `search` requests when you don't need to see the whole document.
- It is easier to debug queries, because you can see exactly what each document contains, rather than having to guess their contents from a list of IDs.

That said, storing the `_source` field does use disk space. If none of the preceding reasons is important to you, you can disable the `_source` field with the following mapping:

# [source,js]

PUT /my_index { "mappings": { "my_type": { "_source": { "enabled": false } } }

# }

In a search request, you can ask for only certain fields by specifying the `_source` parameter in the request body:

# [source,js]

GET /_search { "query": { "match_all": {}}, "_source": [ "title", "created" ]

# }

// SENSE: 070_Index_Mgmt/31_Source_field.json

Values for these fields will be extracted from the `_source` field and returned instead of the full `_source`.

.Stored Fields

---

Besides indexing the values of a field, you (((("stored fields")))(("fields", "stored")))can also choose to `store` the original field value for later retrieval. Users with a Lucene background use stored fields to choose which fields they would like to be able to return in their search results. In fact, the `_source` field is a stored field.

In Elasticsearch, setting individual document fields to be stored is usually a false optimization. The whole document is already stored as the `_source` field. It is almost always better to just extract the fields that you need by using the `_source`

parameter.

[[all-field]] ==== Metadata: _all Field

In <>, we introduced the `_all` field: a special field that indexes the (((("metadata, document", "_all field")))((("_all field", sortas="all field")))values from all other fields as one big string. The `query_string` query clause (and searches performed as `?q=john`) defaults to searching in the `_all` field if no other field is specified.

The `_all` field is useful during the exploratory phase of a new application, while you are still unsure about the final structure that your documents will have. You can throw any query string at it and you have a good chance of finding the document you're after:

# [source,js]

GET /_search { "match": { "_all": "john smith marketing" }

# }

As your application evolves and your search requirements become more exacting, you will find yourself using the `_all` field less and less. The `_all` field is a shotgun approach to search. By querying individual fields, you have more flexbility, power, and fine-grained control over which results are considered to be most relevant.

# [NOTE]

One of the important factors taken into account by the <> is the length of the field: the shorter the field, the more important. A term that appears in a short `title` field is likely to be more important than the same term that appears somewhere in a long `content` field. This distinction

# between field lengths disappears in the `_all` field.

If you decide that you no longer need the `_all` field, you can disable it with this mapping:

# [source,js]

PUT /my_index/_mapping/my_type { "my_type": { "_all": { "enabled": false } }

# }

Inclusion in the `_all` field can be controlled on a field-by-field basis by using the `include_in_all` setting, (((include_in_all setting)))which defaults to `true`. Setting `include_in_all` on an object (or on the root object) changes the default for all fields within that object.

You may find that you want to keep the `_all` field around to use as a catchall full-text field just for specific fields, such as `title`, `overview`, `summary`, and `tags`. Instead of disabling the `_all` field completely, disable `include_in_all` for all fields by default, and enable it only on the fields you choose:

# [source,js]

PUT /my_index/my_type/_mapping { "my_type": { "include_in_all": false, "properties": { "title": { "type": "string",

"include_in_all": true }, ... } }

# }

Remember that the `_all` field is just(("analyzers", "configuring for all field"))) an analyzed `string` field. It uses the default analyzer to analyze its values, regardless of which analyzer has been set on the fields where the values originate. And like any `string` field, you can configure which analyzer the `_all` field should use:

# [source,js]

PUT /my_index/my_type/_mapping { "my_type": { "_all": { "analyzer": "whitespace" } }

# }

==== Metadata: Document Identity

There are four metadata fields (((("metadata, document", "identity")))associated with document identity:

`_id` ::
The string ID of the document

`_type` ::
The type name of the document

`_index` :: The index where the document lives

`_uid` ::
The `_type` and `_id` concatenated together as `type#id`

By default, the `_uid` field is((("id field"))) stored (can be retrieved) and indexed (searchable). The `_type` field((("type field")))((("index field")))((("uid field"))) is indexed but not stored, and the `_id` and `_index` fields are neither indexed nor stored, meaning they don't really exist.

In spite of this, you can query the `_id` field as though it were a real field. Elasticsearch uses the `_uid` field to derive the `_id`. Although you can change the `index` and `store` settings for these fields, you almost never need to do so.

The `_id` field does have one setting that you may want to use: the `path` setting tells((("id field", "path setting")))((("path setting, id field"))) Elasticsearch that it should extract the value for the `_id` from a field within the document itself.

# [source,js]

PUT /my_index { "mappings": { "my_type": { "_id": { "path": "doc_id" <1> }, "properties": { "doc_id": { "type": "string", "index": "not_analyzed" } } } }

# }

// SENSE: 070_Index_Mgmt/33_ID_path.json

<1> Extract the doc `_id` from the `doc_id` field.

Then, when you index a document:

# [source,js]

POST /my_index/my_type { "doc_id": "123"

# }

// SENSE: 070_Index_Mgmt/33_ID_path.json

the `_id` value will be (("doc_id field")))extracted from the `doc_id` field in the document body:

# [source,js]

{ "_index": "my_index", "_type": "my_type", "_id": "123", <1> "_version": 1, "created": true

```
}
```

<1> The `_id` has been extracted correctly.

WARNING: While this is very convenient, be aware that it has a slight performance impact on `bulk` requests (see <>). The node handling the request can no longer use the optimized bulk format to parse just the metadata line in order to decide which shard should receive the request. Instead, it has to parse the document body as well.

[[dynamic-mapping]] === Dynamic Mapping

When Elasticsearch encounters a previously (((("mapping (types)", "dynamic")))((("dynamic mapping")))unknown field in a document, it uses <> to determine the datatype for the field and automatically adds the new field to the type mapping.

Sometimes this is the desired behavior and sometimes it isn't. Perhaps you don't know what fields will be added to your documents later, but you want them to be indexed automatically. Perhaps you just want to ignore them. Or--especially if you are using Elasticsearch as a primary data store--perhaps you want unknown fields to throw an exception to alert you to the problem.

Fortunately, you can control this behavior(((("dynamic setting"))) with the `dynamic` setting, which accepts the following options:

`true` ::
Add new fields dynamically--the default

`false` ::
Ignore new fields

`strict` ::
Throw an exception if an unknown field is encountered

The `dynamic` setting may be applied to the root object or to any field of type `object`. You could set `dynamic` to `strict` by default, but enable it just for a specific inner object:

# [source,js]

PUT /my_index { "mappings": { "my_type": { "dynamic": "strict", <1> "properties": { "title": { "type": "string"}, "stash": { "type": "object", "dynamic": true <2> } } } }

# }

// SENSE: 070_Index_Mgmt/35_Dynamic_mapping.json

<1> The `my_type` object will throw an exception if an unknown field is encountered.

<2> The `stash` object will create new fields dynamically.

With this mapping, you can add new searchable fields into the `stash` object:

# [source,js]

PUT /my_index/my_type/1 { "title": "This doc adds a new field", "stash": { "new_field": "Success!" }

# }

// SENSE: 070_Index_Mgmt/35_Dynamic_mapping.json

But trying to do the same at the top level will fail:

# [source,js]

PUT /my_index/my_type/1 { "title": "This throws a StrictDynamicMappingException", "new_field": "Fail!"

# }

// SENSE: 070_Index_Mgmt/35_Dynamic_mapping.json

NOTE: Setting `dynamic` to `false` doesn't alter the contents of the `_source` field at all. The `_source` will still contain the whole JSON document that you indexed. However, any unknown fields will not be added to the mapping and will not be searchable.

[[custom-dynamic-mapping]] === Customizing Dynamic Mapping

If you know that you are going to be adding new fields on the fly, you probably want to leave dynamic mapping enabled. ((("dynamic mapping", "custom")))(("mapping (types)", "dynamic", "custom"))) At times, though, the dynamic mapping ``rules'' can be a bit blunt. Fortunately, there are settings that you can use to customize these rules to better suit your data.

[[date-detection]] ==== date_detection

When Elasticsearch encounters a new string field, it checks to see if the string contains a recognizable date, like `2014-01-01`.((("date_detection setting")))(("dynamic mapping", "custom", "date_detection setting"))) If it looks like a date, the field is added as type `date`. Otherwise, it is added as type `string`.

Sometimes this behavior can lead to problems. Imagine that you index a document like this:

## [source,js]

## { "note": "2014-01-01" }

Assuming that this is the first time that the `note` field has been seen, it will be added as a `date` field. But what if the next document looks like this:

## [source,js]

## { "note": "Logged out" }

This clearly isn't a date, but it is too late. The field is already a date field and so this ``malformed date'' will cause an exception to be thrown.

Date detection can be turned off by setting `date_detection` to `false` on the ((("root object", "date_detection setting")))root object:

## [source,js]

PUT /my_index { "mappings": { "my_type": { "date_detection": false } }

## }

With this mapping in place, a string will always be a `string`. If you need a `date` field, you have to add it manually.

## [NOTE]

Elasticsearch's idea of which strings look like dates can be altered

## with the http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/mapping-root-object-

# type.html[`dynamic_date_formats` setting].

[[dynamic-templates]] ==== dynamic_templates

With `dynamic_templates`, you can take complete control (((("dynamic_templates setting")))((("dynamic mapping", "custom", "dynamic_templates setting")))over the mapping that is generated for newly detected fields. You can even apply a different mapping depending on the field name or datatype.

Each template has a name, which (((("templates", "dynamic_templates setting")))you can use to describe what the template does, a `mapping` to specify the mapping that should be applied, and at least one parameter (such as `match`) to define which fields the template should apply to.

Templates are checked in order; the first template that matches is applied. For instance, we could specify two templates for `string` fields:

- `es`: Field names ending in `_es` should use the `spanish` analyzer.
- `en`: All others should use the `english` analyzer.

We put the `es` template first, because it is more specific than the catchall `en` template, which matches all string fields:

# [source,js]

PUT /my_index { "mappings": { "my_type": { "dynamic_templates": [ { "es": { "match": "*_es", <1> "match_mapping_type": "string", "mapping": { "type": "string", "analyzer": "spanish" } }}, { "en": { "match": "*", <2> "match_mapping_type": "string", "mapping": { "type": "string", "analyzer": "english" } }} ]

# }}}

// SENSE: 070_Index_Mgmt/40_Custom_dynamic_mapping.json

<1> Match string fields whose name ends in `_es`.

<2> Match all other string fields.

The `match_mapping_type` allows (((("match_mapping_type setting")))you to apply the template only to fields of the specified type, as detected by the standard dynamic mapping rules, (for example `string` or `long`).

The `match` parameter matches just the field name, and the `path_match` parameter((("path_map parameter"))) matches the full path to a field in an object, so the pattern `address.*.name` would match a field like this:

# [source,js]

{ "address": { "city": { "name": "New York" } }

# }

The `unmatch` and `path_unmatch` patterns((("unmatch pattern")))((("path_unmap pattern"))) can be used to exclude fields that would otherwise match.

More configuration options can be found in the http://bit.ly/1wdHOzG[reference documentation for the root object].

[[default-mapping]] === Default Mapping

Often, all types in an index share similar fields and settings. (("mapping (types)", "default")))(("default mapping"))) It can be more convenient to specify these common settings in the `_default_` mapping, instead of having to repeat yourself every time you create a new type. The `_default_` mapping acts as a template for new types. All types created *after* the `_default_` mapping will include all of these default settings, unless explicitly overridden in the type mapping itself.

For instance, we can disable the `_all` field for all types,(("*all field", sortas="all field"))) using the `_default` `mapping, but enable it just for the` blog` type, as follows:

# [source,js]

PUT /my*index { "mappings": { "_default*": { "_all": { "enabled": false } }, "blog": { "_all": { "enabled": true } } }

# }

// SENSE: 070_Index_Mgmt/45_Default_mapping.json

The `_default_` mapping can also be a good place to specify index-wide <>.

[[reindex]] === Reindexing Your Data

Although you can add new types to an index, or add new fields to a type, you can't add new analyzers or make changes to existing fields.((("reindexing")))((("indexing", "reindexing your data"))) If you were to do so, the data that had already been indexed would be incorrect and your searches would no longer work as expected.

The simplest way to apply these changes to your existing data is to reindex: create a new index with the new settings and copy all of your documents from the old index to the new index.

One of the advantages of the `_source` field is that you already have the whole document available to you in Elasticsearch itself. You don't have to rebuild your index from the database, which is usually much slower.

To reindex all of the documents from the old index efficiently, use <> to retrieve batches((("scan-and-scroll", "using in reindexing documents"))) of documents from the old index, and the <> to push them into the new index.

.Reindexing in Batches

---

You can run multiple reindexing jobs at the same time, but you obviously don't want their results to overlap. Instead, break a big reindex down into smaller jobs by filtering on a date or timestamp field:

# [source,js]

GET /old_index/_search?search_type=scan&scroll=1m { "query": { "range": { "date": { "gte": "2014-01-01", "lt": "2014-02-01" } } }, "size": 1000

# }

If you continue making changes to the old index, you will want to make sure that you include the newly added documents in your new index as well. This can be done by rerunning the reindex process, but again filtering on a date field to match only documents that have been added since the last reindex process started.

---

[[index-aliases]] === Index Aliases and Zero Downtime

The problem with the reindexing process described previously is that you need to update your application to use the new index name.((("index aliases"))) Index aliases to the rescue!

An index *alias* is like a shortcut or symbolic link, which can point to one or more indices, and can be used in any API that expects an index name. Aliases(((“aliases, index”))) give us an enormous amount of flexibility. They allow us to do the following:

- Switch transparently between one index and another on a running cluster
- Group multiple indices (for example, `last_three_months`)
- Create ``views'' on a subset of the documents in an index

We will talk more about the other uses for aliases later in the book. For now we will explain how to use them to switch from an old index to a new index with zero downtime.

There are two endpoints for managing aliases: `_alias` for single operations, and `_aliases` to perform multiple operations atomically.

In this scenario, we will assume that your application is talking to an index called `my_index`. In reality, `my_index` will be an alias that points to the current real index. We will include a version number in the name of the real index: `my_index_v1`, `my_index_v2`, and so forth.

To start off, create the index `my_index_v1`, and set up the alias `my_index` to point to it:

# [source,js]

PUT /my_index_v1 <1>

# PUT /my_index_v1/_alias/my_index <2>

// SENSE: 070_Index_Mgmt/55_Aliases.json

<1> Create the index `my_index_v1`.

<2> Set the `my_index` alias to point to `my_index_v1`.

You can check which index the alias points to:

# [source,js]

# GET /*/_alias/my_index

// SENSE: 070_Index_Mgmt/55_Aliases.json

Or which aliases point to the index:

# [source,js]

# GET /my_index_v1/_alias/*

// SENSE: 070_Index_Mgmt/55_Aliases.json

Both of these return the following:

# [source,js]

{ "my_index_v1" : { "aliases" : { "my_index" : { } } }

# }

Later, we decide that we want to change the mappings for a field in our index. Of course, we can't change the existing mapping, so we have to reindex our data.((("reindexing", "using index aliases"))) To start, we create `my_index_v2` with the new mappings:

# [source,js]

PUT /my_index_v2 { "mappings": { "my_type": { "properties": { "tags": { "type": "string", "index": "not_analyzed" } } } }

# }

// SENSE: 070_Index_Mgmt/55_Aliases.json

Then we reindex our data from `my_index_v1` to `my_index_v2`, following the process described in <>. Once we are satisfied that our documents have been reindexed correctly, we switch our alias to point to the new index.

An alias can point to multiple indices, so we need to remove the alias from the old index at the same time as we add it to the new index. The change needs to be atomic, which means that we must use the `_aliases` endpoint:

# [source,js]

POST /_aliases { "actions": [ { "remove": { "index": "my_index_v1", "alias": "my_index" }}, { "add": { "index": "my_index_v2", "alias": "my_index" }} ]

# }

// SENSE: 070_Index_Mgmt/55_Aliases.json

Your application has switched from using the old index to the new index transparently, with zero downtime.

# [TIP]

Even when you think that your current index design is perfect, it is likely that you will need to make some change later, when your index is already being used in production.

Be prepared: use aliases instead of indices in your application. Then you will be able to reindex whenever you need to. Aliases are cheap and should

## be used liberally.

[[inside-a-shard]] == Inside a Shard

In <>, we introduced the *shard*, and described(("shards"))) it as a low-level *worker unit*. But what exactly *is* a shard and how does it work? In this chapter, we answer these questions:

- Why is search *near* real-time?
- Why are document CRUD (create-read-update-delete) operations *real-time*?
- How does Elasticsearch ensure that the changes you make are durable, that they won't be lost if there is a power failure?
- Why does deleting documents not free up space immediately?
- What do the `refresh`, `flush`, and `optimize` APIs do, and when should you use them?

The easiest way to understand how a shard functions today is to start with a history lesson. We will look at the problems that needed to be solved in order to provide a distributed durable data store with near real-time search and analytics.

.Content Warning

---

The information presented in this chapter is for your interest. You are not required to understand and remember all the detail in order to use Elasticsearch. Read this chapter to gain a taste for how things work, and to know where the information is in case you need to refer to it in the future, but don't be overwhelmed by the detail.

---

[[making-text-searchable]] === Making Text Searchable

The first challenge that had to be solved was how to((("text", "making it searchable"))) make text searchable. Traditional databases store a single value per field, but this is insufficient for full-text search. Every word in a text field needs to be searchable, which means that the database needs to be able to index multiple values--words, in this case--in a single field.

The data structure that best supports the *multiple-values-per-field* requirement is the *inverted index*, which(((("inverted index"))) we introduced in <>. The inverted index contains a sorted list of all of the unique values, or *terms*, that occur in any document and, for each term, a list of all the documents that contain it.

```
Term  | Doc 1 | Doc 2 | Doc 3 | ...
-----------------------------------
brown |   X   |       |  X    | ...
fox   |   X   |   X   |  X    | ...
quick |   X   |   X   |       | ...
the   |   X   |       |  X    | ...
```

# [NOTE]

When discussing inverted indices, we talk about indexing *documents* because, historically, an inverted index was used to index whole unstructured text documents. A *document* in Elasticsearch is a structured JSON document with fields and values. In reality, every indexed field in a JSON document has its

# own inverted index.

The inverted index may hold a lot more information than the list of documents that contain a particular term. It may store a count of the number of documents that contain each term, the number of times a term appears in a particular document, the order of terms in each document, the length of each document, the average length of all documents, and more. These statistics allow Elasticsearch to determine which terms are more important than others, and which documents are more important than others, as described in <>.

The important thing to realize is that the inverted index needs to know about *all* documents in the collection in order for it to function as intended.

In the early days of full-text search, one big inverted index was built for the entire document collection and written to disk. As soon as the new index was ready, it replaced the old index, and recent changes became searchable.

[role="pagebreak-before"] ==== Immutability

The inverted index that is written to disk is *immutable*: it doesn't change.((("inverted index", "immutability"))) Ever. This immutability has important benefits:

- There is no need for locking. If you never have to update the index, you never have to worry about multiple processes trying to make changes at the same time.
- Once the index has been read into the kernel's filesystem cache, it stays there, because it never changes. As long as there is enough space in the filesystem cache, most reads will come from memory instead of having to hit disk. This provides a big performance boost.
- Any other caches (like the filter cache) remain valid for the life of the index. They don't need to be rebuilt every time the data changes, because the data doesn't change.
- Writing a single large inverted index allows the data to be compressed, reducing costly disk I/O and the amount of

RAM needed to cache the index.

Of course, an immutable index has its downsides too, primarily the fact that it is immutable! You can't change it. If you want to make new documents searchable, you have to rebuild the entire index. This places a significant limitation either on the amount of data that an index can contain, or the frequency with which the index can be updated.

[[near-real-time]] === Near Real-Time Search

With the development of per-segment search, the ((("searching", "near real-time search")))delay between indexing a document and making it visible to search dropped dramatically. New documents could be made searchable within minutes, but that still isn't fast enough.

The bottleneck is the disk. ((("committing segments to disk")))((("fsync")))((("segments", "committing to disk"))) Commiting a new segment to disk requires an http://en.wikipedia.org/wiki/Fsync[`fsync`] to ensure that the segment is physically written to disk and that data will not be lost if there is a power failure. But an `fsync` is costly; it cannot be performed every time a document is indexed without a big performance hit.

What was needed was a more lightweight way to make new documents visible to search, which meant removing `fsync` from the equation.

Sitting between Elasticsearch and the disk is the filesystem cache.(("filesystem cache"))) As before, documents in the in-memory indexing buffer (<>) are written to a new segment (<>). But the new segment is written to the filesystem cache first--which is cheap--and only later is it flushed to disk--which is expensive. But once a file is in the cache, it can be opened and read, just like any other file.

[[img-pre-refresh]] .A Lucene index with new documents in the in-memory buffer image::images/elas_1104.png["A Lucene index with new documents in the in-memory buffer"]

Lucene allows new segments to be written and opened--making the documents they contain visible to search--without performing a full commit. This is a much lighter process than a commit, and can be done frequently without ruining performance.

[[img-post-refresh]] .The buffer contents have been written to a segment, which is searchable, but is not yet commited image::images/elas_1105.png["The buffer contents have been written to a segment, which is searchable, but is not yet commited"]

[[refresh-api]] ==== refresh API

In Elasticsearch, this lightweight process of writing and opening a new segment is called a *refresh*.((("shards", "refreshes"))) ((("refresh API"))) By default, every shard is refreshed automatically once every second. This is why we say that Elasticsearch has *near* real-time search: document changes are not visible to search immediately, but will become visible within 1 second.

This can be confusing for new users: they index a document and try to search for it, and it just isn't there. The way around this is to perform a manual refresh, with the `refresh` API:

# [source,json]

POST /_refresh <1>

# POST /blogs/_refresh <2>

<1> Refresh all indices.

<2> Refresh just the `blogs` index.

# [TIP]

While a refresh is much lighter than a commit, it still has a performance cost.((("indices", "refresh_interval"))) A manual refresh can be useful when writing tests, but don't do a manual refresh every time you index a document in production; it will hurt your performance. Instead, your application needs to be aware of the near

# real-time nature of Elasticsearch and make allowances for it.

Not all use cases require a refresh every second. Perhaps you are using Elasticsearch to index millions of log files, and you would prefer to optimize for index speed rather than near real-time search. You can reduce the frequency of refreshes on a per-index basis by (((("refresh_interval setting")))setting the `refresh_interval`:

# [source,json]

PUT /my_logs { "settings": { "refresh_interval": "30s" <1> }

# }

<1> Refresh the `my_logs` index every 30 seconds.

The `refresh_interval` can be updated dynamically on an existing index. You can turn off automatic refreshes while you are building a big new index, and then turn them back on when you start using the index in production:

# [source,json]

POST /my_logs/_settings { "refresh_interval": -1 } <1>

POST /my_logs/_settings

# { "refresh_interval": "1s" } <2>

<1> Disable automatic refreshes.

<2> Refresh automatically every second.

CAUTION: The `refresh_interval` expects a *duration* such as `1s` (1 second) or `2m` (2 minutes). An absolute number like `1` means *1 millisecond*--a sure way to bring your cluster to its knees.

[[translog]] === Making Changes Persistent

Without an `fsync` to flush data in the filesystem cache to disk, we cannot be sure that the data will still (((("persistent changes, making")))((("changes, persisting")))be there after a power failure, or even after exiting the application normally. For Elasticsearch to be reliable, it needs to ensure that changes are persisted to disk.

In <>, we said that a full commit flushes segments to disk and writes a commit point, which lists all known segments. (((("commit point"))) Elasticsearch uses this commit point during startup or when reopening an index to decide which segments belong to the current shard.

While we refresh once every second to achieve near real-time search, we still need to do full commits regularly to make sure that we can recover from failure. But what about the document changes that happen between commits? We don't want to lose those either.

Elasticsearch added a *translog*, or transaction log,(((("translog (transaction log)"))) which records every operation in Elasticsearch as it happens. With the translog, the process now looks like this:

1. When a document is indexed, it is added to the in-memory buffer *and* appended to the translog, as shown in <>. + [[img-xlog-pre-refresh]] .New documents are added to the in-memory buffer and appended to the transaction log image::images/elas_1106.png["New documents are added to the in-memory buffer and appended to the transaction log"]

2. The refresh leaves the shard in the state depicted in <>. Once every second, the shard is refreshed:

   +

   **The docs in the in-memory buffer are written to a new segment, without an `fsync`.** The segment is opened to make it visible to search.

   ** The in-memory buffer is cleared.

[[img-xlog-post-refresh]] .After a refresh, the buffer is cleared but the transaction log is not

# image::images/elas_1107.png["After a refresh, the buffer is cleared but the transaction log is not"]

1. This process continues with more documents being added to the in-memory buffer and appended to the transaction log (see <>). + [[img-xlog-pre-flush]] .The transaction log keeps accumulating documents image::images/elas_1108.png["The transaction log keeps accumulating documents"]

1. Every so often--such as when the translog is getting too big--the index is flushed; a new translog is created, and a full commit is performed (see <>):

   +

   **Any docs in the in-memory buffer are written to a new segment.** The buffer is cleared. **A commit point is written to disk.** The filesystem cache is flushed with an `fsync`. ** The old translog is deleted.

--

The translog provides a persistent record of all operations that have not yet been flushed to disk. When starting up, Elasticsearch will use the last commit point to recover known segments from disk, and will then replay all operations in the

translog to add the changes that happened after the last commit.

The translog is also used to provide real-time CRUD. When you try to retrieve, update, or delete a document by ID, it first checks the translog for any recent changes before trying to retrieve the document from the relevant segment. This means that it always has access to the latest known version of the document, in real-time.

[[img-xlog-post-flush]] .After a flush, the segments are fully commited and the transaction log is cleared image::images/elas_1109.png["After a flush, the segments are fully commited and the transaction log is cleared"]

[[flush-api]] ==== flush API

The action of performing a commit and truncating the translog is known in Elasticsearch as a *flush*. (("flushes"))) Shards are flushed automatically every 30 minutes, or when the translog becomes too big. See the http://bit.ly/1E3HKbD[`translog` documentation] for settings that can be used(("translog (transaction log)", "flushes and"))) to control these thresholds:

The http://bit.ly/1ICgxiU[`flush` API] can ((("indices", "flushing")))((("flush API")))be used to perform a manual flush:

# [source,json]

POST /blogs/_flush <1>

# POST /_flush?wait_for_ongoing <2>

<1> Flush the `blogs` index.

<2> Flush all indices and wait until all flushes have completed before returning.

You seldom need to issue a manual `flush` yourself; usually, automatic flushing is all that is required.

That said, it is beneficial to <> your indices before restarting a node or closing an index. When Elasticsearch tries to recover or reopen an index, it has to replay all of the operations in the translog, so the shorter the log, the faster the recovery.

.How Safe Is the Translog?

---

The purpose of the translog is to ensure that operations are not lost. This begs the question: how safe(("translog (transaction log)", "safety of"))) is the translog?

Writes to a file will not survive a reboot until the file has been +fsync+'ed to disk. By default, the translog is +fsync+'ed every 5 seconds. Potentially, we could lose 5 seconds worth of data--if the translog were the only mechanism that we had for dealing with failure.

Fortunately, the translog is only part of a much bigger system. Remember that an indexing request is considered successful only after it has completed on both the primary shard and all replica shards. Even if the node holding the primary shard were to suffer catastrophic failure, it would be unlikely to affect the nodes holding the replica shards at the same time.

While we could force the translog to `fsync` more frequently (at the cost of indexing performance), it is unlikely to provide more reliability.

---

[[merge-process]] === Segment Merging

With the automatic refresh process creating a new segment((("segments", "merging"))) every second, it doesn't take long for the number of segments to explode. Having too many segments is a problem. Each segment consumes file handles, memory, and CPU cycles. More important, every search request has to check every segment in turn; the more segments there are, the slower the search will be.

Elasticsearch solves this problem by merging segments in the background.((("merging segments"))) Small segments are merged into bigger segments, which, in turn, are merged into even bigger segments.

This is the moment when those old deleted documents((("deleted documents", "purging of"))) are purged from the filesystem. Deleted documents (or old versions of updated documents) are not copied over to the new bigger segment.

There is nothing you need to do to enable merging. It happens automatically while you are indexing and searching. The process works like as depicted in <>:

1. While indexing, the refresh process creates new segments and opens them for search.
2. The merge process selects a few segments of similar size and merges them into a new bigger segment in the background. This does not interrupt indexing and searching. + [[img-merge]] .Two commited segments and one uncommited segment in the process of being merged into a bigger segment image::images/elas_1110.png["Two commited segments and one uncommited segment in the process of being merged into a bigger segment"]
3. <> illustrates activity as the merge completes:

   +

   **The new segment is flushed to disk.** A new commit point is written that includes the new segment and

   ```
   excludes the old, smaller segments.
   ```

   **The new segment is opened for search.** The old segments are deleted.

[[img-post-merge]] .Once merging has finished, the old segments are deleted

# image::images/elas_1111.png["Once merging has finished, the old segments are deleted"]

The merging of big segments can use a lot of I/O and CPU, which can hurt search performance if left unchecked. By default, Elasticsearch throttles the merge process so that search still has enough resources available to perform well.

TIP: See <> for advice about tuning merging for your use case.

[[optimize-api]] ==== optimize API

The `optimize` API is best (((("merging segments", "optimize API and")))((("optimize API")))(("segments", "merging", "optimize API")))described as the *forced merge* API. It forces a shard to be merged down to the number of segments specified in the `max_num_segments` parameter. The intention is to reduce the number of segments (usually to one) in order to speed up search performance.

WARNING: The `optimize` API should *not* be used on a dynamic index--an index that is being actively updated. The background merge process does a very good job, and optimizing will hinder the process. Don't interfere!

In certain specific circumstances, the `optimize` API can be beneficial. The typical use case is for logging, where logs are stored in an index per day, week, or month. Older indices are essentially read-only; they are unlikely to change.

In this case, it can be useful to optimize the shards of an old index down to a single segment each; it will use fewer resources and searches will be quicker:

# [source,json]

# POST /logstash-2014-10/_optimize?max_num_segments=1 <1>

<1> Merges each shard in the index down to a single segment

# [WARNING]

Be aware that merges triggered by the `optimize` API are not throttled at all. They can consume all of the I/O on your nodes, leaving nothing for search and potentially making your cluster unresponsive. If you plan on optimizing an index, you should use shard allocation (see <>) to first move the index to a node where it is safe to

# run.

[[structured-search]] == Structured Search

*Structured search* is about interrogating (((("structured search")))data that has inherent structure. Dates, times, and numbers are all structured: they have a precise format that you can perform logical operations on. Common operations include comparing ranges of numbers or dates, or determining which of two values is larger.

Text can be structured too. A box of crayons has a discrete set of colors: `red`, `green`, `blue`. A blog post may be tagged with keywords `distributed` and `search`. Products in an ecommerce store have Universal Product Codes (UPCs) or some other identifier that requires strict and structured formatting.

With structured search, the answer to your question is *always* a yes or no; something either belongs in the set or it does not. Structured search does not worry about document relevance or scoring; it simply includes or excludes documents.

This should make sense logically. A number can't be *more* in a range than any other number that falls in the same range. It is either in the range--or it isn't. Similarly, for structured text, a value is either equal or it isn't. There is no concept of *more similar*.

[[combining-filters]] === Combining Filters

The previous two examples showed a single filter in use.(((("structured search", "combining filters")))((("filters", "combining"))) In practice, you will probably need to filter on multiple values or fields. For example, how would you express this SQL in Elasticsearch?

# [source,sql]

SELECT product FROM products WHERE (price = 20 OR productID = "XHDK-A-1293-#fJ3")

# AND (price != 30)

In these situations, you will need the `bool` filter.((("filters", "combining", "in bool filter")))((("bool filter"))) This is a *compound filter* that accepts other filters as arguments, combining them in various Boolean combinations.

[[bool-filter]] ==== Bool Filter

The `bool` filter is composed of three sections:

# [source,js]

{ "bool" : { "must" : [], "should" : [], "must_not" : [], }

# }

`must` ::
All of these clauses *must* match. The equivalent of `AND`.

`must_not` :: All of these clauses *must not* match. The equivalent of `NOT`.

`should` ::
At least one of these clauses must match. The equivalent of `OR`.

And that's it!((("should clause", "in bool filters")))((("must_not clause", "in bool filters")))((("must clause", "in bool filters"))) When you need multiple filters, simply place them into the different sections of the `bool` filter.

# [NOTE]

Each section of the `bool` filter is optional (for example, you can have a `must` clause and nothing else), and each section can contain a single filter or an

# array of filters.

To replicate the preceding SQL example, we will take the two `term` filters that we used((("term filter", "placing inside bool filter")))((("bool filter", "with two term filters in should clause and must_not clause"))) previously and place them inside the `should` clause of a `bool` filter, and add another clause to deal with the `NOT` condition:

# [source,js]

GET /my_store/products/_search { "query" : { "filtered" : { <1> "filter" : { "bool" : { "should" : [ { "term" : {"price" : 20}}, <2> { "term" : {"productID" : "XHDK-A-1293-#fJ3"}} <2> ], "must_not" : { "term" : {"price" : 30} <3> } } } } }

# }

// SENSE: 080_Structured_Search/10_Bool_filter.json

<1> Note that we still need to use a `filtered` query to wrap everything.

<2> These two `term` filters are *children* of the `bool` filter, and since they are placed inside the `should` clause, at least one of them needs to match.

<3> If a product has a price of `30`, it is automatically excluded because it matches a `must_not` clause.

Our search results return two hits, each document satisfying a different clause in the `bool` filter:

# [source,json]

"hits" : [ { "_id" : "1", "_score" : 1.0, "_source" : { "price" : 10, "productID" : "XHDK-A-1293-#fJ3" <1> } }, { "_id" : "2", "_score" : 1.0, "_source" : { "price" : 20, <2> "productID" : "KDKE-B-9947-#kL5" } }

# ]

<1> Matches the `term` filter for `productID = "XHDK-A-1293-#fJ3"`

<2> Matches the `term` filter for `price = 20`

==== Nesting Boolean Filters

Even though `bool` is a compound filter and accepts children filters, it is important to understand that `bool` is just a filter itself.((("filters", "combining", "nesting bool filters")))((("bool filter", "nesting in another bool filter"))) This means you can nest `bool` filters inside other `bool` filters, giving you the ability to make arbitrarily complex Boolean logic.

Given this SQL statement:

# [source,sql]

SELECT document FROM products WHERE productID = "KDKE-B-9947-#kL5" OR ( productID = "JODL-X-1937-#pV7"

```
    AND price     = 30 )
```

We can translate it into a pair of nested `bool` filters:

# [source,js]

GET /my_store/products/_search { "query" : { "filtered" : { "filter" : { "bool" : { "should" : [ { "term" : {"productID" : "KDKE-B-9947-#kL5"}}, <1> { "bool" : { <1> "must" : [ { "term" : {"productID" : "JODL-X-1937-#pV7"}}, <2> { "term" : {"price" : 30}} <2> ] }} ] } } } }

# }

// SENSE: 080_Structured_Search/10_Bool_filter.json

<1> Because the `term` and the `bool` are sibling clauses inside the first Boolean `should`, at least one of these filters must match for a document to be a hit.

<2> These two `term` clauses are siblings in a `must` clause, so they both have to match for a document to be returned as a hit.

The results show us two documents, one matching each of the `should` clauses:

# [source,json]

"hits" : [ { "_id" : "2", "_score" : 1.0, "_source" : { "price" : 20, "productID" : "KDKE-B-9947-#kL5" <1> } }, { "_id" : "3", "_score" : 1.0, "_source" : { "price" : 30, <2> "productID" : "JODL-X-1937-#pV7" <2> } }

# ]

<1> This `productID` matches the `term` in the first `bool`.

<2> These two fields match the `term` filters in the nested `bool`.

This was a simple example, but it demonstrates how Boolean filters can be used as building blocks to construct complex logical conditions.

=== Finding Multiple Exact Values

The `term` filter is useful for finding a single value, but often you'll want to search for multiple values.((("exact values", "finding multiple")))((("structured search", "finding multiple exact values"))) What if you want to find documents that have a price of $20 or $30?

Rather than using multiple `term` filters, you can instead use a single `terms` filter (note the *s* at the end). The `terms` filter(("terms filter")) is simply the plural version of the singular `term` filter.

It looks nearly identical to a vanilla `term` too. Instead of specifying a single price, we are now specifying an array of values:

# [source,js]

{ "terms" : { "price" : [20, 30] }

# }

And like the `term` filter, we will place it inside a `filtered` query to (("filtered query", "terms filter in")) use it:

# [source,js]

GET /my_store/products/_search { "query" : { "filtered" : { "filter" : { "terms" : { <1> "price" : [20, 30] } } } }

# }

// SENSE: 080_Structured_Search/15_Terms_filter.json

<1> The `terms` filter as seen previously, but placed inside the `filtered` query

The query will return the second, third, and fourth documents:

# [source,json]

"hits" : [ { "_id" : "2", "_score" : 1.0, "_source" : { "price" : 20, "productID" : "KDKE-B-9947-#kL5" } }, { "_id" : "3", "_score" : 1.0, "_source" : { "price" : 30, "productID" : "JODL-X-1937-#pV7" } }, { "_id": "4", "_score": 1.0, "_source": { "price": 30, "productID": "QQPX-R-3956-#aD8" } }

# ]

==== Contains, but Does Not Equal

It is important to understand that `term` and `terms` are *contains* operations, not *equals*.(("structured search", "contains, but does not equal")))(("terms filter", "contains, but does not equal")))(("term filter", "contains, but does not equal"))) What does that mean?

If you have a term filter for `{ "term" : { "tags" : "search" } }`, it will match *both* of the following documents:

# [source,js]

{ "tags" : ["search"] }

# { "tags" : ["search", "open_source"] } <1>

<1> This document is returned, even though it has terms other than `search`.

Recall how the `term` filter works: it checks the inverted index for all documents that contain a term, and then constructs a bitset. In our simple example, we have the following inverted index:

[width="50%",frame="topbot"] |========================== | Token | DocIDs | `open_source` | `2` | `search` | `1`, `2` |==========================

When a `term` filter is executed for the token `search`, it goes straight to the corresponding entry in the inverted index and extracts the associated doc IDs. As you can see, both document 1 and document 2 contain the token in the inverted index. Therefore, they are both returned as a result.

# [NOTE]

The nature of an inverted index also means that entire field equality is rather difficult to calculate. How would you determine whether a particular document contains *only* your request term? You would have to find the term in the inverted index, extract the document IDs, and then scan *every row in the inverted index*, looking for those IDs to see whether a doc has any other terms.

As you might imagine, that would be tremendously inefficient and expensive. For that reason, `term` and `terms` are *must contain* operations, not

# *must equal exactly*.

==== Equals Exactly If you do want that behavior--entire field equality--the best way to accomplish it involves indexing a secondary field. (("structured search", "equals exactly")) In this field, you index the number of values that your field contains. Using our two previous documents, we now include a field that maintains the number of tags:

# [source,js]

{ "tags" : ["search"], "tag_count" : 1 }

# { "tags" : ["search", "open_source"], "tag_count" : 2 }

// SENSE: 080_Structured_Search/20_Exact.json

Once you have the count information indexed, you can construct a `bool` filter that enforces the appropriate number of terms:

# [source,js]

GET /my_index/my_type/_search { "query": { "filtered" : { "filter" : { "bool" : { "must" : [ { "term" : { "tags" : "search" } }, <1> { "term" : { "tag_count" : 1 } } <2> ] } } } }

# }

// SENSE: 080_Structured_Search/20_Exact.json

<1> Find all documents that have the term `search`.

<2> But make sure the document has only one tag.

This query will now match only the document that has a single tag that is `search`, rather than any document that contains `search`.