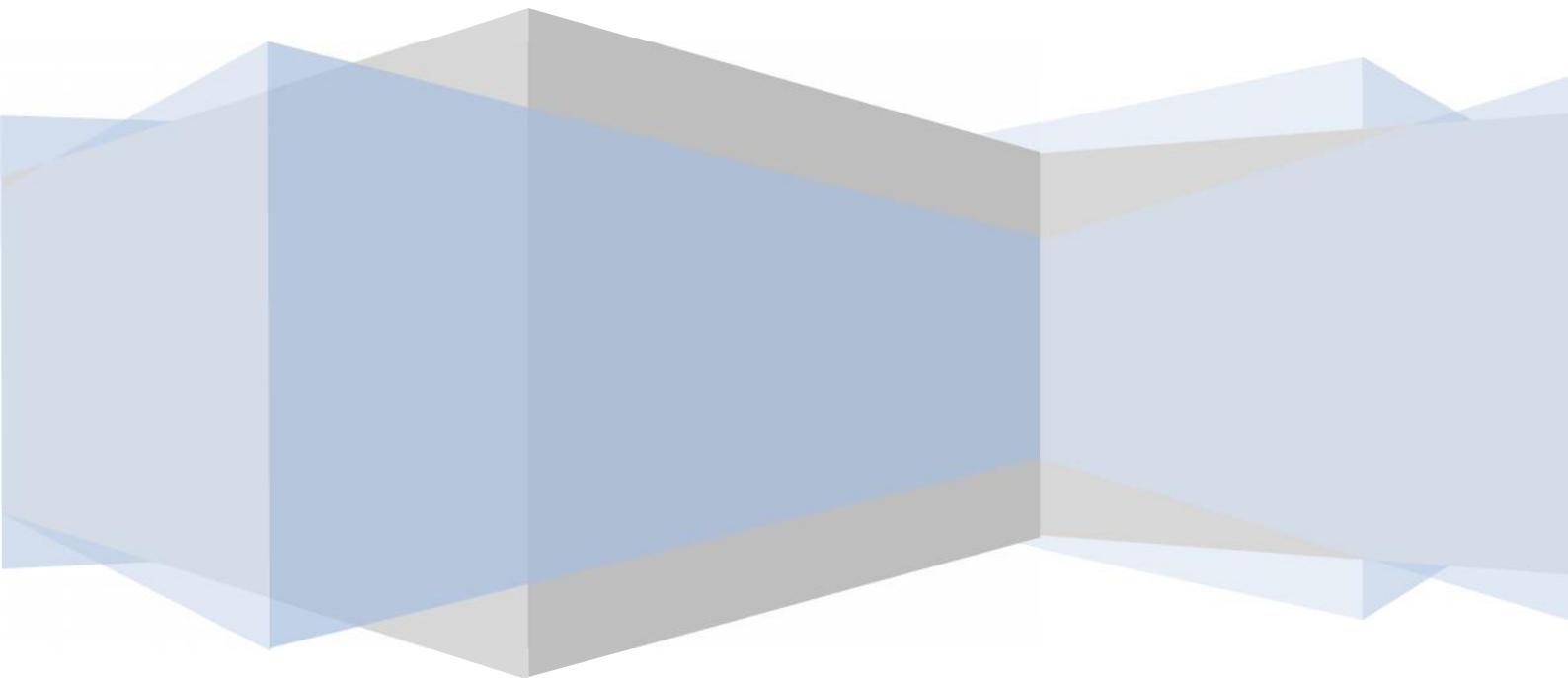




elasticsearch



ElasticSearch

- Elasticsearch is developed by Shay Banon.

Elastic search 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 Elastic search to provide full-text search with highlighted search
- snippets, and search-as-you-type and did-you-mean suggestions.
- The **Guardian** uses Elastic search 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.

Elasticsearch is an open-source search engine built on top of **Apache Lucene™**, a fulltext 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.

Distributed Nature

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

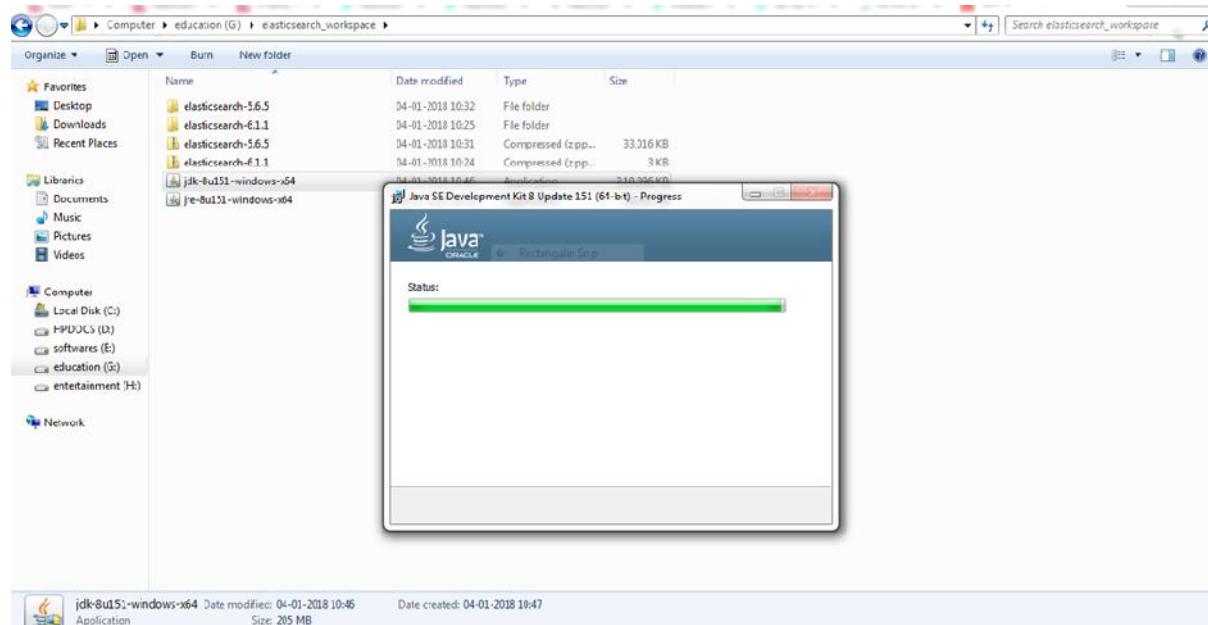
Installing Elasticsearch

Download from <https://www.elastic.co/downloads/past-releases/elasticsearch-6-1-1>

Pre Requirements:

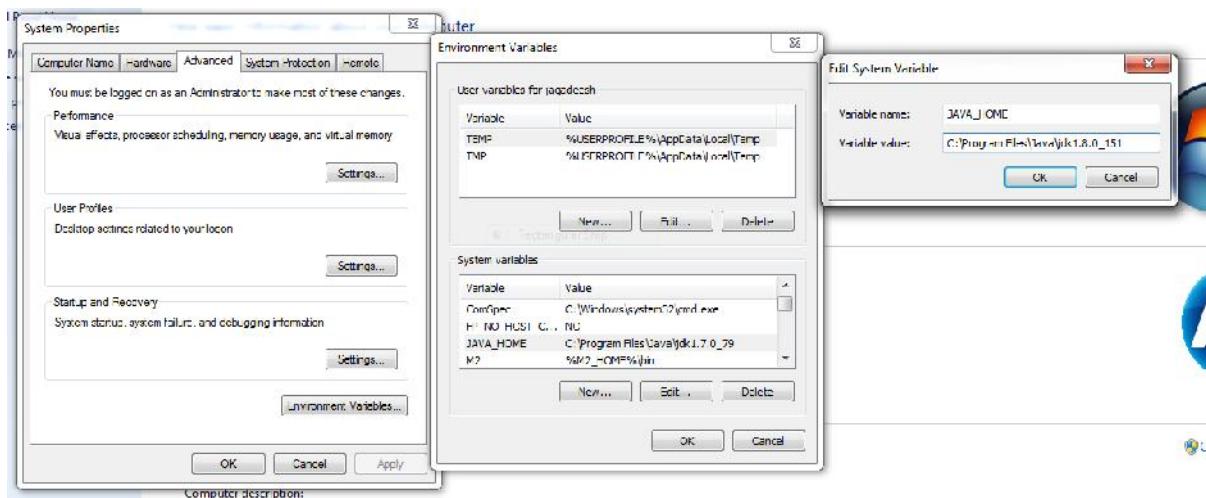
Install java 1.7 or higher version and Set JAVA_HOME

JDK 1.8 installation



JRE 1.8 installation





```

C:\Windows\system32\cmd.exe
implicit:{none,class}
-encoding <encoding>
-source <release>
-target <release>
-profile <profile>
-version
-help
-Akey[=value]
-X
-J<flag>
-Werror
@<filename>      Specify whether or not to generate class files for
                  Specify character encoding used by source files
                  Provide source compatibility with specified release
                  Generate class files for specific VM version
                  Check that API used is available in the specified p
                  Version information
                  Print a synopsis of standard options
                  Options to pass to annotation processors
                  Print a synopsis of nonstandard options
                  Pass <flag> directly to the runtime system
                  Terminate compilation if warnings occur
                  Read options and filenames from file

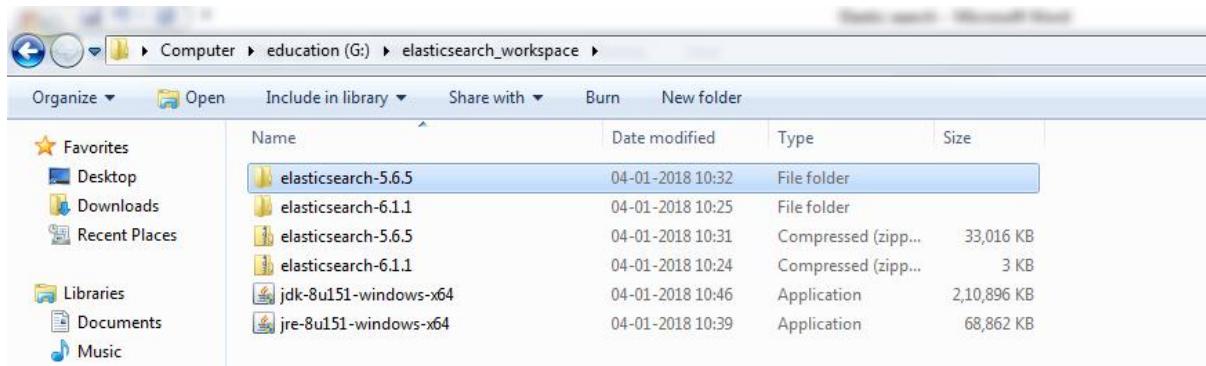
C:\Users\jagadeesh>java - version
Unrecognized option: -
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.

C:\Users\jagadeesh>java -version
java version "1.8.0_151"
Java(TM) SE Runtime Environment (build 1.8.0_151-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.151-b12, mixed mode)

C:\Users\jagadeesh>

```

Extract or Unzip downloaded elastic search folder.

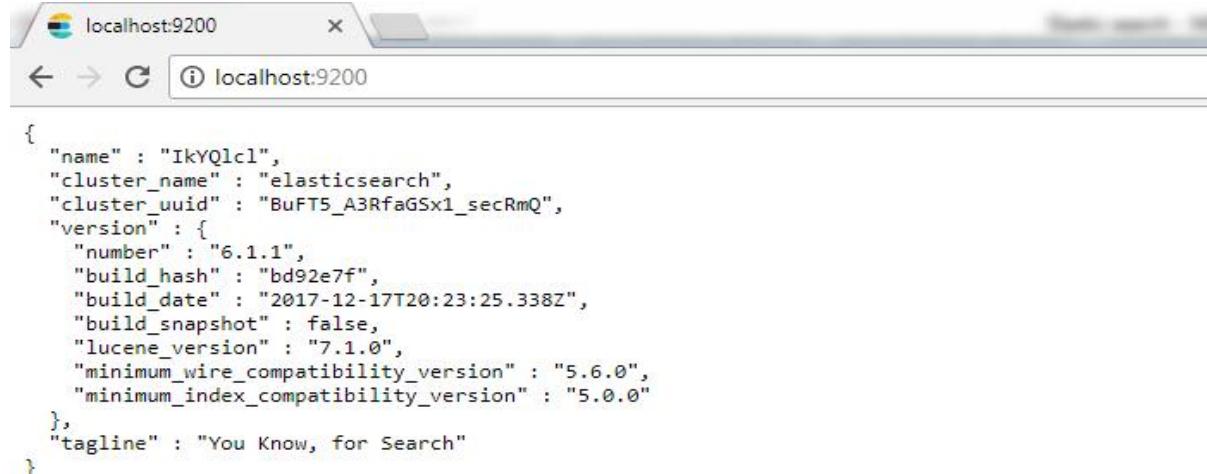


Start elastic search from below path:

G:\elasticsearch_workspace\elasticsearch-6.1.1\bim

Elasticsearch-6.1.1 requires Java 8

Elasticsearch is started now, hit the below URL to access the server:

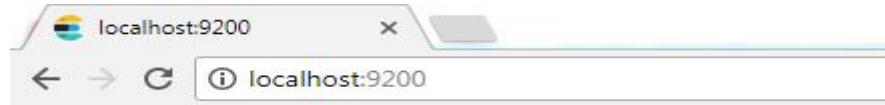


Cutomize the default properties of elastic search from elasticsearch.yml file.

Path: G:\elasticsearch_workspace\elasticsearch-6.1.1\config\ elasticsearch.yml

```
cluster.name: my-application
```

```
node.name: node-1
```



```
{
  "name" : "node-1",
  "cluster_name" : "my-application",
  "cluster_uuid" : "BuFT5_A3RfaGSx1_secRmQ",
  "version" : {
    "number" : "6.1.1",
    "build_hash" : "bd92e7f",
    "build_date" : "2017-12-17T20:23:25.338Z",
    "build_snapshot" : false,
    "lucene_version" : "7.1.0",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

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.

Elastic search x-pack

X-Pack is an Elastic Stack extension that bundles security, alerting, monitoring, reporting, and graph capabilities into one easy-to-install package. While the X-Pack components are designed to work together seamlessly, you can easily enable or disable the features you want to use.

Prior to Elasticsearch 5.0.0, you had to install separate Shield, Watcher, and Marvel plugins to get the features that are bundled together in X-Pack. With X-Pack, you no longer have to worry about whether or not you have the right version of each plugin, just install the X-Pack for the Elasticsearch version you're running, and you're good to go!



X-Pack

One Pack. Loads of Possibilities.

On its own, the Elastic Stack is a force to be reckoned with. X-Pack takes it to a new level by bundling powerful features into a single pack.



Security
(formerly Shield)



Alerting
(via Watcher)



Monitoring
(formerly Marvel)



Reporting



Graph



Machine Learning

Installing X-Pack

You must install Elasticsearch and Kibana before you install X-Pack. If you plan to use the X-Pack features in Logstash, you must also install Logstash before you install X-Pack. For more information, see:

- [Installing Elasticsearch](#)
- [Installing Kibana](#)
- [Installing Logstash](#)

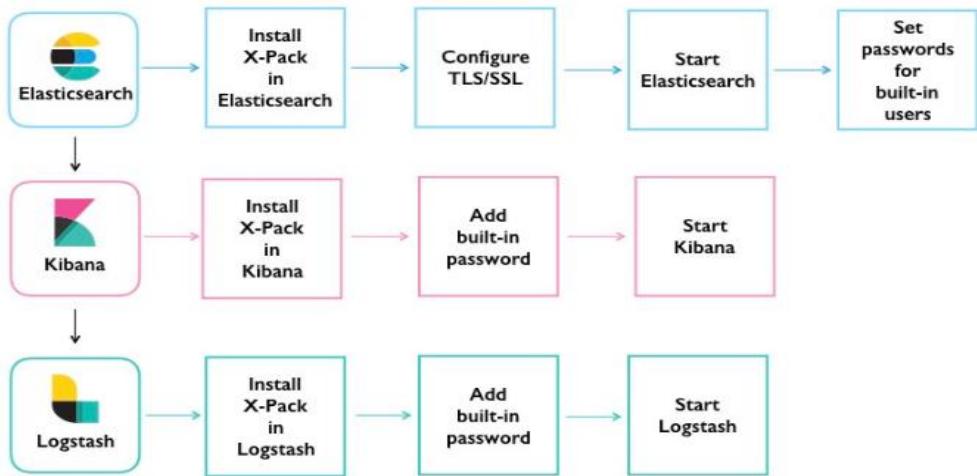
You must install the version of X-Pack that matches the version of each product. For more information about product compatibility, see the [Elastic Support Matrix](#).

To install X-Pack on the Elastic stack:

1. [Install X-Pack on Elasticsearch](#)
2. [Install X-Pack on Kibana](#)
3. [Install X-Pack on Logstash](#)

Note: The X-Pack installation scripts require direct internet access to download and install X-Pack.

The following diagram provides an overview of the steps that are required to set up X-Pack on each product:



We have already installed elastic search, so next we will install kibana.

Kibana

Kibana is an open source analytics and visualization platform designed to work with Elasticsearch. You use Kibana to search, view, and interact with data stored in Elasticsearch indices. You can easily perform advanced data analysis and visualize your data in a variety of charts, tables, and maps.

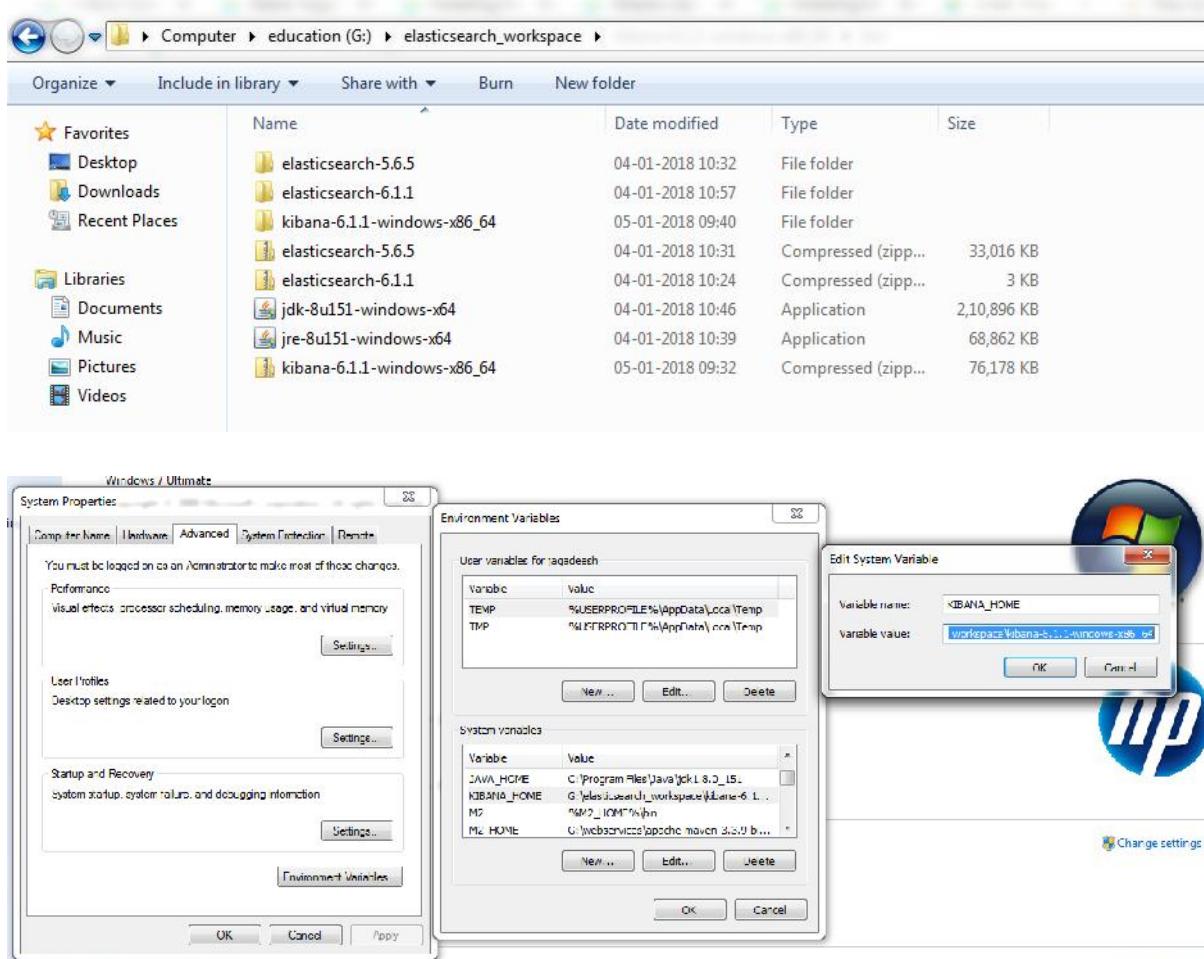
Kibana makes it easy to understand large volumes of data. Its simple, browser-based interface enables you to quickly create and share dynamic dashboards that display changes to Elasticsearch queries in real time.

Setting up Kibana is a snap. You can install Kibana and start exploring your Elasticsearch indices in minutes—no code, no additional infrastructure required.

Installing Kibana

Download: https://artifacts.elastic.co/downloads/kibana/kibana-6.1.1-windows-x86_64.zip

Unzip or extract the downloaded zip file and then set KIBANA_HOME= G:\elasticsearch_workspace\kibana-6.1.1-windows-x86_64



Configure elastic search server in kibana.yml file.

path: G:\elasticsearch_workspace\kibana-6.1.1-windows-x86_64\config\kibana.yml
kibana should point to elastic search server to perform operations.

elasticsearch.url: <http://localhost:9200> //Replace localhost with IPaddress

Run the kibana server:

Path: G:\elasticsearch_workspace\kibana-6.1.1-windows-x86_64\bin

```
[Kibana Server]
Microsoft Windows [Version 6.1.7601]
Copyright © 2009 Microsoft Corporation. All rights reserved.

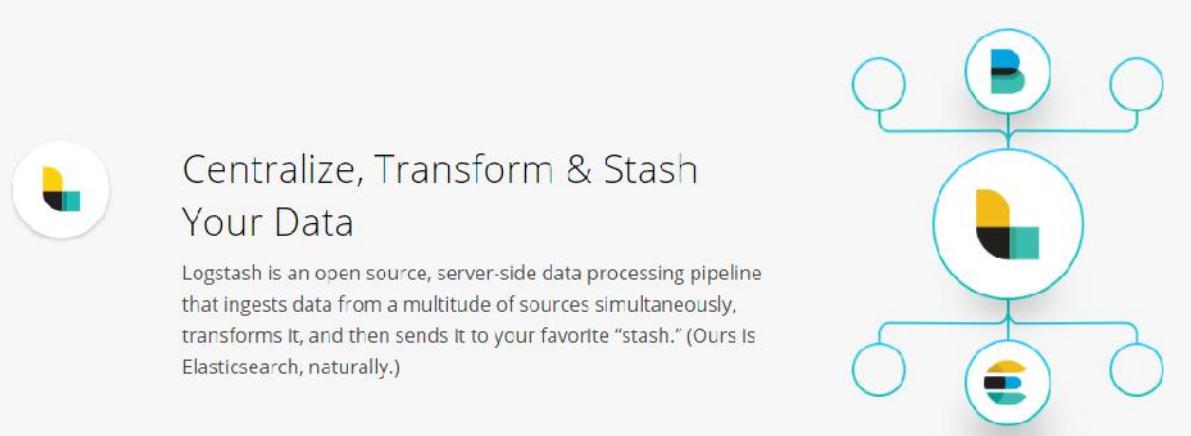
C:\Users\jagadg\elasticsearch_workspace\kibana-6.1.1-windows-x86_64\bin
C:\Users\jagadg>g:
G:\elasticsearch_workspace\kibana-6.1.1-windows-x86_64\bin>kibana.bat
'WIFRE' is not recognized as an internal or external command,
operable program or batch file.

[05:08:07.726] [info][status][l]plugin[kibana@6.1.1] Status changed from uninitialized to green - Ready
[05:08:07.752] [info][status][l]plugin[elasticsearch@6.1.1] Status changed from uninitialized to yellow - Waiting for Elasticsearch
[05:08:07.774] [info][status][l]plugin[console@6.1.1] Status changed from uninitialized to green - Ready
[05:08:07.817] [info][status][l]plugin[metrics@6.1.1] Status changed from uninitialized to green - Ready
[05:08:07.822] [info][status][l]plugin[elasticsearch@6.1.1] Status changed from yellow to green - Ready
[05:08:08.024] [info][status][l]plugin[timelion@6.1.1] Status changed from uninitialized to green - Ready
[05:08:08.030] [info][listening] Server running at http://localhost:5601
```

Kibana is now started, to access this hit the URL: <http://localhost:5601/>

The screenshot shows the Kibana 6.1.1 dashboard. On the left, there is a sidebar with icons for Discover, Visualize, Dashboard, Timelion, Dev Tools, and Management. The main area has two main sections: 'Visualize and Explore Data' and 'Manage and Administer the Elastic Stack'. The 'Visualize and Explore Data' section contains four cards: 'Dashboard' (display and share a collection of visualizations and saved searches), 'Discover' (interactively explore your data by querying and filtering raw documents), 'Timelion' (use an expression language to analyze time series data and visualize the results), and 'Visualize' (create visualizations and aggregate data stores in your Elasticsearch indices). The 'Manage and Administer the Elastic Stack' section contains three cards: 'Console' (skip curl and use this JSON Interface to work with your data directly), 'Index Patterns' (manage the index patterns that help retrieve your data from Elasticsearch), and 'Saved Objects' (import, export, and manage your saved searches, visualizations, and dashboards). At the bottom, there is a search bar with the placeholder 'Didn't find what you were looking for?' and a button labeled 'View full directory of Kibana plugins'.

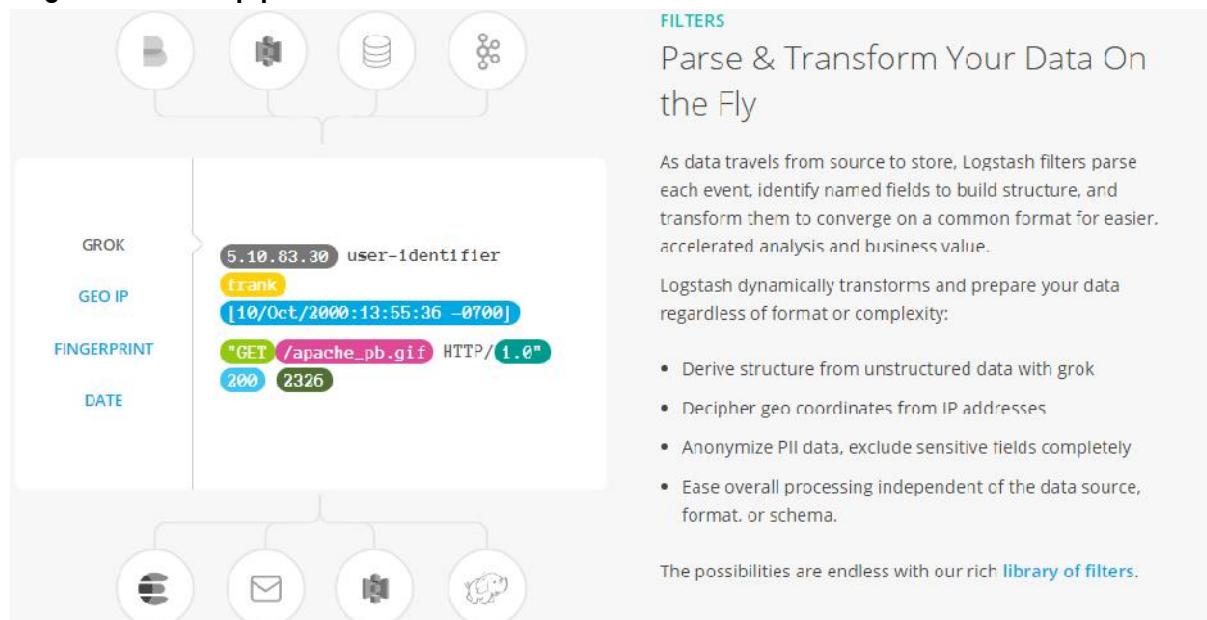
Logstash



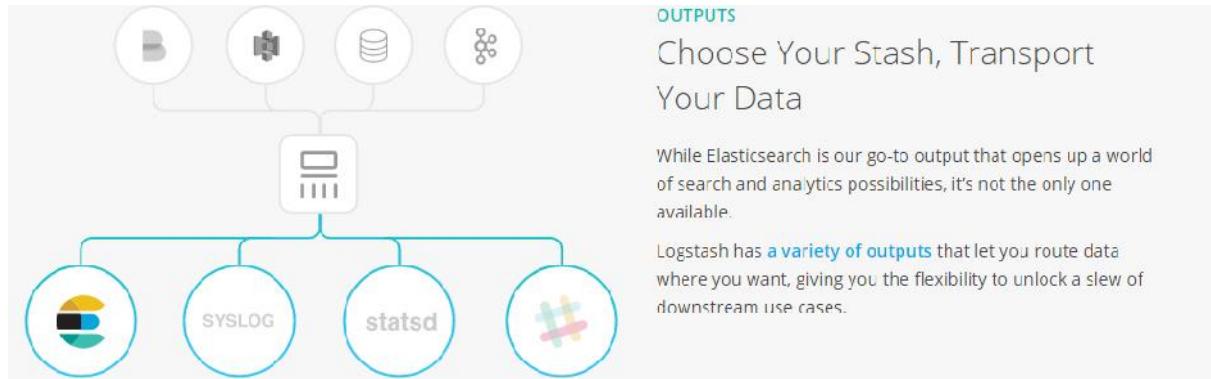
Logstash Input:



Logstash Filters/pipelines:



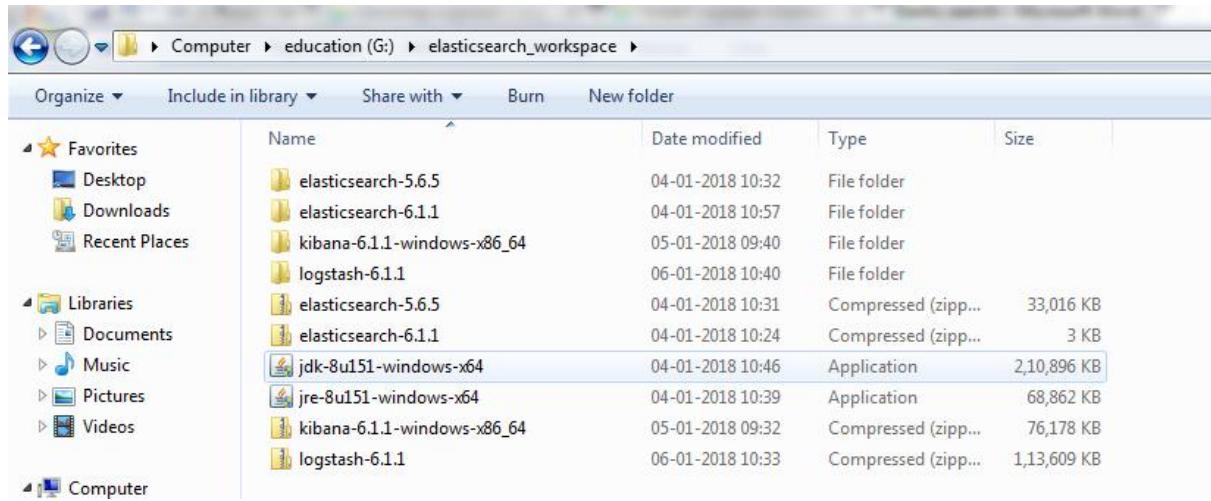
Logstash outputs:



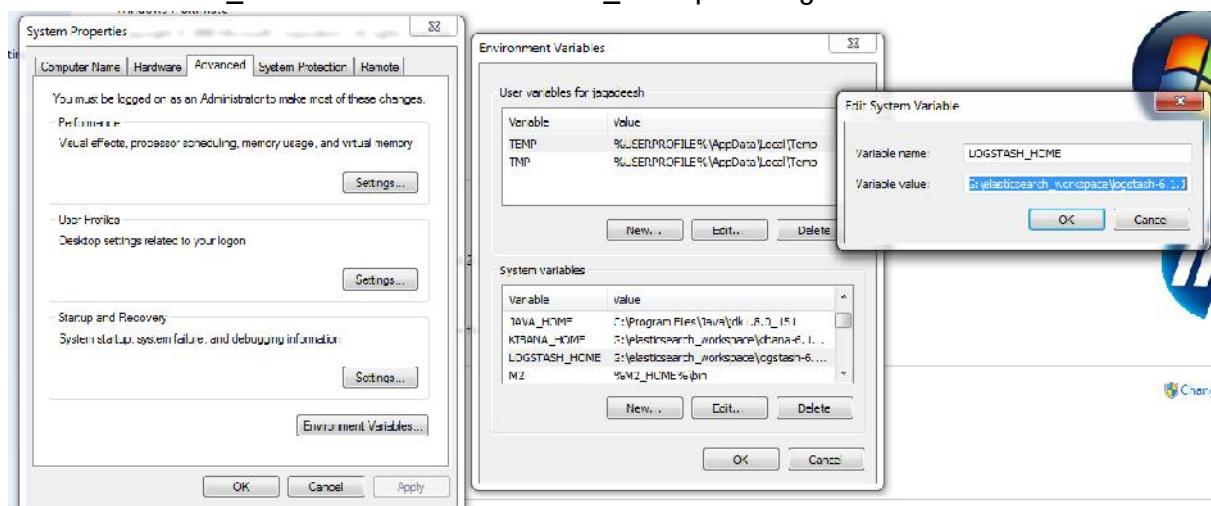
Install Logstash

Download: <https://artifacts.elastic.co/downloads/logstash/logstash-6.1.1.zip>

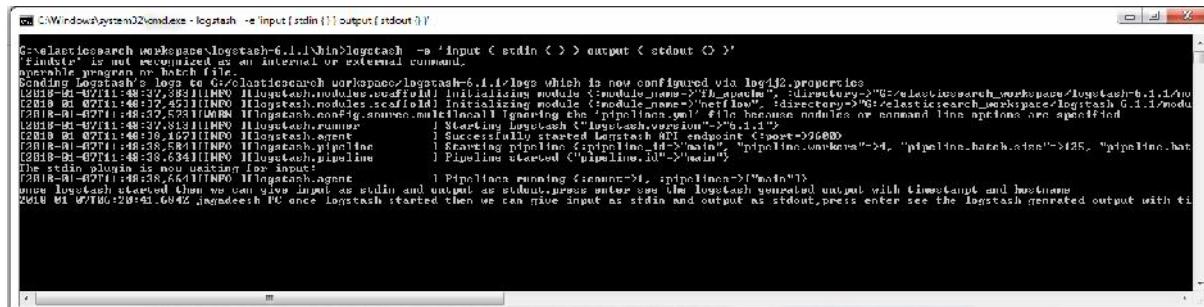
Unzip or extract folder:



Set LOGSTASH_HOME= "G:\elasticsearch_workspace\logstash-6.1.1"



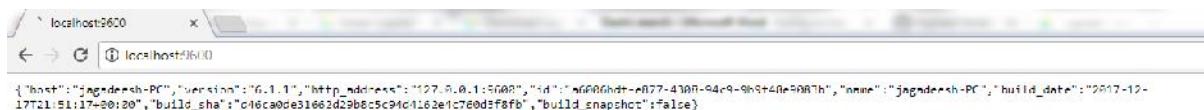
Start logstash:



```
C:\Windows\system32\cmd.exe -logstash -e 'input { stdin } output { stdout }'

G:\elasticsearch_workspace\logstash-6.1.1\bin>logstash -e 'input { stdin } output { stdout }'
[2018-01-07T11:48:39+00:00] [INFO] [logstash.modules.logstash] Initialization module <'logstash'> in directory >'G:\elasticsearch_workspace\logstash-6.1.1\bin\logstash.conf' has been loaded.
[2018-01-07T11:48:40+00:00] [INFO] [logstash.config.source.multifile] Ignoring the 'logstash.conf' file because neither file or command line options are specified.
[2018-01-07T11:48:40+00:00] [INFO] [logstash.runner] Starting Logstash <'logstash.conf'> "6.1.1"
[2018-01-07T11:48:40+00:00] [INFO] [logstash.agent] Successfully started Logstash HTTP endpoint at >7990<
[2018-01-07T11:48:40+00:00] [INFO] [logstash.pipeline] Starting pipeline <@main> "logstash.conf"
[2018-01-07T11:48:40+00:00] [INFO] [logstash.pipeline] Pipeline started <@main>
The stdin plugin is now waiting for input!
[2018-01-07T11:48:56+00:00] [INFO] [logstash.agent] Pipelines running: <@main>
Once logstash starts then we can give input as stdin and output as stdout, press enter see the logstash generated output with timestamp and host name
2018-01-07T10:20:41.604Z [jagadeesh-PC] Once Logstash started then we can give input as stdin and output as stdout, press enter see the logstash generated output with timestamp and host name
```

Logstash can be available as http service. Hit the below URL:



We have installed java, elastic search, kibana and logstash. We will provide security, alerting, monitoring, reporting, and graph capabilities for those tools later.

We can dive into elastic search First.

Talking to Elasticsearch

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

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.

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

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.

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.)

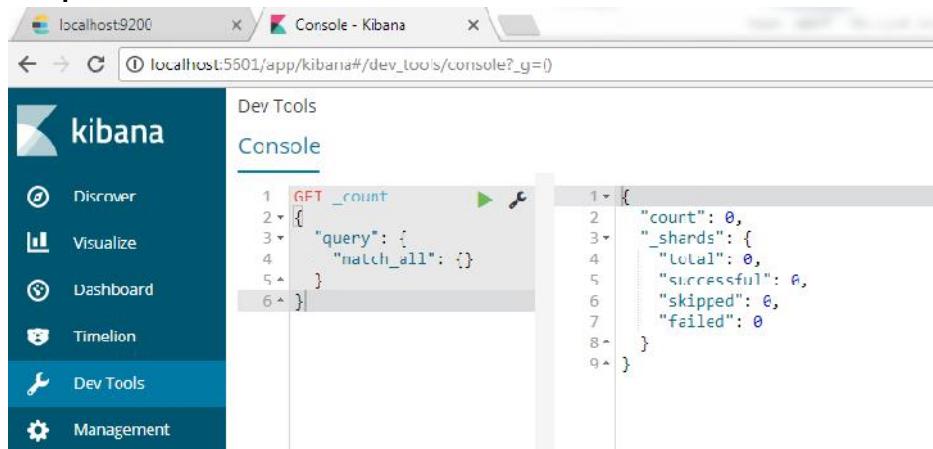
BODY

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": {}  
  }  
}'
```

Response:



The screenshot shows the Kibana Dev Tools Console interface. On the left is a sidebar with icons for Discover, Visualize, Dashboard, Timeline, Dev Tools (which is selected), and Management. The main area has tabs for Dev Tools and Console. In the Dev Tools tab, there is a code editor with the following JSON query:

```
1 GET _count
2 {
3   "query": {
4     "match_all": {}
5   }
6 }
```

On the right, the response is displayed in a monospaced font:

```
1 {
2   "_court": 0,
3   "_shards": {
4     "L0L01": 0,
5     "successful": 0,
6     "skipped": 0,
7     "failed": 0
8   }
9 }
```

Document Oriented

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.

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 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.

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.

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 that is searchable. A field without an inverted index is not searchable.

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

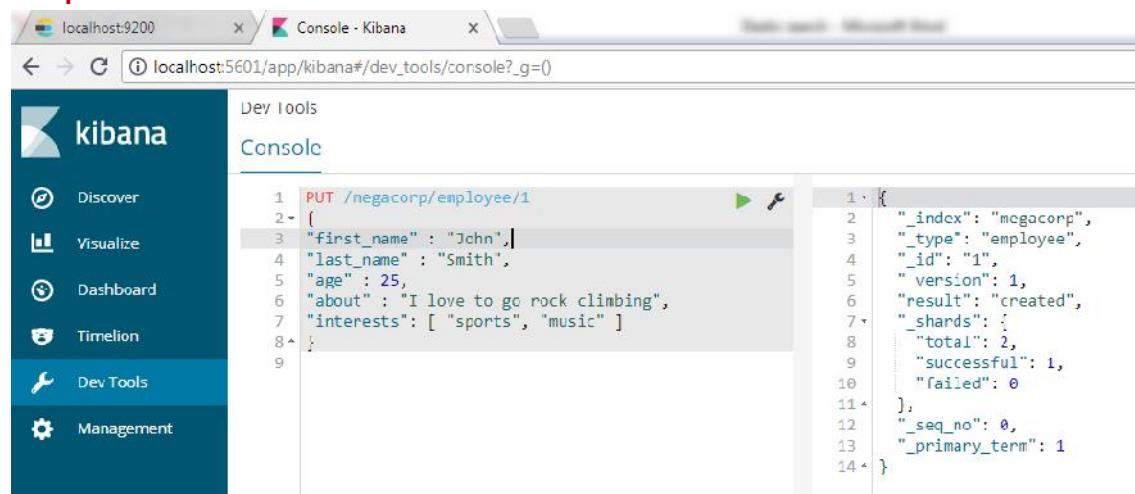
- Index a **document per employee**, which contains all the details of a single employee.
- Each document will be of **type employee**.
- That type will live in the **megacorp index**.
- That index will reside within our **Elasticsearch cluster**.

Request:

```
PUT /megacorp/employee/1
```

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

Response:



The screenshot shows the Kibana interface with the Dev Tools tab selected. In the Dev Tools section, there is a 'Console' tab where a PUT request is being typed and executed. The request is:

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

The response is displayed on the right side of the console, showing a successful creation of the document:

```
1. {  
2.   "_index": "megacorp",  
3.   "_type": "employee",  
4.   "_id": "1",  
5.   "version": 1,  
6.   "result": "created",  
7.   "shards": {  
8.     "total": 2,  
9.     "successful": 1,  
10.    "failed": 0  
11.  },  
12.  "_seq_no": 0,  
13.  "_primary_term": 1  
14. }
```

Request:

```
PUT /megacorp/employee/2
```

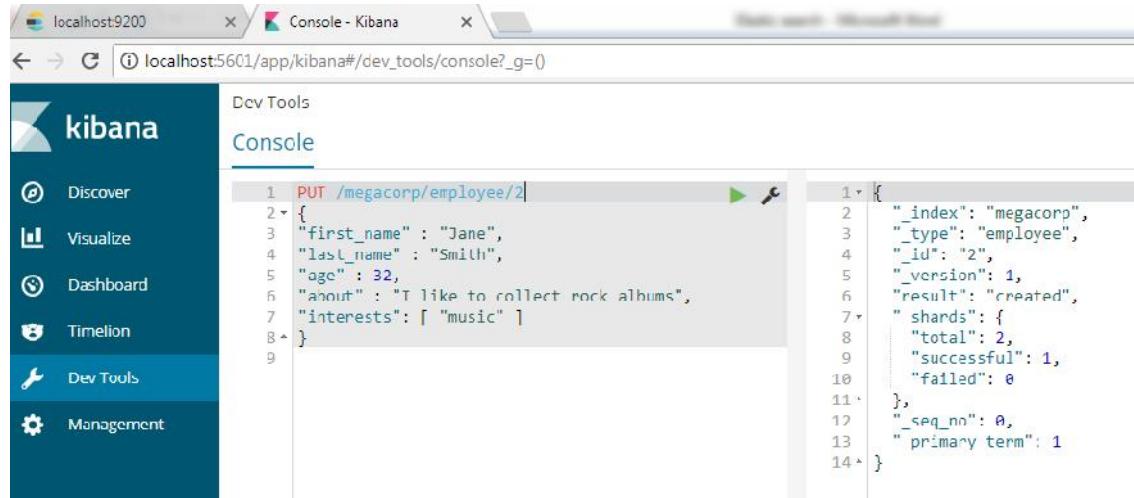
```
{  
  "first_name" : "Jane",  
  "last_name" : "Smith",
```

```

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

```

Response:



The screenshot shows the Kibana interface with the Dev Tools tab selected. In the console, a PUT request is made to the index '/megacorp/employee/2' with the following JSON payload:

```

1 PUT /megacorp/employee/2
2 {
3   "first_name" : "Jane",
4   "last_name" : "Smith",
5   "age" : 32,
6   "about" : "I like to collect rock albums",
7   "interests": [ "music" ]
8 }
9

```

The response on the right side of the console shows a successful creation of the document:

```

1 {
2   "_index": "megacorp",
3   "_type": "employee",
4   "_id": "2",
5   "_version": 1,
6   "result": "created",
7   "shards": {
8     "total": 2,
9     "successful": 1,
10    "failed": 0
11  },
12  "_seq_no": 0,
13  "_primary_term": 1
14 }

```

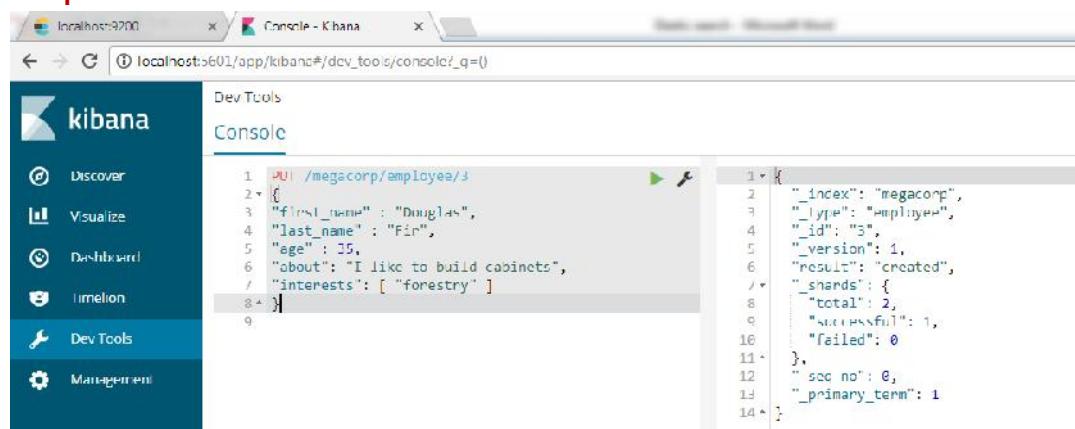
Request:

```

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

```

Response:



The screenshot shows the Kibana interface with the Dev Tools tab selected. In the console, a PUT request is made to the index '/megacorp/employee/3' with the following JSON payload:

```

1 PUT /megacorp/employee/3
2 {
3   "first_name" : "Douglas",
4   "last_name" : "Fir",
5   "age" : 35,
6   "about": "I like to build cabinets",
7   "interests": [ "forestry" ]
8 }
9

```

The response on the right side of the console shows a successful creation of the document:

```

1 {
2   "_index": "megacorp",
3   "_type": "employee",
4   "_id": "3",
5   "_version": 1,
6   "result": "created",
7   "shards": {
8     "total": 2,
9     "successful": 1,
10    "failed": 0
11  },
12  "_seq_no": 0,
13  "_primary_term": 1
14 }

```

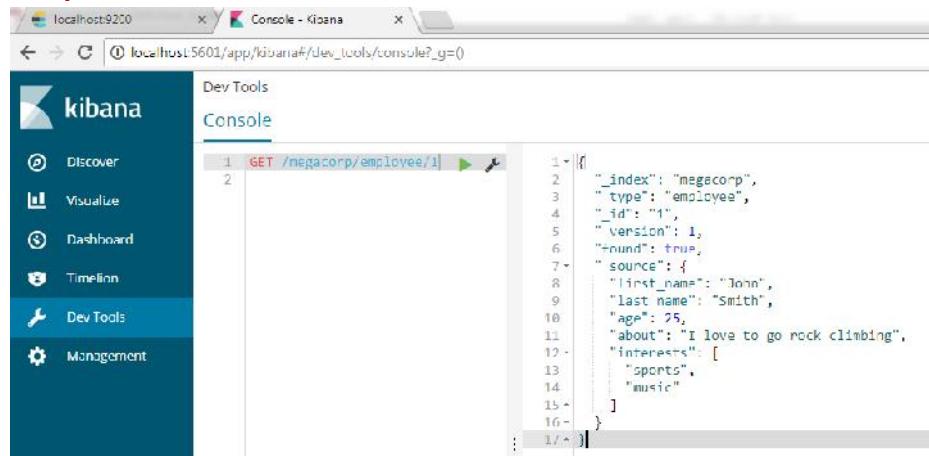
Retrieving a Document:

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.

Request:

GET /megacorp/employee/1

Response:



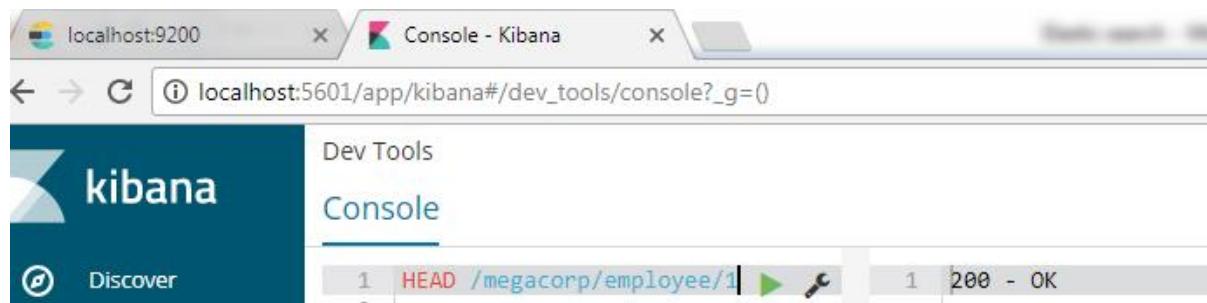
A screenshot of the Kibana Dev Tools Console. The URL in the address bar is `localhost:5601/app/kibana#/dev_tools/console?_g=()`. The console shows a single line of code: `1 GET /megacorp/employee/1`. To the right of the code, the response is displayed as a JSON object. The JSON object contains the following fields and values:

```
1 {  
2   "_index": "megacorp",  
3   "type": "employee",  
4   "_id": "1",  
5   "version": 1,  
6   "+found": true,  
7   "source": {  
8     "first_name": "John",  
9     "last_name": "Smith",  
10    "age": 25,  
11    "about": "I love to go rock climbing",  
12    "interests": [  
13      "sports",  
14      "music"  
15    ]  
16  }  
17}
```

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.



A screenshot of the Kibana Dev Tools Console. The URL in the address bar is `localhost:5601/app/kibana#/dev_tools/console?_g=()`. The console shows a single line of code: `1 HEAD /megacorp/employee/1`. To the right of the code, the response is displayed as a status message: `1 | 404 - Not Found`.

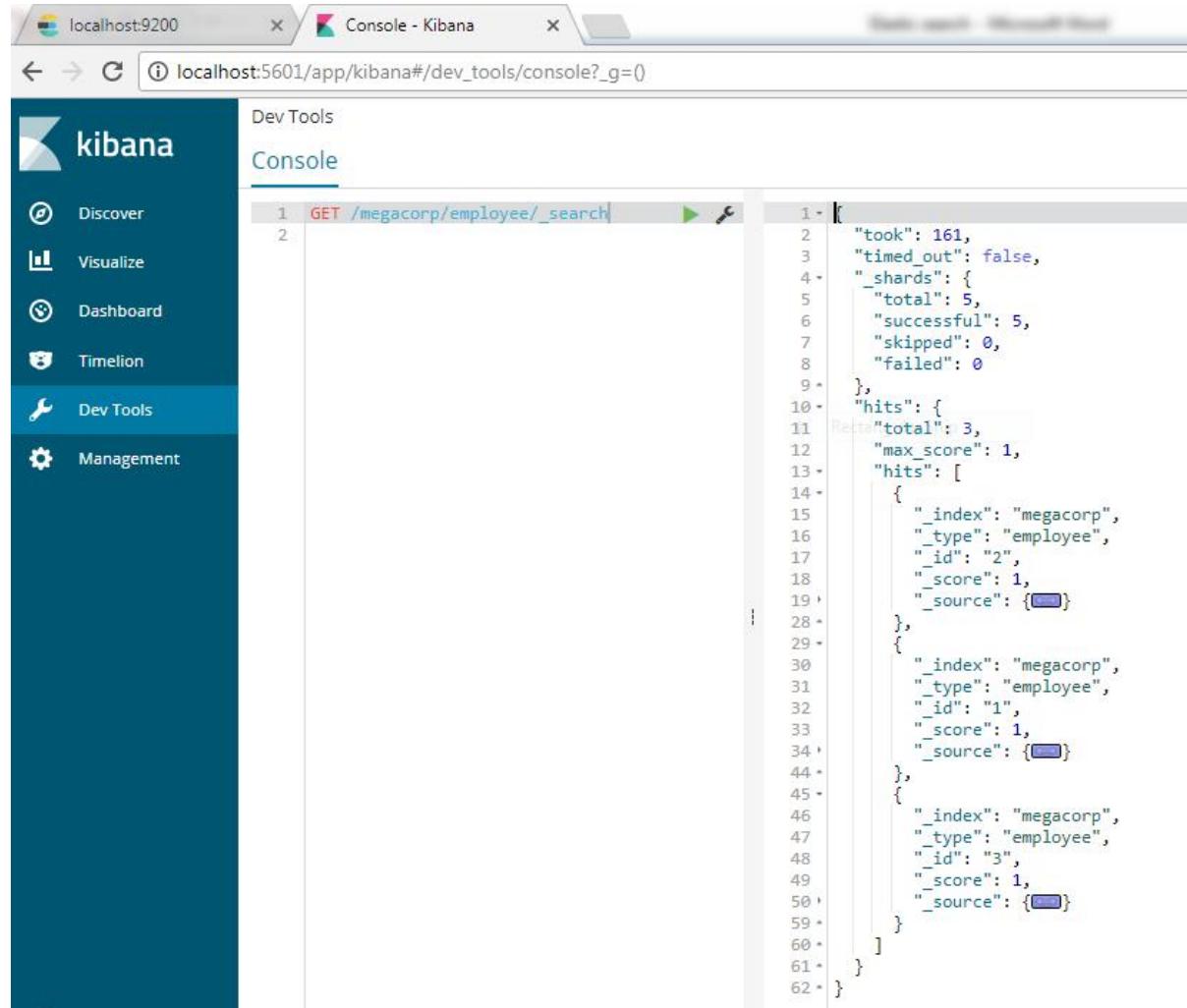


A screenshot of the Kibana Dev Tools Console. The URL in the address bar is `localhost:5601/app/kibana#/dev_tools/console?_g=()`. The console shows a single line of code: `1 HEAD /megacorp/employee/1`. To the right of the code, the response is displayed as a status message: `1 | 200 - OK`.

We will search for all employees, with this request:

GET /megacorp/employee/_search

Response:



The screenshot shows the Kibana interface with the 'Dev Tools' tab selected in the sidebar. In the 'Console' section, a command is being run: 'GET /megacorp/employee/_search'. The response is displayed as a JSON object with the following structure:

```
1 [ { 2 "took": 161, 3 "timed_out": false, 4 "shards": { 5 "total": 5, 6 "successful": 5, 7 "skipped": 0, 8 "failed": 0 9 }, 10 "hits": { 11 "total": 3, 12 "max_score": 1, 13 "hits": [ 14 { 15 "index": "megacorp", 16 "type": "employee", 17 "id": "2", 18 "score": 1, 19 "source": { 20 } 21 }, 22 { 23 "index": "megacorp", 24 "type": "employee", 25 "id": "1", 26 "score": 1, 27 "source": { 28 } 29 }, 30 { 31 "index": "megacorp", 32 "type": "employee", 33 "id": "3", 34 "score": 1, 35 "source": { 36 } 37 } 38 ], 39 } 40 }
```

By default, a search will return the top 10 results.

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.

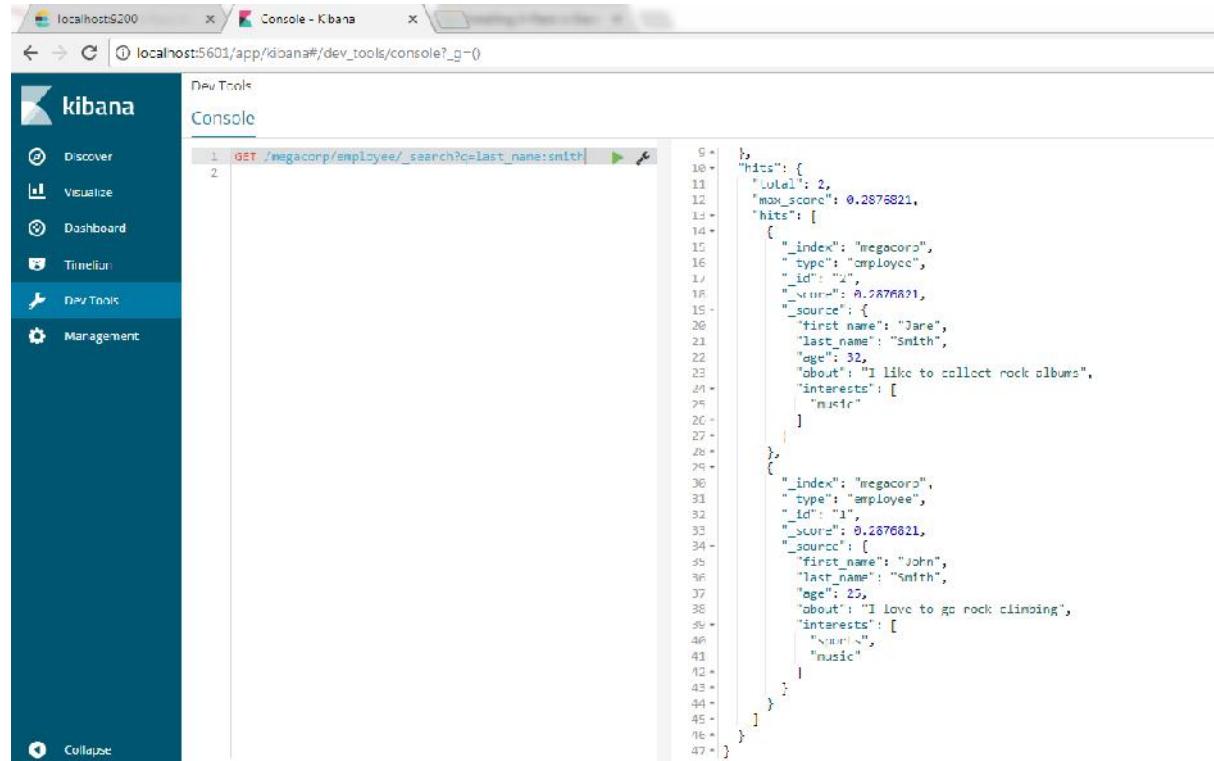
Let's try searching for employees who have "Smith" in their last name. To do this, we'll use a lightweight search method that is easy to use from the command line.

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
```

Response:



The screenshot shows the Kibana Dev Tools Console interface. On the left is a sidebar with icons for Discover, Visualize, Dashboard, Timeline, Dev Tools (which is selected), and Management. The main area has tabs for Dev Tools and Console. The Console tab is active, showing a command-line interface with the following content:

```
1 GET /megacorp/employee/_search?q=last_name:Smith
2
9 +
10 +
11 +
12 +
13 +
14 +
15 +
16 +
17 +
18 +
19 +
20 +
21 +
22 +
23 +
24 +
25 +
26 +
27 +
28 +
29 +
30 +
31 +
32 +
33 +
34 +
35 +
36 +
37 +
38 +
39 +
40 +
41 +
42 +
43 +
44 +
45 +
46 +
47 + }
```

The response shows two hits from the "megacorp" index, type "employee". The first hit is for "Jane Smith" (id: 2) and the second for "John Smith" (id: 1). Both documents have a score of 0.2876821 and include fields like first_name, last_name, age, about, and interests.

Search with Query DSL

There are two forms of the search API: a “lite” query-string version that expects all its 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.

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.

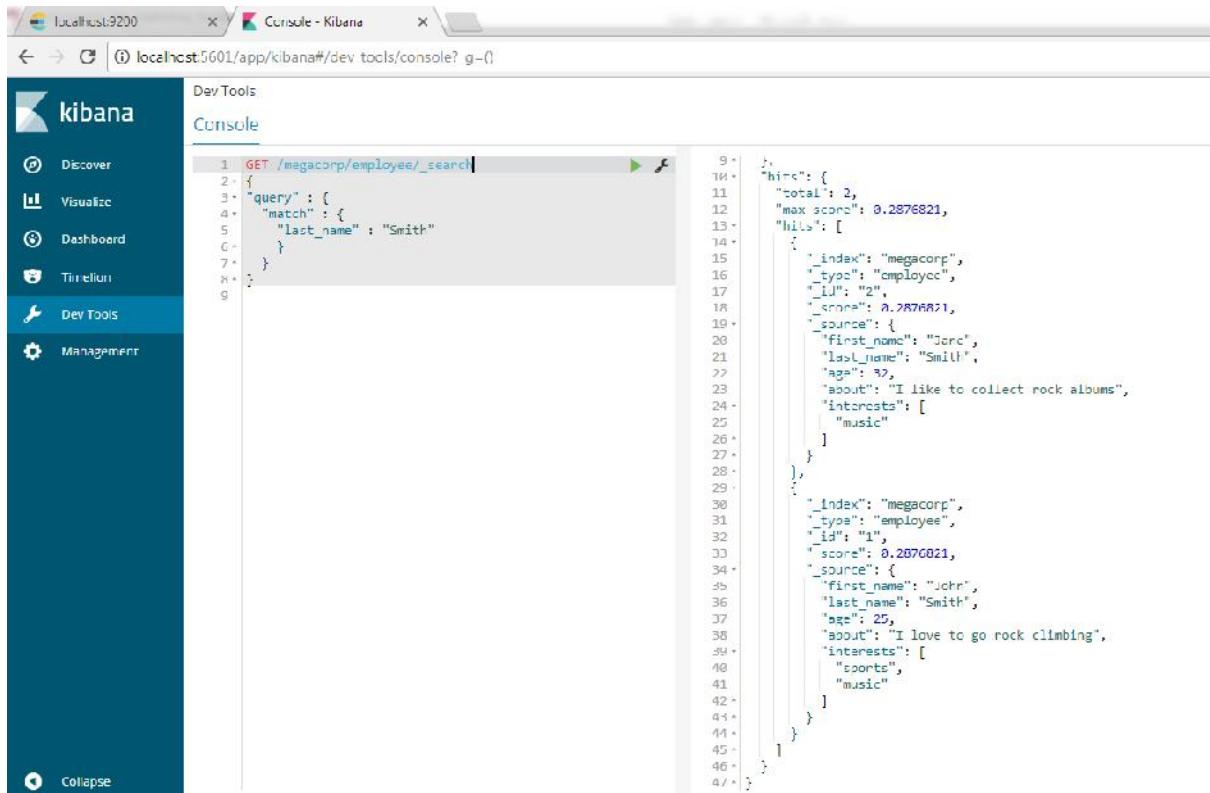
Request:

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

```
{
  "query" : {
    "match" : {
      "last_name" : "Smith"
    }
}
```

}

Response:



The screenshot shows the Kibana interface with the Dev Tools tab selected. In the console, a GET request is being run against the '/megacorp/employee/_search' index. The query is a simple match on the 'last_name' field, specifically 'Smith'. The response shows two hits, each representing an employee document from the 'megacorp' index. The first hit has an ID of '2' and a score of 0.2876821. The second hit has an ID of '1' and a score of 0.2876821. Both documents contain fields like 'first_name', 'last_name', 'age', 'about', and 'interests'.

```
1 GET /megacorp/employee/_search
2 {
3   "query": {
4     "match": {
5       "last_name": "Smith"
6     }
7   }
8 }
9
10 [
11   {
12     "_index": "megacorp",
13     "_type": "employee",
14     "_id": "2",
15     "_score": 0.2876821,
16     "_source": {
17       "first_name": "Jenc",
18       "last_name": "Smith",
19       "age": 32,
20       "about": "I like to collect rock albums",
21       "interests": [
22         "music"
23       ]
24     }
25   },
26   {
27     "_index": "megacorp",
28     "_type": "employee",
29     "_id": "1",
30     "_score": 0.2876821,
31     "_source": {
32       "first_name": "John",
33       "last_name": "Smith",
34       "age": 25,
35       "about": "I love to go rock climbing",
36       "interests": [
37         "sports",
38         "music"
39       ]
40     }
41   }
42 ],
43 [
44   {
45     "_index": "megacorp",
46     "_type": "employee",
47     "_id": "1",
48     "_score": 0.2876821,
49     "_source": {
50       "first_name": "John",
51       "last_name": "Smith",
52       "age": 25,
53       "about": "I love to go rock climbing",
54       "interests": [
55         "sports",
56         "music"
57       ]
58     }
59   }
60 ]
```

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:

Request:

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

```
{
```

```
  "query": {
    "bool": {
      "must": {
        "match": {
          "last_name": "smith"
        }
      }
    },
    "filter": {
      "range": {
        "age": {
          "gt": 30
        }
      }
    }
  }
```

})
})
})
})
})

Response:

The screenshot shows the Kibana Dev Tools Console interface. On the left, a sidebar lists navigation options: Discover, Visualize, Dashboard, Timelion, Dev Tools (which is selected), and Management. The main area is titled "Console". A code editor on the left contains a search query:

```
1 GET /megacorp/employee/_search
2 {
3   "query": {
4     "bool": {
5       "must": [
6         "match": {
7           "last_name" : "smith"
8         }
9       ],
10      "filter": {
11        "range": {
12          "age" : { "gt" : 30 }
13        }
14      }
15    }
16  }
17 }
```

The results of the search are displayed on the right, showing a single hit:

```
1 {
2   "took": 87,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 1,
12    "max_score": 0.2876821,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_type": "employee",
17        "_id": "2",
18        "_score": 0.2876821,
19        "_source": {
20          "first_name": "Jane",
21          "last_name": "Smith",
22          "age": 32,
23          "about": "I like to collect rock albums",
24          "interests": [
25            "music"
26          ]
27        }
28      }
29    ]
30  }
31 }
```

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" : {

"a

}

}

}

Response:

The screenshot shows the Kibana interface with the 'Dev Tools' tab selected. In the 'Console' section, a search query is run against the 'megacorp/employee/_search' index. The response shows two hits, both of which have an '_id' of '1' and a '_score' of 0.5753642. The first hit is for an employee named John Smith, 25 years old, who loves rock climbing and has interests in sports and music. The second hit is for an employee named Jane Smith, 32 years old, who likes to collect rock albums and has an interest in music.

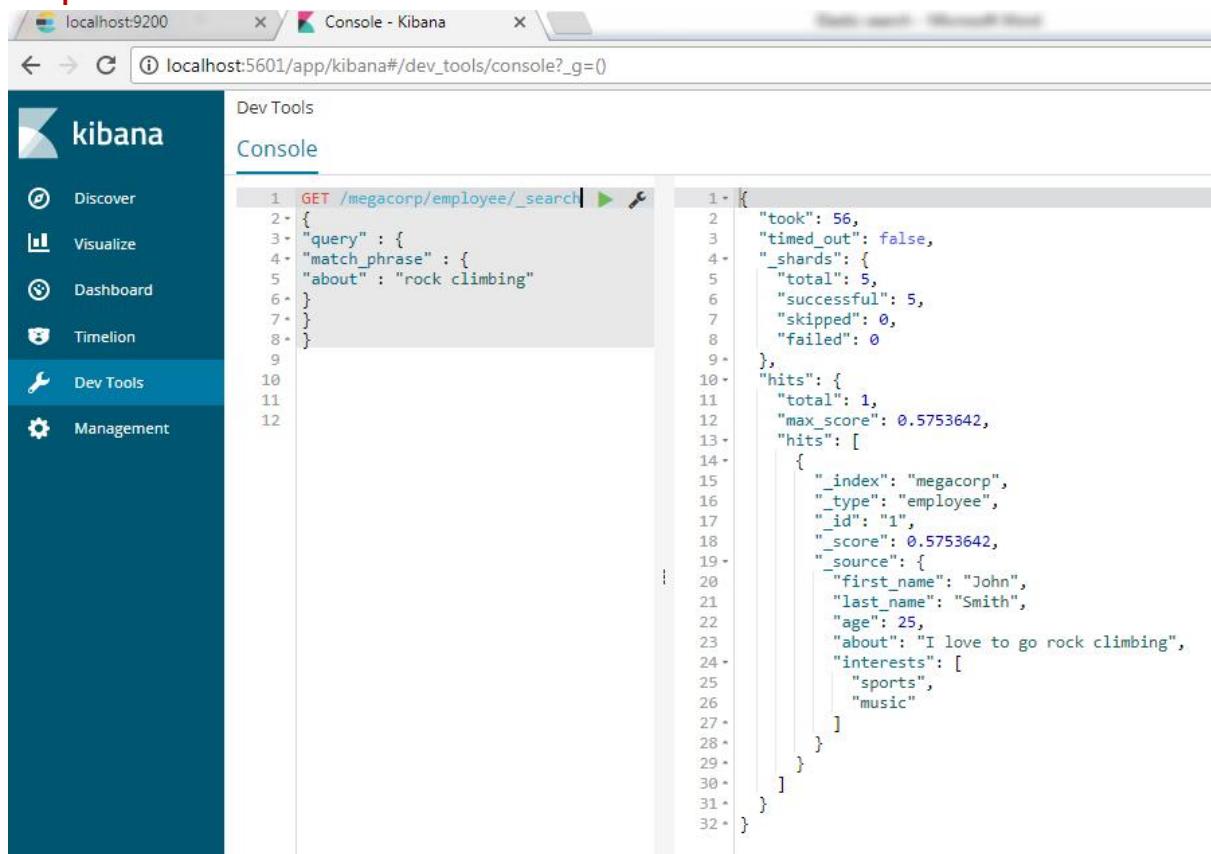
```
1  GET /megacorp/employee/_search
2  {
3    "query" : {
4      "match" : {
5        "about" : "rock climbing"
6      }
7    }
8  }
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

```
{
  "hits": [
    {
      "_index": "megacorp",
      "_type": "employee",
      "_id": "1",
      "_score": 0.5753642,
      "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": 0.2876821,
      "source": {
        "first_name": "Jane",
        "last_name": "Smith",
        "age": 32,
        "about": "I like to collect rock albums",
        "interests": [
          "music"
        ]
      }
    }
  ]
}
```

Request: exact match

```
{
  "query" : {
    "match_phrase" : {
      "about" : "rock climbing"
    }
  }
}
```

Response:



The screenshot shows the Kibana interface with the 'Dev Tools' tab selected. In the 'Console' section, a code editor displays a GET request to the '/megacorp/employee/_search' endpoint. The query uses a 'match_phrase' search for 'about': 'rock climbing'. The response is a JSON object containing search statistics and a single hit. The hit details an employee named John Smith, aged 25, with interests in sports and music, and a specific highlight for the phrase 'rock climbing'.

```
1 GET /megacorp/employee/_search| ▶ 🔍
2 {
3   "query" : {
4     "match_phrase" : {
5       "about" : "rock climbing"
6     }
7   }
8 }
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32 }
```

```
1 {
2   "took": 56,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 1,
12    "max_score": 0.5753642,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_type": "employee",
17        "_id": "1",
18        "_score": 0.5753642,
19        "_source": {
20          "first_name": "John",
21          "last_name": "Smith",
22          "age": 25,
23          "about": "I love to go rock climbing",
24          "interests": [
25            "sports",
26            "music"
27          ]
28        }
29      }
30    ]
31  }
32 }
```

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.

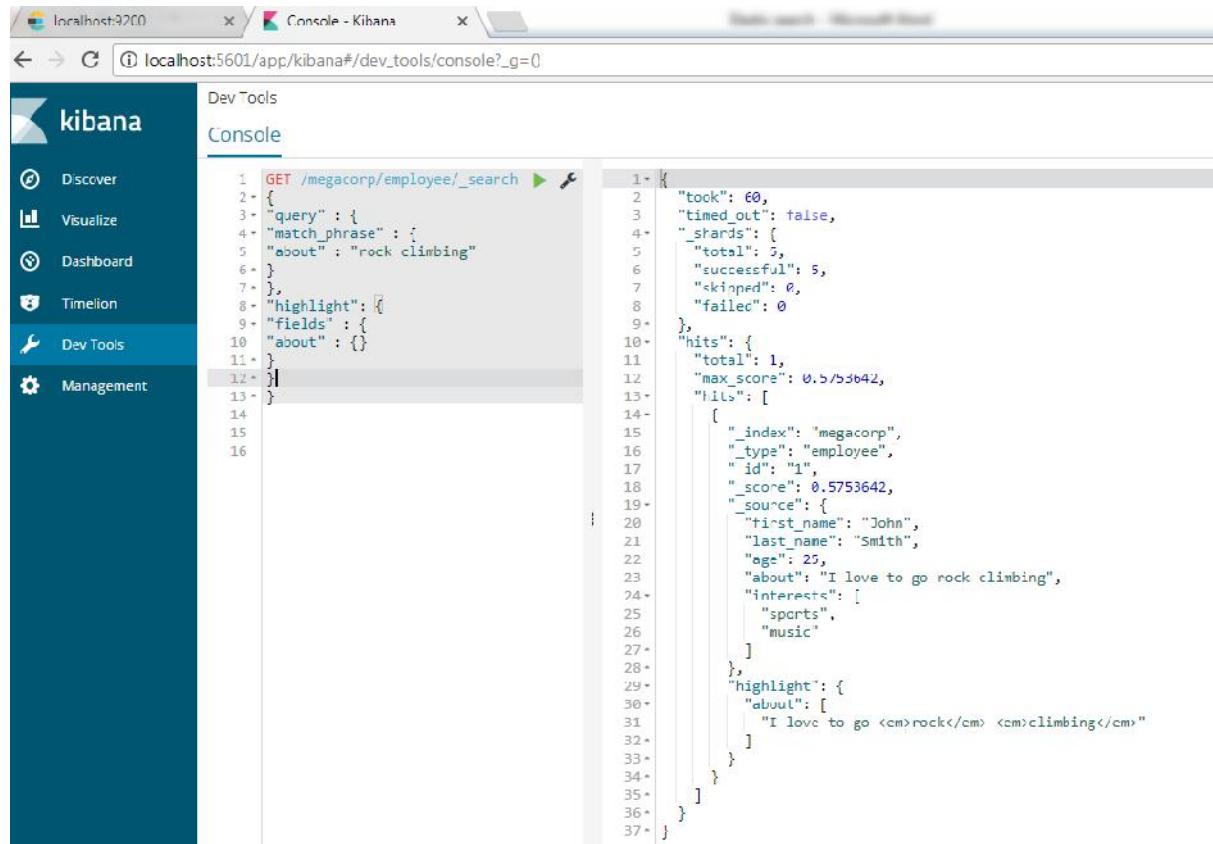
Let's rerun our previous query, but add a new highlight parameter:

Request:

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

Response:

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 `` HTML tags:



The screenshot shows the Kibana interface with the Dev Tools tab selected. In the console, a GET request is being run against the /megacorp/employee/_search endpoint. The response is displayed in JSON format, showing a successful search with one hit. The hit's source includes fields like first_name, last_name, age, about, and interests. A 'highlight' section is present in the response, containing the text "I love to go rock climbing" with the word "rock climbing" wrapped in `` tags.

```
1- {
2-   "took": 60,
3-   "timed_out": false,
4-   "_shards": [
5-     {"total": 5,
6-      "successful": 5,
7-      "skipped": 0,
8-      "failed": 0
9-    },
10-   "hits": [
11-     {"total": 1,
12-      "max_score": 0.573642,
13-      "hits": [
14-        {
15-          "_index": "megacorp",
16-          "_type": "employee",
17-          "id": "1",
18-          "score": 0.573642,
19-          "_source": {
20-            "first_name": "John",
21-            "last_name": "Smith",
22-            "age": 25,
23-            "about": "I love to go rock climbing",
24-            "interests": [
25-              "sports",
26-              "music"
27-            ],
28-            "highlight": {
29-              "about": [
30-                "I love to go rock climbing"
31-              ]
32-            }
33-          }
34-        }
35-      ]
36-    }
37-  }
```

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.

Note: Fielddata should be enabled to perform aggregations and sorting.

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

GET /megacorp/employee/_search

```
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

The screenshot shows the Kibana Dev Tools Console interface. On the left is a dark sidebar with navigation links: Discover, Visualize, Dashboard, Timeline, Dev Tools (which is selected), and Management. The main area is titled 'Console' and contains a code editor with the following Elasticsearch query:

```
1 GET /megacorp/employee/_search
2 {
3   "query": {
4     "match_phrase": {
5       "about": "rock climbing"
6     }
7   },
8   "highlight": {
9     "fields": {
10      "about": {}
11    }
12  }
13}
14
15
16
```

To the right of the code editor is the response from the Elasticsearch search request. It includes the following JSON structure:

```
1 {
2   "took": 60,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 1,
12    "max_score": 0.5753642,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_type": "employee",
17        "id": "1",
18        "_score": 0.5753642,
19        "_source": {
20          "first_name": "John",
21          "last_name": "Smith",
22          "age": 25,
23          "about": "I love to go rock climbing",
24          "interests": [
25            "sports",
26            "music"
27          ],
28          "highlight": {
29            "about": [
30              "I love to go <cm>rock</cm> <cm>climbing</cm>"
31            ]
32          }
33        }
34      }
35    ]
36  }
37}
```

Enabling fielddata on text fields :

You can enable fielddata on an existing text field using the PUT mapping API as follows:

The screenshot shows the Kibana Dev Tools Console interface. On the left is a dark sidebar with navigation links: Discover, Visualize, Dashboard, Timeline, Dev Tools (selected), and Management. The main area is titled 'Console' and contains a code editor with the following PUT mapping API call:

```
1 PUT megacorp/_mapping/employee
2 {
3   "properties": {
4     "interests": {
5       "type": "text",
6       "fielddata": true
7     }
8   }
9 }
```

To the right of the code editor is the response from the Elasticsearch PUT mapping request. It includes the following JSON structure:

```
1 {
2   "acknowledged": true
3 }
```

The screenshot shows the Kibana interface with the Dev Tools tab selected. In the console, a search request is being run:

```
1 GET /megacorp/employee/_search
2 {
3   "aggs": {
4     "all_interests": {
5       "terms": { "field": "interests" }
6     }
7   }
8 }
```

The response returned is:

```
1 {
2   "took": 69,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 3,
12    "max_score": 1,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_id": "AVWZGqIyBQKJLwzA",
17        "_score": 1,
18        "_source": {
19          "name": "John Doe",
20          "age": 30,
21          "interests": "music"
22        }
23      },
24      {
25        "_index": "megacorp",
26        "_id": "AVWZGqIyBQKJLwzB",
27        "_score": 1,
28        "_source": {
29          "name": "Jane Doe",
30          "age": 25,
31          "interests": "forestry"
32        }
33      },
34      {
35        "_index": "megacorp",
36        "_id": "AVWZGqIyBQKJLwzC",
37        "_score": 1,
38        "_source": {
39          "name": "Bob Smith",
40          "age": 40,
41          "interests": "sports"
42        }
43      }
44    ]
45  },
46  " aggregations": {
47    "all_interests": {
48      "doc_count_error_upper_bound": 0,
49      "sum_other_doc_count": 0,
50      "buckets": [
51        {
52          "key": "music",
53          "doc_count": 2
54        },
55        {
56          "key": "forestry",
57          "doc_count": 1
58        },
59        {
60          "key": "sports",
61          "doc_count": 1
62        }
63      ]
64    }
65  }
66 }
```

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

Request:

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

Response:

The screenshot shows the Kibana interface with the Dev Tools tab selected. In the left sidebar, the Dev Tools option is highlighted. The main area is titled "Console" and contains a code editor with a syntax-highlighted Elasticsearch query. The results of the query are displayed in a table-like format on the right, showing aggregated data for three categories: "music", "forestry", and "sports".

```
1 GET /megacorp/employee/_search
2 {
3     "aggs" : [
4         "all_interests" : {
5             "terms" : { "field" : "interests" },
6             "aggs" : {
7                 "avg_age" : {
8                     "avg" : { "field" : "age" }
9                 }
10            }
11        }
12    }
13 }
14
15
```

Category	doc_count	avg_age
music	2	28.5
forestry	1	35
sports	1	25

Life Inside a Cluster

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.

An Empty Cluster

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

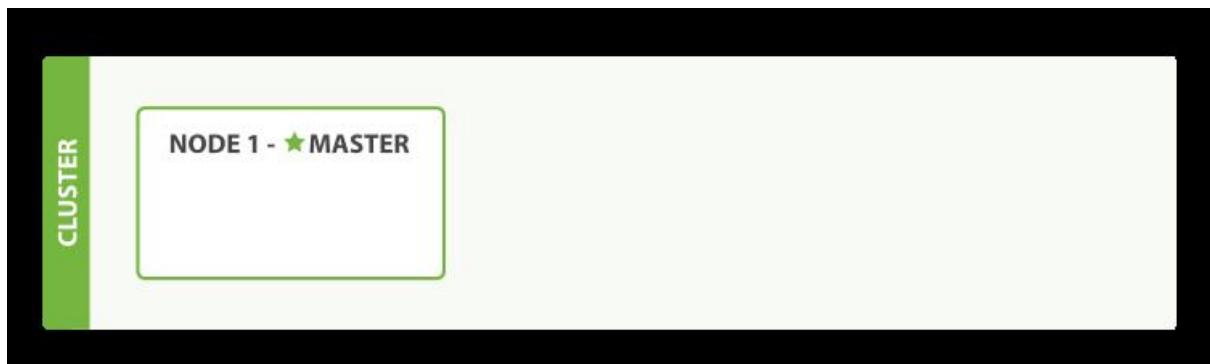


Figure – A cluster with one empty node

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.

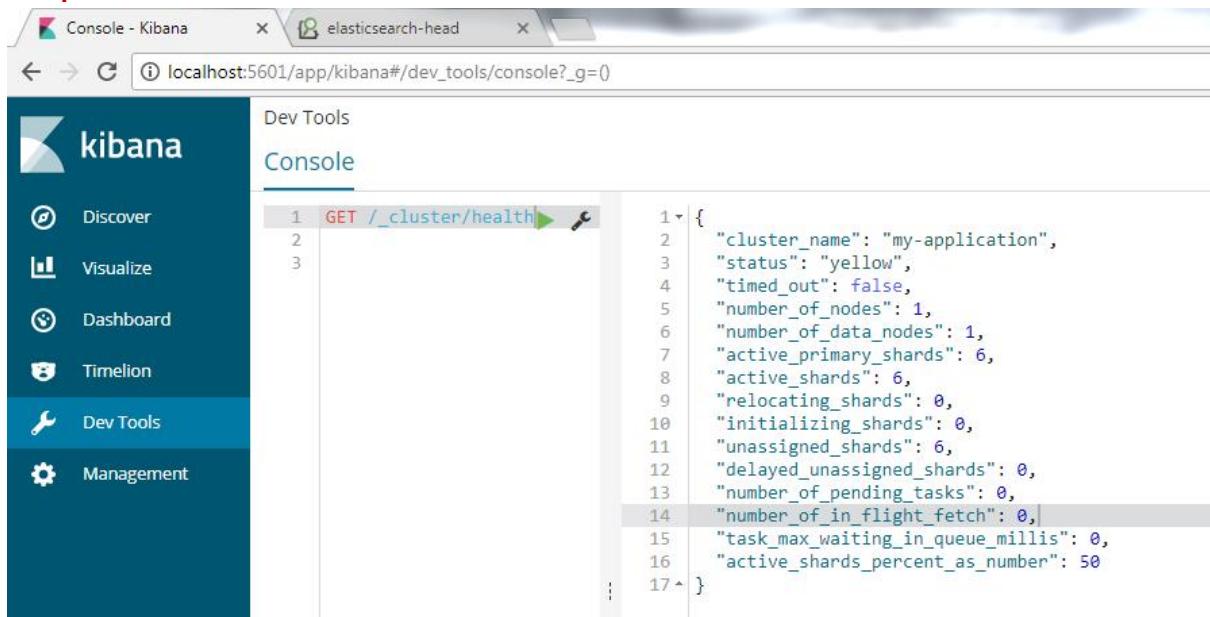
Cluster Health

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:

Request:

GET /_cluster/health

Response:

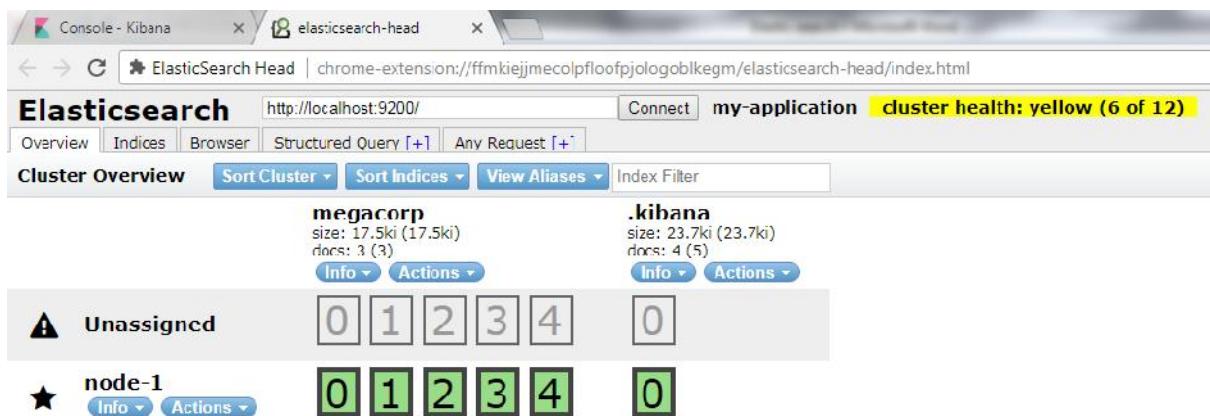


The screenshot shows the Kibana interface with the 'Dev Tools' tab selected. In the 'Console' section, a command is being run: `GET /_cluster/health`. The response is displayed as a JSON object:

```
1 {  
2   "cluster_name": "my-application",  
3   "status": "yellow",  
4   "timed_out": false,  
5   "number_of_nodes": 1,  
6   "number_of_data_nodes": 1,  
7   "active_primary_shards": 6,  
8   "active_shards": 6,  
9   "relocating_shards": 0,  
10  "initializing_shards": 0,  
11  "unassigned_shards": 6,  
12  "delayed_unassigned_shards": 0,  
13  "number_of_pending_tasks": 0,  
14  "number_of_in_flight_fetch": 0,  
15  "task_max_waiting_in_queue_millis": 0,  
16  "active_shards_percent_as_number": 50  
17 }
```

Head plug-in GUI:

To see the cluster and nodes with their status add head plug in extension to chrome.



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.

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.

In the above snippet, `megacorp` is an index. It is a logical namespace and that points to 5 primary shards in node-1 server. And 5 replica shards (copy of primary shards). At present these replicas are un assigned as we have only 1 node in the cluster (my-application). If we add another node to the cluster then these replicas will be active to primary shards. **Shard is nothing but some part of node.** By default a node can be partitioned to 5 parts or 5 shards. All primary shards can have original data and replicas can have carbon copy of this primary shards data.

A shard is a low-level worker unit that holds just a slice of all the data in the index.

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 (`megacorp`).

Elasticsearch distributes data around your cluster through Shards. 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.

Note: 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.

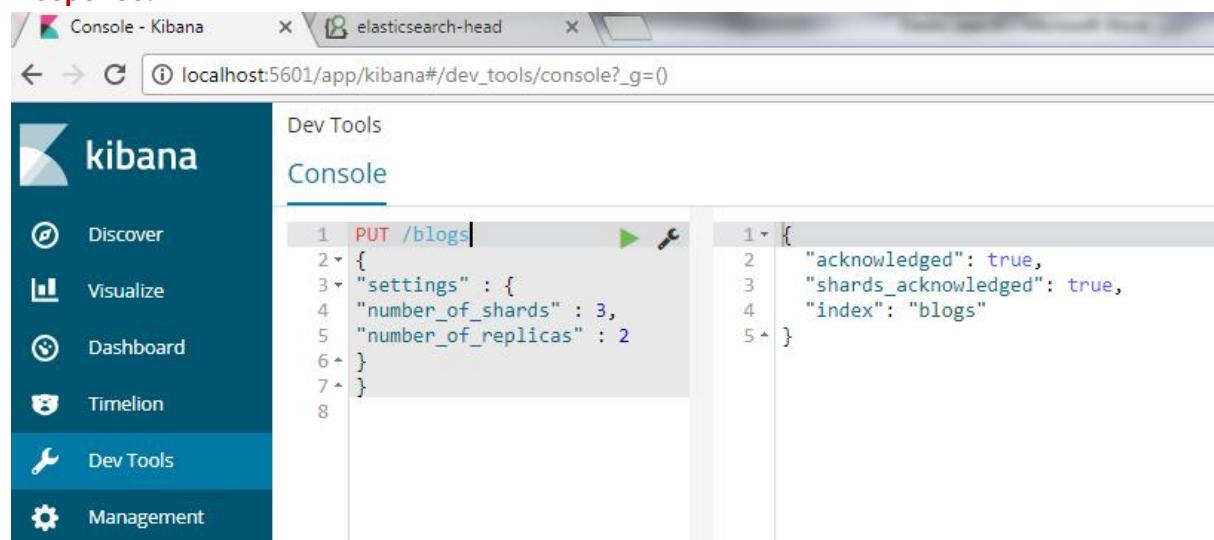
Let's create an index called `blogs` in our my-application one-node cluster. By default, indices are assigned five primary shards, but for the purpose of this

demonstration, we'll assign just 3 primary shards and 2 replica (2 replica of every primary shard):

Request:

```
PUT /blogs
{
  "settings" : {
    "number_of_shards" : 3,
    "number_of_replicas" : 2
  }
}
```

Response:



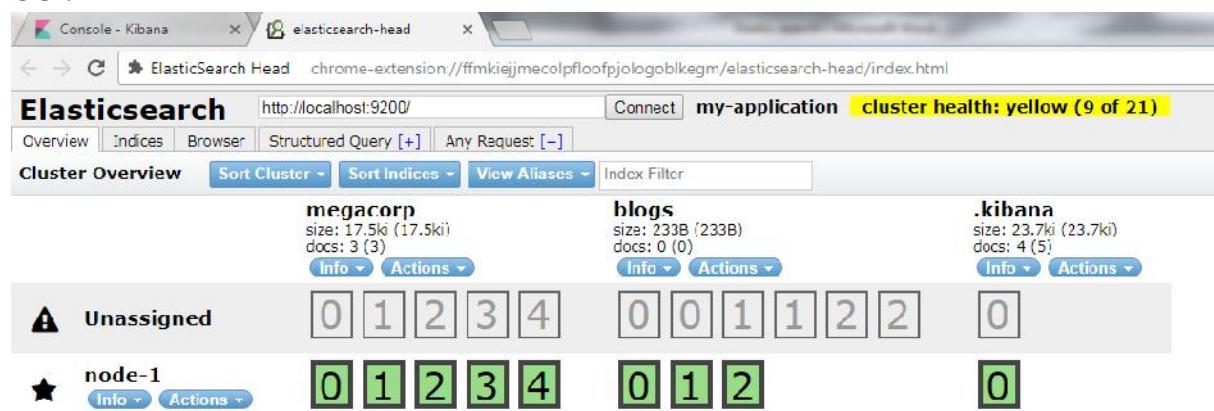
The screenshot shows the Kibana Dev Tools interface. On the left, there's a sidebar with icons for Discover, Visualize, Dashboard, Timelion, Dev Tools (which is selected), and Management. The main area has tabs for Dev Tools and Console. In the Dev Tools tab, a code editor shows the following JSON:

```
1 PUT /blogs| ↗ 🔒
2 {
3   "settings" : {
4     "number_of_shards" : 3,
5     "number_of_replicas" : 2
6   }
7 }
```

To the right, the Console tab shows the response from Elasticsearch:

```
1 { "acknowledged": true,
2   "shards_acknowledged": true,
3   "index": "blogs"
4 }
```

GUI:



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 6 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.

Request:

```
PUT /blogs1
{
  "settings" : {
    "number_of_shards" : 3,
    "number_of_replicas" : 1
  }
}
```

Response:

```
{
  "cluster_name": "elasticsearch",
  "status": "yellow",
  "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
}
```

Our cluster now looks like [below](#). All three primary shards have been allocated to Node 1.

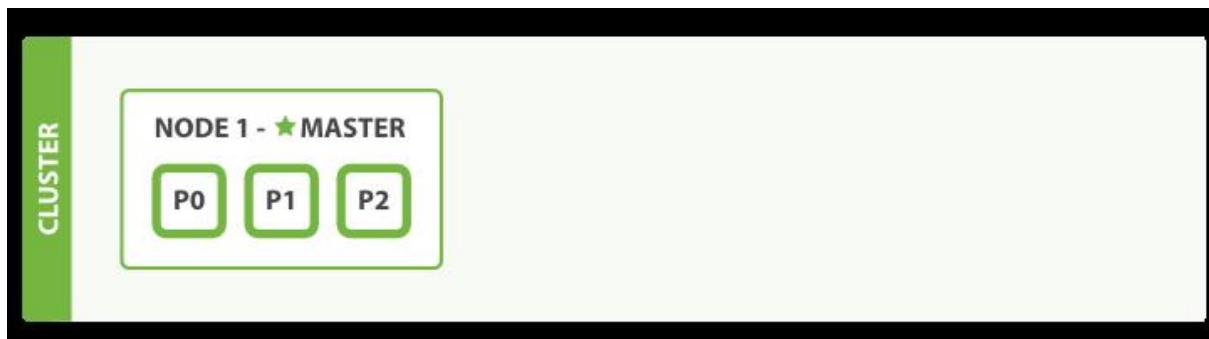


Fig: A single-node cluster with an index

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.

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 below.

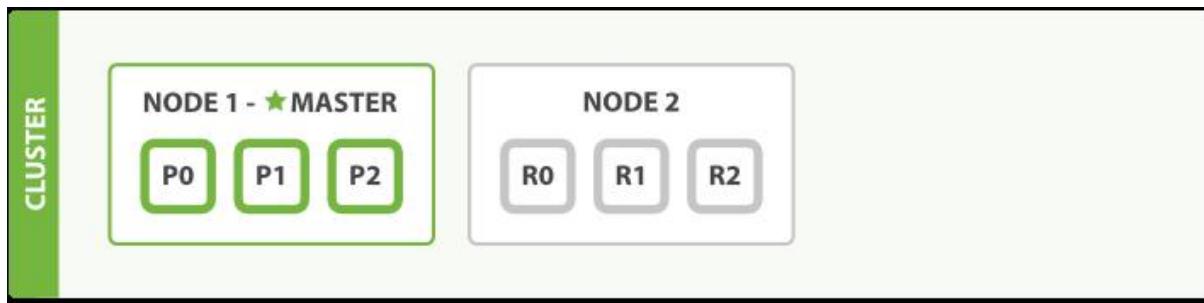


Fig: A two-node cluster—all primary and replica shards are allocated

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",  
  "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
}

```

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 [below](#).

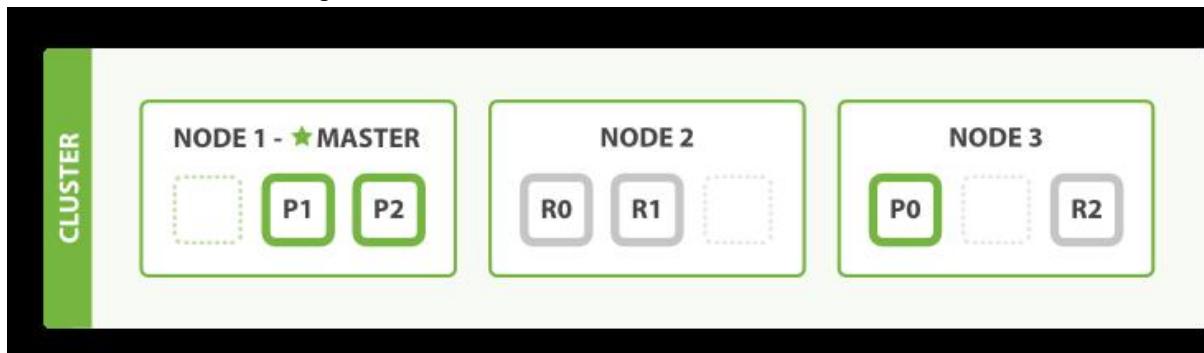


Fig: A three-node cluster—shards have been reallocated to spread the load

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 /blogs1/_settings
{
  "number_of_replicas" : 2
}
```

As can be seen in [the below](#), 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.

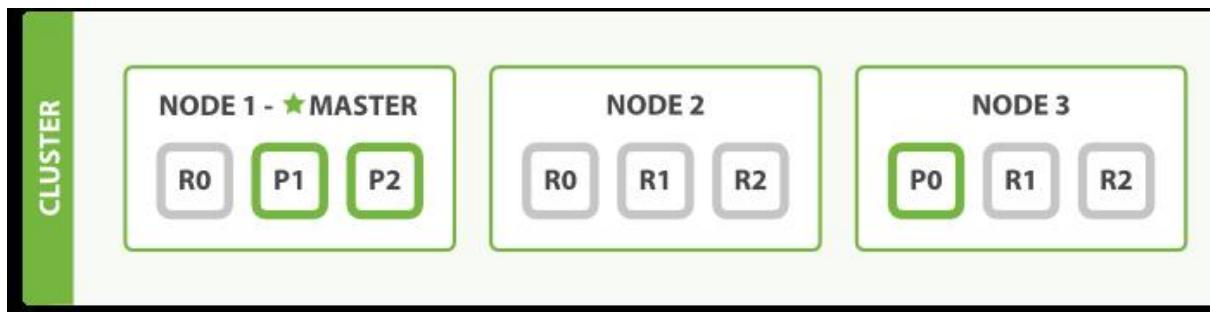


Fig: Increasing the *number_of_replicas* to 2

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 [below](#).

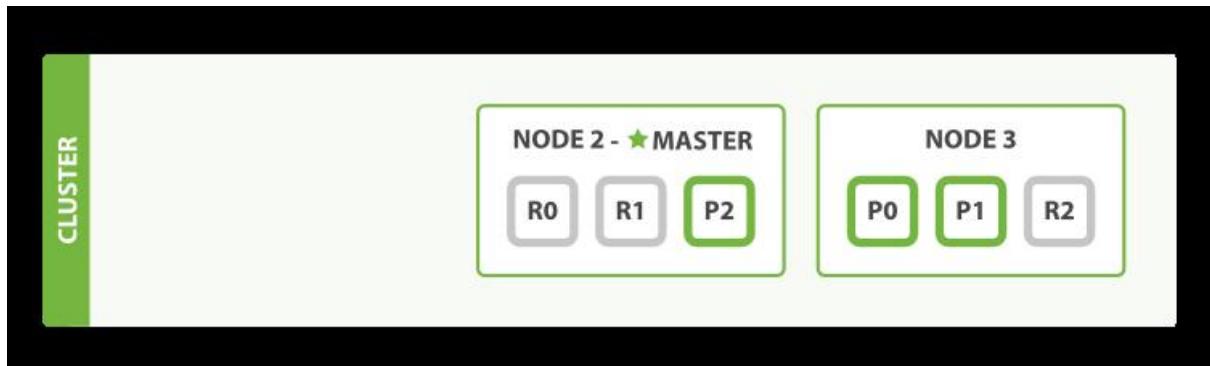


Fig: Cluster after killing one node

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.

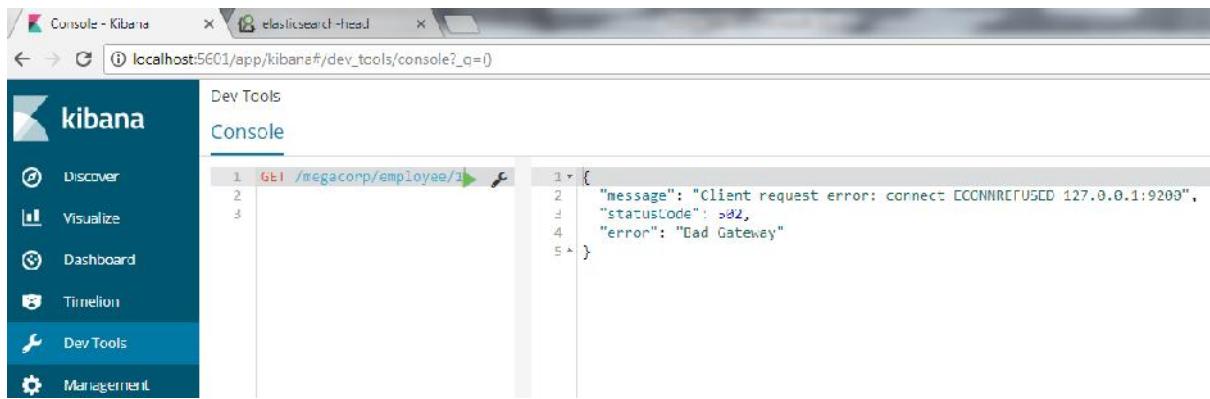
Primary shards 1 and 2 were lost when we killed Node 1, and our index cannot function properly if it is missing primary shards. If we had checked the cluster health at this point, we would have seen status red: not all primary shards are active!

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 [below](#). 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.

Suppose, If we lose all data means all shards and full cluster is down, then we will see below response from Elastic search server.



The screenshot shows the Kibana Dev Tools Console interface. On the left is a sidebar with icons for Discover, Visualize, Dashboard, Timeline, Dev Tools (which is selected), and Management. The main area has tabs for Dev Tools and Console. In the Dev Tools tab, there is a code editor window containing the following JSON response:

```
1 GET /megacorp/employee/1
2
3
4 {
5   "message": "Client request error: connect ECONNREFUSED 127.0.0.1:9200",
6   "statuscode": 502,
7   "error": "Bad Gateway"
8 }
```

Data In, Data Out

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.

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:

Example:

```
{  
  "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"  
    }  
  ]  
}
```

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.

Note: 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.

_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.

_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.

Indexing a Document

Documents are indexed—stored and made 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 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:

```
PUT /{index}/{type}/{id}
```

```
{  
  "field": "value",  
  ...  
}
```

```
PUT /website/blog/123
```

```
{  
  "title": "My first blog entry",  
  "text": "Just trying this out...",
```

```
"date": "2014/01/01"  
}
```

Elasticsearch responds as follows:

```
{  
  "_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.

Autogenerating IDs

The URL now contains just the _index and the _type:

```
POST /website/blog/
```

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

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

for us:

```
{  
  "_index": "website",  
  "_type": "blog",  
  "_id": "wMOOSFhDQXGZAWDf0-drSA",  
  "_version": 1,  
  "created": true  
}
```

Autogenerated IDs are 22 character long, URL-safe, Base64-encoded string universally unique identifiers, or **UUIDs**.

Retrieving a Document

```
GET /website/blog/123?pretty
```

Response:

```
{  
  "_index": "website",  
  "_type": "blog",  
  "_id": "123",  
  "_score": 1.0, "text": "Still trying this out...",  
  "date": "2014/01/01", "title": "My second blog entry",  
  "_version": 1  
}
```

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

Adding pretty to the query-string parameters for any request, as in the preceding example, causes Elasticsearch to pretty-print the 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. 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 causes it to display the response headers:

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

The response now looks like this:

```
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

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

Response:

```
{
  "_index" : "website",
```

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

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

GET /website/blog/123/_source

Response:

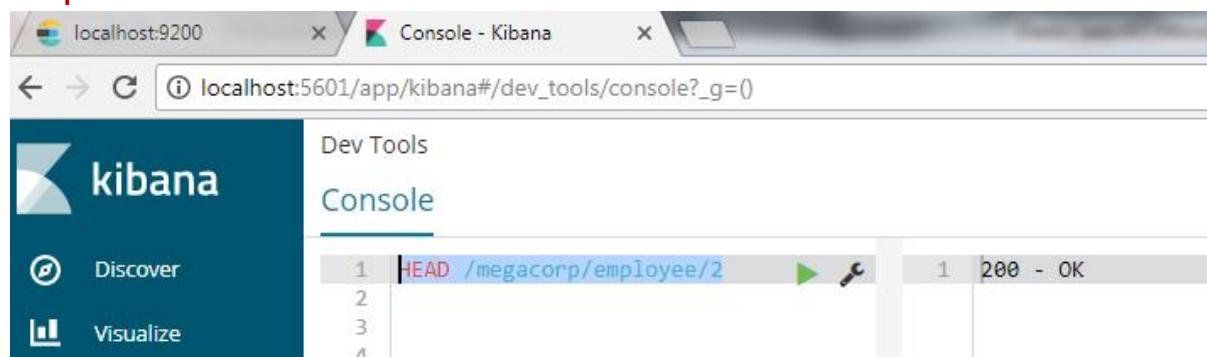
```
{
  "title": "My first blog entry",
  "text": "Just trying this out...",
  "date": "2014/01/01"
}
```

Checking Whether a Document Exists

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

Request: HEAD /megacorp/employee/2

Response:



The screenshot shows the Kibana Dev Tools Console interface. On the left, there's a sidebar with the Kibana logo and links for 'Discover' and 'Visualize'. The main area has tabs for 'Dev Tools' and 'Console'. In the 'Console' tab, there's a text input field containing the command 'HEAD /megacorp/employee/2'. Below the input field, there's a list of numbers (1, 2, 3, 4) and a status bar indicating '1 200 - OK'.

Request: HEAD /megacorp/employee/5

Response:

The screenshot shows the Kibana Dev Tools Console interface. On the left, there's a sidebar with the Kibana logo and links for Discover, Visualize, and Dev Tools. The main area is titled "Dev Tools" and "Console". A code editor window displays the following command:

```
1 HEAD /megacorp/employee/5
```

Below the code editor, the response status is shown as 1 404 - Not Found.

Request: GET /megacorp/employee/2

Response:

The screenshot shows the Kibana Dev Tools Console interface. The sidebar includes Discover, Visualize, Dashboard, Timelion, Dev Tools (which is selected), and Management. The main area shows a code editor with the following command:

```
1 GET /megacorp/employee/2
```

On the right, the response is displayed in a JSON format:

```
1 {  
2   "_index": "megacorp",  
3   "_type": "employee",  
4   "_id": "2",  
5   "_version": 1,  
6   "found": true,  
7   "_source": {  
8     "first_name": "Jane",  
9     "last_name": "Smith",  
10    "age": 32,  
11    "about": "I like to collect rock albums",  
12    "interests": [  
13      "music"  
14    ]  
15  }  
16 }
```

Updating a Whole Document

Documents in Elasticsearch are immutable; we cannot change them. Instead, if we need to update an existing document, we reindex or replace it, which we can do using the same index API.

The screenshot shows the Kibana Dev Tools Console interface. On the left is a sidebar with icons for Discover, Visualize, Dashboard, Timelion, Dev Tools (which is selected), and Management. The main area has tabs for Dev Tools and Console, with Console being active. In the Console tab, there is a code editor containing a PUT request:

```
1 PUT /megacorp/employee/2
2 {
3     "first_name": "Jane2",
4     "last_name": "Smith2",
5     "age": 32,
6     "about": "I like to collect rock
albums",
7     "interests": [
8         "music"
9     ]
10 }
```

To the right of the code editor is the response pane, which displays the JSON response from Elasticsearch:

```
1 {
2     "_index": "megacorp",
3     "_type": "employee",
4     "_id": "2",
5     "_version": 2,
6     "result": "updated",
7     "_shards": {
8         "total": 2,
9         "successful": 1,
10        "failed": 0
11    },
12    "_seq_no": 1,
13    "_primary_term": 4
14 }
```

Creating a New Document

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

PUT /megacorp/employee/5/_create

Or

PUT /megacorp/employee/5?op_type=create

Response:

The screenshot shows the Kibana Dev Tools Console interface. The sidebar and tabs are identical to the previous screenshot. The code editor contains a PUT request with op_type=create:

```
1 PUT /megacorp/employee/5?op_type=create
2 {
3     "first_name": "Jane2",
4     "last_name": "Smith2",
5     "age": 32,
6     "about": "I like to collect rock
albums",
7     "interests": [
8         "music"
9     ]
10 }
```

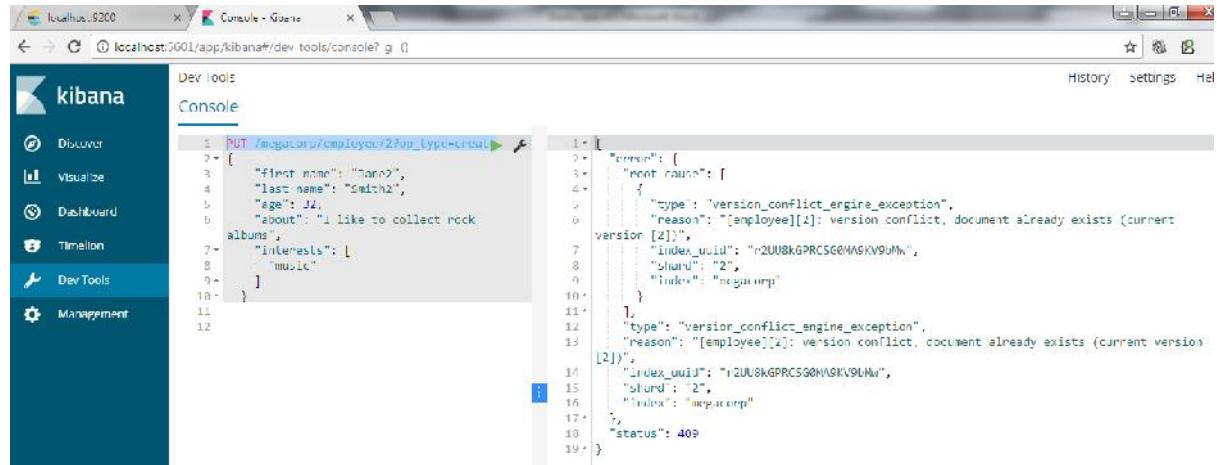
The response pane shows the successful creation of the document:

```
1 {
2     "_index": "megacorp",
3     "_type": "employee",
4     "_id": "5",
5     "_version": 1,
6     "result": "created",
7     "_shards": {
8         "total": 2,
9         "successful": 1,
10        "failed": 0
11    },
12    "_seq_no": 0,
13    "_primary_term": 4
14 }
```

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 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:

Response:



The screenshot shows the Kibana Dev Tools Console interface. On the left, there's a sidebar with options: Discover, Visualize, Dashboard, Timeline, DevTools (which is selected), and Management. The main area has tabs for Dev Tools and Console. In the Dev Tools tab, a command is being run: `PUT /employee/_id/2?op_type=create`. The request body is a JSON object with fields: first name, last name, age, about, albums, and interests. In the Console tab, the response is displayed as a multi-line JSON object. It starts with an "error" field containing a stack trace. The "type" field is "version_conflict_engine_exception". The "reason" field indicates a conflict because the document already exists (current version [2]). It also shows another entry for version [2] with the same details. Finally, the "status" field is set to 409.

```
1 PUT /employee/_id/2?op_type=create
2 {
3   "first_name": "Darth",
4   "last_name": "Smith2",
5   "age": 21,
6   "about": "I like to collect rock
7   "albums": [
8     "music"
9   ]
10 }
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
```

Dealing with Conflicts

When updating a document 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 [the below fig.](#)

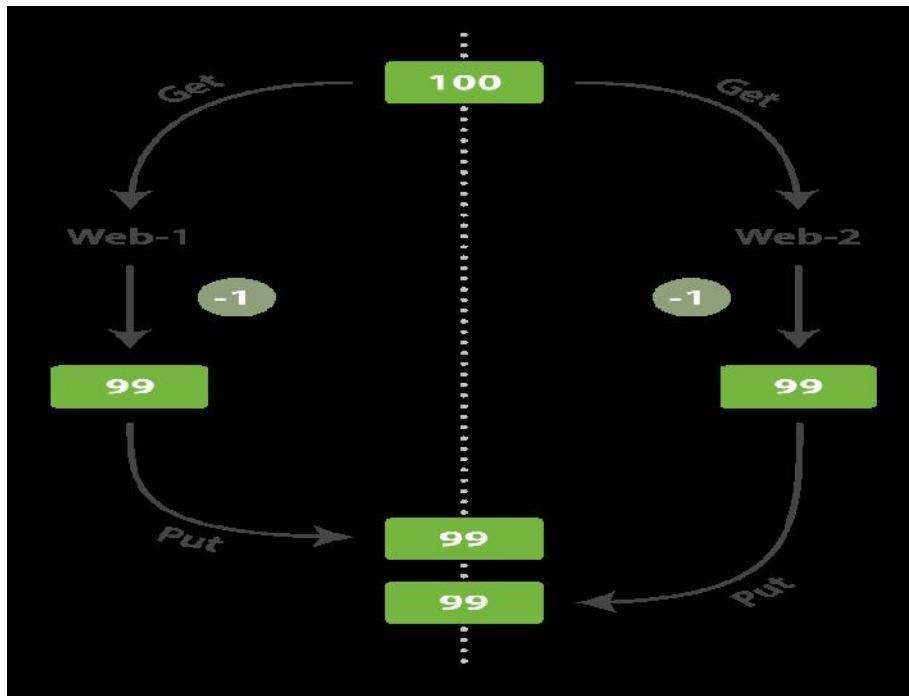


Fig: Consequence of no concurrency control

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 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, 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

Elasticsearch is distributed. When documents 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 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:

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

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:

```
GET /website/blog/1
```

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

```
{
  "_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:

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

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:

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

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:

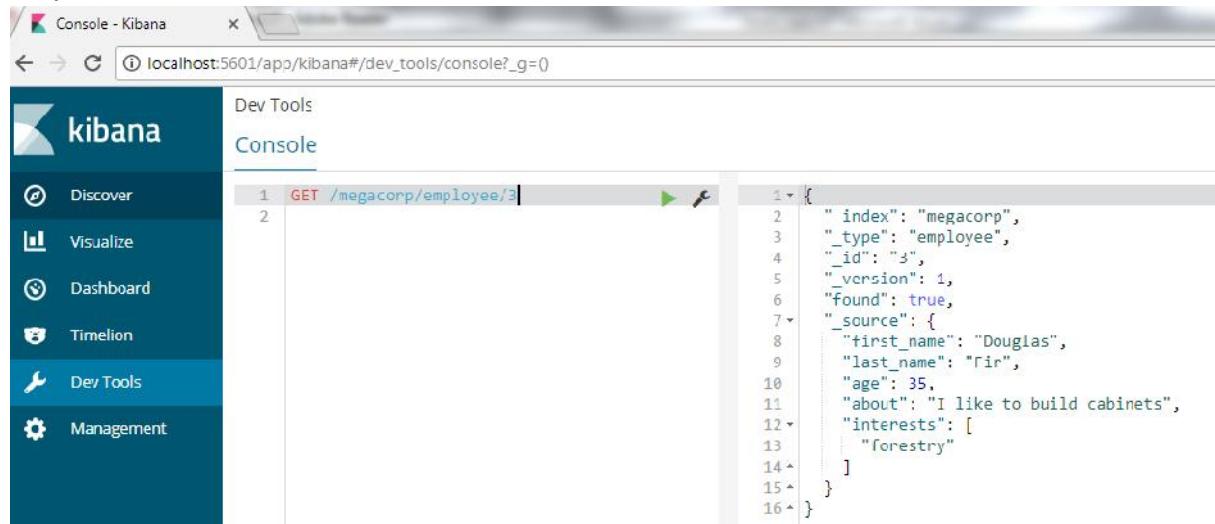
```
{  
"error" : "VersionConflictEngineException[[website][2] [blog][1]:  
version conflict, current [2], provided [1]]",  
"status" : 409  
}
```

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.

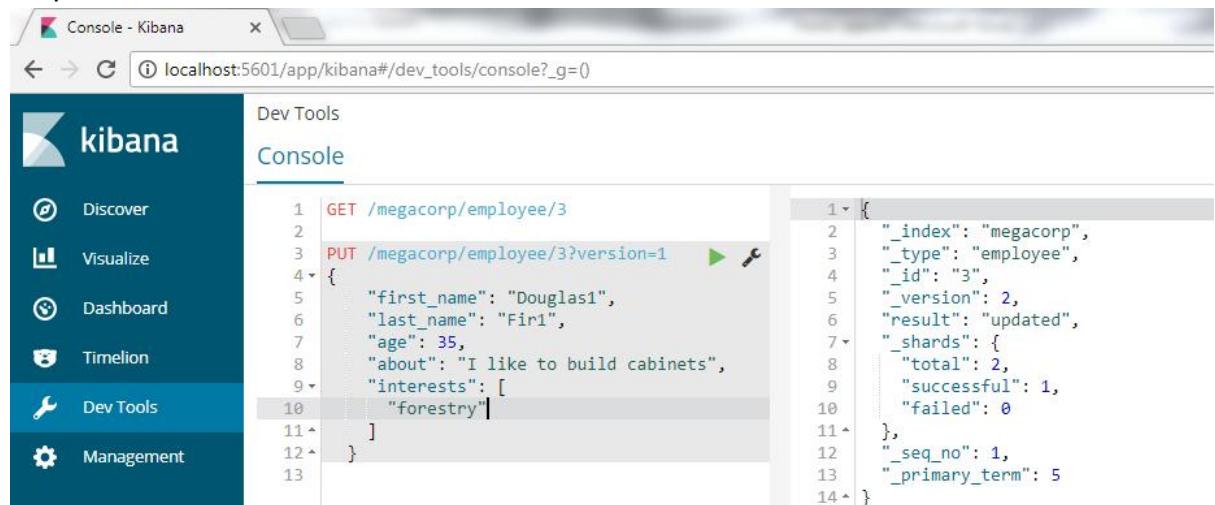
Step1:



A screenshot of the Kibana Dev Tools Console. The URL is `localhost:5601/app/kibana#/dev_tools/console?_g=()`. The left sidebar shows the Dev Tools tab is selected. In the main area, a GET request is shown with the path `/megacorp/employee/3`. The response is a JSON object representing an employee document:

```
1 {  
2   "index": "megacorp",  
3   "_type": "employee",  
4   "_id": "3",  
5   "_version": 1,  
6   "found": true,  
7   "_source": {  
8     "first_name": "Douglas",  
9     "last_name": "Fir",  
10    "age": 35,  
11    "about": "I like to build cabinets",  
12    "interests": [  
13      "forestry"  
14    ]  
15  }  
16}
```

Step2:



A screenshot of the Kibana Dev Tools Console. The URL is `localhost:5601/app/kibana#/dev_tools/console?_g=()`. The left sidebar shows the Dev Tools tab is selected. In the main area, a PUT request is shown with the path `/megacorp/employee/3?version=1`. The request body contains updated employee information:

```
1 GET /megacorp/employee/3  
2  
3 PUT /megacorp/employee/3?version=1  
4 {  
5   "first_name": "Douglas1",  
6   "last_name": "Fir1",  
7   "age": 35,  
8   "about": "I like to build cabinets",  
9   "interests": [  
10     "forestry"  
11   ]  
12 }  
13
```

The response is a JSON object indicating the update was successful:

```
1 {  
2   "_index": "megacorp",  
3   "_type": "employee",  
4   "_id": "3",  
5   "_version": 2,  
6   "result": "updated",  
7   "_shards": {  
8     "total": 2,  
9     "successful": 1,  
10    "failed": 0  
11  },  
12  "_seq_no": 1,  
13  "_primary_term": 5  
14 }
```

Step3:

The screenshot shows the Kibana Dev Tools Console interface. On the left, there's a sidebar with options: Discover, Visualize, Dashboard, Timeline, Dev Tools (which is selected), and Management. The main area has tabs for Dev Tools and Console. In the Dev Tools tab, there's a code editor with the following JSON:

```

1  PUT /megacorp/employee/3
2
3  {
4    "first_name": "Douglas",
5    "last_name": "Fir",
6    "age": 35,
7    "about": "I like to build websites",
8    "Interests": [
9      "forestry"
10     ]
11   }
12
13
14
15
16
17
18
19

```

When you run this, the response in the Console tab is:

```

1  {
2    "error": true,
3    "root_cause": [
4      {
5        "type": "version_conflict_engine_exception",
6        "reason": "[employee][3]: version conflict, current version [2] is different than the one provided [1]",
7        "index_uuid": "n2U8kGPRCSG0N9KV9LWw",
8        "shard": "4",
9        "index": "megacorp"
10      }
11    ],
12    "type": "version_conflict_engine_exception",
13    "reason": "[employee][3]: version conflict, current version [2] is different than the one provided [1]",
14    "index_uuid": "n2U8kGPRCSG0N9KV9LWw",
15    "shard": "4",
16    "index": "megacorp"
17  },
18  "status": 409
19

```

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, 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. 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:

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

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

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

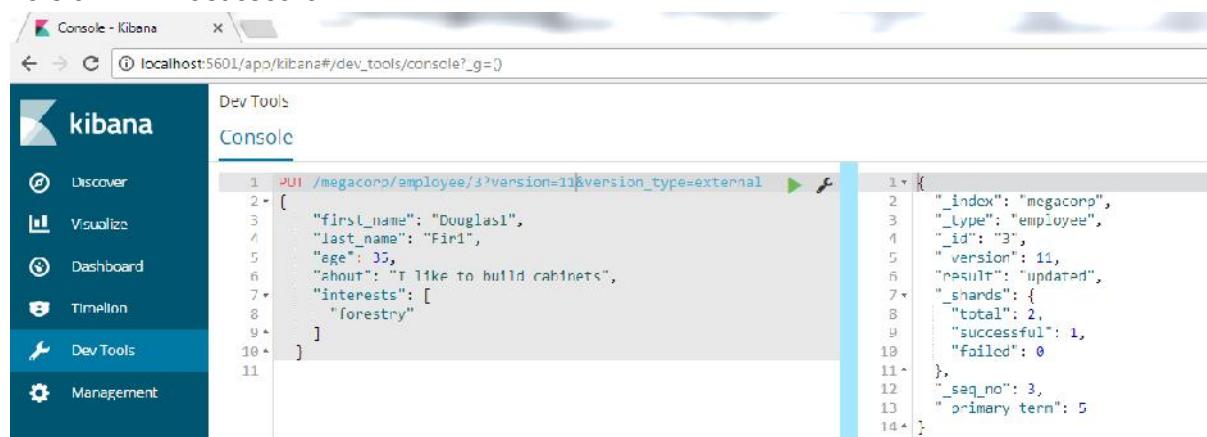
Now we update this document, specifying a new version number of 10:

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

The request succeeds and sets the current _version to 10:

```
{  
  "_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.



The screenshot shows the Kibana Dev Tools Console interface. The URL bar indicates the request is to localhost:5601/app/kibana#/dev_tools/console?_g=0. The console window displays a successful PUT request to the '/megacorp/employee/3' endpoint with a version of 11 and a version type of external. The response shows the document was updated, with its _index set to 'megacorp', _type to 'employee', _id to '3', _version to 11, and a result of 'updated'. The '_shards' object shows a total of 2 shards, 1 successful, and 0 failed. The '_seq_no' is 3, and the '_primary term' is 5. The document itself is shown with fields like first_name, last_name, age, about, and interests.

```
PUT /megacorp/employee/3?version=11&version_type=external  
{  
  "first_name": "Douglas1",  
  "last_name": "Fir1",  
  "age": 35,  
  "about": "I like to build cabinets",  
  "interests": [  
    "forestry"  
  ]  
}  
  
1 {  
  "_index": "megacorp",  
  "_type": "employee",  
  "_id": "3",  
  "_version": 11,  
  "result": "updated",  
  "_shards": {  
    "total": 2,  
    "successful": 1,  
    "failed": 0  
  },  
  "_seq_no": 3,  
  "_primary_term": 5  
}
```

Partial Updates to Documents

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:

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

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

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

Retrieving the document shows the updated _source field:

```
{
  "_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" ],  
"views": 0  
}  
}
```

Step1:

The screenshot shows the Kibana Dev Tools Console interface. On the left, the sidebar has 'Dev Tools' selected. The main area is titled 'Console'. A code editor window displays a POST request to '/megacorp/employee/3/_update?version=11'. The request body is a JSON object with a 'doc' field containing a modified name. To the right, the response is shown as a JSON object, indicating a successful update with version 12, a total of 2 documents, and one successful update.

```
POST /megacorp/employee/3/_update?version=11
{
  "doc": {
    "name": "updated name will come here"
  }
}

{
  "_index": "megacorp",
  "_type": "employee",
  "_id": "3",
  "_version": 12,
  "result": "updated",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 4,
  "_primary_term": 5
}
```

Step2:

The screenshot shows the Kibana Dev Tools Console interface. On the left, the sidebar includes links for Discover, Visualize, Dashboard, Timeline, Dev Tools (which is selected), and Management. The main area has tabs for Dev Tools and Console, with the Console tab active. In the Dev Tools section, a POST request is shown to update a document:

```
POST /megacorp/employee/3/_update?version=11
{
  "doc": {
    "name": "updated name will come here"
  }
}
```

Below this, a GET request is shown to retrieve the updated document:

```
GET /megacorp/employee/3
```

The response body shows the updated document with the new name:

```
{
  "_index": "megacorp",
  "_type": "employee",
  "_id": "3",
  "_version": 12,
  "_found": true,
  "_source": {
    "first_name": "Douglas",
    "last_name": "Hirn",
    "age": 35,
    "about": "I like to build cabinets",
    "interests": [
      "forestry"
    ],
    "name": "updated name will come here"
  }
}
```

“retry_on_conflict” parameter to the number of times that update should retry before failing; it defaults to 0.

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

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 [index API](#), the update API adopts a last-write-wins approach by default, but it also accepts a version parameter that allows you to use [optimistic concurrency control](#) to specify which version of the document you intend to update.

Deleting a Document

The syntax for deleting a document follows the same pattern that we have seen already, but uses the DELETE method :

```
DELETE /website/blog/123
```

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:

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

If the document isn't found, we get a 404 Not Found response code and a body like this:

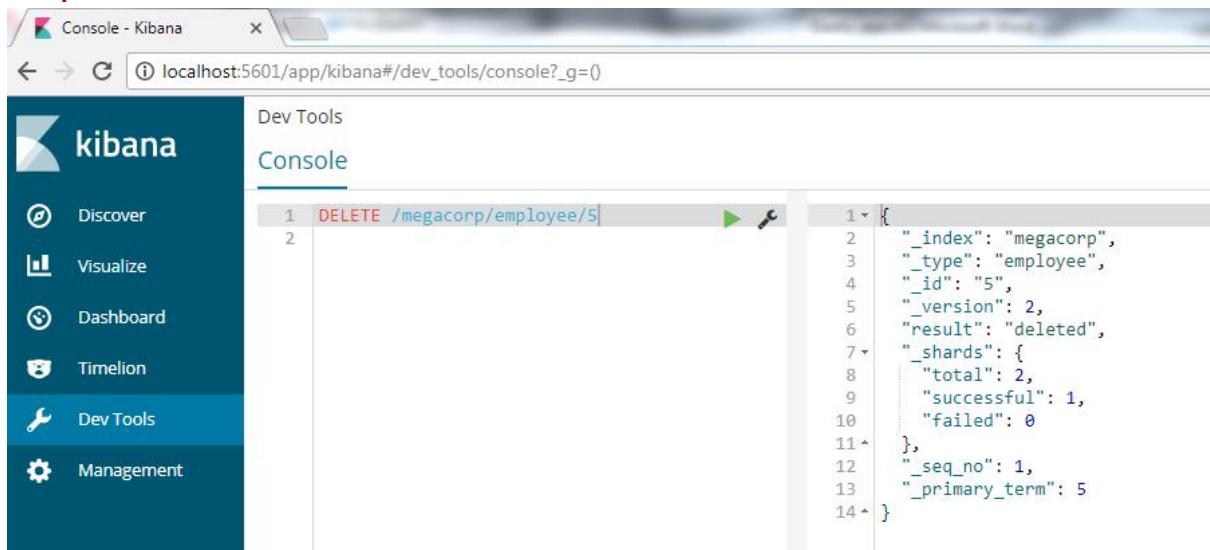
```
{
  "found" : false,
  "_index" : "website",
  "_type" : "blog",
  "_id" : "123",
  "_version" : 4
}
```

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.

Request

```
DELETE /megacorp/employee/5
```

Response:



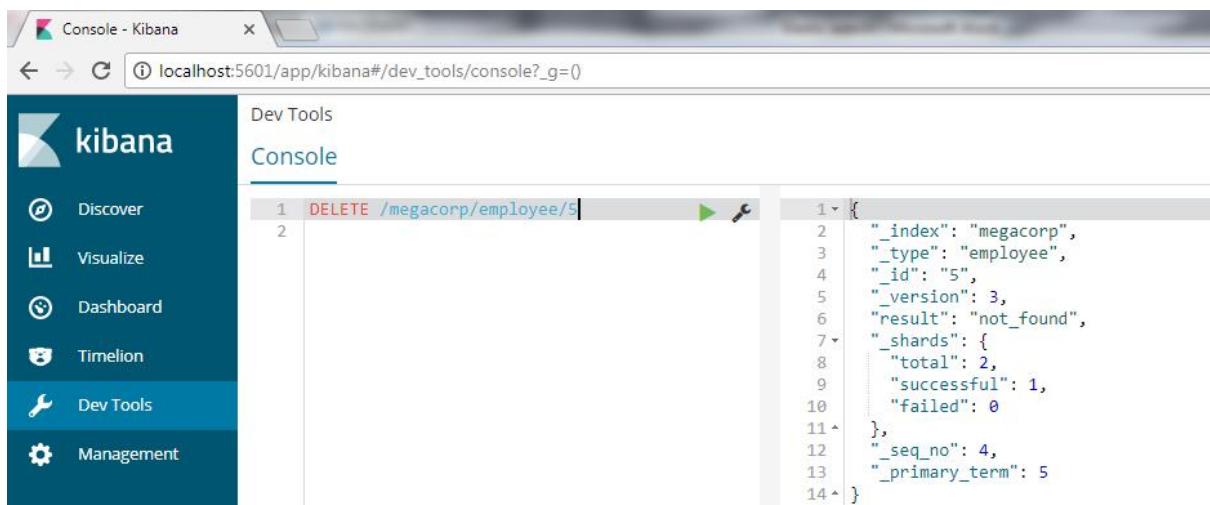
The screenshot shows the Kibana Dev Tools Console interface. On the left is a sidebar with icons for Discover, Visualize, Dashboard, Timelion, Dev Tools (which is selected), and Management. The main area is titled "Console". A command line window shows the following:

```
1 DELETE /megacorp/employee/5
```

On the right, the response is displayed in a JSON editor:

```
1 {  
2   "_index": "megacorp",  
3   "_type": "employee",  
4   "_id": "5",  
5   "_version": 2,  
6   "result": "deleted",  
7   "_shards": {  
8     "total": 2,  
9     "successful": 1,  
10    "failed": 0  
11  },  
12  "_seq_no": 1,  
13  "_primary_term": 5  
14 }
```

If not found:



The screenshot shows the Kibana Dev Tools Console interface. On the left is a sidebar with icons for Discover, Visualize, Dashboard, Timelion, Dev Tools (selected), and Management. The main area is titled "Console". A command line window shows the following:

```
1 DELETE /megacorp/employee/5
```

On the right, the response is displayed in a JSON editor:

```
1 {  
2   "_index": "megacorp",  
3   "_type": "employee",  
4   "_id": "5",  
5   "_version": 3,  
6   "result": "not_found",  
7   "_shards": {  
8     "total": 2,  
9     "successful": 1,  
10    "failed": 0  
11  },  
12  "_seq_no": 4,  
13  "_primary_term": 5  
14 }
```

Note: 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.

Retrieving Multiple Documents

As fast as Elasticsearch is, it can be faster still. 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, instead of document by document.

The mget API expects a docs array, each 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:

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

The response body also contains a docs array 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 `get` request:

```
{
  "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  
  }  
}  
}  
]  
}
```

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:

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

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:

```
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:

```
{  
  "docs" : [  
    {  
      "_index" : "website",  
      "_type" : "blog",  
      "_id" : "1",  
      "_score" : 0, "found" : false  
    }  
  ]  
}
```

```
"_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
}
]
```

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 the found flag.

```

1  GET /megacorp/employee/_mget
2  {
3    "ids": ["1", "3", "4"]
4  }
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44

```

```

1 { "docs": [
2   {
3     "_index": "megacorp",
4     "_type": "employee",
5     "_id": "1",
6     "_version": 1,
7     "found": true,
8     "_source": {}
9   },
10  {
11    "_index": "megacorp",
12    "_type": "employee",
13    "_id": "3",
14    "_version": 12,
15    "found": true,
16    "_source": {}
17  },
18  {
19    "_index": "megacorp",
20    "_type": "employee",
21    "_id": "4",
22    "_version": 1,
23    "found": false
24  }
25 ]
26 }
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44

```

Cheaper in Bulk

In the same way that mget allows us to retrieve multiple documents at once, the bulk API allows 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:

```

{ 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. 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.

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

The action must be one of the following:

Create : Create a document only if the document does not already exist.

Index : Create a new document or replace an existing document.

Update : Do a partial update on a document.

Delete: Delete a document.

The metadata should 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:

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

The request body line consists of the 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.

```
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title": "My first blog post" }
```

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

```
{ "index": { "_index": "website", "_type": "blog" }}
{ "title": "My second blog post" }
```

To put it all together, a complete bulk request has this form:

`POST /_bulk`

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

The Elasticsearch response contains the `items` array, which lists the result of each request, in the same order as we requested them:

```
{
"took": 4,
"errors": false,
```

```

"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
}}
]
}

```

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:

`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" }

```

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:

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

Step1:

```

POST / bulk
1 { "delete": { "_index": "website", "_type": "blog", "_id": "123" } }
2 { "create": { "_index": "website", "_type": "blog", "_id": "123" } }
3 { "title": "My first blog post" }
4 { "index": { "_index": "website", "_type": "blog" } }
5 { "title": "My second blog post" }
6 { "update": { "_index": "website", "_type": "blog", "_id": "123", "retry_on_conflict": 3 } }
7 { "doc": { "title": "The update blog post" } }
8

```

Response:

```

{ "took": 217,
  "errors": false,
  "items": [
    {
      "delete": {}
    },
    {
      "create": {}
    },
    {
      "index": {}
    },
    {
      "update": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 4,
        "result": "updated",
        "shards": {},
        "seq_no": 5,
        "primary_term": 1,
        "status": 200
      }
    }
  ]
}

```

Step2:

```

POST / bulk
1 { "create": { "_index": "website", "_type": "blog", "_id": "123" } }
2 { "title": "Cannot create it already exists" }
3 { "index": { "_index": "website", "_type": "blog", "_id": "123" } }
4 { "title": "But we can update it" }
5

```

Response:

```

{ "took": 105,
  "errors": true,
  "items": [
    {
      "create": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "status": 409,
        "error": {
          "type": "version_conflict_engine_exception",
          "reason": "[blog][123]: version conflict, document already exists"
        },
        "current_version": 6
      }
    },
    {
      "index": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 7,
        "result": "updated",
        "shards": {},
        "seq_no": 6,
        "primary_term": 1,
        "status": 200
      }
    }
  ]
}

```

Distributed Document Store

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

Routing a Document to a Shard

When you index a document, it is stored on a single primary 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:

Routing = id of document

shard = hash(routing) % number_of_primary_shards

`hash("1") % 3=0 (shard 0)`

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 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.

All document APIs (`get`, `index`, `delete`, `bulk`, `update`, and `mget`) accept a 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.

How Primary and Replica Shards Interact?

For explanation purposes, let's 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 [below](#).



Fig: A cluster with three nodes and one index

We can send our requests to any node in the cluster. 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.

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

Creating, Indexing, and Deleting a Document

Create, index, and delete requests are write operations, which must be successfully completed on the primary shard before they can be copied to any associated replica shards, as shown in [the](#) below fig.

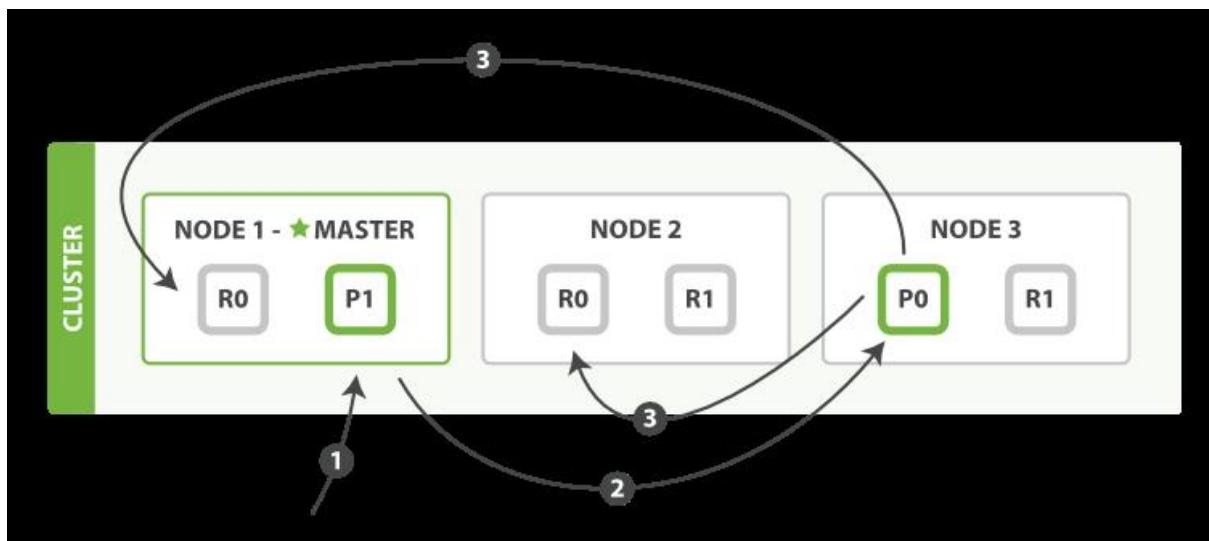


Fig: Creating, indexing, or deleting a single document

Here is the sequence 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 is sync. This causes the primary shard to wait for successful responses from the replica shards before returning. If you set replication to async, 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 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 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.

Retrieving a Document

A document can be retrieved from a primary shard or from any of its replicas, as shown in the below fig.

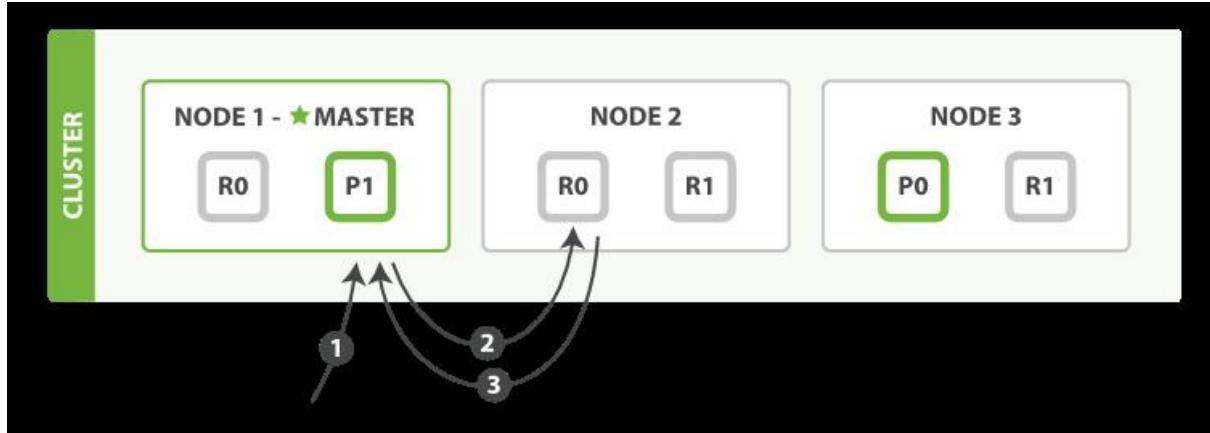


Fig: Retrieving a single document

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-robs 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 below fig, combines the read and write patterns explained previously.

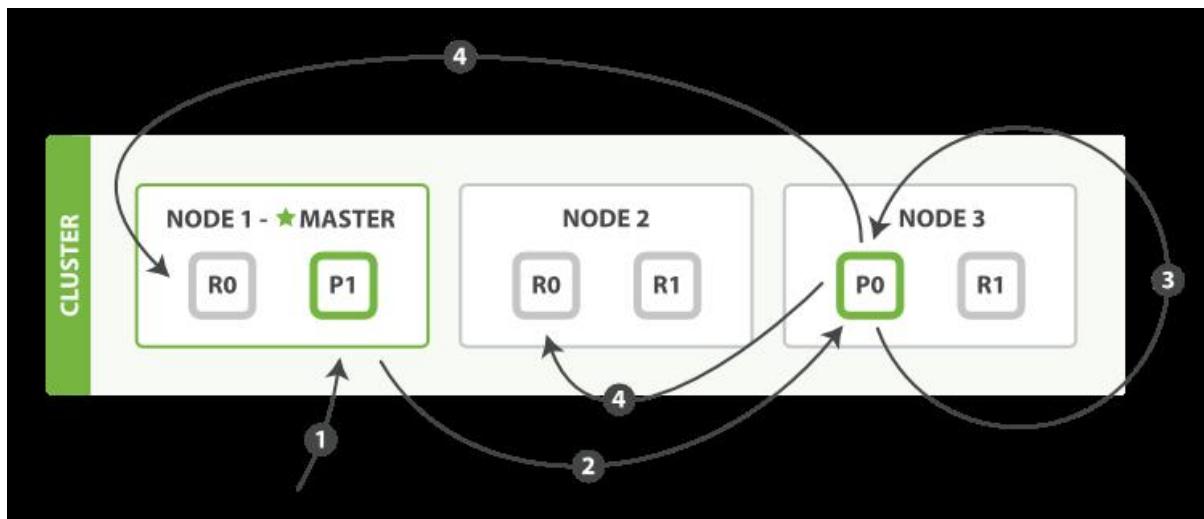


Fig: Partial updates to a document

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.

Note:

When a primary shard forwards changes to its 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.

Multidocument Patterns

The patterns for the mget and bulk APIs 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 the below fig.

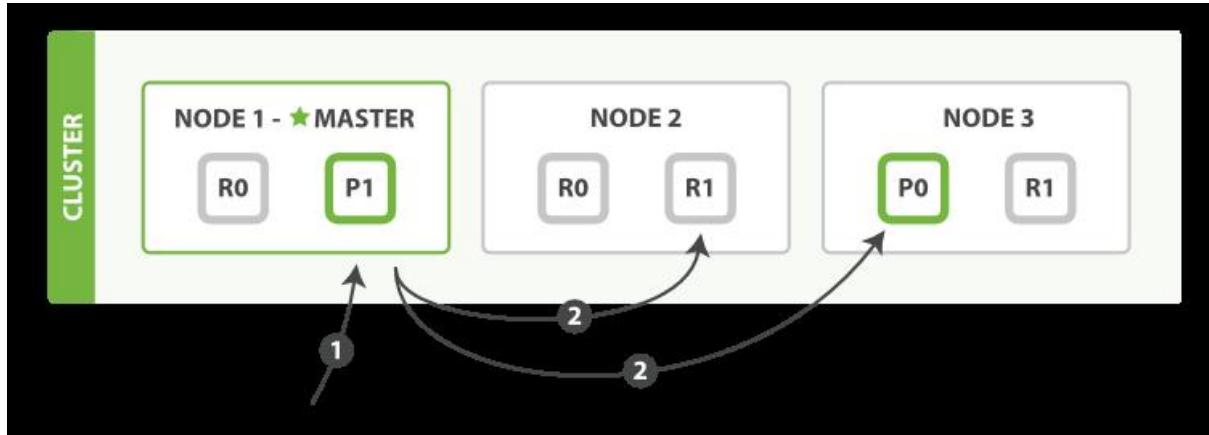


Fig: Retrieving multiple documents with *mget*

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 be set for each document in the docs array. The bulk API, as depicted in [the below fig](#), allows the execution of multiple create, index, delete, and update requests within a single bulk request.

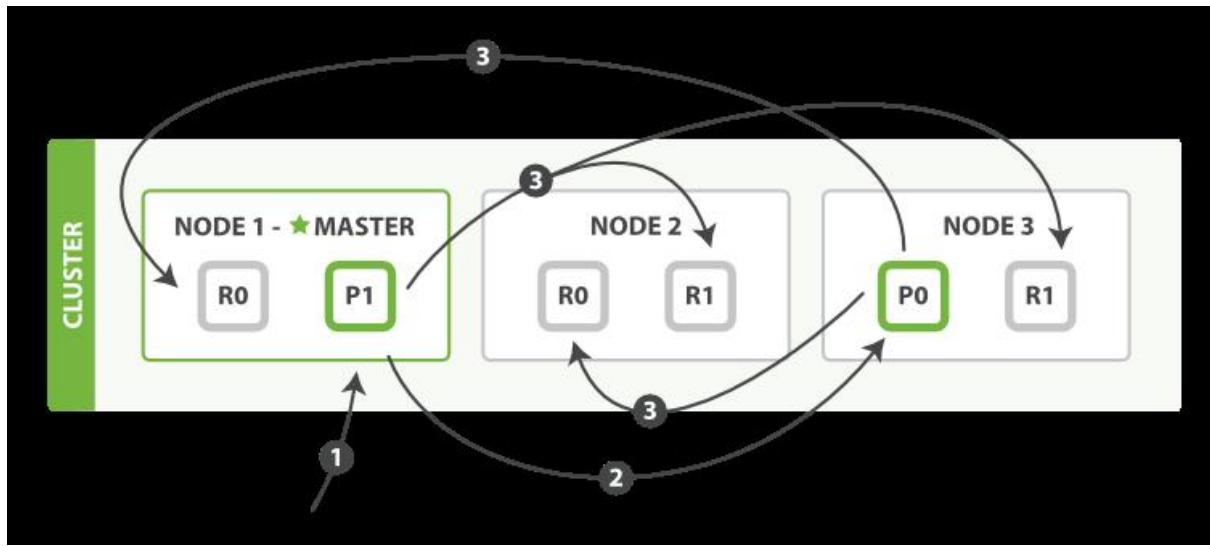


Fig: Multiple document changes with *bulk*

The sequence of steps 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 the replication and consistency parameters at the top level for the whole bulk request, and the routing parameter in the metadata for each request.

Why the Funny Format?

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

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

“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 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.

Searching—The Basic Tools

So far, we have learned how to use Elasticsearch as a simple NoSQL-style distributed document store. We can 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. And it's not just that. During a single query, Elasticsearch can use all of these indices, to return results at breathtaking 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 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 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

The Empty Search

`GET /_search`

The most basic form of the search API is the empty search, which doesn't specify any query but simply returns all documents in all indices in the cluster.

Hits – The most important section of the response is hits, which contains the total number of documents that matched our query, and a hits array containing the first 10 of those matching documents—the results.

Took – The took value tells us how many milliseconds the entire search request took to execute.

Shards – The _shards element tells us the total number of shards that were involved in the query and, of them, how many were successful and how many failed.

Timeout – The timed_out value tells us whether the query timed out. By default, search requests do not time out. 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):

`GET /_search?timeout=10ms`

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

Note: It should be noted that this timeout does not 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, Multitype

`/_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.

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

Pagination:

In the same way as SQL uses the LIMIT keyword to return a single “page” of results, Elasticsearch accepts 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:

`GET /_search?size=5`

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

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

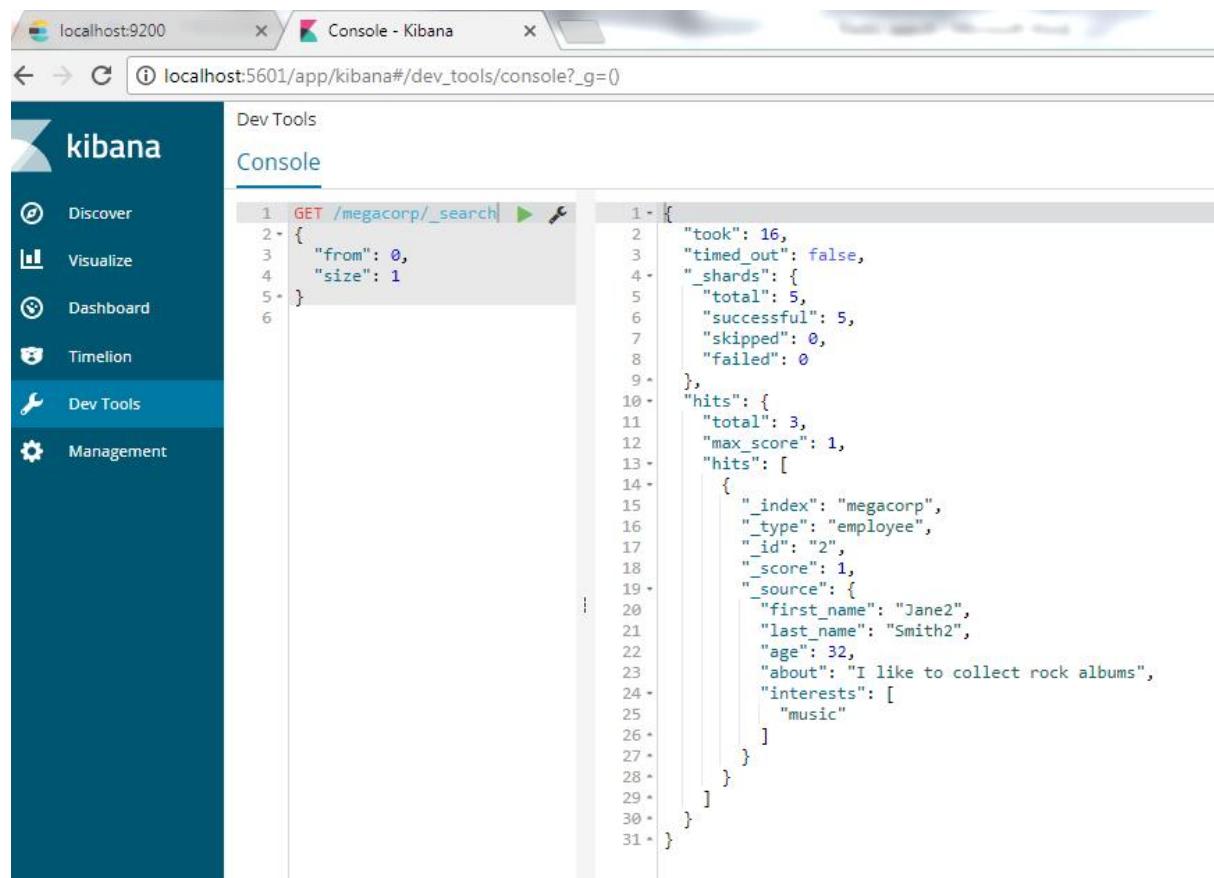
Beware of paging too deep or requesting too many results at once. Results are sorted before being returned. But remember that a search request usually spans

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 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.



The screenshot shows the Kibana interface with the Dev Tools tab selected. In the left sidebar, the Dev Tools option is highlighted. The main area is titled "Console" and contains a code editor and a results panel. The code editor shows a GET request to "/megacorp/_search" with parameters: "from": 0, "size": 1. The results panel displays the JSON response from Elasticsearch:

```
1 - {  
2 -   "took": 16,  
3 -   "timed_out": false,  
4 -   "_shards": {  
5 -     "total": 5,  
6 -     "successful": 5,  
7 -     "skipped": 0,  
8 -     "failed": 0  
9 -   },  
10 -  "hits": {  
11 -    "total": 3,  
12 -    "max_score": 1,  
13 -    "hits": [  
14 -      {  
15 -        "_index": "megacorp",  
16 -        "_type": "employee",  
17 -        "_id": "2",  
18 -        "_score": 1,  
19 -        "_source": {  
20 -          "first_name": "Jane2",  
21 -          "last_name": "Smith2",  
22 -          "age": 32,  
23 -          "about": "I like to collect rock albums",  
24 -          "interests": [  
25 -            "music"  
26 -          ]  
27 -        }  
28 -      }  
29 -    ]  
30 -  }  
31 - }
```

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:

```
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 !
```

Why does querying the `_all` field 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 the mapping (or schema definition) for the tweet type in the `gb` index:

```
GET /gb/_mapping/tweet
```

This gives us the following:

```
{
  "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. The _all field isn't mentioned because it is a default field, but we know that the _all field is of type string. So fields of type date and fields of type string are indexed differently, and can thus be searched differently. That's not entirely surprising. You might expect that each of the 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 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. 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 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:

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:

Elasticsearch uses a structure called 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 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 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 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.

Analysis and Analyzers

Analysis is a process that consists of the following:

1. First, tokenizing a block of text into individual terms suitable for use in an inverted index.
2. Then normalizing these terms into a standard form to improve their “searchability,” or recall. This job is performed by analyzers.

An **analyzer** is really just a wrapper that combines three functions into a 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, tokenizers, and token filters out of the box. These can be combined to create custom analyzers suitable for different purposes.

Built-in Analyzers

However, Elasticsearch also ships with prepackaged analyzers that you can use directly. 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 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 defined by the [Unicode Consortium](#), and removes most punctuation. Finally, it lowerscases all terms. It would produce
set, the, shape, to, semi, transparent, by, calling, set_trans, 5

Simple analyzer

The simple analyzer splits the text on anything that isn't a letter, and lowerscases the terms. It would produce

set, the, shape, to, semi, transparent, by, calling, set, trans

Whitespace analyzer

The whitespace analyzer splits 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 are available for [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 (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 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. 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 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, but instead search for the exact value that you have specified.

Now you can understand why the queries that we demonstrated at the [start of this chapter](#) 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:

```
GET /_search?q=2014 # 12 results
```

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:

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

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

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

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

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

Testing Analyzers

Especially when you are new 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:

```
GET /_analyze?analyzer=standard
```

Text to analyze

Each element in the result represents a single term:

```
{
  "tokens": [
    {
      "token": "text",
      "start_offset": 0,
      "end_offset": 4,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "to",
      "start_offset": 5,
      "end_offset": 7,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "analyze",
      "start_offset": 8,
      "end_offset": 15,
      "type": "<ALPHANUM>",
      "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.

The type values like <ALPHANUM> vary per analyzer and can be ignored. The only place that they are used in Elasticsearch is in the [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 in your documents, it automatically configures it as a full-text string field and analyzes it with the 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

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. This information is contained in the mapping.

Core Simple Field Types

Elasticsearch supports the following simple field types:

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

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 for one or more types in one or more indices by using the `/_mapping` endpoint. At the [start of this chapter](#), we already retrieved the mapping for type `tweet` in index `gb`:

```
GET /gb/_mapping/tweet
```

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

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

Note: Incorrect mappings, such as 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!

Customizing Field Mappings

While the basic field datatypes are sufficient for many cases, you will often need to customize the mapping 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:

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

index

The index attribute controls 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:

```
{
  "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 the analyzer attribute to specify which analyzer to apply both at search time and at index time. By default, Elasticsearch uses the standard analyzer, but you can change this by specifying one of the built-in analyzers, such as whitespace, simple, or english:

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

Updating a Mapping

You can specify the mapping for a type when you first 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:

```
DELETE /gb
```

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

```
PUT /gb  
{  
  "mappings": {  
    "tweet" : {  
      "properties" : {  
        "tweet" : {  
          "type" : "string",  
          "analyzer": "english"  
        },  
        "date" : {  
          "type" : "date"  
        },  
        "name" : {  
          "type" : "string"  
        },  
        "user_id" : {  
          "type" : "string"  
        }  
      }  
    }  
  }  
}
```

```
"type" : "long"
}
}
}
}
}
```

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

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

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 test the mapping for string fields by name. Compare the output of these two requests:

```
GET /gb/_analyze?field=tweet
Black-cats
GET /gb/_analyze?field=tag
Black-cats
```

Note: 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 Field Types

Besides the simple scalar datatypes that we have mentioned, JSON also has null values, arrays, and objects, all of which are supported by Elasticsearch.

It is quite possible that we want our tag field to contain more than one tag. Instead of a single string, we could index an array of tags:

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

There is no special mapping required for 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. 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.

Empty Fields

Arrays can, of course, be 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.

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

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

Multilevel Objects

The last native JSON datatype that we need to discuss is the object — known in other languages as a hash, hashmap, dictionary or associative array.

Inner objects are often used 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:

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

```
    "last": "Smith"
  }
}
}
```

Mapping for Inner Objects

Elasticsearch will detect new object fields dynamically and map them as type object, with each inner field listed under properties:

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

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. 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. 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:

```
{  
  "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 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.

Arrays of Inner Objects

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

```
{  
  "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:

```
{  
  "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?”

Nested Objects

Given the fact that creating, deleting, and updating a single document in Elasticsearch is atomic, it makes sense to store closely related entities within the same document.

For instance, we could store an order and all of its order lines in one document, or we could store a blog post and all of its comments together, by passing an array of comments:

```
PUT /my_index/blogpost/1
{
  "title": "Nest eggs",
  "body": "Making your money work...",
  "tags": [ "cash", "shares" ],
  "comments": [
    {
      "name": "John Smith",
      "comment": "Great article",
      "age": 28,
      "stars": 4,
      "date": "2014-09-01"
    },
    {
      "name": "Alice White",
      "comment": "More like this please",
      "age": 31,
      "stars": 5,
      "date": "2014-10-22"
    }
  ]
}
```

Note: If we rely on [dynamic mapping](#), the comments field will be autocreated as an object field.

Because all of the content is in the same document, there is no need to join blog posts and comments at query time, so searches perform well.

The problem is that the preceding document would match a query like this:

```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "name": "Alice" } },
        { "match": { "age": 28 } }
      ]
    }
  }
}
```

Alice is 31, not 28!

The reason for this cross-object matching, is that our beautifully structured JSON document is flattened into a simple key-value format in the index that looks like this:

```
{
  "title": [ eggs, nest ],
  "body": [ making, money, work, your ],
  "tags": [ cash, shares ],
  "comments.name": [ alice, john, smith, white ],
  "comments.comment": [ article, great, like, more, please, this ],
  "comments.age": [ 28, 31 ],
  "comments.stars": [ 4, 5 ],
  "comments.date": [ 2014-09-01, 2014-10-22 ]
}
```

The correlation between Alice and 31, or between John and 2014-09-01, has been irretrievably lost. While fields of type object are useful for storing a single object, they are useless, from a search point of view, for storing an array of objects.

This is the problem that nested objects are designed to solve. By mapping the comments field as type nested instead of type object, each nested object is indexed as a hidden separate document, something like this:

```
{
  "comments.name": [ john, smith ],
  "comments.comment": [ article, great ],
  "comments.age": [ 28 ],
  "comments.stars": [ 4 ],
  "comments.date": [ 2014-09-01 ]
}
```

```
{
  "comments.name": [ "alice", "white" ],
  "comments.comment": [ "like", "more", "please", "this" ],
  "comments.age": [ 31 ],
  "comments.stars": [ 5 ],
  "comments.date": [ "2014-10-22" ]
}
{
  "title": [ "eggs", "nest" ],
  "body": [ "making", "money", "work", "your" ],
  "tags": [ "cash", "shares" ]
}
```

By indexing each nested object separately, the fields within the object maintain their relationships. We can run queries that will match only if the match occurs within the same nested object.

Not only that, because of the way that nested objects are indexed, joining the nested documents to the root document at query time is fast—almost as fast as if they were a single document.

These extra nested documents are hidden; we can't access them directly. To update, add, or remove a nested object, we have to reindex the whole document. It's important to note that, the result returned by a search request is not the nested object alone; it is the whole document.

Nested Object Mapping

Setting up a nested field is simple—where you would normally specify type object, make it type nested instead:

```
PUT /my_index
{
  "mappings": {
    "blogpost": {
      "properties": {
        "comments": {
          "type": "nested",
          "properties": {
            "name": { "type": "string" },
            "comment": { "type": "string" },
            "age": { "type": "short" },
            "stars": { "type": "short" },
            "date": { "type": "date" }
          }
        }
      }
    }
  }
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Note: A nested field accepts the same parameters as a field of type object

That's all that is required. Any comments objects would now be indexed as separate nested documents. See the [nested type reference docs](#) for more.

Querying a Nested Object

Because nested objects are indexed as separate hidden documents, we can't query them directly. Instead, we have to use the [nested query](#) or [nested filter](#) to access them:

```
GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "eggs" }},
        {
          "nested": {
            "path": "comments",
            "query": {
              "bool": {
                "must": [
                  { "match": { "comments.name": "john" }},
                  { "match": { "comments.age": 28 }}
                ]
              }
            }}}
        ]
      }
    }
}
```

Note:

- The title clause operates on the root document.
- The nested clause “steps down” into the nested comments field. It no longer has access to fields in the root document, nor fields in any other nested document.
- The comments.name and comments.age clauses operate on the same nested document.

A nested field can contain other nested fields. Similarly, a nested query can contain other nested queries. The nesting hierarchy is applied as you would expect.

Of course, a nested query could match several nested documents. Each matching nested document would have its own relevance score, but these multiple scores need to be reduced to a single score that can be applied to the root document. By default, it averages the scores of the matching nested documents. This can be controlled by setting the `score_mode` parameter to `avg`, `max`, `sum`, or even `none` (in which case the root document gets a constant score of 1.0).

```
GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "eggs" }},
        {
          "nested": {
            "path": "comments",
            "score_mode": "max",
            "query": {
              "bool": {
                "must": [
                  { "match": { "comments.name": "john" }},
                  { "match": { "comments.age": 28 }}
                ]
              }
            }
          }
        }
      ]
    }
  }
}
```

Give the root document the `_score` from the best-matching nested document.

Note: A nested filter behaves much like a nested query, except that it doesn't accept the `score_mode` parameter. It can be used only in filter context—such as inside a filtered query—and it behaves like any other filter: it includes or excludes, but it doesn't score. While the results of the nested filter itself are not cached, the usual caching rules apply to the filter inside the nested filter.

Sorting by Nested Fields

It is possible to sort by the value of a nested field, even though the value exists in a separate nested document. To make the result more interesting, we will add another record:

```

PUT /my_index/blogpost/2
{
  "title": "Investment secrets",
  "body": "What they don't tell you ...",
  "tags": [ "shares", "equities" ],
  "comments": [
    {
      "name": "Mary Brown",
      "comment": "Lies, lies, lies",
      "age": 42,
      "stars": 1,
      "date": "2014-10-18"
    },
    {
      "name": "John Smith",
      "comment": "You're making it up!",
      "age": 28,
      "stars": 2,
      "date": "2014-10-16"
    }
  ]
}

```

Imagine that we want to retrieve blog posts that received comments in October, ordered by the lowest number of stars that each blog post received. The search request would look like this:

```

GET /_search
{
  "query": {
    "nested": {
      "path": "comments",
      "filter": {
        "range": {
          "comments.date": {
            "gte": "2014-10-01",
            "lt": "2014-11-01"
          }
        }
      }
    }
  }
}

```

```
    }
  },
  "sort": {
    "comments.stars": {
      "order": "asc",
      "mode": "min",
      "nested_filter": {
        "range": {
          "comments.date": {
            "gte": "2014-10-01",
            "lt": "2014-11-01"
          }
        }
      }
    }
  }
}
```

- The nested query limits the results to blog posts that received a comment in October.
 - Results are sorted in ascending (asc) order by the lowest value (min) in the comment.stars field in any matching comments.
 - The nested_filter in the sort clause is the same as the nested query in the main query clause. The reason is explained next.

Why do we need to repeat the query conditions in the nested_filter? The reason is that sorting happens after the query has been executed. The query matches blog posts that received comments in October, but it returns blog post documents as the result.

If we didn't include the `nested_filter` clause, we would end up sorting based on any comments that the blog post has ever received, not just those received in October.

Nested Aggregations

In the same way as we need to use the special nested query to gain access to nested objects at search time, the dedicated nested aggregation allows us to aggregate fields in nested objects:

```
GET /my_index/blogpost/_search?search_type=count
{
  "aggs": {
```

```
"comments": {  
    "nested": {  
        "path": "comments"  
    },  
    "aggs": {  
        "by_month": {  
            "date_histogram": {  
                "field": "comments.date",  
                "interval": "month",  
                "format": "yyyy-MM"  
            },  
            "aggs": {  
                "avg_stars": {  
                    "avg": {  
                        "field": "comments.stars"  
                    }  
                }  
            }  
        }  
    }  
}
```

Note: The nested aggregation “steps down” into the nested comments object. Comments are bucketed into months based on the comments.date field. The average number of stars is calculated for each bucket.

The results show that aggregation has happened at the nested document level:

```
...  
"aggregations": {  
  "comments": {  
    "doc_count": 4,  
    "by_month": {  
      "buckets": [  
        {  
          "key_as_string": "2014-09",  
          "key": 1409529600000,  
          "doc_count": 1,  
          "avg_stars": {
```

```

"value": 4
}
},
{
"key_as_string": "2014-10",
"key": 1412121600000,
"doc_count": 3,
"avg_stars": {
"value": 2.6666666666666665
}
}
]
}
}
}
...

```

reverse_nested Aggregation

A nested aggregation can access only the fields within the nested document. It can't see fields in the root document or in a different nested document. However, we can step out of the nested scope back into the parent with a reverse_nested aggregation.

For instance, we can find out which tags our commenters are interested in, based on the age of the commenter. The comment.age is a nested field, while the tags are in the root document:

```

GET /my_index/blogpost/_search?search_type=count
{
  "aggs": {
    "comments": {
      "nested": {
        "path": "comments"
      },
      "aggs": {
        "age_group": {
          "histogram": {
            "field": "comments.age",
            "interval": 10
          },
          "aggs": {
            "blogposts": {
              "reverse_nested": {}
            }
          }
        }
      }
    }
  }
}
```

```
"tags": {  
  "terms": {  
    "field": "tags"  
  }  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}
```

Note: The nested agg steps down into the comments object. The histogram agg groups on the comments.age field, in buckets of 10 years. The reverse_nested agg steps back up to the root document. The terms agg counts popular terms per age group of the commenter.

The abbreviated results show us the following:

```
..  
  "aggregations": {  
    "comments": {  
      "doc_count": 4,  
      "age_group": {  
        "buckets": [  
          {  
            "key": 20,  
            "doc_count": 2,  
            "blogposts": {  
              "doc_count": 2,  
              "tags": {  
                "doc_count_error_upper_bound": 0,  
                "buckets": [  
                  { "key": "shares", "doc_count": 2 },  
                  { "key": "cash", "doc_count": 1 },  
                  { "key": "equities", "doc_count": 1 }  
                ]  
              }  
            }  
          },  
        ],  
      },  
    },  
  },  
  ...
```

Note: There are four comments. There are two comments by commenters between the ages of 20 and 30. Two blog posts are associated with those comments. The popular tags in those blog posts are shares, cash, and equities.

When to Use Nested Objects

Nested objects are useful when there is one main entity, like our blogpost, with a limited number of closely related but less important entities, such as comments. It is useful to be able to find blog posts based on the content of the comments, and the nested query and filter provide for fast query-time joins.

The disadvantages of the nested model are as follows:

- To add, change, or delete a nested document, the whole document must be reindexed. This becomes more costly the more nested documents there are.
- Search requests return the whole document, not just the matching nested documents. Although there are plans afoot to support returning the best-matching nested documents with the root document, this is not yet supported.

Sometimes you need a complete separation between the main document and its associated entities. This separation is provided by the parent-child relationship.

Parent-Child Relationship

The parent-child relationship is similar in nature to the **nested model**: both allow you to associate one entity with another. The difference is that, with nested objects, all entities live within the same document while, with parent-child, the parent and children are completely separate documents.

The parent-child functionality allows you to associate one document type with another, in a one-to-many relationship—one parent to many children. The advantages that parent-child has over **nested objects** are as follows:

- The parent document can be updated without reindexing the children.
- Child documents can be added, changed, or deleted without affecting either the parent or other children. This is especially useful when child documents are large in number and need to be added or changed frequently.
- Child documents can be returned as the results of a search request.

Elasticsearch maintains a map of which parents are associated with which children. It is thanks to this map that query-time joins are fast, but it does place a limitation on the parent-child relationship: the parent document and all of its children must live on the same shard.

Note: At the time of going to press, the parent-child ID map is held in memory as part of `fielddata`. There are plans afoot to change the default setting to use `doc values` by default instead.

Parent-Child Mapping

All that is needed in order to establish the parent-child relationship is to specify which document type should be the parent of a child type. This must be done at index creation time, or with the update-mapping API before the child type has been created.

As an example, let's say that we have a company that has branches in many cities. We would like to associate employees with the branch where they work. We need to be able to search for branches, individual employees, and employees who work for particular branches, so the nested model will not help. We could, of course, use `application-side-joins` or `data denormalization` here instead, but for demonstration purposes we will use parent-child.

All that we have to do is to tell Elasticsearch that the employee type has the branch document type as its `_parent`, which we can do when we create the index:

```
PUT /company
{
  "mappings": {
    "branch": {},
    "employee": {
      "_parent": {
        "type": "branch"
      }
    }
  }
}
```

Note: Documents of type employee are children of type branch.

Indexing Parents and Children

Indexing parent documents is no different from any other document. Parents don't need to know anything about their children:

POST /company/branch/_bulk

```
{ "index": { "_id": "london" } }
{ "name": "London Westminster", "city": "London", "country": "UK" }
{ "index": { "_id": "liverpool" } }
{ "name": "Liverpool Central", "city": "Liverpool", "country": "UK" }
{ "index": { "_id": "paris" } }
{ "name": "Champs Élysées", "city": "Paris", "country": "France" }
```

When indexing child documents, you must specify the ID of the associated parent document:

PUT /company/employee/1?parent=london

```
{ "name": "Alice Smith",
  "dob": "1970-10-24",
  "hobby": "hiking"
}
```

Note: This employee document is a child of the london branch.

This parent ID serves two purposes: it creates the link between the parent and the child, and it ensures that the child document is stored on the same shard as the parent.

In “[Routing a Document to a Shard](#)”, we explained how Elasticsearch uses a routing value, which defaults to the `_id` of the document, to decide which shard a document should belong to. The routing value is plugged into this simple formula: `shard = hash(routing) % number_of_primary_shards`. However, if a parent ID is specified, it is used as the routing value instead of the `_id`.

In other words, both the parent and the child use the same routing value—the `_id` of the parent—and so they are both stored on the same shard.

The parent ID needs to be specified on all single-document requests: when retrieving a child document with a GET request, or when indexing, updating, or deleting a child document. Unlike a search request, which is forwarded to all shards in an index, these single-document requests are forwarded only to the shard that holds the document—if the parent ID is not specified, the request will probably be forwarded to the wrong shard.

The parent ID should also be specified when using the bulk API:

POST /company/employee/_bulk

```
{ "index": { "_id": 2, "parent": "london" } }
{ "name": "Mark Thomas", "dob": "1982-05-16", "hobby": "diving" }
{ "index": { "_id": 3, "parent": "liverpool" } }
{ "name": "Barry Smith", "dob": "1979-04-01", "hobby": "hiking" }
{ "index": { "_id": 4, "parent": "paris" } }
{ "name": "Adrien Grand", "dob": "1987-05-11", "hobby": "horses" }
```

Note: If you want to change the parent value of a child document, it is not sufficient to just reindex or update the child document—the new parent document may be on a different shard. Instead, you must first delete the old child, and then index the new child.

Finding Parents by Their Children

The has_child query and filter can be used to find parent documents based on the contents of their children. For instance, we could find all branches that have employees born after 1980 with a query like this:

```
GET /company/branch/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "query": {
        "range": {
          "dob": {
            "gte": "1980-01-01"
          }
        }
      }
    }
  }
}
```

Like the [nested query](#), the has_child query could match several child documents, each with a different relevance score. How these scores are reduced to a single score for the parent document depends on the score_mode parameter. The default setting is none, which ignores the child scores and assigns a score of 1.0 to the parents, but it also accepts avg, min, max, and sum.

The following query will return both london and liverpool, but london will get a better score because Alice Smith is a better match than Barry Smith:

```
GET /company/branch/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "score_mode": "max"
    }
  }
}
```

Note: The default score_mode of none is significantly faster than the other modes because Elasticsearch doesn't need to calculate the score for each child document. Set it to avg, min, max, or sum only if you care about the score.

min_children and max_children

The has_child query and filter both accept the min_children and max_children parameters, which will return the parent document only if the number of matching children is within the specified range.

This query will match only branches that have at least two employees:

```
GET /company/branch/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "min_children": 2,
      "query": {
        "match_all": {}
      }
    }
  }
}
```

A branch must have at least two employees in order to match.

The performance of a has_child query or filter with the min_children or max_children parameters is much the same as a has_child query with scoring enabled.

has_child Filter:

The has_child filter works in the same way as the has_child query, except that it doesn't support the score_mode parameter. It can be used only in filter context—such as inside a filtered query—and behaves like any other filter: it includes or excludes, but doesn't score.

While the results of a has_child filter are not cached, the usual caching rules apply to the filter inside the has_child filter.

Finding Children by Their Parents

While a nested query can always return only the root document as a result, parent and child documents are independent and each can be queried independently. The has_child query allows us to return parents based on data in their children, and the has_parent query returns children based on data in their parents.

It looks very similar to the has_child query. This example returns employees who work in the UK:

```
GET /company/employee/_search
{
  "query": {
    "has_parent": {
      "type": "branch",
      "query": {
        "match": {
          "country": "UK"
        }
      }
    }
  }
}
```

Returns children who have parents of type branch

The has_parent query also supports the score_mode, but it accepts only two settings: none (the default) and score. Each child can have only one parent, so there is no need to reduce multiple scores into a single score for the child. The choice is simply between using the score (score) or not (none).

has_parent Filter:

The has_parent filter works in the same way as the has_parent query, except that it doesn't support the score_mode parameter. It can be used only in filter context—such as inside a filtered query—and behaves like any other filter: it includes or excludes, but doesn't score. While the results of a has_parent filter are not cached, the usual caching rules apply to the filter inside the has_parent filter.

Children Aggregation

Parent-child supports a children aggregation as a direct analog to the nested aggregation discussed in “[Nested Aggregations](#)”. A parent aggregation (the equivalent of reverse_nested) is not supported.

This example demonstrates how we could determine the favorite hobbies of our employees by country:

```
GET /company/branch/_search?search_type=count
{
  "aggs": {
    "country": {
      "terms": {
        "field": "country"
      },
      "aggs": {
        "employees": {
          "children": {
            "type": "employee"
          },
          "aggs": {
            "hobby": {
              "terms": {
                "field": "employee.hobby"
              }
            }
          }
        }
      }
    }
  }
}
```

- The country field in the branch documents.

- The children aggregation joins the parent documents with their associated children of type employee.
- The hobby field from the employee child documents.

Grandparents and Grandchildren

The parent-child relationship can extend across more than one generation—grandchildren can have grandparents—but it requires an extra step to ensure that documents from all generations are indexed on the same shard.

Let's change our previous example to make the country type a parent of the branch type:

```
PUT /company
```

```
{
  "mappings": {
    "country": {},
    "branch": {
      "_parent": {
        "type": "country"
      }
    },
    "employee": {
      "_parent": {
        "type": "branch"
      }
    }
  }
}
```

Note: branch is a child of country. employee is a child of branch.

Countries and branches have a simple parent-child relationship, so we use the same process as we used in “[Indexing Parents and Children](#)” :

```
POST /company/country/_bulk
{ "index": { "_id": "uk" } }
{ "name": "UK" }
{ "index": { "_id": "france" } }
{ "name": "France" }

POST /company/branch/_bulk
{ "index": { "_id": "london", "parent": "uk" } }
{ "name": "London Westminster" }
{ "index": { "_id": "liverpool", "parent": "uk" } }
```

```
{ "name": "Liverpool Central" }
{ "index": { "_id": "paris", "parent": "france" } }
{ "name": "Champs Élysées" }
```

The parent ID has ensured that each branch document is routed to the same shard as its parent country document. However, look what would happen if we were to use the same technique with the employee grandchildren:

```
PUT /company/employee/1?parent=london
```

```
{
  "name": "Alice Smith",
  "dob": "1970-10-24",
  "hobby": "hiking"
}
```

The shard routing of the employee document would be decided by the parent ID—london—but the london document was routed to a shard by its own parent ID—uk. It is very likely that the grandchild would end up on a different shard from its parent and grandparent, which would prevent the same-shard parent-child mapping from functioning. Instead, we need to add an extra routing parameter, set to the ID of the grandparent, to ensure that all three generations are indexed on the same shard. The indexing request should look like this:

```
PUT /company/employee/1?parent=london&routing=uk
```

```
{
  "name": "Alice Smith",
  "dob": "1970-10-24",
  "hobby": "hiking"
}
```

Note: The routing value overrides the parent value.

The parent parameter is still used to link the employee document with its parent, but the routing parameter ensures that it is stored on the same shard as its parent and grandparent. The routing value needs to be provided for all single-document requests.

Querying and aggregating across generations works, as long as you step through each generation. For instance, to find countries where employees enjoy hiking, we need to join countries with branches, and branches with employees:

```
GET /company/country/_search
```

```
{
  "query": {
    "has_child": {
      "type": "branch",
```

```
"query": {  
  "has_child": {  
    "type": "employee",  
    "query": {  
      "match": {  
        "hobby": "hiking"  
      }  
    }  
  }  
}  
}  
}  
}  
}  
}  
}  
}
```

Practical Considerations

Parent-child joins can be a useful technique for managing relationships when index time performance is more important than search-time performance, but it comes at a significant cost. Parent-child queries can be 5 to 10 times slower than the equivalent nested query!

Memory Use

At the time of going to press, the parent-child ID map is still held in memory. There are plans to change the map to use doc values instead, which will be a big memory saving. Until that happens, you need to be aware of the following: the string `_id` field of every parent document has to be held in memory, and every child document requires 8 bytes (a long value) of memory. Actually, it's a bit less thanks to compression, but this gives you a rough idea.

You can check how much memory is being used by the parent-child cache by consulting the `indices-stats` API (for a summary at the index level) or the `node-stats` API (for a summary at the node level):

`GET /_nodes/stats/indices/id_cache?human`

Returns memory use of the ID cache summarized by node in a human-friendly format.

Global Ordinals and Latency

Parent-child uses [global ordinals](#) to speed up joins. Regardless of whether the parentchild map uses an in-memory cache or on-disk doc values, global ordinals still need to be rebuilt after any change to the index.

The more parents in a shard, the longer global ordinals will take to build. Parentchild is best suited to situations where there are many children for each parent, rather than many parents and few children.

Global ordinals, by default, are built lazily: the first parent-child query or aggregation after a refresh will trigger building of global ordinals. This can introduce a significant latency spike for your users. You can use [eager_global_ordinals](#) to shift the cost of building global ordinals from query time to refresh time, by mapping the `_parent` field as follows:

```
PUT /company
{
  "mappings": {
    "branch": {},
    "employee": {
      "_parent": {
        "type": "branch",
        "fielddata": {
          "loading": "eager_global_ordinals"
        }
      }
    }
  }
}
```

Global ordinals for the `_parent` field will be built before a new segment becomes visible to search.

With many parents, global ordinals can take several seconds to build. In this case, it makes sense to increase the `refresh_interval` so that refreshes happen less often and global ordinals remain valid for longer. This will greatly reduce the CPU cost of rebuilding global ordinals every second.

Multigenerations and Concluding Thoughts

The ability to join multiple generations sounds attractive until you think of the costs involved:

- The more joins you have, the worse performance will be.

- Each generation of parents needs to have their string `_id` fields stored in memory, which can consume a lot of RAM.

As you consider your relationship schemes and whether parent-child is right for you, consider this advice about parent-child relationships:

- Use parent-child relationships sparingly, and only when there are many more children than parents.
- Avoid using multiple parent-child joins in a single query.
- Avoid scoring by using the `has_child` filter, or the `has_child` query with `score_mode` set to `none`.
- Keep the parent IDs short, so that they require less memory.

Above all: think about the other relationship techniques that we have discussed before reaching for parent-child.

Full-Body Search

Search lite—a [query-string search](#)—is useful for ad hoc queries from the command line. To harness the full power of search, however, you should use the request body search API, 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 the search API, the empty search, which returns all documents in all indices:

```
GET /_search
{}
```

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

```
GET /index_2014*/type1,type2/_search
{}
```

And you can use the `from` and `size` parameters for pagination:

```
GET /_search
{
  "from": 30,
```

```
"size": 10  
}
```

A GET Request with a Body?

The HTTP libraries of certain languages (notably JavaScript) don't allow GET requests to have a request body. In fact, some users are surprised that GET requests are ever allowed to have a body.

The truth is that [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 accepts POST requests:

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

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

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.

Query DSL

The query DSL is a flexible, expressive search 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 in the query parameter:

```
GET /_search  
{  
  "query": YOUR_QUERY_HERE  
}
```

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

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

```
}
```

The full search request would look like this:

```
GET /_search
{
  "query": {
    "match": {
      "tweet": "elasticsearch"
    }
  }
}
```

Combining Multiple Clauses

Query clauses are simple building blocks that can be combined with each other to create complex queries. Clauses can be as follows:

- Leaf clauses (like the match clause) that are used to compare a field (or fields) to a query string.
- Compound clauses that are used to combine other query clauses. For instance, a bool clause allows you to combine other clauses that either must match, must_not match, or should match if possible:

```
{
  "bool": {
    "must": { "match": { "tweet": "elasticsearch" }},
    "must_not": { "match": { "name": "mary" }},
    "should": { "match": { "tweet": "full text" } }
}
```

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:

```
{
  "bool": {
    "must": { "match": { "email": "business opportunity" }},
    "should": [
      { "match": { "starred": true }},
      { "bool": {
        "must": { "folder": "inbox" }
      }}
    ]
}
```

```
"must_not": { "spam": true } }  
}  
],  
"minimum_should_match": 1  
}  
}
```

Queries and Filters

Although we refer to the query DSL, in reality there are two DSLs: the query DSL and the filter DSL. Query clauses and filter clauses are similar in nature, but have slightly different purposes.

A filter asks a yes/no question of 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 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 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 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 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 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.

term Filter

The term filter is used to filter by exact values, be they numbers, dates, Booleans, or not_analyzed exact-value string fields:

```
{ "term": { "age": 26 } }
{ "term": { "date": "2014-09-01" } }
{ "term": { "public": true } }
{ "term": { "tag": "full_text" } }
```

terms Filter

The terms filter is 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:

```
{ "terms": { "tag": [ "search", "full_text", "nosql" ] } }
```

range Filter

The range filter allows you to find numbers or dates that fall into a specified range:

```
{
  "range": {
    "age": {
      "gte": 20,
      "lt": 30
    }
  }
}
```

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 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:

```
{  
  "exists": {  
    "field": "title"  
  }  
}
```

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 to combine multiple filter clauses using Boolean logic. 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:

```
{  
  "bool": {  
    "must": { "term": { "folder": "inbox" }},  
    "must_not": { "term": { "tag": "spam" }},  
    "should": [  
      { "term": { "starred": true }},  
      { "term": { "unread": true }}  
    ]  
  }  
}
```

match_all Query

The match_all query simply matches all documents. It is the default query that is used if no query has been specified:

```
{ "match_all": {}}
```

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 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:

```
{ "match": { "tweet": "About Search" } }
```

If you use it on a field containing an exact value, such as a number, a date, a Boolean, or a not_analyzed string field, then it will search for that exact value:

```
{ "match": { "age": 26 } }
{ "match": { "date": "2014-09-01" } }
{ "match": { "public": true } }
{ "match": { "tag": "full_text" } }
```

Note: For exact-value searches, you probably want to use a filter instead of a query, as a filter will be cached.

multi_match Query

The multi_match query allows to run the same match query on multiple fields:

```
{
  "multi_match": {
    "query": "full text search",
    "fields": [ "title", "body" ]
  }
}
```

bool Query

The bool query, like the bool filter, 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. 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:

```
{  
  "bool": {  
    "must": { "match": { "title": "how to make millions" }},  
    "must_not": { "match": { "tag": "spam" }},  
    "should": [  
      { "match": { "tag": "starred" }},  
      { "range": { "date": { "gte": "2014-01-01" }}}  
    ]  
  }  
}
```

Note: 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. 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.

Filtering a Query

Let's say we have this query:

```
{ "match": { "email": "business opportunity" }}
```

We want to combine it with the following term filter, which will match only documents that are in our inbox:

```
{ "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:

```
{
  "filtered": {
    "query": { "match": { "email": "business opportunity" } },
    "filter": { "term": { "folder": "inbox" } }
  }
}
```

We can now pass this query to the query parameter of the search API:

```
GET /_search
{
  "query": {
    "filtered": {
      "query": { "match": { "email": "business opportunity" } },
      "filter": { "term": { "folder": "inbox" } }
    }
  }
}
```

Just a Filter

While in query context, if you need to use a filter without a query (for instance, to match all emails in the inbox), you can just omit the query:

```
GET /_search
{
  "query": {
    "filtered": {
      "filter": { "term": { "folder": "inbox" } }
    }
  }
}
```

If a query is not specified it defaults to using the match_all query, so the preceding query is equivalent to the following:

```
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 just wraps a query. The following example shows one way we could exclude emails that look like spam:

```
GET /_search
{
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "must": { "term": { "folder": "inbox" } },
          "must_not": {
            "query": {
              "match": { "email": "urgent business proposal" }
            }
          }
        }
      }
    }
  }
}
```

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 when combined with different analyzers and field mappings, can become a bit difficult to follow. The validatequery API can be used to check whether a query is valid.

```
GET /gb/tweet/_validate/query
{
  "query": {
    "tweet" : {
      "match" : "really powerful"
    }
  }
}
```

```
}
```

The response to the preceding validate request tells us that the query is invalid:

```
{
  "valid" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  }
}
```

Understanding Errors

To find out why it is invalid, add the explain parameter to the query string:

```
GET /gb/tweet/_validate/query?explain
{
  "query": {
    "tweet" : {
      "match" : "really powerful"
    }
  }
}
```

Apparently, we've mixed up the type of query (match) with the name of the field (tweet):

```
{
  "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 humanreadable description of the (valid) query, which can be useful for understanding exactly how your query has been interpreted by Elasticsearch:

```
GET /_validate/query?explain
{
  "query": {
```

```
"match" : {  
  "tweet" : "really powerful"  
}  
}  
}  
}
```

An explanation is returned for each index that we query, because each index can have different mappings and analyzers:

```
{  
  "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.

Sorting and Relevance

By default, results are returned sorted by relevance—with the most relevant docs first.

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, 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:

```
GET /_search  
{  
  "query" : {  
    "filtered" : {
```

```
"filter" : {  
  "term" : {  
    "user_id" : 1  
  }  
}  
}  
}  
}  
}
```

Filters have no bearing on `_score`, and the 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. We can do this with the `sort` parameter:

```
GET /_search  
{  
  "query" : {  
    "filtered" : {  
      "filter" : { "term" : { "user_id" : 1 } }  
    }  
  },  
  "sort": { "date": { "order": "desc" } }  
}
```

You will notice two differences in the results:

```
"hits" : {  
  "total" : 6,  
  "max_score" : null,  
  "hits" : [ {  
    "_index" : "us",  
    "_type" : "tweet",  
    "_id" : "14",  
    "_score" : null,  
    "_source" : {  
      "date": "2014-09-24",  
      ...  
    },  
    "sort" : [ 1411516800000 ]  
  },
```

...

}

The `_score` is not calculated, because it is not being used for sorting.

The value of the date field, expressed as milliseconds since the epoch, is returned in the sort values.

Multilevel Sorting

Perhaps we want to combine the `_score` from a query with the date, and show all matching results sorted first by date, then by relevance:

GET /_search

```
{  
  "query" : {  
    "filtered" : {  
      "query": { "match": { "tweet": "manage text search" }},  
      "filter" : { "term" : { "user_id" : 2 }}  
    }  
  },  
  "sort": [  
    { "date": { "order": "desc" }},  
    { "_score": { "order": "desc" }}  
  ]  
}
```

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, on geo-distance or on a custom value calculated in a script.

Sorting on Multivalue Fields

When sorting on fields with more than one value, 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. For instance, you could sort on the earliest date in each `dates` field by using the following:

```
"sort": {  
  "dates": {  
    "order": "asc",  
    "mode": "min"
```

```
 }  
 }
```

String Sorting and Multifields

Analyzed string fields are also multivalue fields, 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. 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 that allows you to transform a simple mapping like

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

into a multifield mapping like this:

```
"tweet": {  
  "type": "string",  
  "analyzer": "english",  
  "fields": {  
    "raw": {  
      "type": "string",  
      "index": "not_analyzed"  
    }  
  }  
}
```

The main tweet field is just the same as before: an analyzed full-text field. 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:

```
GET /_search
{
  "query": {
    "match": {
      "tweet": "elasticsearch"
    }
  },
  "sort": "tweet.raw"
}
```

What Is Relevance?

The relevance score of each document is represented by a positive floating-point number called the _score. 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. 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 fulltext field are to a full-text query string.

The standard similarity algorithm used in Elasticsearch is known as term frequency/inverse document frequency, or TF/IDF, which takes the following factors into account:

```
GET /_search?explain
{
  "query" : { "match" : { "tweet" : "honeymoon" } }
}
```

The explain parameter adds an explanation of how the _score was calculated to every result.

The first part is the summary of the calculation. It tells us that it has calculated the weight—the 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 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.

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.

When you sort on a field, Elasticsearch needs access to the value of that field for every document that matches the query. 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`.

Note: 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.

`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

Distributed Search Execution

A CRUD operation deals with a single document that has a unique combination of `_index`, `_type`, and `routing values` (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
}
```

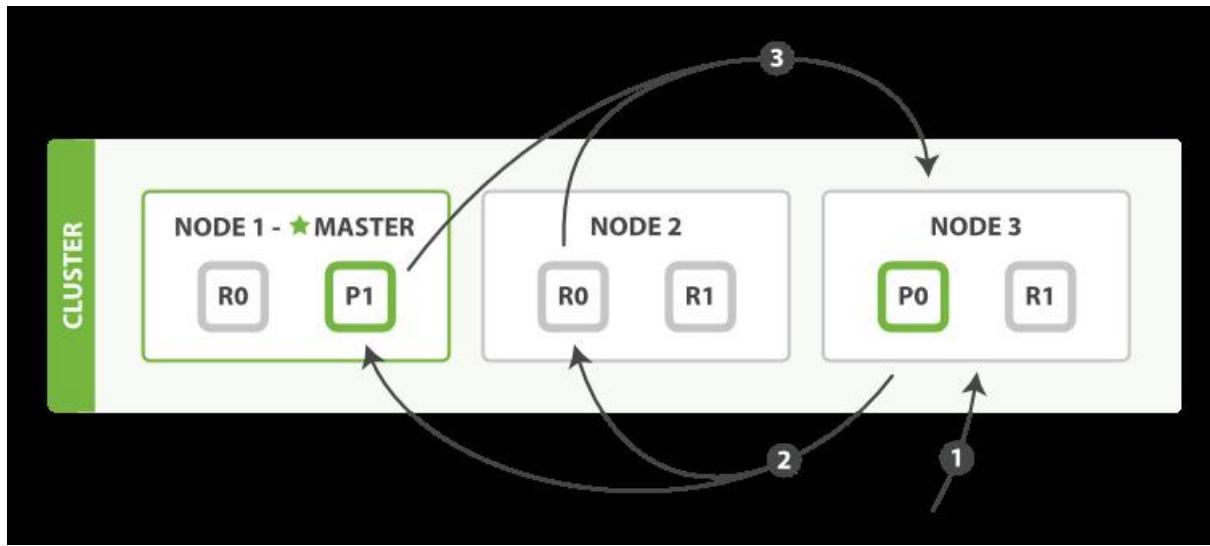


Fig: Query phase of distributed search

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 $\text{from} + \text{size}$.
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 $\text{from} + \text{size}$.
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 $\text{from} + \text{size}$ —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, 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 satisfy the search request, but we still need to retrieve the documents themselves. This is the job of the fetch phase, shown in the below Fig.

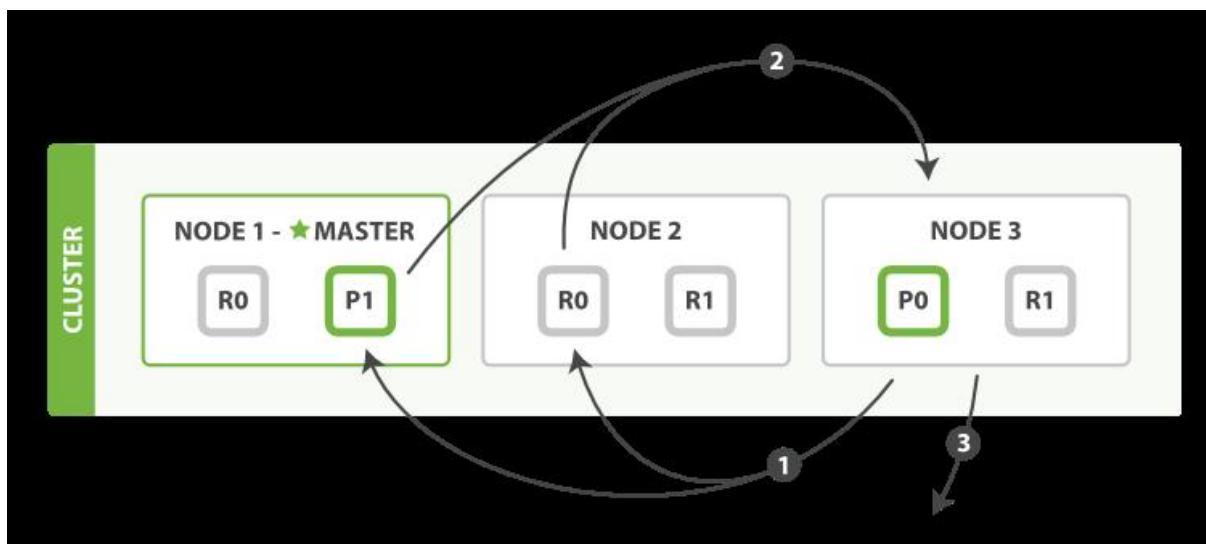


Fig: Fetch phase of distributed search

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.

scan and scroll

The `scan` search type and the `scroll` API 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 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 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 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 request setting search_type to scan, and passing a scroll parameter telling Elasticsearch how long it should keep the scroll open:

```
GET /old_index/_search?search_type=scan&scroll=1m
{
  "query": { "match_all": {} },
  "size": 1000
}
```

The response to this request doesn't include any hits, but does include a _scroll_id, which is a long Base-64 encoded string. Now we can pass the _scroll_id to the _search/scroll endpoint to retrieve the first batch of results:

```
GET /_search/scroll?scroll=1m
c2Nhbs1OzExODpRNV9aY1VyUVM4U0NMd2pjWIJ3YWIBOzExOTpRNV9aY1VyUVM4U0
NMd2pjWIJ3YWIBOzExNjpRNV9aY1VyUVM4U0NMd2pjWIJ3YWIBOzExNzpRNV9aY1Vy
UVM4U0NMd2pjWIJ3YWIBOzEyMDpRNV9aY1VyUVM4U0NMd2pjWIJ3YWIBOzE7dG90Y
WxfaGl0czoxOw==
```

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. When scanning, the size is applied to each shard, so you will get back a maximum of size * num_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.

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,
"_shards": {
  "total": 5,
  "successful": 4,
  "failed": 1
},
...
```

Routing

In “[Routing a Document to a Shard](#)”, 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
```

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.

Scan

The scan search type is used in conjunction with the scroll API to retrieve large numbers of results efficiently. It does this by disabling sorting.

dfs_query_then_fetch and dfs_query_and_fetch

The dfs search types have a prequery phase that fetches the term frequencies from all involved shards in order to calculate global term frequencies.

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.

Creating an Index

Until now, we have created a new index 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:

```
PUT /my_index
{
  "settings": { ... any settings ... },
  "mappings": {
    "type_one": { ... any mappings ... },
    "type_two": { ... any mappings ... },
    ...
  }
}
```

In fact, if you want to, you can prevent the automatic creation of indices by adding the following setting to the config/elasticsearch.yml file on each node:

```
action.auto_create_index: false
```

Deleting an Index

To delete an index, use the following request:

```
DELETE /my_index
```

You can delete multiple indices with this:

```
DELETE /index_one,index_two
```

```
DELETE /index_*
```

You can even delete all indices with this:

```
DELETE/_all
```

Index Settings

There are many many knobs that you can twiddle to customize index behavior, which you can read about in the [Index Modules reference documentation](#).

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 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 one primary shard—and no replica shards with the following request:

```
PUT /my_temp_index
{
  "settings": {
    "number_of_shards" : 1,
    "number_of_replicas" : 0
  }
}
```

Later, we can change the number of replica shards dynamically using the updateindex-settings API as follows:

```
PUT /my_temp_index/_settings
{
  "number_of_replicas": 1
}
```

In the following example, we create a new analyzer called the es_std analyzer, which uses the predefined list of Spanish stopwords:

```
PUT /spanish_docs
{
  "settings": {
    "analysis": {
      "analyzer": {
        "es_std": {
          "type": "standard",
          "stopwords": "_spanish_"
        }
      }
    }
  }
}
```

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:

```
GET /spanish_docs/_analyze?analyzer=es_std
El veloz zorro marrón
```

The abbreviated results show that the Spanish stopword El has been removed correctly:

```
{
  "tokens" : [
```

```
{ "token" : "veloz", "position" : 2 },
{ "token" : "zorro", "position" : 3 },
{ "token" : "marrón", "position" : 4 }
]
}
```

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 “[Analysis and Analyzers](#)”, we said that an analyzer is a wrapper that combines three functions into a single package, which are executed in sequence:

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 <p> or <div> that we don’t want to be indexed. We can use the `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. The tokenizer breaks up the string into individual terms or tokens. The `standard` tokenizer, which is used 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 `keyword` tokenizer outputs exactly the same string as it received, without any tokenization. The `whitespace` tokenizer splits text on whitespace only. The `pattern` tokenizer can be used to split text on a matching regular expression.

Token filters

After tokenization, the resulting token stream is passed through any specified token filters, in the order in which they are specified.

Token filters may change, add, or remove tokens. We have already mentioned the `lowercase` and `stop` token filters, but there are many more available in Elasticsearch. `Stemming` token filters “stem” words to their root form. The `ascii_folding` filter removes diacritics, converting a term like “très” into “tres”. The

ngram and edge_ngram token filters can produce tokens suitable for partial matching or autocomplete.

Creating a Custom Analyzer

In the same way as we configured the es_std analyzer previously, we can configure character filters, tokenizers, and token filters in their respective sections under analysis:

```
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:

```
"char_filter": {
  "&_to_and": {
    "type": "mapping",
    "mappings": [ "&=> and "]
  }
}
```

3. Tokenize words, using the standard tokenizer.
4. Lowercase terms, using the lowercase token filter.
5. Remove a custom list of stopwords, using a custom stop token filter:

```
"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:

```
"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 looks like this:

```
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" ]  
            }  
          }  
    }  
  }
```

After creating the index, use the analyze API to test the new analyzer:

```
GET /my_index/_analyze?analyzer=my_analyzer
```

The quick & brown fox

The following abbreviated results show that our analyzer is working correctly:

```
{  
  "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 Elasticsearch where to use it. We can

apply it to a string field with a mapping such as the following:

```
PUT /my_index/_mapping/my_type
{
  "properties": {
    "title": {
      "type": "string",
      "analyzer": "my_analyzer"
    }
  }
}
```

Types and Mappings

A type in Elasticsearch represents a class of similar documents. A type consists of a name—such as user or blogpost—and a mapping. The mapping, like a database schema, describes the fields or properties that documents of that type may have, the datatype of each field—such as string, integer, or date—and how those fields should be indexed and stored by Lucene.

How Lucene Sees Documents

A document in Lucene consists of a simple list of field-value pairs. 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.

The Root Object

The uppermost level of a mapping is known 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.

Metadata: _source Field

By default, Elasticsearch 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:

```
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:

```
GET /_search
```

```
{  
  "query": { "match_all": {}},  
  "_source": [ "title", "created" ]  
}
```

Metadata: Document Identity

There are four metadata fields 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 stored (can be retrieved) and indexed (searchable). The **_type** 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 Elasticsearch that it should extract the value for the **_id** from a field within the document itself.

```
PUT /my_index  
{  
  "mappings": {  
    "my_type": {  
      "_id": {  
        "path": "doc_id"  
      },  
      "properties": {  
        "doc_id": {  
          "type": "string",  
          "index": "not_analyzed"  
        }  
      }  
    }  
  }  
}
```

Then, when you index a document:

```
POST /my_index/my_type
```

```
{  
  "doc_id": "123"  
}
```

the _id value will be extracted from the doc_id field in the document body:

```
{  
  "_index": "my_index",  
  "_type": "my_type",  
  "_id": "123",  
  "_version": 1,  
  "created": true  
}
```

Dynamic Mapping

When Elasticsearch encounters a previously unknown field in a document, it uses **dynamic mapping** 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 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:

```
PUT /my_index
```

```
{  
  "mappings": {  
    "my_type": {  
      "dynamic": "strict",  
      "properties": {  
        "title": { "type": "string"},  
        "stash": {  
          "type": "object",  
        }  
      }  
    }  
  }  
}
```

```
"dynamic": true
}
}
}
}
}
```

With this mapping, you can add new searchable fields into the stash object:

```
PUT /my_index/my_type/1
{
  "title": "This doc adds a new field",
  "stash": { "new_field": "Success!" }
}
```

But trying to do the same at the top level will fail:

```
PUT /my_index/my_type/1
{
  "title": "This throws a StrictDynamicMappingException",
  "new_field": "Fail!"
}
```

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. 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

When Elasticsearch encounters a new string field, it checks to see if the string contains a recognizable date, like 2014-01-01. 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:

```
{ "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:

```
{ "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:

```
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.

dynamic_templates

With dynamic_templates, you can take complete control 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 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:

```
PUT /my_index
{
  "mappings": {
    "my_type": {
      "dynamic_templates": [
        { "es": {
          "match": "*_es",
          "match_mapping_type": "string",
          "mapping": {
            "type": "string",
            "analyzer": "spanish"
          }
        }},
        { "en": {
          "match": "*",
          "match_mapping_type": "string",

```

```
"mapping": {  
  "type": "string",  
  "analyzer": "english"  
}  
}  
]  
}}
```

The `match_mapping_type` allows 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 matches the full path to a field in an object, so the pattern `address.*.name` would match a field like this:

```
{  
  "address": {  
    "city": {  
      "name": "New York"  
    }  
  }  
}
```

The `unmatch` and `path_unmatch` patterns can be used to exclude fields that would otherwise match.

Default Mapping

Often, all types in an index share similar fields and settings. 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, using the `_default_` mapping, but enable it just for the `blog` type, as follows:

```
PUT /my_index  
{  
  "mappings": {  
    "_default": {  
      "_all": { "enabled": false }  
    },  
    "blog": {
```

```
"_all": { "enabled": true }
}
}
}
```

The `_default_` mapping can also be a good place to specify index-wide [dynamic templates](#).

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. 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 [scan-and-scroll](#) to retrieve batches of documents from the old index, and the [bulk API](#) 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:

```
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 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 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 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:

```
PUT /my_index_v1
PUT /my_index_v1/_alias/my_index
```

Create the index `my_index_v1`.

Set the `my_index` alias to point to `my_index_v1`.

You can check which index the alias points to:

```
GET /*/_alias/my_index
```

Or which aliases point to the index:

```
GET /my_index_v1/_alias/*
```

Both of these return the following:

```
{
  "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. To start, we create `my_index_v2` with the new mappings:

```
PUT /my_index_v2
{
  "mappings": {
    "my_type": {
      "properties": {
        "tags": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

Then we reindex our data from `my_index_v1` to `my_index_v2`.

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:

```
POST /_aliases
{
  "actions": [
    { "remove": { "index": "my_index_v1", "alias": "my_index" } },
    { "add": { "index": "my_index_v2", "alias": "my_index" } }
  ]
}
```

Your application has switched from using the old index to the new index transparently, with zero downtime.

Note: 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

we introduced the shard, and described 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?

Making Text Searchable

The first challenge that had to be solved was how to 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. 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 “[What Is Relevance?](#)”.

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.

Immutability

The inverted index that is written to disk is immutable: it doesn’t change. 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.

Dynamically Updatable Indices

The next problem that needed to be solved was how to make an inverted index updatable without losing the benefits of immutability? The answer turned out to be: use more than one index.

Instead of rewriting the whole inverted index, add new supplementary indices to reflect more-recent changes. Each inverted index can be queried in turn—starting with the oldest—and the results combined.

Lucene, the Java libraries on which Elasticsearch is based, introduced the concept of per-segment search. A segment is an inverted index in its own right, but now the word index in Lucene came to mean a collection of segments plus a commit point—a file that lists all known segments, as depicted in Fig1. New documents are first added to an in-memory indexing buffer, as shown in [Fig2](#), before being written to an on-disk segment, as in [Fig3](#)

Index Versus Shard

To add to the confusion, a Lucene index is what we call a shard in Elasticsearch, while an index in Elasticsearch is a collection of shards. When Elasticsearch searches an index, it sends the query out to a copy of every shard (Lucene index) that belongs to the index, and then reduces the per-shards results to a global result set.

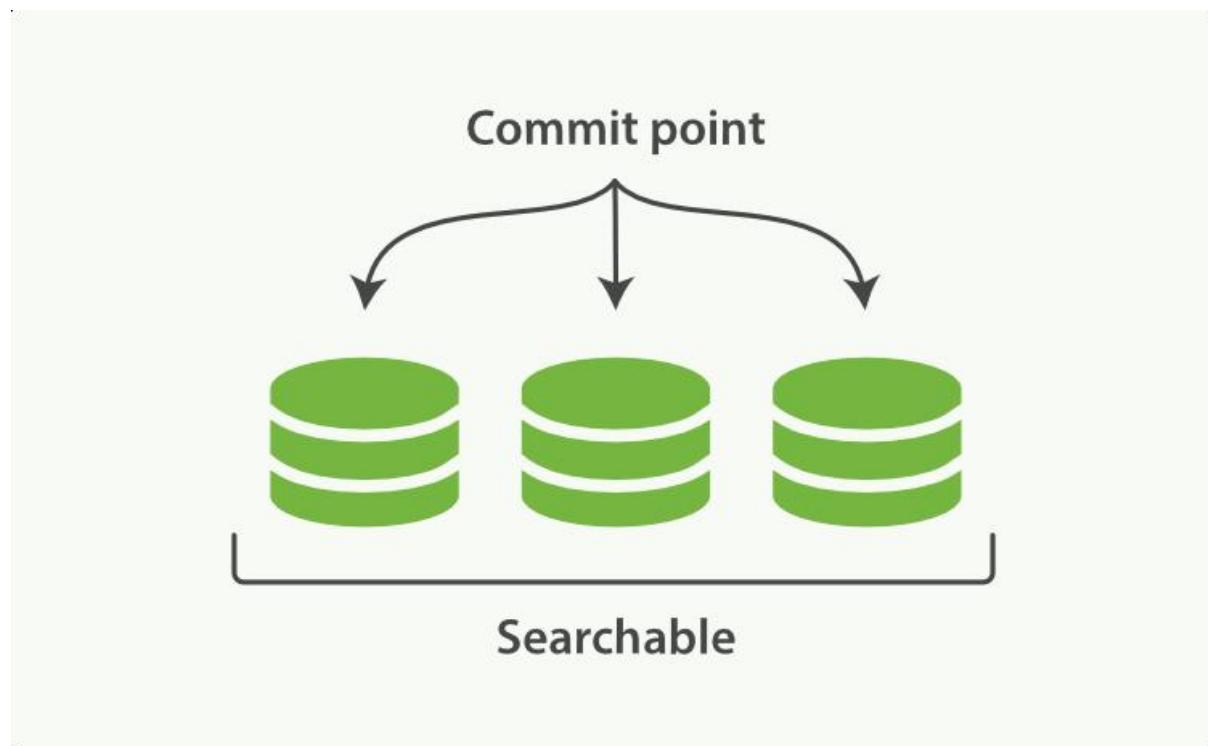


Fig1. A Lucene index with a commit point and three segments

A per-segment search works as follows:

1. New documents are collected in an in-memory indexing buffer. See [Fig2](#).
2. Every so often, the buffer is committed:

3. A new segment—a supplementary inverted index—is written to disk.
4. A new commit point is written to disk, which includes the name of the new segment.
5. The disk is fsync’ed—all writes waiting in the filesystem cache are flushed to disk, to ensure that they have been physically written.
6. The new segment is opened, making the documents it contains visible to search.
7. The in-memory buffer is cleared, and is ready to accept new documents.

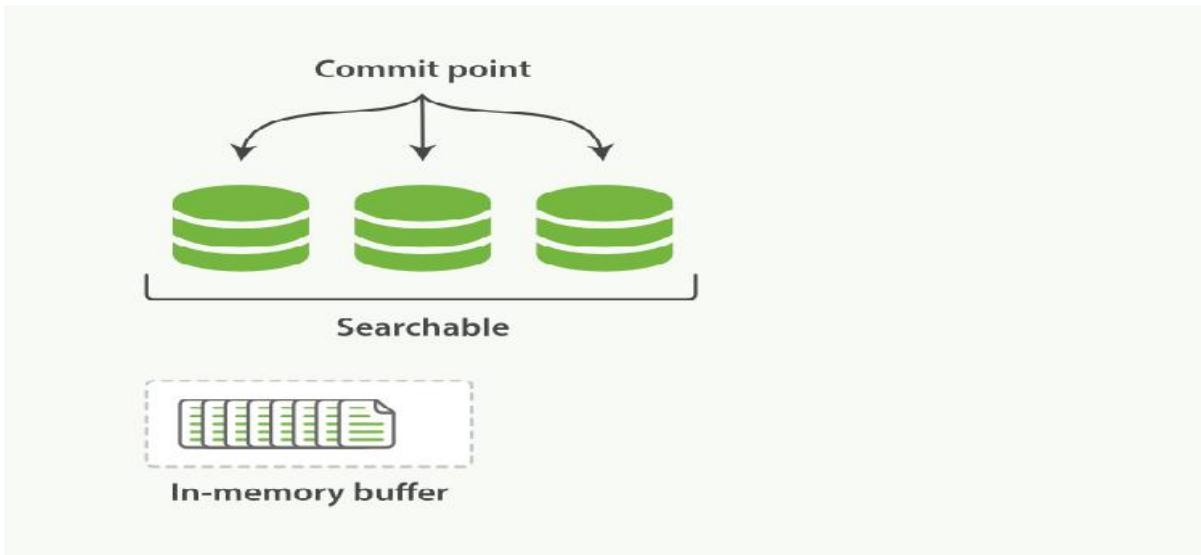


Fig2. A Lucene index with new documents in the in-memory buffer, ready to Commit

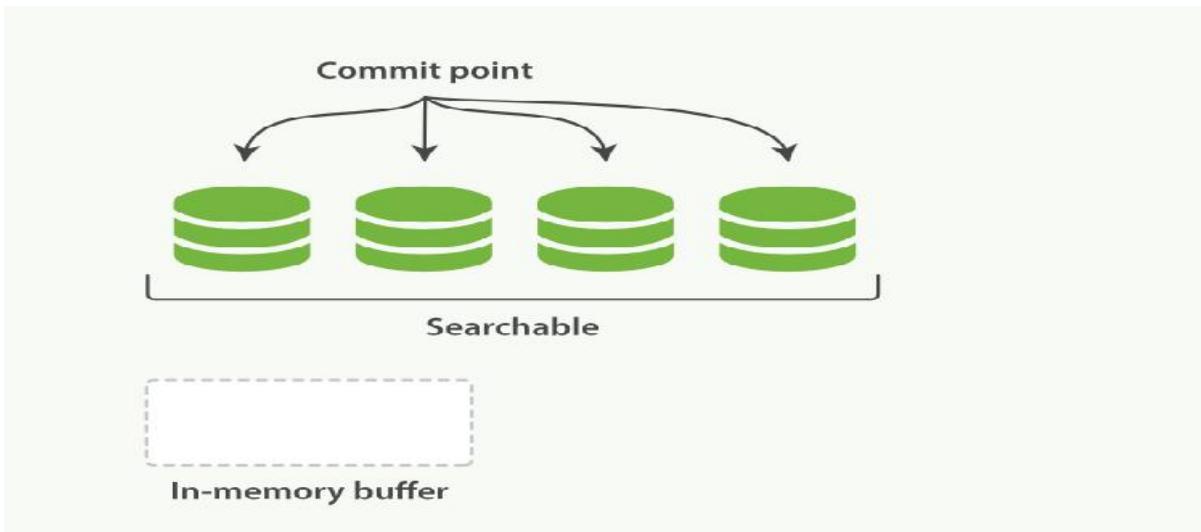


Fig3. After a commit, a new segment is added to the commit point and the buffer is cleared

When a query is issued, all known segments are queried in turn. Term statistics are aggregated across all segments to ensure that the relevance of each term and

each document is calculated accurately. In this way, new documents can be added to the index relatively cheaply.

Deletes and Updates

Segments are immutable, so documents cannot be removed from older segments, nor can older segments be updated to reflect a newer version of a document. Instead, every commit point includes a .del file that lists which documents in which segments have been deleted.

When a document is “deleted,” it is actually just marked as deleted in the .del file. A document that has been marked as deleted can still match a query, but it is removed from the results list before the final query results are returned.

Document updates work in a similar way: when a document is updated, the old version of the document is marked as deleted, and the new version of the document is indexed in a new segment. Perhaps both versions of the document will match a query, but the older deleted version is removed before the query results are returned.

Near Real-Time Search

With the development of per-segment search, the 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 a new segment to disk requires an `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. As before, documents in the in-memory indexing buffer (Fig4) are written to a new segment (Fig5). 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.

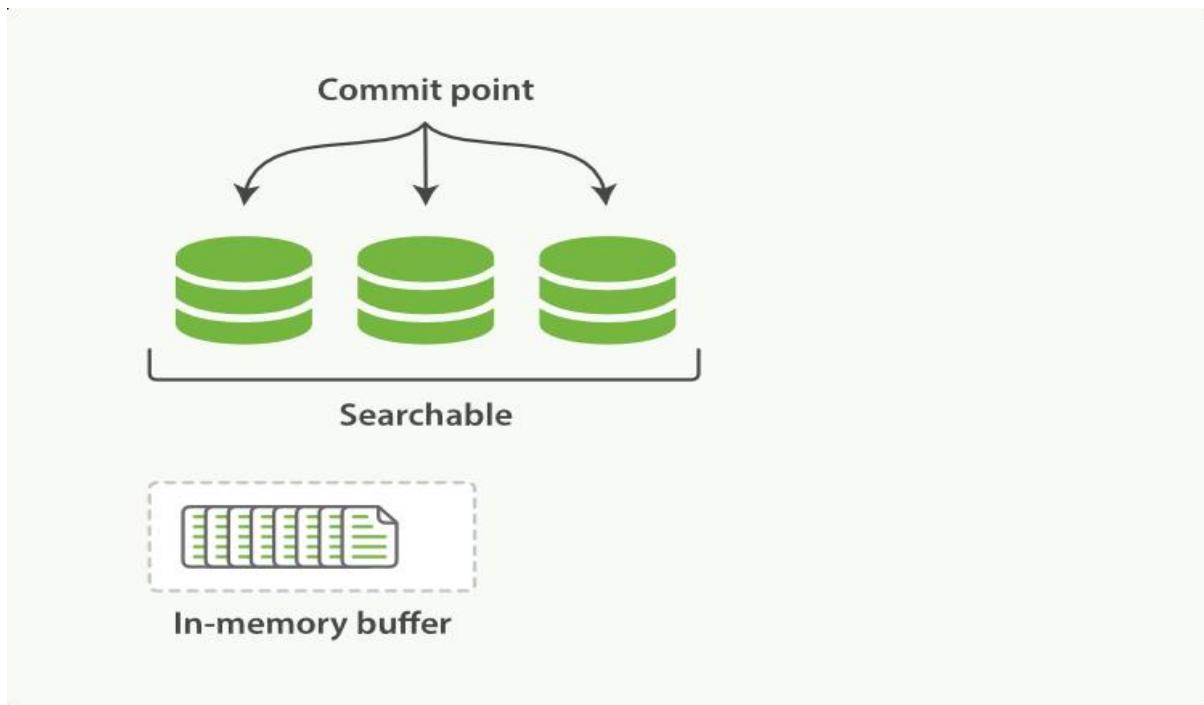


Fig4. A Lucene index with new documents in the in-memory buffer

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.

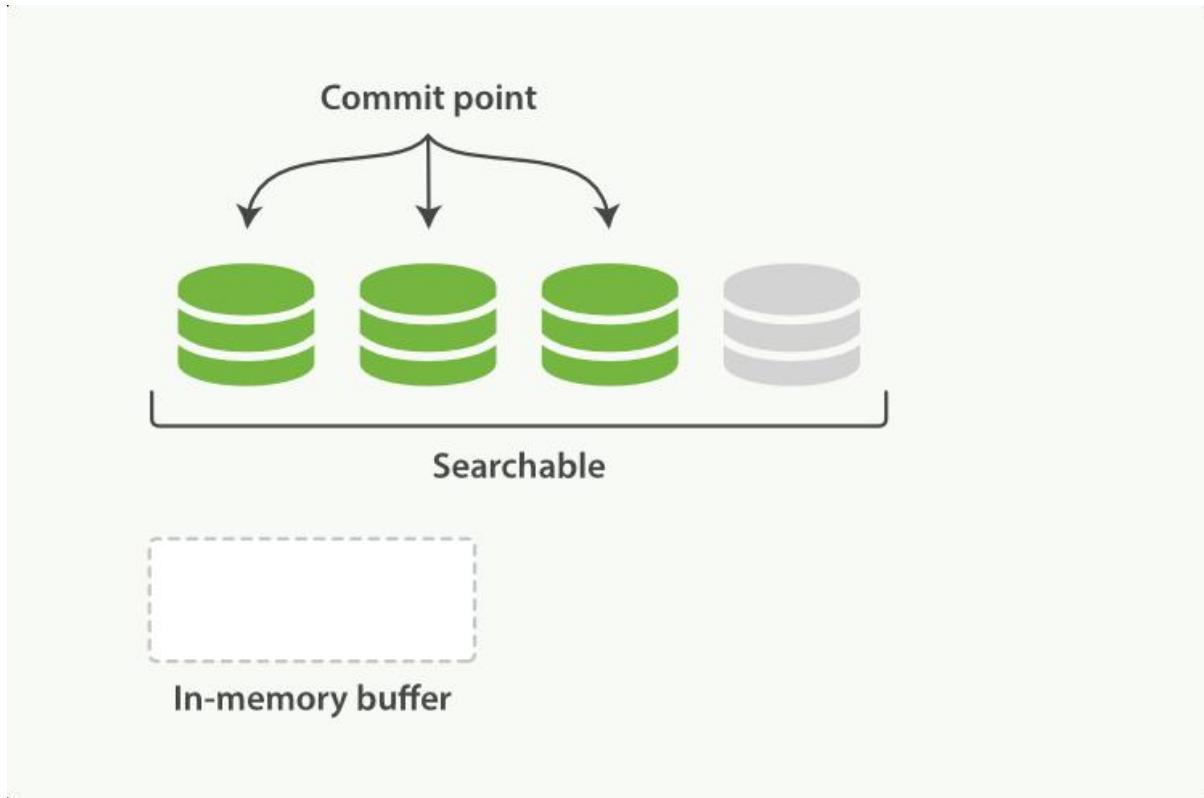


Fig5. The buffer contents have been written to a segment, which is searchable, but is not yet committed

Refresh API

In Elasticsearch, this lightweight process of writing and opening a new segment is called a refresh. 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:

```
POST /_refresh          - Refresh all indices  
POST /blogs/_refresh    - Refresh just the blogs index
```

Note: While a refresh is much lighter than a commit, it still has a performance cost. 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 setting the refresh_interval:

```
PUT /my_logs  
{  
  "settings": {  
    "refresh_interval": "30s"  - 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:

```
POST /my_logs/_settings  
{ "refresh_interval": -1 }          - Disable automatic refreshes.  
POST /my_logs/_settings  
{ "refresh_interval": "1s" }         - Refresh automatically every second
```

Note: The refresh_interval expects a duration such as 1s (1 second) or 2m (2 minutes). An absolute number like 1 means 1millisecond--a sure way to bring your cluster to its knees.

Making Changes Persistent

Without an fsync to flush data in the filesystem cache to disk, we cannot be sure that the data will still 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.

we said that a full commit flushes segments to disk and writes a commit point, which lists all known segments. 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, which records every operation in Elasticsearch as it happens. With the translog, the process now looks like this:

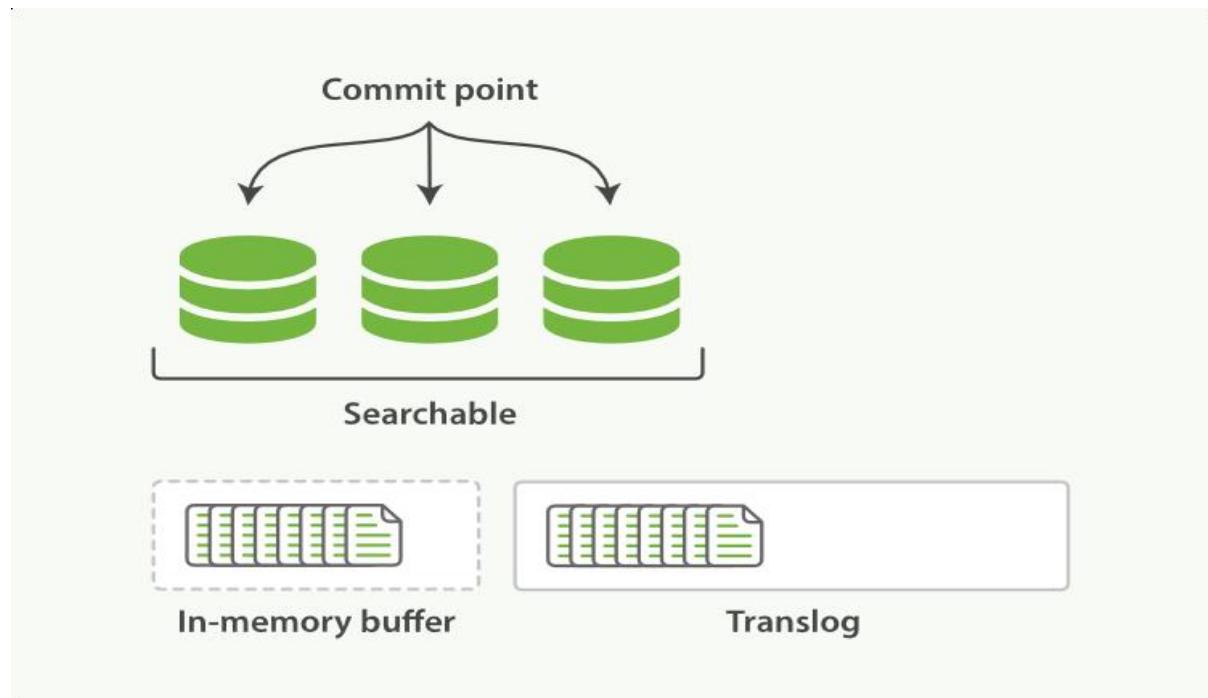


Fig6. New documents are added to the in-memory buffer and appended to the transaction log

- When a document is indexed, it is added to the in-memory buffer and appended to the translog, as shown in Fig6.
- The refresh leaves the shard in the state depicted in Fig7. 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.

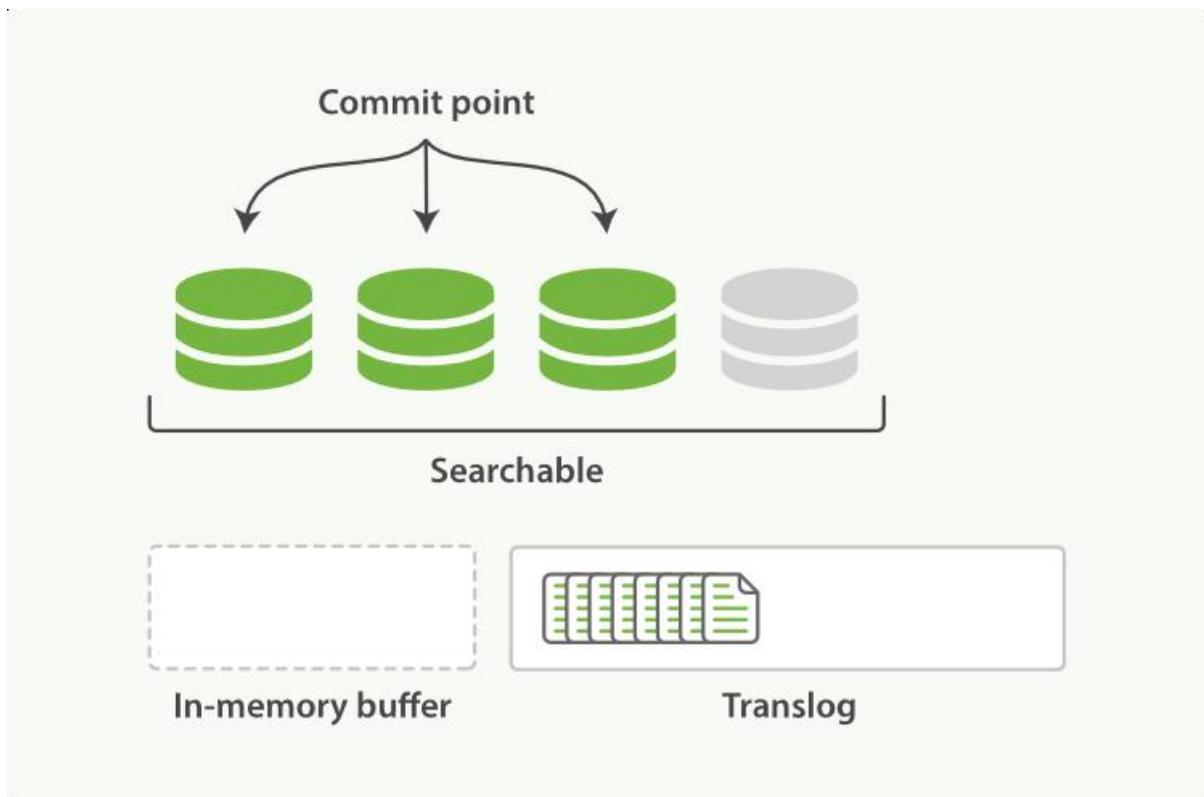


Fig7. After a refresh, the buffer is cleared but the transaction log is not

- This process continues with more documents being added to the in-memory buffer and appended to the transaction log (Fig8).

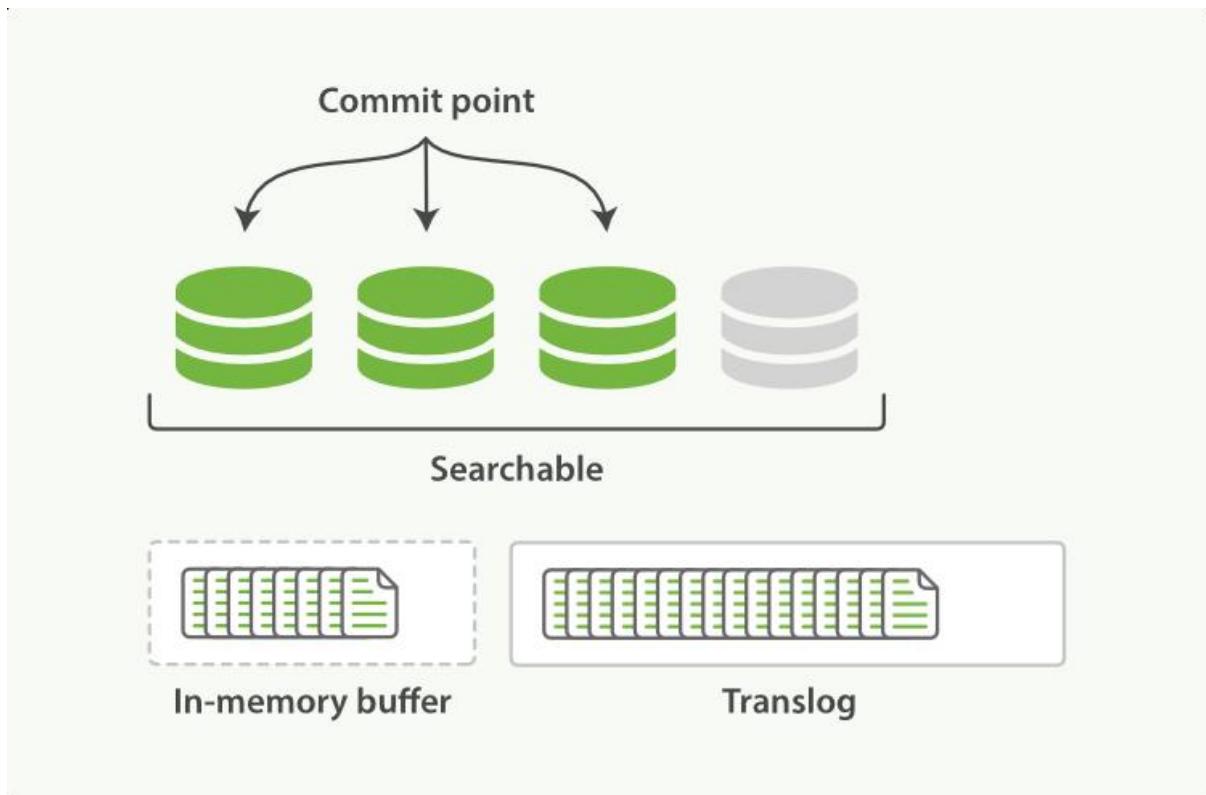


Fig8. The transaction log keeps accumulating documents

4. 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 Fig9):

- 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.

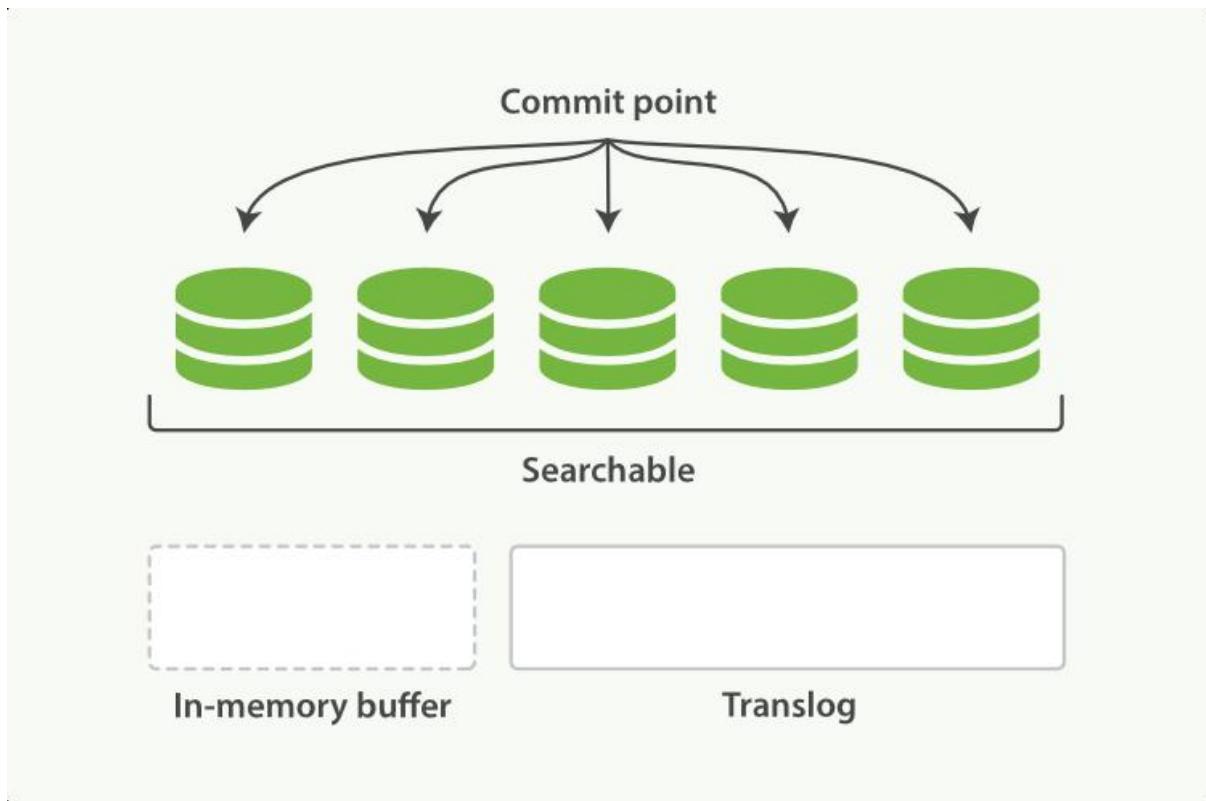


Fig9. After a flush, the segments are fully committed and the transaction log is cleared

flush API

The action of performing a commit and truncating the translog is known in Elasticsearch as a flush. Shards are flushed automatically every 30 minutes, or when the translog becomes too big. See the [translog documentation](#) for settings that can be used to control these thresholds:

The [flush API](#) can be used to perform a manual flush:

`POST /blogs/_flush` – Flush the blogs index.

`POST /_flush?wait_for_ongoing` – 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 [flush](#) 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 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.

Segment Merging

With the automatic refresh process creating a new segment 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. 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 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 Fig10:

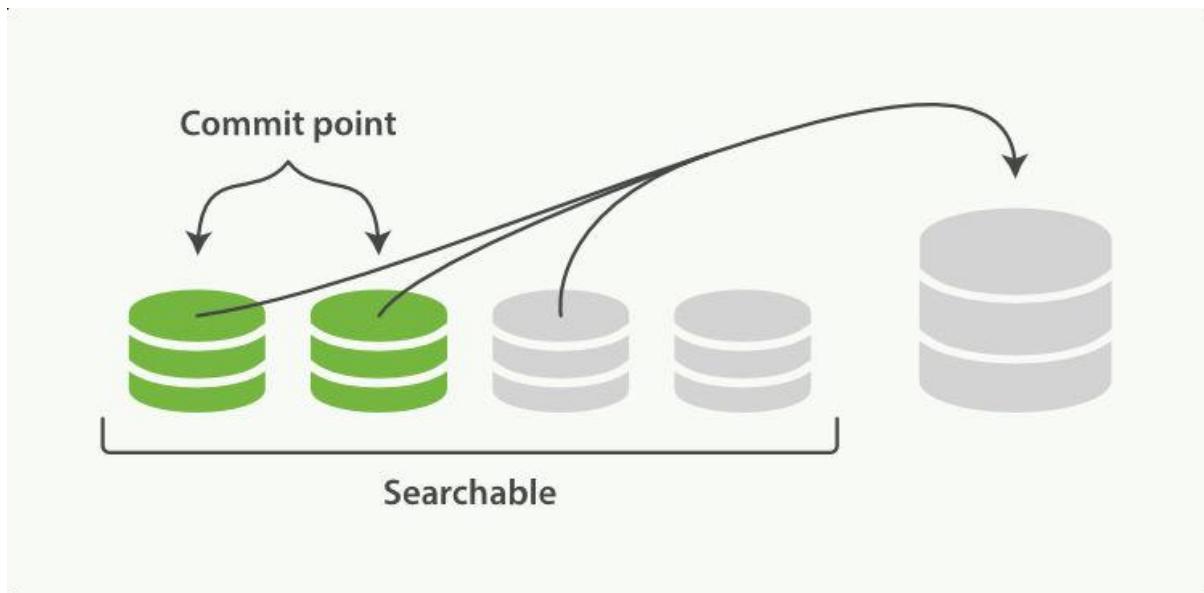


Fig10. Two committed segments and one uncommitted segment in the process of being merged into a bigger segment

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.
3. [Fig11](#) illustrates activity as the merge completes:

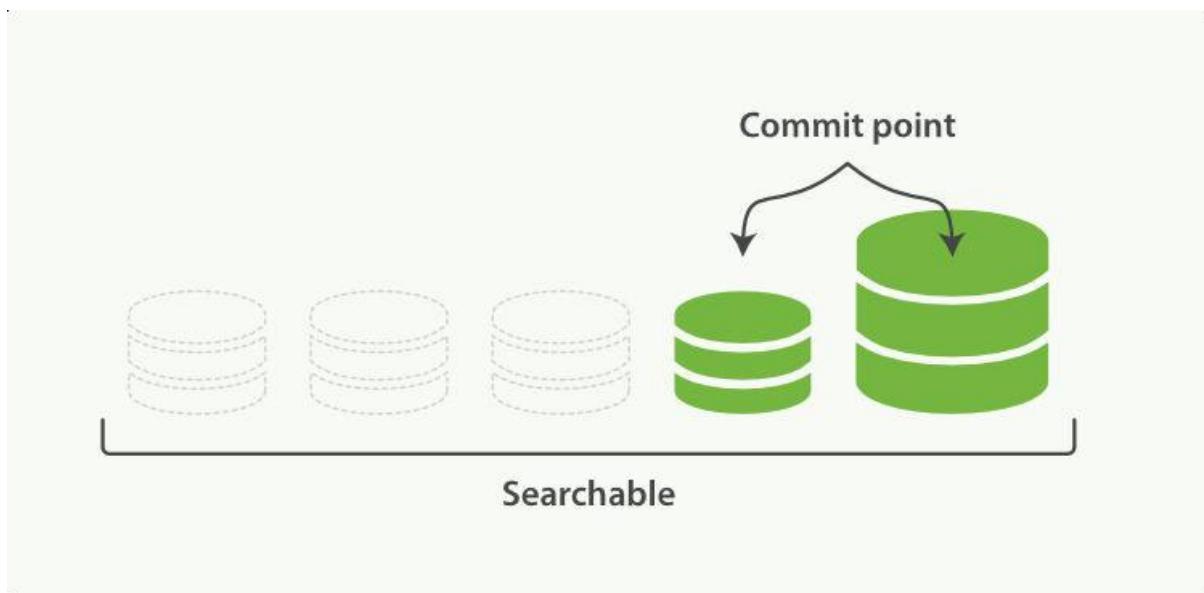


Fig11. Once merging has finished, the old segments are deleted

- 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.

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.

Optimize API

The optimize API is best 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.

Note: 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:

`POST /logstash-2014-10/_optimize?max_num_segments=1`

Merges each shard in the index down to a single segment

Note: 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 to first move the index to a node where it is safe to run.

Structured Search

Structured search is about interrogating 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.

Finding Exact Values

When working with exact values, you will be working with filters. Filters are important because they are very, very fast. Filters do not calculate relevance (avoiding the entire scoring phase) and are easily cached.

term Filter with Numbers

We are going to explore the term filter first because you will use it often. This filter is capable of handling numbers, Booleans, dates, and text.

Let's look at an example using numbers first by indexing some products. These documents have a price and a productID:

```
POST /my_store/products/_bulk
{ "index": { "_id": 1 } }
{ "price" : 10, "productID" : "XHDK-A-1293-#fJ3" }
{ "index": { "_id": 2 } }
{ "price" : 20, "productID" : "KDKE-B-9947-#kL5" }
{ "index": { "_id": 3 } }
{ "price" : 30, "productID" : "JODL-X-1937-#pV7" }
{ "index": { "_id": 4 } }
{ "price" : 30, "productID" : "QQPX-R-3956-#aD8" }
```

Our goal is to find all products with a certain price. You may be familiar with SQL if you are coming from a relational database background. If we expressed this query as an SQL query, it would look like this:

```
SELECT document FROM products WHERE price = 20
```

In the Elasticsearch query DSL, we use a term filter to accomplish the same thing. The term filter will look for the exact value that we specify. By itself, a term filter is simple. It accepts a field name and the value that we wish to find:

```
{  
  "term" : {  
    "price" : 20  
  }  
}
```

The term filter isn't very useful on its own, though. the search API expects a query, not a filter. To use our term filter, we need to wrap it with a **filtered query**:

```
GET /my_store/products/_search  
{  
  "query" : {  
    "filtered" : {  
      "query" : {  
        "match_all" : {}  
      },  
      "filter" : {  
        "term" : {}  
      }  
    }  
  }  
}  
}  
}  
}  
}  
}  
}
```

The filtered query accepts both a query and a filter.
A match_all is used to return all matching documents.
Notice how it is placed inside the filter clause.

Once executed, the search results from this query are exactly what you would expect: only document 2 is returned as a hit (because only 2 had a price of 20):

```
"hits" : [  
  {  
    "_index" : "my_store",  
    "_type" : "products",  
    "_id" : "2",  
    "_score" : 1.0,  
    "_source" : {
```

```
"price" : 20,  
"productID" : "KDKE-B-9947-#kL5"  
}  
}  
]
```

term Filter with Text

As mentioned at the top of this section, the term filter can match strings just as easily as numbers. Instead of price, let's try to find products that have a certain UPC identification code. To do this with SQL, we might use a query like this:

```
SELECT product  
FROM products  
WHERE productID = "XHDK-A-1293-#fJ3"
```

Translated into the query DSL, we can try a similar query with the term filter, like so:

```
GET /my_store/products/_search  
{  
  "query" : {  
    "filtered" : {  
      "filter" : {  
        "term" : {  
          "productID" : "XHDK-A-1293-#fJ3"  
        }  
      }  
    }  
  }  
}
```

Except there is a little hiccup: we don't get any results back! Why is that? The problem isn't with the term query; it is with the way the data has been indexed. If we use the analyze API we can see that our UPC has been tokenized into smaller tokens:

```
GET /my_store/_analyze?field=productID  
XHDK-A-1293-#fJ3
```

```
{  
  "tokens" : [ {  
    "token" : "xhdk",  
    "start_offset" : 0,  
    "end_offset" : 4,
```

```

"type" : "<ALPHANUM>",
"position" : 1
}, {
"token" : "a",
"start_offset" : 5,
"end_offset" : 6,
"type" : "<ALPHANUM>",
"position" : 2
}, {
"token" : "1293",
"start_offset" : 7,
"end_offset" : 11,
"type" : "<NUM>",
"position" : 3
}, {
"token" : "fj3",
"start_offset" : 13,
"end_offset" : 16,
"type" : "<ALPHANUM>",
"position" : 4
} ]
}

```

There are a few important points here:

- We have four distinct tokens instead of a single token representing the UPC.
- All letters have been lowercased.
- We lost the hyphen and the hash (#) sign.

So when our term filter looks for the exact value XHDK-A-1293-#fJ3, it doesn't find anything, because that token does not exist in our inverted index. Instead, there are the four tokens listed previously.

Obviously, this is not what we want to happen when dealing with identification codes, or any kind of precise enumeration.

To prevent this from happening, we need to tell Elasticsearch that this field contains an exact value by setting it to be `not_analyzed`. To do this, we need to first delete our old index (because it has the incorrect mapping) and create a new one with the correct mappings:

```

DELETE /my_store
PUT /my_store
{
  "mappings" : {
    "products" : {
      "properties" : {
        "productID" : {
          "type" : "string",
          "index" : "not_analyzed"
        }
      }
    }
  }
}

```

Now we can go ahead and reindex our documents:

```

POST /my_store/products/_bulk
{ "index": { "_id": 1 } }
{ "price" : 10, "productID" : "XHDK-A-1293-#fJ3" }
{ "index": { "_id": 2 } }
{ "price" : 20, "productID" : "KDKE-B-9947-#KL5" }
{ "index": { "_id": 3 } }
{ "price" : 30, "productID" : "JODL-X-1937-#pV7" }
{ "index": { "_id": 4 } }
{ "price" : 30, "productID" : "QQPX-R-3956-#aD8" }

```

Only now will our term filter work as expected. Let's try it again on the newly indexed data (notice, the query and filter have not changed at all, just how the data is mapped):

```

GET /my_store/products/_search
{
  "query" : {
    "filtered" : {
      "filter" : {
        "term" : {
          "productID" : "XHDK-A-1293-#fJ3"
        }
      }
    }
  }
}

```

Since the productID field is not analyzed, and the term filter performs no analysis, the query finds the exact match and returns document 1 as a hit. Success!

Internal Filter Operation

Internally, Elasticsearch is performing several operations when executing a filter:

1. Find matching docs.

The term filter looks up the term XHDK-A-1293-#fJ3 in the inverted index and retrieves the list of documents that contain that term. In this case, only document 1 has the term we are looking for.

2. Build a bitset.

The filter then builds a bitset—an array of 1s and 0s—that describes which documents contain the term. Matching documents receive a 1 bit. In our example, the bitset would be [1,0,0,0].

3. Cache the bitset.

Last, the bitset is stored in memory, since we can use this in the future and skip steps 1 and 2. This adds a lot of performance and makes filters very fast.

When executing a filtered query, the filter is executed before the query. The resulting bitset is given to the query, which uses it to simply skip over any documents that have already been excluded by the filter. This is one of the ways that filters can improve performance. Fewer documents evaluated by the query means faster response times.

Combining Filters

The previous two examples showed a single filter in use. In practice, you will probably need to filter on multiple values or fields. For example, how would you express this SQL in Elasticsearch?

```
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. This is a compound filter that accepts other filters as arguments, combining them in various Boolean combinations.

Bool Filter

The bool filter is composed of three sections:

```
{  
"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! 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 previously and place them inside the should clause of a bool filter, and add another clause to deal with the NOT condition:

GET /my_store/products/_search

```
{
  "query" : {
    "filtered" : {
      "filter" : {
        "bool" : {
          "should" : [
            { "term" : {"price" : 20}},
            { "term" : {"productID" : "XHDK-A-1293-#fJ3"}}
          ],
          "must_not" : {
            "term" : {"price" : 30}
          }
        }
      }
    }
  }
}
```

Our search results return two hits, each document satisfying a different clause in the bool filter:

```
"hits" : [
{
  "_id" : "1",
  "_score" : 1.0,
  "_source" : {
    "price" : 10,
    "productID" : "XHDK-A-1293-#fJ3"
  }
},
{
  "_id" : "2",
  "_score" : 1.0,
  "_source" : {
    "price" : 20,
    "productID" : "KDKE-B-9947-#kL5"
  }
}
]
```

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. 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:

```
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:

```
GET /my_store/products/_search
{
  "query" : {
    "filtered" : {
      "filter" : {
        "bool" : {
          "should" : [
            { "term" : {"productID" : "KDKE-B-9947-#kL5"} },
            { "bool" : {
              "must" : [
```

```

    { "term" : {"productID" : "JODL-X-1937-#pV7"},  

    { "term" : {"price" : 30}}  

]  

}  

]  

}  

}  

}  

}  

}

```

The results show us two documents, one matching each of the should clauses:

```

"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"  

  }  

}  

]

```

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. 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 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:

```
{  
  "terms" : {  
    "price" : [20, 30]  
  }  
}
```

And like the term filter, we will place it inside a filtered query to use it:

```
GET /my_store/products/_search
```

```
{  
  "query" : {  
    "filtered" : {  
      "filter" : {  
        "terms" : {  
          "price" : [20, 30]  
        }  
      }  
    }  
  }  
}
```

The query will return the second, third, and fourth documents:

```
"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. What does that mean?

If you have a term filter for { "term" : { "tags" : "search" } }, it will match both of the following documents:

```

{ "tags" : ["search"] }
{ "tags" : ["search", "open_source"] }

```

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:

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. 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:

```
{ "tags" : [ "search" ], "tag_count" : 1 }
{ "tags" : [ "search", "open_source" ], "tag_count" : 2 }
```

Once you have the count information indexed, you can construct a bool filter that enforces the appropriate number of terms:

```
GET /my_index/my_type/_search
{
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "must": [
            { "term" : { "tags" : "search" } },
            { "term" : { "tag_count" : 1 } }
          ]
        }
      }
    }
  }
}
```

This query will now match only the document that has a single tag that is search, rather than any document that contains search.

Ranges

When dealing with numbers in this chapter, we have so far searched for only exact numbers. In practice, filtering on ranges is often more useful. For example, you might want to find all products with a price greater than \$20 and less than \$40.

In SQL terms, a range can be expressed as follows:

SELECT document

FROM products

WHERE price **BETWEEN** 20 **AND** 40

Elasticsearch has a range filter, which, unsurprisingly, allows you to filter ranges:

```
"range" : {
  "price" : {
    "gt" : 20,
```

```
"lt" : 40
}
}
```

Ranges on Dates

The range filter can be used on date fields too:

```
"range" : {
  "timestamp" : {
    "gt" : "2014-01-01 00:00:00",
    "lt" : "2014-01-07 00:00:00"
  }
}
```

When used on date fields, the range filter supports date math operations. For example, if we want to find all documents that have a timestamp sometime in the last hour:

```
"range" : {
  "timestamp" : {
    "gt" : "now-1h"
  }
}
```

This filter will now constantly find all documents with a timestamp greater than the current time minus 1 hour, making the filter a sliding window across your documents.

Date math can also be applied to actual dates, rather than a placeholder like now. Just add a double pipe (||) after the date and follow it with a date math expression:

```
"range" : {
  "timestamp" : {
    "gt" : "2014-01-01 00:00:00",
    "lt" : "2014-01-01 00:00:00||+1M"
  }
}
```

Ranges on Strings

The range filter can also operate on string fields. String ranges are calculated lexicographically or alphabetically. For example, these values are sorted in lexicographic order:

- 5, 50, 6, B, C, a, ab, abb, abc, b

Note: Terms in the inverted index are sorted in lexicographical order, which is why string ranges use this order.

If we want a range from a up to but not including b, we can use the same range filter syntax:

```
"range" : {  
  "title" : {  
    "gte" : "a",  
    "lt" : "b"  
  }  
}
```

Be Careful of Cardinality

Numeric and date fields are indexed in such a way that ranges are efficient to calculate. This is not the case for string fields, however. To perform a range on a string field, Elasticsearch is effectively performing a term filter for every term that falls in the range. This is much slower than a date or numeric range.

String ranges are fine on a field with low cardinality—a small number of unique terms. But the more unique terms you have, the slower the string range will be.

Dealing with Null Values

exists Filter

The first tool in your arsenal is the exists filter. This filter will return documents that have any value in the specified field. Let's use the tagging example and index some example documents:

```
POST /my_index/posts/_bulk  
{ "index": { "_id": "1" }}  
{ "tags" : [ "search" ] } - The tags field has one value.  
{ "index": { "_id": "2" }}  
{ "tags" : [ "search", "open_source" ] } --The tags field has two values.  
  
{ "index": { "_id": "3" }}  
{ "other_field" : "some data" } - The tags field is missing altogether.  
{ "index": { "_id": "4" }}  
{ "tags" : null } - The tags field is set to null.  
{ "index": { "_id": "5" }}  
{ "tags" : [ "search", null ] } -The tags field has one value and a null.
```

The resulting inverted index for our tags field will look like this:

Token	DocIDs
-------	--------

open_source	2
-------------	---

search	1,2,5
--------	-------

Our objective is to find all documents where a tag is set. We don't care what the tag is, so long as it exists within the document. In SQL parlance, we would use an IS NOT NULL query:

```
SELECT tags
FROM posts
WHERE tags IS NOT NULL
```

In Elasticsearch, we use the exists filter:

```
GET /my_index/posts/_search
{
  "query" : {
    "filtered" : {
      "filter" : {
        "exists" : { "field" : "tags" }
      }
    }
  }
}
```

Our query returns three documents:

```
"hits" : [
  {
    "_id" : "1",
    "_score" : 1.0,
    "_source": { "tags" : [ "search" ] }
  },
  {
    "_id" : "5",
    "_score" : 1.0,
    "_source": { "tags" : [ "search", null ] }
  },
  {
    "_id" : "2",
    "_score" : 1.0,
    "_source": { "tags" : [ "search", "open source" ] }
  }
]
```

```
}
```

```
]
```

The results are easy to understand. Any document that has terms in the tags field was returned as a hit. The only two documents that were excluded were documents 3 and 4.

missing Filter

The missing filter is essentially the inverse of exists: it returns documents where there is no value for a particular field, much like this SQL:

```
SELECT tags  
FROM posts  
WHERE tags IS NULL
```

Let's swap the exists filter for a missing filter from our previous example:

```
GET /my_index/posts/_search
```

```
{  
  "query" : {  
    "filtered" : {  
      "filter": {  
        "missing" : { "field" : "tags" }  
      }  
    }  
  }  
}
```

And, as you would expect, we get back the two docs that have no real values in the tags field—documents 3 and 4:

```
"hits" : [  
  {  
    "_id" : "3",  
    "_score" : 1.0,  
    "_source": { "other_field" : "some data" }  
  },  
  {  
    "_id" : "4",  
    "_score" : 1.0,  
    "_source": { "tags" : null }  
  }  
]
```

When null Means null

Sometimes you need to be able to distinguish between a field that doesn't have a

value, and a field that has been explicitly set to null. With the default behavior that we saw previously, this is impossible; the data is lost. Luckily, there is an option that we can set that replaces explicit null values with a placeholder value of our choosing. When specifying the mapping for a string, numeric, Boolean, or date field, you can also set a `null_value` that will be used whenever an explicit null value is encountered.

A field without a value will still be excluded from the inverted index.

When choosing a suitable `null_value`, ensure the following:

- It matches the field's type. You can't use a string `null_value` in a field of type date.
- It is different from the normal values that the field may contain, to avoid confusing real values with null values.

All About Caching

Earlier in this chapter, we briefly discussed how filters are calculated. At their heart is a bitset representing which documents match the filter. Elasticsearch aggressively caches these bitsets for later use. Once cached, these bitsets can be reused wherever the same filter is used, without having to reevaluate the entire filter again. These cached bitsets are “smart”: they are updated incrementally. As you index new documents, only those new documents need to be added to the existing bitsets, rather than having to recompute the entire cached filter over and over. Filters are real-time like the rest of the system; you don't need to worry about cache expiry.

Controlling Caching

Most leaf filters—those dealing directly with fields like the term filter—are cached, while compound filters, like the bool filter, are not.

Note: Leaf filters have to consult the inverted index on disk, so it makes sense to cache them. Compound filters, on the other hand, use fast bit logic to combine the bitsets resulting from their inner clauses, so it is efficient to recalculate them every time.

Certain leaf filters, however, are not cached by default, because it doesn't make sense to do so:

Script filters

The results from `script filters` cannot be cached because the meaning of the script is opaque to Elasticsearch.

Geo-filters

The geolocation filters, which we cover in more detail in [Part V](#), are usually used to filter results based on the geolocation of a specific user. Since each user has a unique geolocation, it is unlikely that geo-filters will be reused, so it makes no sense to cache them.

Date ranges

Date ranges that use the now function (for example "now-1h"), result in values accurate to the millisecond. Every time the filter is run, now returns a new time. Older filters will never be reused, so caching is disabled by default. However, when using now with rounding (for example, now/d rounds to the nearest day), caching is enabled by default.

Sometimes the default caching strategy is not correct. Perhaps you have a complicated bool expression that is reused several times in the same query. Or you have a filter on a date field that will never be reused. The default caching strategy can be overridden on almost any filter by setting the `_cache` flag:

```
{  
  "range" : {  
    "timestamp" : {  
      "gt" : "2014-01-02 16:15:14"  
    },  
    "_cache" : false  
  }  
}
```

Filter Order

The order of filters in a bool clause is important for performance. More-specific filters should be placed before less-specific filters in order to exclude as many documents as possible, as early as possible.

If Clause A could match 10 million documents, and Clause B could match only 100 documents, then Clause B should be placed before Clause A. Cached filters are very fast, so they should be placed before filters that are not cacheable. Imagine that we have an index that contains one month's worth of log events.

However, we're mostly interested only in log events from the previous hour:

```
GET /logs/2014-01/_search
```

```
{  
  "query" : {  
    "filtered" : {  
      "filter" : {  
        "range" : {  
          "timestamp" : {  
            "gt" : "2014-01-01T23:00:00Z"  
          }  
        }  
      }  
    }  
  }  
}
```

```
"timestamp" : {  
  "gt" : "now-1h"  
}  
}  
}  
}  
}  
}  
}
```

This filter is not cached because it uses the now function, the value of which changes every millisecond. That means that we have to examine one month's worth of log events every time we run this query!

We could make this much more efficient by combining it with a cached filter: we can exclude most of the month's data by adding a filter that uses a fixed point in time, such as midnight last night:

```
"bool": {  
  "must": [  
    { "range" : {  
      "timestamp" : {  
        "gt" : "now-1h/d" - This filter is cached because it uses now rounded to midnight.  
      }  
    }},  
    { "range" : {  
      "timestamp" : {  
        "gt" : "now-1h" - This filter is not cached because it uses now without rounding.  
      }  
    }  
  ]  
}
```

The now-1h/d clause rounds to the previous midnight and so excludes all documents created before today. The resulting bitset is cached because now is used with rounding, which means that it is executed only once a day, when the value for midnight-lastnight changes. The now-1h clause isn't cached because now produces a time accurate to the nearest millisecond. However, thanks to the first filter, this second filter need only check documents that have been created since midnight.

The order of these clauses is important. This approach works only because the sincemidnight clause comes before the last-hour clause. If they were the other way

around, then the last-hour clause would need to examine all documents in the index, instead of just documents created since midnight.

Full-Text Search

Now that we have covered the simple case of searching for structured data, it is time to explore full-text search: how to search within full-text fields in order to find the most relevant documents.

The two most important aspects of full-text search are as follows:

Relevance

The ability to rank results by how relevant they are to the given query, whether relevance is calculated using TF/IDF (see “[What Is Relevance?](#)”), proximity to a geolocation, fuzzy similarity, or some other algorithm.

Analysis

The process of converting a block of text into distinct, normalized tokens in order to (a) create an inverted index and (b) query the inverted index.

As soon as we talk about either relevance or analysis, we are in the territory of queries, rather than filters.

Term-Based Versus Full-Text

While all queries perform some sort of relevance calculation, not all queries have an analysis phase. Besides specialized queries like the bool or function_score queries, which don’t operate on text at all, textual queries can be broken down into two families:

Term-based queries

Queries like the term or fuzzy queries are low-level queries that have no analysis phase. They operate on a single term. A term query for the term Foo looks for that exact term in the inverted index and calculates the TF/IDF relevance _score for each document that contains the term.

It is important to remember that the term query looks in the inverted index for the exact term only; it won’t match any variants like foo or FOO. It doesn’t matter

how the term came to be in the index, just that it is. If you were to index ["Foo", "Bar"] into an exact value not_analyzed field, or Foo Bar into an analyzed field with the whitespace analyzer, both would result in having the two terms Foo and Bar in the inverted index.

Full-text queries

Queries like the match or query_string queries are high-level queries that understand the mapping of a field:

- If you use them to query a date or integer field, they will treat the querystring as a date or integer, respectively.
- If you query an exact value (`not_analyzed`) string field, they will treat the whole query string as a single term.
- But if you query a full-text (`analyzed`) field, they will first pass the query string through the appropriate analyzer to produce the list of terms to be queried.

Once the query has assembled a list of terms, it executes the appropriate lowlevel query for each of these terms, and then combines their results to produce the final relevance score for each document.

Note: If you do find yourself wanting to use a query on an exact value `not_analyzed` field, think about whether you really want a query or a filter. Single-term queries usually represent binary yes/no questions and are almost always better expressed as a filter, so that they can benefit from [filter caching](#):

```
GET /_search
{
  "query": {
    "filtered": {
      "filter": {
        "term": { "gender": "female" }
      }
    }
  }
}
```

The match Query

The match query is the go-to query—the first query that you should reach for whenever you need to query any field. It is a high-level full-text query, meaning that it knows how to deal with both full-text fields and exact-value fields.

That said, the main use case for the match query is for full-text search. So let's take a look at how full-text search works with a simple example.

Index Some Data

First, we'll create a new index and index some documents using the [bulk API](#):

```
DELETE /my_index
PUT /my_index
{ "settings": { "number_of_shards": 1 } }
POST /my_index/_bulk
```

```
{ "index": { "_id": 1 } }
{ "title": "The quick brown fox" }
{ "index": { "_id": 2 } }
{ "title": "The quick brown fox jumps over the lazy dog" }
{ "index": { "_id": 3 } }
{ "title": "The quick brown fox jumps over the quick dog" }
{ "index": { "_id": 4 } }
{ "title": "Brown fox brown dog" }
```

A Single-Word Query

Our first example explains what happens when we use the match query to search within a full-text field for a single word:

```
GET /my_index/my_type/_search
```

```
{
  "query": {
    "match": {
      "title": "QUICK!"
    }
  }
}
```

Elasticsearch executes the preceding match query as follows:

1. Check the field type.

a. The title field is a full-text (analyzed) string field, which means that the query string should be analyzed too.

2. Analyze the query string.

The query string QUICK! is passed through the standard analyzer, which results in the single term quick. Because we have just a single term, the match query can be executed as a single low-level term query.

3. Find matching docs.

The term query looks up quick in the inverted index and retrieves the list of documents that contain that term—in this case, documents 1, 2, and 3.

4. Score each doc.

The term query calculates the relevance _score for each matching document, by combining the term frequency (how often quick appears in the title field of each document), with the inverse document frequency (how often quick appears in the title field in all documents in the index), and the length of each field (shorter fields are considered more relevant).

This process gives us the following (abbreviated) results:

```
"hits": [
{
```

```
"_id": "1",
"_score": 0.5,
"_source": {
"title": "The quick brown fox"
},
},
{
"_id": "3",
"_score": 0.44194174,
"_source": {
"title": "The quick brown fox jumps over the quick dog"
},
},
{
"_id": "2",
"_score": 0.3125,
"_source": {
"title": "The quick brown fox jumps over the lazy dog"
}
}
]
```

Multiword Queries

If we could search for only one word at a time, full-text search would be pretty inflexible. Fortunately, the match query makes multiword queries just as simple:

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "BROWN DOG!"
    }
  }
}
```

The preceding query returns all four documents in the results list:

```
{
  "hits": [
    {
      "_id": "4",
      "_score": 0.73185337,
      "_source": {
```

```

    "title": "Brown fox brown dog"
  },
  },
  {
    "_id": "2",
    "_score": 0.47486103,
    "_source": {
      "title": "The quick brown fox jumps over the lazy dog"
    }
  },
  {
    "_id": "3",
    "_score": 0.47486103,
    "_source": {
      "title": "The quick brown fox jumps over the quick dog"
    }
  },
  {
    "_id": "1",
    "_score": 0.11914785,
    "_source": {
      "title": "The quick brown fox"
    }
  }
]
}

```

Because the match query has to look for two terms—["brown","dog"]—internally it has to execute two term queries and combine their individual results into the overall result.

Improving Precision

Matching any document that contains any of the query terms may result in a long tail of seemingly irrelevant results. It's a shotgun approach to search. Perhaps we want to show only documents that contain all of the query terms. In other words, instead of brown OR dog, we want to return only documents that match brown AND dog.

The match query accepts an operator parameter that defaults to or. You can change it to and to require that all specified terms must match:

`GET /my_index/my_type/_search`

```
{
}
```

```
"query": {  
  "match": {  
    "title": {  
      "query": "BROWN DOG!",  
      "operator": "and"  
    }  
  }  
}  
}  
}  
}
```

Controlling Precision

The choice between all and any is a bit too black-or-white. What if the user specified five query terms, and a document contains only four of them? Setting operator to and would exclude this document.

The match query supports the minimum_should_match parameter, which allows you to specify the number of terms that must match for a document to be considered relevant. While you can specify an absolute number of terms, it usually makes sense to specify a percentage instead, as you have no control over the number of words the user may enter:

```
GET /my_index/my_type/_search  
{  
  "query": {  
    "match": {  
      "title": {  
        "query": "quick brown dog",  
        "minimum_should_match": "75%"  
      }  
    }  
  }  
}
```

When specified as a percentage, minimum_should_match does the right thing: in the preceding example with three terms, 75% would be rounded down to 66.6%, or two out of the three terms. No matter what you set it to, at least one term must match for a document to be considered a match.

Combining Queries

Like the filter equivalent, the bool query accepts multiple query clauses under the must, must_not, and should parameters. For instance:

```
GET /my_index/my_type/_search
```

```
{
  "query": {
    "bool": {
      "must": { "match": { "title": "quick" } },
      "must_not": { "match": { "title": "lazy" } },
      "should": [
        { "match": { "title": "brown" } },
        { "match": { "title": "dog" } }
      ]
    }
  }
}
```

The results from the preceding query include any document whose title field contains the term quick, except for those that also contain lazy. So far, this is pretty similar to how the bool filter works.

The difference comes in with the two should clauses, which say that: a document is not required to contain either brown or dog, but if it does, then it should be considered more relevant:

```
{
  "hits": [
    {
      "_id": "3",
      "_score": 0.70134366,
      "_source": {
        "title": "The quick brown fox jumps over the quick dog"
      }
    },
    {
      "_id": "1",
      "_score": 0.3312608,
      "_source": {
        "title": "The quick brown fox"
      }
    }
  ]
}
```

Score Calculation

The bool query calculates the relevance _score for each document by adding together the _score from all of the matching must and should clauses, and then

dividing by the total number of must and should clauses.

The must_not clauses do not affect the score; their only purpose is to exclude documents that might otherwise have been included.

Controlling Precision

Just as we can control the precision of the match query, we can control how many should clauses need to match by using the minimum_should_match parameter, either as an absolute number or as a percentage:

```
GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "brown" } },
        { "match": { "title": "fox" } },
        { "match": { "title": "dog" } }
      ],
      "minimum_should_match": 2
    }
  }
}
```

The results would include only documents whose title field contains "brown" AND "fox", "brown" AND "dog", or "fox" AND "dog". If a document contains all three, it would be considered more relevant than those that contain just two of the three.

How match Uses bool

```
{
  "match": { "title": "brown fox" }
}
```

Equivalent to

```
{
  "bool": {
    "should": [
      { "term": { "title": "brown" } },
      { "term": { "title": "fox" } }
    ]
}
```

```
}
```

```
}
```

```
{  
  "match": {  
    "title": {  
      "query": "brown fox",  
      "operator": "and"  
    }  
  }  
}
```

Equivalent to

```
{  
  "bool": {  
    "must": [  
      { "term": { "title": "brown" }},  
      { "term": { "title": "fox" }}  
    ]  
  }  
}
```

```
{  
  "match": {  
    "title": {  
      "query": "quick brown fox",  
      "minimum_should_match": "75%"  
    }  
  }  
}
```

Equivalent to

```
{  
  "bool": {  
    "should": [  
      { "term": { "title": "brown" }},  
      { "term": { "title": "fox" }},  
      { "term": { "title": "quick" }}  
],  
    "minimum_should_match": 2  
}
```

```
}
```

Boosting Query Clauses

Of course, the bool query isn't restricted to combining simple one-word match queries.

It can combine any other query, including other bool queries. It is commonly used to fine-tune the relevance _score for each document by combining the scores from several distinct queries.

Imagine that we want to search for documents about "full-text search," but we want to give more weight to documents that also mention "Elasticsearch" or "Lucene." By more weight, we mean that documents mentioning "Elasticsearch" or "Lucene" will receive a higher relevance _score than those that don't, which means that they will appear higher in the list of results.

A simple bool query allows us to write this fairly complex logic as follows:

```
GET /_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": {
            "query": "full text search",
            "operator": "and"
          }
        }
      },
      "should": [
        { "match": { "content": "Elasticsearch" } },
        { "match": { "content": "Lucene" } }
      ]
    }
  }
}
```

The more should clauses that match, the more relevant the document. So far, so good. But what if we want to give more weight to the docs that contain Lucene and even more weight to the docs containing Elasticsearch?

We can control the relative weight of any query clause by specifying a boost value, which defaults to 1. A boost value greater than 1 increases the relative weight of that clause. So we could rewrite the preceding query as follows:

```

GET /_search
{
  "query": {
    "bool": {
      "must": [
        "match": {
          "content": {
            "query": "full text search",
            "operator": "and"
          }
        }
      ],
      "should": [
        { "match": {
          "content": {
            "query": "Elasticsearch",
            "boost": 3
          }
        }},
        { "match": {
          "content": {
            "query": "Lucene",
            "boost": 2
          }
        }}
      ]
    }
  }
}

```

Note: The boost parameter is used to increase the relative weight of a clause (with a boost greater than 1) or decrease the relative weight (with a boost between 0 and 1), but the increase or decrease is not linear. In other words, a boost of 2 does not result in double the _score.

Instead, the new _score is normalized after the boost is applied. Each type of query has its own normalization algorithm, and the details are beyond the scope of this book. Suffice to say that a higher boost value results in a higher _score.

If you are implementing your own scoring model not based on TF/IDF and you need more control over the boosting process, you can use the `function_score` query to manipulate a document's boost without the normalization step.

Default Analyzers

While we can specify an analyzer at the field level, how do we determine which analyzer is used for a field if none is specified at the field level?

Analyzers can be specified at several levels. Elasticsearch works through each level until it finds an analyzer that it can use. At index time, the order is as follows:

- The analyzer defined in the field mapping, else
- The analyzer defined in the `_analyzer` field of the document, else
- The default analyzer for the type, which defaults to
- The analyzer named `default` in the index settings, which defaults to
- The analyzer named `default` at node level, which defaults to
- The standard analyzer

At search time, the sequence is slightly different:

- The `analyzer` defined in the query itself, else
- The analyzer defined in the field mapping, else
- The default analyzer for the type, which defaults to
- The analyzer named `default` in the index settings, which defaults to
- The analyzer named `default` at node level, which defaults to
- The standard analyzer

Multifield Search

Queries are seldom simple one-clause match queries. We frequently need to search for the same or different query strings in one or more fields, which means that we need to be able to combine multiple query clauses and their relevance scores in a way that makes sense.

Multiple Query Strings

The simplest multifield query to deal with is the one where we can map search terms to specific fields. If we know that War and Peace is the title, and Leo Tolstoy is the author, it is easy to write each of these conditions as a match clause and to combine them with a `bool query`:

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "War and Peace" }},
        { "match": { "author": "Leo Tolstoy" }}
      ]
    }
  }
}
```

```
}
```

The bool query takes a more-matches-is-better approach, so the score from each match clause will be added together to provide the final _score for each document. Documents that match both clauses will score higher than documents that match just one clause.

Of course, you're not restricted to using just match clauses: the bool query can wrap any other query type, including other bool queries. We could add a clause to specify that we prefer to see versions of the book that have been translated by specific translators:

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "War and Peace" }},
        { "match": { "author": "Leo Tolstoy" }},
        { "bool": {
          "should": [
            { "match": { "translator": "Constance Garnett" }},
            { "match": { "translator": "Louise Maude" }}
          ]
        }}
      ]
    }
  }
}
```

Prioritizing Clauses

The simplest weapon in our tuning arsenal is the boost parameter. To increase the weight of the title and author fields, give them a boost value higher than 1:

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": {
          "title": {
            "query": "War and Peace",
            "boost": 2
          }}
        },
        { "match": {}}
```

```
"author": {  
  "query": "Leo Tolstoy",  
  "boost": 2  
}},  
{ "bool": {  
  "should": [  
    { "match": { "translator": "Constance Garnett" }},  
    { "match": { "translator": "Louise Maude" }}  
  ]  
}  
}  
]  
}  
}  
}  
}
```

The “best” value for the boost parameter is most easily determined by trial and error: set a boost value, run test queries, repeat. A reasonable range for boost lies between 1 and 10, maybe 15. Boosts higher than that have little more impact because scores are [normalized](#).

Single Query String

The bool query is the mainstay of mult clause queries. It works well for many cases, especially when you are able to map different query strings to individual fields.

The problem is that, these days, users expect to be able to type all of their search terms into a single field, and expect that the application will figure out how to give them the right results. It is ironic that the multifield search form is known as Advanced Search—it may appear advanced to the user, but it is much simpler to implement.

There is no simple one-size-fits-all approach to multiword, multifield queries. To get the best results, you have to know your data and know how to use the appropriate tools.

Know Your Data

When your only user input is a single query string, you will encounter three scenarios frequently:

Best fields

When searching for words that represent a concept, such as “brown fox,” the words mean more together than they do individually. Fields like the title and body, while related, can be considered to be in competition with each other.

Documents should have as many words as possible in the same field, and the score should come from the best-matching field.

Most fields

A common technique for fine-tuning relevance is to index the same data into multiple fields, each with its own analysis chain. The main field may contain words in their stemmed form, synonyms, and words stripped of their diacritics, or accents. It is used to match as many documents as possible.

The same text could then be indexed in other fields to provide more-precise matching. One field may contain the unstemmed version, another the original word with accents, and a third might use shingles to provide information about **word proximity**.

These other fields act as signals to increase the relevance score of each matching document. The more fields that match, the better.

Cross fields

For some entities, the identifying information is spread across multiple fields, each of which contains just a part of the whole:

- Person: first_name and last_name
- Book: title, author, and description
- Address: street, city, country, and postcode

In this case, we want to find as many words as possible in any of the listed fields. We need to search across multiple fields as if they were one big field.

All of these are multiword, multifield queries, but each requires a different strategy. We will examine each strategy in turn in the rest of this chapter.

Best Fields

Imagine that we have a website that allows users to search blog posts, such as these two documents:

```
PUT /my_index/my_type/1
{
  "title": "Quick brown rabbits",
  "body": "Brown rabbits are commonly seen."
}
PUT /my_index/my_type/2
{
  "title": "Keeping pets healthy",
  "body": "My quick brown fox eats rabbits on a regular basis."
}
```

The user types in the words “Brown fox” and clicks Search. We don’t know ahead of time if the user’s search terms will be found in the title or the body field of the post, but it is likely that the user is searching for related words. To

our eyes, document 2 appears to be the better match, as it contains both words that we are looking for.

Now we run the following bool query:

```
{  
  "query": {  
    "bool": {  
      "should": [  
        { "match": { "title": "Brown fox" }},  
        { "match": { "body": "Brown fox" }}  
      ]  
    }  
  }  
}
```

And we find that this query gives document 1 the higher score:

```
{  
  "hits": [  
    {  
      "_id": "1",  
      "_score": 0.14809652,  
      "_source": {  
        "title": "Quick brown rabbits",  
        "body": "Brown rabbits are commonly seen."  
      }  
    },  
    {  
      "_id": "2",  
      "_score": 0.09256032,  
      "_source": {  
        "title": "Keeping pets healthy",  
        "body": "My quick brown fox eats rabbits on a regular basis."  
      }  
    }  
  ]  
}
```

To understand why, think about how the bool query calculates its score:

1. It runs both of the queries in the should clause.
2. It adds their scores together.
3. It multiplies the total by the number of matching clauses.
4. It divides the result by the total number of clauses (two).

Document 1 contains the word brown in both fields, so both match clauses are successful

and have a score. Document 2 contains both brown and fox in the body field but neither word in the title field. The high score from the body query is added to the zero score from the title query, and multiplied by one-half, resulting in a lower overall score than for document 1.

In this example, the title and body fields are competing with each other. We want to find the single best-matching field. What if, instead of combining the scores from each field, we used the score from the best-matching field as the overall score for the query? This would give preference to a single field that contains both of the words we are looking for, rather than the same word repeated in different fields.

dis_max Query

Instead of the bool query, we can use the dis_max or Disjunction Max Query. Disjunction means or (while conjunction means and) so the Disjunction Max Query simply means return documents that match any of these queries, and return the score of the best matching query:

```
{  
  "query": {  
    "dis_max": {  
      "queries": [  
        { "match": { "title": "Brown fox" }},  
        { "match": { "body": "Brown fox" }}  
      ]  
    }  
  }  
}
```

This produces the results that we want:

```
{  
  "hits": [  
    {  
      "_id": "2",  
      "_score": 0.21509302,  
      "_source": {  
        "title": "Keeping pets healthy",  
        "body": "My quick brown fox eats rabbits on a regular basis."  
      }  
    },  
    {  
      "_id": "1",  
      "_score": 0.12713557,  
      "_source": {  
        "title": "Quick brown rabbits",  
        "body": "A quick brown fox jumps over the lazy dog."  
      }  
    }  
  ]  
}
```

```
        "body": "Brown rabbits are commonly seen."
    }
}
]
}
```

Tuning Best Fields Queries

What would happen if the user had searched instead for “quick pets”? Both documents contain the word quick, but only document 2 contains the word pets. Neither document contains both words in the same field.

A simple dis_max query like the following would choose the single best matching field, and ignore the other:

```
{
  "query": {
    "dis_max": {
      "queries": [
        { "match": { "title": "Quick pets" }},
        { "match": { "body": "Quick pets" }}
      ]
    }
  }
}

{
  "hits": [
    {
      "_id": "1",
      "_score": 0.12713557,
      "_source": {
        "title": "Quick brown rabbits",
        "body": "Brown rabbits are commonly seen."
      }
    },
    {
      "_id": "2",
      "_score": 0.12713557,
      "_source": {
        "title": "Keeping pets healthy",
        "body": "My quick brown fox eats rabbits on a regular basis."
      }
    }
  ]
}
```

Note that the scores are exactly the same.

We would probably expect documents that match on both the title field and the body field to rank higher than documents that match on just one field, but this isn't the case. Remember: the dis_max query simply uses the _score from the single bestmatching clause.

tie_breaker

It is possible, however, to also take the _score from the other matching clauses into account, by specifying the tie_breaker parameter:

```
{  
  "query": {  
    "dis_max": {  
      "queries": [  
        { "match": { "title": "Quick pets" }},  
        { "match": { "body": "Quick pets" }}  
      ],  
      "tie_breaker": 0.3  
    }  
  }  
}
```

This gives us the following results:

```
{  
  "hits": [  
    {  
      "_id": "2",  
      "_score": 0.14757764,  
      "_source": {  
        "title": "Keeping pets healthy",  
        "body": "My quick brown fox eats rabbits on a regular basis."  
      }  
    },  
    {  
      "_id": "1",  
      "_score": 0.124275915,  
      "_source": {  
        "title": "Quick brown rabbits",  
        "body": "Brown rabbits are commonly seen."  
      }  
    }  
  ]  
}
```

Document 2 now has a small lead over document 1.

The tie_breaker parameter makes the dis_max query behave more like a halfway house between dis_max and bool. It changes the score calculation as follows:

1. Take the _score of the best-matching clause.
2. Multiply the score of each of the other matching clauses by the tie_breaker.
3. Add them all together and normalize.

With the tie_breaker, all matching clauses count, but the best-matching clause counts most.

Note: The tie_breaker can be a floating-point value between 0 and 1, where 0 uses just the best-matching clause and 1 counts all matching clauses equally. The exact value can be tuned based on your data and queries, but a reasonable value should be close to zero, (for example, 0.1 – 0.4), in order not to overwhelm the bestmatching nature of dis_max.

multi_match Query

The multi_match query provides a convenient shorthand way of running the same query against multiple fields.

By default, this query runs as type best_fields, which means that it generates a match query for each field and wraps them in a dis_max query. This dis_max query

```
{  
  "dis_max": {  
    "queries": [  
      {  
        "match": {  
          "title": {  
            "query": "Quick brown fox",  
            "minimum_should_match": "30%"  
          }  
        }  
      },  
      {  
        "match": {  
          "body": {  
            "query": "Quick brown fox",  
            "minimum_should_match": "30%"  
          }  
        }  
      },  
      {  
        "match": {  
          "text": {  
            "query": "Quick brown fox",  
            "minimum_should_match": "30%"  
          }  
        }  
      },  
      {  
        "match": {  
          "category": {  
            "query": "Quick brown fox",  
            "minimum_should_match": "30%"  
          }  
        }  
      }  
    ]  
  }  
}
```

```
"tie_breaker": 0.3
}
}
```

could be rewritten more concisely with multi_match as follows:

```
{
"multi_match": {
"query": "Quick brown fox",
"type": "best_fields",
"fields": [ "title", "body" ],
"tie_breaker": 0.3,
"minimum_should_match": "30%"
}
}
```

Using Wildcards in Field Names

Field names can be specified with wildcards: any field that matches the wildcard pattern will be included in the search. You could match on the book_title, chapter_title, and section_title fields, with the following:

```
{
"multi_match": {
"query": "Quick brown fox",
"fields": "*_title"
}
}
```

Boosting Individual Fields

Individual fields can be boosted by using the caret (^) syntax: just add ^boost after the field name, where boost is a floating-point number:

```
{
"multi_match": {
"query": "Quick brown fox",
"fields": [ "*_title", "chapter_title^2" ]
}
}
```

The chapter_title field has a boost of 2, while the book_title and section_title fields have a default boost of 1.

Multifield Mapping

The first thing to do is to set up our field to be indexed twice: once in a stemmed form and once in an unstemmed form.

```

DELETE /my_index
PUT /my_index
{
  "settings": { "number_of_shards": 1 },
  "mappings": {
    "my_type": {
      "properties": {
        "title": {
          "type": "string",
          "analyzer": "english",
          "fields": {
            "std": {
              "type": "string",
              "analyzer": "standard"
            }
          }
        }
      }
    }
  }
}

```

Next we index some documents:

```

PUT /my_index/my_type/1
{ "title": "My rabbit jumps" }
PUT /my_index/my_type/2
{ "title": "Jumping jack rabbits" }

```

Here is a simple match query on the title field for jumping rabbits:

```

GET /my_index/_search
{
  "query": {
    "match": {
      "title": "jumping rabbits"
    }
  }
}

```

This becomes a query for the two stemmed terms jump and rabbit, thanks to the english analyzer. The title field of both documents contains both of those terms, so

both documents receive the same score:

```
{  
  "hits": [  
    {  
      "_id": "1",  
      "_score": 0.42039964,  
      "_source": {  
        "title": "My rabbit jumps"  
      }  
    },  
    {  
      "_id": "2",  
      "_score": 0.42039964,  
      "_source": {  
        "title": "Jumping jack rabbits"  
      }  
    }  
  ]  
}
```

If we were to query just the title.std field, then only document 2 would match. However, if we were to query both fields and to combine their scores by using the bool query, then both documents would match (thanks to the title field) and document 2 would score higher (thanks to the title.std field):

```
GET /my_index/_search  
{  
  "query": {  
    "multi_match": {  
      "query": "jumping rabbits",  
      "type": "most_fields",  
      "fields": [ "title", "title.std" ]  
    }  
  }  
}
```

We want to combine the scores from all matching fields, so we use the most_fields type. This causes the multi_match query to wrap the two field clauses in a bool query instead of a dis_max query.

```
{  
  "hits": [  
    {
```

```
{
  "_id": "2",
  "_score": 0.8226396,
  "_source": {
    "title": "Jumping jack rabbits"
  }
},
{
  "_id": "1",
  "_score": 0.10741998,
  "_source": {
    "title": "My rabbit jumps"
  }
}
]
```

We are using the broad-matching title field to include as many documents as possible—to increase recall—but we use the title.std field as a signal to push the most relevant results to the top.

The contribution of each field to the final score can be controlled by specifying custom boost values. For instance, we could boost the title field to make it the most important field, thus reducing the effect of any other signal fields:

```
GET /my_index/_search
{
  "query": {
    "multi_match": {
      "query": "jumping rabbits",
      "type": "most_fields",
      "fields": [ "title^10", "title.std" ]
    }
  }
}
```

Cross-fields Entity Search

Now we come to a common pattern: cross-fields entity search. With entities like person, product, or address, the identifying information is spread across several fields.

We may have a person indexed as follows:

```
{
```

```
"firstname": "Peter",
"lastname": "Smith"
}

Or an address like this:
{

"street": "5 Poland Street",
"city": "London",
"country": "United Kingdom",
"postcode": "W1V 3DG"
}
```

Our user might search for the person “Peter Smith” or for the address “Poland Street W1V.” Each of those words appears in a different field, so using a `dis_max` /`best_fields` query to find the single best-matching field is clearly the wrong approach.

A Naive Approach Really, we want to query each field in turn and add up the scores of every field that matches, which sounds like a job for the `bool` query:

```
{
  "query": {
    "bool": {
      "should": [
        { "match": { "street": "Poland Street W1V" }},
        { "match": { "city": "Poland Street W1V" }},
        { "match": { "country": "Poland Street W1V" }},
        { "match": { "postcode": "Poland Street W1V" }}
      ]
    }
  }
}
```

Repeating the query string for every field soon becomes tedious. We can use the `multi_match` query instead, and set the type to `most_fields` to tell it to combine the scores of all matching fields:

```
{
  "query": {
    "multi_match": {
      "query": "Poland Street W1V",
      "type": "most_fields",
      "fields": [ "street", "city", "country", "postcode" ]
    }
  }
}
```

Problems with the most_fields Approach

The most_fields approach to entity search has some problems that are not immediately obvious:

- It is designed to find the most fields matching any words, rather than to find the most matching words across all fields.
- It can't use the operator or minimum_should_match parameters to reduce the long tail of less-relevant results.
- Term frequencies are different in each field and could interfere with each other to produce badly ordered results.

Field-Centric Queries

All three of the preceding problems stem from most_fields being field-centric rather than term-centric: it looks for the most matching fields, when really what we're interested is the most matching terms.

Note: The best_fields type is also field-centric and suffers from similar problems.

First we'll look at why these problems exist, and then how we can combat them.

Problem 1: Matching the Same Word in Multiple Fields

Think about how the most_fields query is executed: Elasticsearch generates a separate match query for each field and then wraps these match queries in an outer bool query.

We can see this by passing our query through the validate-query API:

```
GET /_validate/query?explain
{
  "query": {
    "multi_match": {
      "query": "Poland Street W1V",
      "type": "most_fields",
      "fields": [ "street", "city", "country", "postcode" ]
    }
  }
}
```

which yields this explanation:

```
(street:poland street:street street:w1v)
(city:poland city:street city:w1v)
(country:poland country:street country:w1v)
(postcode:poland postcode:street postcode:w1v)
```

You can see that a document matching just the word poland in two fields could score higher than a document matching poland and street in one field.

Problem 2: Trimming the Long Tail

In “[Controlling Precision](#)”, we talked about using the and operator or the minimum_should_match parameter to trim the long tail of almost irrelevant results.

Perhaps we could try this:

```
{  
  "query": {  
    "multi_match": {  
      "query": "Poland Street W1V",  
      "type": "most_fields",  
      "operator": "and",  
      "fields": [ "street", "city", "country", "postcode" ]  
    }  
  }  
}
```

All terms must be present.

However, with best_fields or most_fields, these parameters are passed down to the generated match queries. The explanation for this query shows the following:

```
( +street:poland +street:street +street:w1v)  
( +city:poland +city:street +city:w1v)  
( +country:poland +country:street +country:w1v)  
( +postcode:poland +postcode:street +postcode:w1v)
```

In other words, using the and operator means that all words must exist in the same field, which is clearly wrong! It is unlikely that any documents would match this query.

Problem 3: Term Frequencies

In “[What Is Relevance?](#)”, we explained that the default similarity algorithm used to calculate the relevance score for each term is TF/IDF:

Term frequency – The more often a term appears in a field in a single document, the more relevant the document.

Inverse document frequency

The more often a term appears in a field in all documents in the index, the less relevant is that term.

When searching against multiple fields, TF/IDF can introduce some surprising results.

Consider our example of searching for “Peter Smith” using the first_name and last_name fields. Peter is a common first name and Smith is a common last name— both will have low IDFs. But what if we have another person in the index

whose name is Smith Williams? Smith as a first name is very uncommon and so will have a high IDF!

A simple query like the following may well return Smith Williams above Peter Smith in spite of the fact that the second person is a better match than the first.

```
{  
  "query": {  
    "multi_match": {  
      "query": "Peter Smith",  
      "type": "most_fields",  
      "fields": [ "*_name" ]  
    }  
  }  
}
```

The high IDF of smith in the first name field can overwhelm the two low IDFs of peter as a first name and smith as a last name.

Solution

These problems only exist because we are dealing with multiple fields. If we were to combine all of these fields into a single field, the problems would vanish. We could achieve this by adding a full_name field to our person document:

```
{  
  "first_name": "Peter",  
  "last_name": "Smith",  
  "full_name": "Peter Smith"  
}
```

When querying just the full_name field:

- Documents with more matching words would trump documents with the same word repeated.
- The minimum_should_match and operator parameters would function as expected.
- The inverse document frequencies for first and last names would be combined so it wouldn't matter whether Smith were a first or last name anymore.

While this would work, we don't like having to store redundant data. Instead, Elasticsearch offers us two solutions—one at index time and one at search time—which we discuss next.

Proximity Matching

This is the province of phrase matching, or proximity matching.

Phrase Matching

In the same way that the match query is the go-to query for standard full-text search, the match_phrase query is the one you should reach for when you want to find words that are near each other:

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": "quick brown fox"
    }
  }
}
```

Like the match query, the match_phrase query first analyzes the query string to produce a list of terms. It then searches for all the terms, but keeps only documents that contain all of the search terms, in the same positions relative to each other. A query for the phrase quick fox would not match any of our documents, because no document contains the word quick immediately followed by fox.

Note: The match_phrase query can also be written as a match query with type phrase:

```
"match": {
  "title": {
    "query": "quick brown fox",
    "type": "phrase"
  }
}
```

Term Positions

When a string is analyzed, the analyzer returns not only a list of terms, but also the position, or order, of each term in the original string:

```
GET /_analyze?analyzer=standard
```

```
Quick brown fox
```

This returns the following:

```
{
  "tokens": [
    {
      "token": "quick",
      "start_offset": 0,
      "end_offset": 5,
      "type": "<ALPHANUM>",
    }
  ]
}
```

```
"position": 1
},
{
"token": "brown",
"start_offset": 6,
"end_offset": 11,
"type": "<ALPHANUM>",
"position": 2
},
{
"token": "fox",
"start_offset": 12,
"end_offset": 15,
"type": "<ALPHANUM>",
"position": 3
}
]
}
```

The position of each term in the original string.

Positions can be stored in the inverted index, and position-aware queries like the `match_phrase` query can use them to match only documents that contain all the words in exactly the order specified, with no words in-between.

What Is a Phrase

For a document to be considered a match for the phrase “quick brown fox,” the following must be true:

- quick, brown, and fox must all appear in the field.
- The position of brown must be 1 greater than the position of quick.
- The position of fox must be 2 greater than the position of quick.

If any of these conditions is not met, the document is not considered a match.

Note: Internally, the `match_phrase` query uses the low-level span query family to do position-aware matching. Span queries are term-level queries, so they have no analysis phase; they search for the exact term specified.

Thankfully, most people never need to use the span queries directly, as the `match_phrase` query is usually good enough. However, certain specialized fields, like patent searches, use these lowlevel queries to perform very specific, carefully constructed positional searches.

Mixing It Up

Requiring exact-phrase matches may be too strict a constraint. Perhaps we do want documents that contain “quick brown fox” to be considered a match for the query “quick fox,” even though the positions aren’t exactly equivalent.

We can introduce a degree of flexibility into phrase matching by using the slop parameter:

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": {
        "query": "quick fox",
        "slop": 1
      }
    }
  }
}
```

The slop parameter tells the match_phrase query how far apart terms are allowed to be while still considering the document a match. By how far apart we mean how many times do you need to move a term in order to make the query and document match?

We’ll start with a simple example. To make the query quick fox match a document containing quick brown fox we need a slop of just 1:

Pos 1 Pos 2 Pos 3

Doc: quick brown fox

Query: quick fox

Slop 1: quick fox

Although all words need to be present in phrase matching, even when using slop, the words don’t necessarily need to be in the same sequence in order to match. With a high enough slop value, words can be arranged in any order.

To make the query fox quick match our document, we need a slop of 3:

Pos 1 Pos 2 Pos 3

Doc: quick brown fox

Query: fox quick

Slop 1: fox\quick

Slop 2: quick ↴ fox

Slop 3: quick ↴ fox

Note that fox and quick occupy the same position in this step. Switching word order from fox quick to quick fox thus requires two steps, or a slop of 2.

Multivalue Fields

A curious thing can happen when you try to use phrase matching on multivalue fields. Imagine that you index this document:

```
PUT /my_index/groups/1
{
  "names": [ "John Abraham", "Lincoln Smith" ]
}
```

Then run a phrase query for Abraham Lincoln:

```
GET /my_index/groups/_search
{
  "query": {
    "match_phrase": {
      "names": "Abraham Lincoln"
    }
  }
}
```

Surprisingly, our document matches, even though Abraham and Lincoln belong to two different people in the names array. The reason for this comes down to the way arrays are indexed in Elasticsearch.

When John Abraham is analyzed, it produces this:

- Position 1: john
- Position 2: abraham

Then when Lincoln Smith is analyzed, it produces this:

- Position 3: lincoln
- Position 4: smith

In other words, Elasticsearch produces exactly the same list of tokens as it would have for the single string John Abraham Lincoln Smith. Our example query looks for abraham directly followed by lincoln, and these two terms do indeed exist, and they are right next to each other, so the query matches.

Fortunately, there is a simple workaround for cases like these, called the position_offset_gap, which we need to configure in the field mapping:

```
DELETE /my_index/groups/
PUT /my_index/_mapping/groups
{
  "properties": {
    "names": {
      "type": "string",
```

```
"position_offset_gap": 100  
}  
}  
}
```

First delete the groups mapping and all documents of that type.

Then create a new groups mapping with the correct values.

The position_offset_gap setting tells Elasticsearch that it should increase the current term position by the specified value for every new array element. So now, when we index the array of names, the terms are emitted with the following positions:

- Position 1: john
- Position 2: abraham
- Position 103: lincoln
- Position 104: smith

Our phrase query would no longer match a document like this because abraham and lincoln are now 100 positions apart. You would have to add a slop value of 100 in order for this document to match.

Improving Performance

Phrase and proximity queries are more expensive than simple match queries.

Whereas a match query just has to look up terms in the inverted index, a match_phrase query has to calculate and compare the positions of multiple possibly repeated terms.

The [Lucene nightly benchmarks](#) show that a simple term query is about 10 times as fast as a phrase query, and about 20 times as fast as a proximity query (a phrase query with slop). And of course, this cost is paid at search time instead of at index time.

Note: Usually the extra cost of phrase queries is not as scary as these numbers suggest. Really, the difference in performance is a testimony to just how fast a simple term query is. Phrase queries on typical full-text data usually complete within a few milliseconds, and are perfectly usable in practice, even on a busy cluster.

In certain pathological cases, phrase queries can be costly, but this is unusual. An example of a pathological case is DNA sequencing, where there are many many identical terms repeated in many positions. Using higher slop values in this case results in a huge growth in the number of position calculations.

So what can we do to limit the performance cost of phrase and proximity queries? One useful approach is to reduce the total number of documents that need to be examined by the phrase query.

Rescoring Results

In [the preceding section](#), we discussed using proximity queries just for relevance purposes, not to include or exclude results from the result set. A query may match millions of results, but chances are that our users are interested in only the first few pages of results.

A simple match query will already have ranked documents that contain all search terms near the top of the list. Really, we just want to rerank the top results to give an extra relevance bump to those documents that also match the phrase query.

The search API supports exactly this functionality via rescoring. The rescore phase allows you to apply a more expensive scoring algorithm—like a phrase query—to just the top K results from each shard. These top results are then resorted according to their new scores.

The request looks like this:

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": {
        "query": "quick brown fox",
        "minimum_should_match": "30%"
      }
    }
  },
  "rescore": {
    "window_size": 50,
    "query": {
      "rescore_query": {
        "match_phrase": {
          "title": {
            "query": "quick brown fox",
            "slop": 50
          }
        }
      }
    }
  }
}
```

The match query decides which results will be included in the final result set and ranks results according to TF/IDF.

The window_size is the number of top results to rescore, per shard.

The only rescoring algorithm currently supported is another query, but there are plans to add more algorithms later.

This is a theme that you will encounter frequently in Elasticsearch: enables you to achieve a lot at search time, without requiring any up-front setup. Once you understand your requirements more clearly, you can achieve better results with better performance by modeling your data correctly at index time.

Partial Matching

A keen observer will notice that all the queries so far in this book have operated on whole terms. To match something, the smallest unit had to be a single term. You can find only terms that exist in the inverted index.

But what happens if you want to match parts of a term but not the whole thing? Partial matching allows users to specify a portion of the term they are looking for and find any words that contain that fragment.

The requirement to match on part of a term is less common in the full-text searchengine world than you might think. If you have come from an SQL background, you likely have, at some stage of your career, implemented a poor man's full-text search using SQL constructs like this:

```
WHERE text LIKE "*quick*"
```

```
AND text LIKE "*brown*"
```

```
AND text LIKE "*fox*"
```

fox would match “fox” and “foxes.”

Of course, with Elasticsearch, we have the analysis process and the inverted index that remove the need for such brute-force techniques. To handle the case of matching both “fox” and “foxes,” we could simply use a stemmer to index words in their root form. There is no need to match partial terms.

That said, on some occasions partial matching can be useful. Common use cases include the following:

- Matching postal codes, product serial numbers, or other not_analyzed values that start with a particular prefix or match a wildcard pattern or even a regular expression
- search-as-you-type—displaying the most likely results before the user has finished typing the search terms
- Matching in languages like German or Dutch, which contain long compound words, like Weltgesundheitsorganisation (World Health Organization) We will start by examining prefix matching on exact-value not_analyzed fields.

Postcodes and Structured Data

We will use United Kingdom postcodes (postal codes in the United States) to illustrate how to use partial matching with structured data. UK postcodes have a welldefined structure. For instance, the postcode W1V 3DG can be broken down as follows:

W1V: This outer part identifies the postal area and district:

—W indicates the area (one or two letters)

—1V indicates the district (one or two numbers, possibly followed by a letter)

3DG: This inner part identifies a street or building:

—3 indicates the sector (one number)

—DG indicates the unit (two letters)

Let's assume that we are indexing postcodes as exact-value `not_analyzed` fields, so we could create our index as follows:

```
PUT /my_index
{
  "mappings": {
    "address": {
      "properties": {
        "postcode": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

And index some postcodes:

```
PUT /my_index/address/1
{ "postcode": "W1V 3DG" }
PUT /my_index/address/2
{ "postcode": "W2F 8HW" }
PUT /my_index/address/3
{ "postcode": "W1F 7HW" }
PUT /my_index/address/4
{ "postcode": "WC1N 1LZ" }
PUT /my_index/address/5
{ "postcode": "SW5 0BE" }
```

Now our data is ready to be queried.

prefix Query

To find all postcodes beginning with W1, we could use a simple prefix query:

```
GET /my_index/address/_search
```

```
{
  "query": {
    "prefix": {
      "postcode": "W1"
    }
  }
}
```

The prefix query is a low-level query that works at the term level. It doesn't analyze the query string before searching. It assumes that you have passed it the exact prefix that you want to find.

Note: By default, the prefix query does no relevance scoring. It just finds matching documents and gives them all a score of 1. Really, it behaves more like a filter than a query. The only practical difference between the prefix query and the prefix filter is that the filter can be cached.

Previously, we said that “you can find only terms that exist in the inverted index,” but we haven’t done anything special to index these postcodes; each postcode is simply indexed as the exact value specified in each document. So how does the prefix query work?

Remember that the inverted index consists of a sorted list of unique terms (in this case, postcodes). For each term, it lists the IDs of the documents containing that term in the postings list. The inverted index for our example documents looks something like this:

Term:	Doc IDs:
"SW5 0BE"	5
"W1F 7HW"	3
"W1V 3DG"	1
"W2F 8HW"	2
"WC1N 1LZ"	4

To support prefix matching on the fly, the query does the following:

1. Skips through the terms list to find the first term beginning with W1.
2. Collects the associated document IDs.
3. Moves to the next term.
4. If that term also begins with W1, the query repeats from step 2; otherwise, we’re finished.

While this works fine for our small example, imagine that our inverted index contains a million postcodes beginning with W1. The prefix query would need to visit all one million terms in order to calculate the result!

And the shorter the prefix, the more terms need to be visited. If we were to look for the prefix W instead of W1, perhaps we would match 10 million terms instead of just one million.

Note: The prefix query or filter are useful for ad hoc prefix matching, but should be used with care. They can be used freely on fields with a small number of terms, but they scale poorly and can put your cluster under a lot of strain. Try to limit their impact on your cluster by using a long prefix; this reduces the number of terms that need to be visited.

Later in this chapter, we present an alternative index-time solution that makes prefix matching much more efficient. But first, we'll take a look at two related queries: the wildcard and regexp queries.

wildcard and regexp Queries

The wildcard query is a low-level, term-based query similar in nature to the prefix query, but it allows you to specify a pattern instead of just a prefix. It uses the standard shell wildcards: ? matches any character, and * matches zero or more characters.

This query would match the documents containing W1F 7HW and W2F 8HW:

```
GET /my_index/address/_search
{
  "query": {
    "wildcard": {
      "postcode": "W?F*HW"
    }
  }
}
```

The ? matches the 1 and the 2, while the * matches the space and the 7 and 8.

Imagine now that you want to match all postcodes just in the W area. A prefix match would also include postcodes starting with WC, and you would have a similar problem with a wildcard match. We want to match only postcodes that begin with a W, followed by a number. The regexp query allows you to write these more complicated patterns:

```
GET /my_index/address/_search
{
  "query": {
```

```
"regexp": {  
  "postcode": "W[0-9].+"  
}  
}  
}
```

The regular expression says that the term must begin with a W, followed by any number from 0 to 9, followed by one or more other characters. The wildcard and regexp queries work in exactly the same way as the prefix query.

They also have to scan the list of terms in the inverted index to find all matching terms, and gather document IDs term by term. The only difference between them and the prefix query is that they support more-complex patterns.

This means that the same caveats apply. Running these queries on a field with many unique terms can be resource intensive indeed. Avoid using a pattern that starts with a wildcard (for example, *foo or, as a regexp, .*foo).

Whereas prefix matching can be made more efficient by preparing your data at index time, wildcard and regular expression matching can be done only at query time. These queries have their place but should be used sparingly.

Note: The prefix, wildcard, and regexp queries operate on terms. If you use them to query an analyzed field, they will examine each term in the field, not the field as a whole. For instance, let's say that our title field contains "Quick brown fox" which produces the terms quick, brown, and fox.

This query would match:

```
{ "regexp": { "title": "br.*" } }
```

But neither of these queries would match:

```
{ "regexp": { "title": "Qu.*" } } - The term in the index is quick, not Quick.
```

```
{ "regexp": { "title": "quick br*" } } - quick and brown are separate terms.
```

Getting Started with Languages

Elasticsearch ships with a collection of language analyzers that provide good, basic, out-of-the-box support for many of the world's most common languages:

Arabic, Armenian, Basque, Brazilian, Bulgarian, Catalan, Chinese, Czech, Danish, Dutch, English, Finnish, French, Galician, German, Greek, Hindi, Hungarian, Indonesian, Irish, Italian, Japanese, Korean, Kurdish, Norwegian, Persian, Portuguese, Romanian, Russian, Spanish, Swedish, Turkish, and Thai.

These analyzers typically perform four roles:

- Tokenize text into individual words:

The quick brown foxes → [The, quick, brown, foxes]

- Lowercase tokens:

The → the

- Remove common stopwords:

[The, quick, brown, foxes] → [quick, brown, foxes]

- Stem tokens to their root form:

foxes → fox

- Each analyzer may also apply other transformations specific to its language in order to make words from that language more searchable:

- The english analyzer removes the possessive 's:

John's → john

- The french analyzer removes elisions like l' and qu' and diacritics like " or ^:

l'église → eglis

- The german analyzer normalizes terms, replacing ä and ae with a, or ß with ss, among others:

äußerst → ausserst

Using Language Analyzers

The built-in language analyzers are available globally and don't need to be configured before being used. They can be specified directly in the field mapping:

PUT /my_index

```
{  
  "mappings": {  
    "blog": {  
      "properties": {  
        "title": {  
          "type": "string",  
          "analyzer": "english"  
        }  
      }  
    }  
  }  
}
```

The title field will use the english analyzer instead of the default standard analyzer.

Of course, by passing text through the english analyzer, we lose information:

GET /my_index/_analyze?field=title

I'm not happy about the foxes

Emits token: i'm, happi, about, fox

We can't tell if the document mentions one fox or many foxes; the word not is a

stopword and is removed, so we can't tell whether the document is happy about foxes or not. By using the english analyzer, we have increased recall as we can match more loosely, but we have reduced our ability to rank documents accurately. To get the best of both worlds, we can use **multifields** to index the title field twice:

once with the english analyzer and once with the standard analyzer:

```
PUT /my_index
```

```
{  
  "mappings": {  
    "blog": {  
      "properties": {  
        "title": {  
          "type": "string",  
          "fields": {  
            "english": {  
              "type": "string",  
              "analyzer": "english"  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

The main title field uses the standard analyzer.

The title.english subfield uses the english analyzer.

With this mapping in place, we can index some test documents to demonstrate how to use both fields at query time:

```
PUT /my_index/blog/1
```

```
{ "title": "I'm happy for this fox" }
```

```
PUT /my_index/blog/2
```

```
{ "title": "I'm not happy about my fox problem" }
```

```
GET /_search
```

```
{  
  "query": {  
    "multi_match": {  
      "type": "most_fields",  
      "query": "not happy foxes",  
      "fields": [ "title", "title.english" ]  
    }  
  }  
}
```

```
}
```

Use the `most_fields` query type to match the same text in as many fields as possible.

Even though neither of our documents contain the word `foxes`, both documents are returned as results thanks to the word stemming on the `title.english` field. The second document is ranked as more relevant, because the word `not` matches on the `title` field.

Configuring Language Analyzers

While the language analyzers can be used out of the box without any configuration, most of them do allow you to control aspects of their behavior, specifically:

Stem-word exclusion

Imagine, for instance, that users searching for the “World Health Organization” are instead getting results for “organ health.” The reason for this confusion is that both “organ” and “organization” are stemmed to the same root word: `organ`. Often this isn’t a problem, but in this particular collection of documents, this leads to confusing results. We would like to prevent the words `organization` and `organizations` from being stemmed.

Custom stopwords

The default list of stopwords used in English are as follows:
`a`, `an`, `and`, `are`, `as`, `at`, `be`, `but`, `by`, `for`, `if`, `in`, `into`, `is`, `it`,
`no`, `not`, `of`, `on`, `or`, `such`, `that`, `the`, `their`, `then`, `there`, `these`,
`they`, `this`, `to`, `was`, `will`, `with`

The unusual thing about `no` and `not` is that they invert the meaning of the words that follow them. Perhaps we decide that these two words are important and that we shouldn’t treat them as stopwords.

To customize the behavior of the english analyzer, we need to create a custom analyzer that uses the english analyzer as its base but adds some configuration:

```
PUT /my_index
```

```
{  
  "settings": {  
    "analysis": {  
      "analyzer": {  
        "my_english": {  
          "type": "english",  
          "stem_exclusion": [ "organization", "organizations" ],  
          "stopwords": [  
            "a", "an", "and", "are", "as", "at", "be", "but", "by", "for",  
            "if", "in", "into", "is", "it", "of", "on", "or", "such", "that",  
            "this", "to", "was", "will", "with"  
          ]  
        }  
      }  
    }  
  }  
}
```

```
"the", "their", "then", "there", "these", "they", "this", "to",
"was", "will", "with"
]
}
}
}
}
}
}
```

GET /my_index/_analyze?analyzer=my_english

The World Health Organization does not sell organs.

Prevents organization and organizations from being stemmed

Specifies a custom list of stopwords

Emits tokens world, health, organization, does, not, sell, organ

Analyze Multiple Times

If you primarily deal with a limited number of languages, you could use multi-fields to analyze the text once per language:

```
PUT /movies
{
  "mappings": {
    "title": {
      "properties": {
        "title": {
          "type": "string",
          "fields": {
            "de": {
              "type": "string",
              "analyzer": "german"
            },
            "en": {
              "type": "string",
              "analyzer": "english"
            },
            "fr": {
              "type": "string",
              "analyzer": "french"
            },
            "es": {
              "type": "string",

```

```
        "analyzer": "spanish"
    }
}
}
}
}
}
}
}
```

The main title field uses the standard analyzer.

Each subfield applies a different language analyzer to the text in the title field.

Synonyms

While stemming helps to broaden the scope of search by simplifying inflected words to their root form, synonyms broaden the scope by relating concepts and ideas. Perhaps

no documents match a query for “English queen,” but documents that contain “British monarch” would probably be considered a good match.

A user might search for “the US” and expect to find documents that contain United States, USA, U.S.A., America, or the States. However, they wouldn’t expect to see results about the states of matter or state machines.

This example provides a valuable lesson. It demonstrates how simple it is for a human to distinguish between separate concepts, and how tricky it can be for mere machines. The natural tendency is to try to provide synonyms for every word in the language, to ensure that any document is findable with even the most remotely related terms.

This is a mistake. In the same way that we prefer light or minimal stemming to aggressive stemming, synonyms should be used only where necessary. Users understand why their results are limited to the words in their search query. They are less understanding when their results seems almost random.

Synonyms can be used to conflate words that have pretty much the same meaning, such as jump, leap, and hop, or pamphlet, leaflet, and brochure. Alternatively, they can be used to make a word more generic. For instance, bird could be used as a more general synonym for owl or pigeon, and adult could be used for man or woman.

Synonyms appear to be a simple concept but they are quite tricky to get right. In this chapter, we explain the mechanics of using synonyms and discuss the limitations and gotchas.

Using Synonyms

Synonyms can replace existing tokens or be added to the token stream by using the

synonym token filter:

```
PUT /my_index
```

```
{  
  "settings": {  
    "analysis": {  
      "filter": {  
        "my_synonym_filter": {  
          "type": "synonym", - First, we define a token filter of type synonym.  
          "synonyms": [           - formats in Formatting Synonyms  
            "british,english",  
            "queen,monarch"  
          ]  
        }  
      },  
      "analyzer": {  
        "my_synonyms": {  
          "tokenizer": "standard",  
          "filter": [  
            "lowercase",  
            "my_synonym_filter"       - Then we create a custom analyzer that uses the  
              my_synonym_filter.  
          ]  
        }  
      }  
    }  
  }  
}
```

Testing our analyzer with the analyze API shows the following:

```
GET /my_index/_analyze?analyzer=my_synonyms
```

```
Elizabeth is the English queen
```

```
Pos 1: (elizabeth)
```

```
Pos 2: (is)
```

```
Pos 3: (the)
```

```
Pos 4: (british,english)
```

```
Pos 5: (queen,monarch)
```

All synonyms occupy the same position as the original term.

A document like this will match queries for any of the following: English queen,

British queen, English monarch, or British monarch. Even a phrase query will work, because the position of each term has been preserved.

Note: Using the same synonym token filter at both index time and search time is redundant. If, at index time, we replace English with the two terms english and british, then at search time we need to search for only one of those terms. Alternatively, if we don't use synonyms at index time, then at search time, we would need to convert a query for English into a query for english OR british. Whether to do synonym expansion at search or index time can be a difficult choice.

Formatting Synonyms

In their simplest form, synonyms are listed as comma-separated values:

"jump,leap,hop"

If any of these terms is encountered, it is replaced by all of the listed synonyms. For instance:

Original terms: Replaced by:

jump → (jump,leap,hop)

leap → (jump,leap,hop)

hop → (jump,leap,hop)

Alternatively, with the => syntax, it is possible to specify a list of terms to match (on

the left side), and a list of one or more replacements (on the right side):

"u s a,united states,united states of america => usa"

"g b,gb,great britain => britain,england,scotland,wales"

Original terms: Replaced by:

u s a → (usa)

united states → (usa)

great britain → (britain,england,scotland,wales)

If multiple rules for the same synonyms are specified, they are merged together. The order of rules is not respected. Instead, the longest matching rule wins. Take the following rules as an example:

"united states => usa",

"united states of america => usa"

If these rules conflicted, Elasticsearch would turn United States of America into the terms (usa),(of),(america). Instead, the longest sequence wins, and we end up

with just the term (usa).

Expand or contract

we have seen that it is possible to replace synonyms by simple expansion, simple contraction, or generic expansion. We will look at the trade-offs each of these techniques in this section. This section deals with single-word synonyms only.

Simple Expansion

With simple expansion, any of the listed synonyms is expanded into all of the listed synonyms:

"jump, hop, leap"

Simple Contraction

Simple contraction maps a group of synonyms on the left side to a single value on the right side:

"leap, hop => jump"

Genre Expansion

Genre expansion is quite different from simple contraction or expansion. Instead of treating all synonyms as equal, genre expansion widens the meaning of a term to be more generic. Take these rules, for example:

"cat => cat, pet",

"kitten => kitten, cat, pet",

"dog => dog, pet"

"puppy => puppy, dog, pet"

By applying genre expansion at index time:

- A query for kitten would find just documents about kittens.
- A query for cat would find documents about kittens and cats.
- A query for pet would find documents about kittens, cats, puppies, dogs, or pets.

Alternatively, by applying genre expansion at query time, a query for kitten would be expanded to return documents that mention kittens, cats, or pets specifically.

You could also have the best of both worlds by applying expansion at index time to ensure that the genres are present in the index. Then, at query time, you can choose to not apply synonyms (so that a query for kitten returns only documents about kittens) or to apply synonyms in order to match kittens, cats and pets (including the canine variety).

With the preceding example rules above, the IDF for kitten will be correct, while the IDF for cat and pet will be artificially deflated. However, this works in your

favor—a genre-expanded query for kitten OR cat OR pet will rank documents with kitten highest, followed by documents with cat, and documents with pet would be right at the bottom.

Multiword Synonyms and Phrase Queries

So far, synonyms appear to be quite straightforward. Unfortunately, this is where things start to go wrong. For **phrase queries** to function correctly, Elasticsearch needs to know the position that each term occupies in the original text. Multiword synonyms can play havoc with term positions, especially when the injected synonyms are of differing lengths.

To demonstrate, we'll create a synonym token filter that uses this rule:

"usa,united states,u s a,united states of america"

```
PUT /my_index
```

```
{  
  "settings": {  
    "analysis": {  
      "filter": {  
        "my_synonym_filter": {  
          "type": "synonym",  
          "synonyms": [  
            "usa,united states,u s a,united states of america"  
          ]  
        }  
      }  
    },  
    "analyzer": {  
      "my_synonyms": {  
        "tokenizer": "standard",  
        "filter": [  
          "lowercase",  
          "my_synonym_filter"  
        ]  
      }  
    }  
  }  
}
```

```
GET /my_index/_analyze?analyzer=my_synonyms&text=
```

The United States is wealthy

The tokens emitted by the analyze request look like this:

Pos 1: (the)

Pos 2: (usa,united,u,united)

Pos 3: (states,s,states)

Pos 4: (is,a,of)

Pos 5: (wealthy,america)

If we were to index a document analyzed with synonyms as above, and then run a

phrase query without synonyms, we'd have some surprising results. These phrases would not match:

- The usa is wealthy
- The united states of america is wealthy
- The U.S.A. is wealthy

However, these phrases would:

- United states is wealthy
- Usa states of wealthy
- The U.S. of wealthy
- U.S. is america

If we were to use synonyms at query time instead, we would see even more-bizarre matches. Look at the output of this validate-query request:

`GET /my_index/_validate/query?explain`

```
{  
  "query": {  
    "match_phrase": {  
      "text": {  
        "query": "usa is wealthy",  
        "analyzer": "my_synonyms"  
      }  
    }  
  }  
}
```

The explanation is as follows:

`"(usa united u united) (is states s states) (wealthy a of) america"`

This would match documents containing u is of america but wouldn't match any document that didn't contain the term america.

Use Simple Contraction for Phrase Queries

The way to avoid this mess is to use `simple contraction` to inject a single term that represents all synonyms, and to use the same synonym token filter at query time:

`PUT /my_index`

```
{  
  "settings": {  
    "analysis": {
```

```
"filter": {  
  "my_synonym_filter": {  
    "type": "synonym",  
    "synonyms": [  
      "united states,u s a,united states of america=>usa"  
    ]  
  }  
},  
"analyzer": {  
  "my_synonyms": {  
    "tokenizer": "standard",  
    "filter": [  
      "lowercase",  
      "my_synonym_filter"  
    ]  
  }  
}  
}  
}  
}  
}  
}  
}
```

GET /my_index/_analyze?analyzer=my_synonyms

The United States is wealthy

The result of the preceding analyze request looks much more sane:

Pos 1: (the)
Pos 2: (usa)
Pos 3: (is)
Pos 5: (wealthy)

And repeating the validate-query request that we made previously yields a simple, sane explanation:

"usa is wealthy"

The downside of this approach is that, by reducing united states of america down to the single term usa, you can't use the same field to find just the word united or states. You would need to use a separate field with a different analysis chain for that purpose.

Symbol Synonyms

The final part of this chapter is devoted to symbol synonyms, which are unlike the synonyms we have discussed until now. Symbol synonyms are string aliases used to represent symbols that would otherwise be removed during tokenization.

While most punctuation is seldom important for full-text search, character combinations like emoticons may be very significant, even changing the meaning of the the text. Compare these:

two sentences that have quite different intent.

We can use the `mapping character filter` to replace emoticons with symbol synonyms like `emoticon_happy` and `emoticon_sad` before the text is passed to the tokenizer:

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": {
        "emoticons": {
          "type": "mapping",
          "mappings": [
            ": ) =>emoticon_happy",
            ":( =>emoticon_sad"
          ]
        }
      },
      "analyzer": {
        "my_emoticons": {
          "char_filter": "emoticons",
          "tokenizer": "standard",
          "filter": [ "lowercase" ]
        }
      }
    }
  }
}
```

```
GET /my_index/_analyze?analyzer=my_emoticons
```

```
I am :) not :(
```

The mappings filter replaces the characters to the left of `=>` with those to the right.

Emits tokens `i`, `am`, `emoticon_happy`, `not`, `emoticon_sad`.

It is unlikely that anybody would ever search for `emoticon_happy`, but ensuring that important symbols like emoticons are included in the index can be helpful when doing sentiment analysis. Of course, we could equally have used real words, like `happy` and `sad`.

Typoes and Mispelings

We expect a query on structured data like dates and prices to return only documents that match exactly. However, good full-text search shouldn't have the same restriction.

Instead, we can widen the net to include words that may match, but use the relevance score to push the better matches to the top of the result set.

In fact, full-text search that only matches exactly will probably frustrate your users. Wouldn't you expect a search for "quick brown fox" to match a document containing "fast brown foxes," "Johnny Walker" to match "Johnnie Walker," or "Arnold Shcwazenneger" to match "Arnold Schwarzenegger"?

If documents exist that do contain exactly what the user has queried, they should appear at the top of the result set, but weaker matches can be included further down the list. If no documents match exactly, at least we can show the user potential matches; they may even be what the user originally intended!

Fuzzy matching allows for query-time matching of misspelled words, while phonetic token filters at index time can be used for sounds-like matching.

Fuzziness

Fuzzy matching treats two words that are "fuzzily" similar as if they were the same word. First, we need to define what we mean by fuzziness.

In 1965, Vladimir Levenshtein developed the [Levenshtein distance](#), which measures the number of single-character edits required to transform one word into the other. He proposed three types of one-character edits:

- Substitution of one character for another: `_f_ox` → `_b_ox`
- Insertion of a new character: `sic` → `sic_k_`
- Deletion of a character: `b_l_ack` → `back`

[Frederick Damerau](#) later expanded these operations to include one more:

- Transposition of two adjacent characters: `_st_ar` → `_ts_ar`

For example, to convert the word `bieber` into `beaver` requires the following steps:

1. Substitute `v` for `b`: `bie_b_er` → `bie_v_er`
2. Substitute `a` for `i`: `b_i_ever` → `b_a_ever`
3. Transpose `a` and `e`: `b_ae_ver` → `b_ea_ver`

These three steps represent a [Damerau-Levenshtein edit distance](#) of 3.

Clearly, `bieber` is a long way from `beaver`—they are too far apart to be considered a simple misspelling. Damerau observed that 80% of human misspellings have an

edit distance of 1. In other words, 80% of misspellings could be corrected with a single edit to the original string.

Elasticsearch supports a maximum edit distance, specified with the fuzziness parameter, of 2.

Of course, the impact that a single edit has on a string depends on the length of the string. Two edits to the word hat can produce mad, so allowing two edits on a string of length 3 is overkill. The fuzziness parameter can be set to AUTO, which results in the following maximum edit distances:

- 0 for strings of one or two characters
- 1 for strings of three, four, or five characters
- 2 for strings of more than five characters

Of course, you may find that an edit distance of 2 is still overkill, and returns results that don't appear to be related. You may get better results, and better performance, with a maximum fuzziness of 1.

Fuzzy Query

The **fuzzy query** is the fuzzy equivalent of the term query. You will seldom use it directly yourself, but understanding how it works will help you to use fuzziness in the higher-level match query.

To understand how it works, we will first index some documents:

```
POST /my_index/my_type/_bulk
{
  "index": { "_id": 1 },
  "text": "Surprise me!",
  "index": { "_id": 2 },
  "text": "That was surprising.",
  "index": { "_id": 3 },
  "text": "I wasn't surprised."
```

Now we can run a fuzzy query for the term surprize:

```
GET /my_index/my_type/_search
{
  "query": {
    "fuzzy": {
      "text": "surprize"
    }
  }
}
```

The fuzzy query is a term-level query, so it doesn't do any analysis. It takes a single term and finds all terms in the term dictionary that are within the specified fuzziness. The default fuzziness is AUTO.

In our example, surprize is within an edit distance of 2 from both surprise and surprised, so documents 1 and 3 match. We could reduce the matches to just surprise with the following query:

```
GET /my_index/my_type/_search
{
  "query": {
    "fuzzy": {
      "text": {
        "value": "surprise",
        "fuzziness": 1
      }
    }
  }
}
```

Improving Performance

The fuzzy query works by taking the original term and building a Levenshtein automaton— like a big graph representing all the strings that are within the specified edit distance of the original string.

The fuzzy query then uses the automation to step efficiently through all of the terms in the term dictionary to see if they match. Once it has collected all of the matching terms that exist in the term dictionary, it can compute the list of matching documents.

Of course, depending on the type of data stored in the index, a fuzzy query with an edit distance of 2 can match a very large number of terms and perform very badly.

Two parameters can be used to limit the performance impact:

prefix_length

The number of initial characters that will not be “fuzzified.” Most spelling errors occur toward the end of the word, not toward the beginning. By using a prefix_length of 3, for example, you can significantly reduce the number of matching terms.

max_expansions

If a fuzzy query expands to three or four fuzzy options, the new options may be meaningful. If it produces 1,000 options, they are essentially meaningless. Use max_expansions to limit the total number of options that will be produced. The fuzzy query will collect matching terms until it runs out of terms or reaches the max_expansions limit.

Fuzzy match Query

The match query supports fuzzy matching out of the box:

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "text": {
        "query": "SURPRISE ME!",
        "fuzziness": "AUTO",
        "operator": "and"
      }
    }
  }
}
```

The query string is first analyzed, to produce the terms [surprise, me], and then each term is fuzzified using the specified fuzziness.

Similarly, the multi_match query also supports fuzziness, but only when executing with type best_fields or most_fields:

```
GET /my_index/my_type/_search
{
  "query": {
    "multi_match": {
      "fields": [ "text", "title" ],
      "query": "SURPRISE ME!",
      "fuzziness": "AUTO"
    }
  }
}
```

Both the match and multi_match queries also support the prefix_length and max_expansions parameters.

Note: Fuzziness works only with the basic match and multi_match queries. It doesn't work with phrase matching, common terms, or cross_fields matches.

Scoring Fuzziness

Users love fuzzy queries. They assume that these queries will somehow magically find the right combination of proper spellings. Unfortunately, the truth is somewhat more prosaic.

Imagine that we have 1,000 documents containing “Schwarzenegger,” and just one

document with the misspelling “Schwarzeneger.” According to the theory of **term frequency/ inverse document frequency**, the misspelling is much more relevant than the

correct spelling, because it appears in far fewer documents!

In other words, if we were to treat fuzzy matches like any other match, we would favor misspellings over correct spellings, which would make for grumpy users.

Note: Fuzzy matching should not be used for scoring purposes—only to widen the net of matching terms in case there are misspellings.

By default, the match query gives all fuzzy matches the constant score of 1. This is sufficient to add potential matches onto the end of the result list, without interfering with the relevance scoring of nonfuzzy queries.

Note: Fuzzy queries alone are much less useful than they initially appear. They are better used as part of a “bigger” feature, such as the search-as-you-type completion suggester or the did-you-mean phrase suggester.

Aggregations

Until this point, this book has been dedicated to search. With search, we have a query and we want to find a subset of documents that match the query. We are looking for the proverbial needle(s) in the haystack.

With aggregations, we zoom out to get an overview of our data. Instead of looking for individual documents, we want to analyze and summarize our complete set of data:

- How many needles are in the haystack?
- What is the average length of the needles?
- What is the median length of the needles, broken down by manufacturer?
- How many needles were added to the haystack each month?

Aggregations can answer more subtle questions too:

- What are your most popular needle manufacturers?
- Are there any unusual or anomalous clumps of needles?

Aggregations allow us to ask sophisticated questions of our data. And yet, while the functionality is completely different from search, it leverages the same datastructures.

This means aggregations execute quickly and are near real-time, just like search. This is extremely powerful for reporting and dashboards. Instead of performing rollups of your data (that crusty Hadoop job that takes a week to run), you can

visualize your data in real time, allowing you to respond immediately. Your report changes as your data changes, rather than being pre-calculated, out of date and irrelevant.

Finally, aggregations operate alongside search requests. This means you can both search/filter documents and perform analytics at the same time, on the same data, in a single request. And because aggregations are calculated in the context of a user's search, you're not just displaying a count of four-star hotels—you're displaying a count of four-star hotels that match their search criteria.

Aggregations are so powerful that many companies have built large Elasticsearch clusters solely for analytics.

High-Level Concepts

Like the query DSL, aggregations have a composable syntax: independent units of functionality can be mixed and matched to provide the custom behavior that you need. This means that there are only a few basic concepts to learn, but nearly limitless combinations of those basic components.

To master aggregations, you need to understand only two main concepts:

Buckets – Collections of documents that meet a criterion

Metrics – Statistics calculated on the documents in a bucket

That's it! Every aggregation is simply a combination of one or more buckets and zero or more metrics. To translate into rough SQL terms:

`SELECT COUNT(color)`

`FROM table`

`GROUP BY color`

`COUNT(color)` is equivalent to a metric.

`GROUP BY color` is equivalent to a bucket.

Buckets are conceptually similar to grouping in SQL, while metrics are similar to `COUNT()`, `SUM()`, `MAX()`, and so forth.

Let's dig into both of these concepts and see what they entail.

Buckets

A bucket is simply a collection of documents that meet a certain criteria:

- An employee would land in either the male or female bucket.
- The city of Albany would land in the New York state bucket.
- The date 2014-10-28 would land within the October bucket.

As aggregations are executed, the values inside each document are evaluated to determine whether they match a bucket's criteria. If they match, the document is placed inside the bucket and the aggregation continues.

Buckets can also be nested inside other buckets, giving you a hierarchy or conditional partitioning scheme. For example, Cincinnati would be placed inside the

Ohio state bucket, and the entire Ohio bucket would be placed inside the USA country bucket.

Elasticsearch has a variety of buckets, which allow you to partition documents in many ways (by hour, by most-popular terms, by age ranges, by geographical location, and more). But fundamentally they all operate on the same principle: partitioning documents based on a criteria.

Metrics

Buckets allow us to partition documents into useful subsets, but ultimately what we want is some kind of metric calculated on those documents in each bucket. Bucketing is the means to an end: it provides a way to group documents in a way that you can calculate interesting metrics.

Most metrics are simple mathematical operations (for example, min, mean, max, and sum) that are calculated using the document values. In practical terms, metrics allow you to calculate quantities such as the average salary, or the maximum sale price, or the 95th percentile for query latency.

Combining the Two

An aggregation is a combination of buckets and metrics. An aggregation may have a single bucket, or a single metric, or one of each. It may even have multiple buckets nested inside other buckets. For example, we can partition documents by which country they belong to (a bucket), and then calculate the average salary per country (a metric).

Because buckets can be nested, we can derive a much more complex aggregation:

1. Partition documents by country (bucket).
2. Then partition each country bucket by gender (bucket).
3. Then partition each gender bucket by age ranges (bucket).
4. Finally, calculate the average salary for each age range (metric)

This will give you the average salary per <country, gender, age> combination. All in one request and with one pass over the data!

Aggregation Test-Drive

So let's just dive in and start with an example. We are going to build some aggregations that might be useful to a car dealer. Our data will be about car transactions: the car model, manufacturer, sale price, when it sold, and more.

First we will bulk-index some data to work with:

`POST /cars/transactions/_bulk`

`{ "index": {}}`

```
{
  "price" : 10000, "color" : "red", "make" : "honda", "sold" : "2014-10-28"
}
{
  "index": {}
}
{
  "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05"
}
{
  "index": {}
}
{
  "price" : 30000, "color" : "green", "make" : "ford", "sold" : "2014-05-18"
}
{
  "index": {}
}
{
  "price" : 15000, "color" : "blue", "make" : "toyota", "sold" : "2014-07-02"
}
{
  "index": {}
}
{
  "price" : 12000, "color" : "green", "make" : "toyota", "sold" : "2014-08-19"
}
{
  "index": {}
}
{
  "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05"
}
{
  "index": {}
}
{
  "price" : 80000, "color" : "red", "make" : "bmw", "sold" : "2014-01-01" }
{
  "index": {}
}
{
  "price" : 25000, "color" : "blue", "make" : "ford", "sold" : "2014-02-12" }
```

Now that we have some data, let's construct our first aggregation. A car dealer may want to know which color car sells the best. This is easily accomplished using a simple aggregation. We will do this using a terms bucket:

```
GET /cars/transactions/_search?search_type=count
{
  "aggs" : { - 1
  "colors" : { - 2
  "terms" : {
    "field" : "color" - 3
  }
  }
  }
}
```

1. Aggregations are placed under the top-level aggs parameter (the longer aggregations will also work if you prefer that).
2. We then name the aggregation whatever we want: colors, in this example
3. Finally, we define a single bucket of type terms.

Let's execute that aggregation and take a look at the results:

```
{  
...  
"hits": { "hits": []  
},  
"aggregations": {  
"colors": {  
"buckets": [  
{  
"key": "red",  
"doc_count": 4  
},  
{  
"key": "blue",  
"doc_count": 2  
},  
{  
"key": "green",  
"doc_count": 2  
}  
]  
}  
}  
}
```

Adding a Metric to the Mix

The previous example told us the number of documents in each bucket, which is useful.

But often, our applications require more-sophisticated metrics about the documents.

For example, what is the average price of cars in each bucket?

To get this information, we need to tell Elasticsearch which metrics to calculate, and on which fields. This requires nesting metrics inside the buckets. Metrics will calculate mathematical statistics based on the values of documents within a bucket.

Let's go ahead and add an average metric to our car example:

```
GET /cars/transactions/_search?search_type=count  
{  
"aggs": {  
"colors": {  
"terms": {  
"field": "color"
```

```
},
"aggs": {
  "avg_price": {
    "avg": {
      "field": "price"
    }
  }
}
}
}
}
}
}
}
```

Result:

```
{
...
"aggregations": {
  "colors": {
    "buckets": [
      {
        "key": "red",
        "doc_count": 4,
        "avg_price": {
          "value": 32500
        }
      },
      {
        "key": "blue",
        "doc_count": 2,
        "avg_price": {
          "value": 20000
        }
      },
      {
        "key": "green",
        "doc_count": 2,
        "avg_price": {
          "value": 21000
        }
      }
    ]
  }
}
```

```
}
```

```
...
```

```
}
```

Buckets Inside Buckets

The true power of aggregations becomes apparent once you start playing with different nesting schemes. In the previous examples, we saw how you could nest a metric inside a bucket, which is already quite powerful.

But the real exciting analytics come from nesting buckets inside other buckets. This time, we want to find out the distribution of car manufacturers for each color:

```
GET /cars/transactions/_search?search_type=count
```

```
{
```

```
  "aggs": {
```

```
    "colors": {
```

```
      "terms": {
```

```
        "field": "color"
```

```
      },
```

```
      "aggs": {
```

```
        "avg_price": {
```

```
          "avg": {
```

```
            "field": "price"
```

```
          }
```

```
        },
```

```
        "make": {
```

```
          "terms": {
```

```
            "field": "make"
```

```
          }
```

```
        }
```

```
      }
```

```
    }
```

```
  }
```

```
  }
```

```
  }
```

```
  }
```

```
  }
```

```
  }
```

Let's take a look at the response (truncated for brevity, since it is now growing quite long):

```
{
```

```
...
```

```
  "aggregations": {
```

```
    "colors": {
```

```
      "buckets": [
```

```
        {
```

```
"key": "red",
"doc_count": 4,
"make": {
  "buckets": [
    {
      "key": "honda",
      "doc_count": 3
    },
    {
      "key": "bmw",
      "doc_count": 1
    }
  ]
},
"avg_price": {
  "value": 32500
},
},
...  
}
```

One Final Modification

Just to drive the point home, let's make one final modification to our example before moving on to new topics. Let's add two metrics to calculate the min and max price for each make:

```
GET /cars/transactions/_search?search_type=count
{
  "aggs": {
    "colors": {
      "terms": {
        "field": "color"
      },
      "aggs": {
        "avg_price": { "avg": { "field": "price" } }
      },
      "make" : {
        "terms" : {
          "field" : "make"
        },
        "aggs" : {
          "min_price" : { "min": { "field": "price" } },
          "max_price" : { "max": { "field": "price" } }
        }
      }
    }
  }
}
```

```
"max_price" : { "max": { "field": "price"} }
}
}
}
}
}
}
```

Which gives us the following output (again, truncated):

```
{
...
"aggregations": {
"colors": {
"buckets": [
{
"key": "red",
"doc_count": 4,
"make": {
"buckets": [
{
"key": "honda",
"doc_count": 3,
"min_price": {
"value": 10000
},
"max_price": {
"value": 20000
}
},
{
"key": "bmw",
"doc_count": 1,
"min_price": {
"value": 80000
},
"max_price": {
"value": 80000
}
}
]
},
{
"key": "blue",
"doc_count": 2,
"min_price": {
"value": 15000
},
"max_price": {
"value": 30000
}
}
],
}
}
}
}
```

```
"avg_price": {  
  "value": 32500  
}  
},  
...
```

Filtering Queries and Aggregations

A natural extension to aggregation scoping is filtering. Because the aggregation operates in the context of the query scope, any filter applied to the query will also apply to the aggregation.

Filtered Query

If we want to find all cars over \$10,000 and also calculate the average price for those cars, we can simply use a filtered query:

```
GET /cars/transactions/_search?search_type=count
```

```
{  
  "query" : {  
    "filtered": {  
      "filter": {  
        "range": {  
          "price": {  
            "gte": 10000  
          }  
        }  
      }  
    }  
  },  
  "aggs" : {  
    "single_avg_price": {  
      "avg" : { "field" : "price" }  
    }  
  }  
}
```

Filter Bucket

```
GET /cars/transactions/_search?search_type=count
```

```
{  
  "query":{  
    "match": {  
      "make": "ford"  
    }  
  },  
  "aggs":{
```

```
"recent_sales": {  
  "filter": {  
    "range": {  
      "sold": {  
        "from": "now-1M"  
      }  
    }  
  },  
  "aggs": {  
    "average_price":{  
      "avg": {  
        "field": "price"  
      }  
    }  
  }  
}
```

Geolocation

Gone are the days when we wander around a city with paper maps. Thanks to smartphones, we now know exactly where we are all the time, and we expect websites to use that information. I'm not interested in restaurants in Greater London—I want to know about restaurants within a 5-minute walk of my current location. But geolocation is only one part of the puzzle. The beauty of Elasticsearch is that it allows you to combine geolocation with full-text search, structured search, and analytics.

For instance: show me restaurants that mention vitello tonnato, are within a 5-minute walk, and are open at 11 p.m., and then rank them by a combination of user rating, distance, and price. Another example: show me a map of vacation rental properties available in August throughout the city, and calculate the average price per zone.

Elasticsearch offers two ways of representing geolocations: latitude-longitude points using the geo_point field type, and complex shapes defined in [GeoJSON](#), using the geo_shape field type.

Geo-points allow you to find points within a certain distance of another point, to calculate distances between two points for sorting or relevance scoring, or to aggregate into a grid to display on a map. Geo-shapes, on the other hand, are

used purely for filtering. They can be used to decide whether two shapes overlap, or whether one shape completely contains other shapes.

Geo-Points

A geo-point is a single latitude/longitude point on the Earth's surface. Geo-points can be used to calculate distance from a point, to determine whether a point falls within a bounding box, or in aggregations.

Geo-points cannot be automatically detected with [dynamic mapping](#). Instead, geo_point fields should be mapped explicitly:

PUT /attractions

```
{  
  "mappings": {  
    "restaurant": {  
      "properties": {  
        "name": {  
          "type": "string"  
        },  
        "location": {  
          "type": "geo_point"  
        }  
      }  
    }  
  }  
}
```

Lat/Lon Formats

With the location field defined as a geo_point, we can proceed to index documents containing latitude/longitude pairs, which can be formatted as strings, arrays, or objects:

PUT /attractions/restaurant/1

```
{  
  "name": "Chipotle Mexican Grill",  
  "location": "40.715, -74.011"  
}
```

PUT /attractions/restaurant/2

```
{  
  "name": "Pala Pizza",  
  "location": {  
    "lat": 40.722,  
    "lon": -73.989  
  }  
}
```

```
PUT /attractions/restaurant/3
{
  "name": "Mini Munchies Pizza",
  "location": [ -73.983, 40.719 ]
}
```

Filtering by Geo-Point

Four geo-point filters can be used to include or exclude documents by geolocation:

`geo_bounding_box`

Find geo-points that fall within the specified rectangle.

`geo_distance`

Find geo-points within the specified distance of a central point.

`geo_distance_range`

Find geo-points within a specified minimum and maximum distance from a central point.

`geo_polygon`

Find geo-points that fall within the specified polygon. This filter is very expensive.

If you find yourself wanting to use it, you should be looking at [geo-shapes](#) instead.

All of these filters work in a similar way: the lat/lon values are loaded into memory for all documents in the index, not just the documents that match the query. Each filter performs a slightly different calculation to check whether a point falls into the containing area.

Note: Geo-filters are expensive — they should be used on as few documents as possible. First remove as many documents as you can with cheaper filters, like term or range filters, and apply the geofilters last.

The [bool filter](#) will do this for you automatically. First it applies any bitset-based to exclude as many documents as it can as cheaply as possible. Then it applies the more expensive geo or script filters to each remaining document in turn.

`geo_bounding_box` Filter

This is by far the most efficient geo-filter because its calculation is very simple. You provide it with the top, bottom, left, and right coordinates of a rectangle, and all it does is compare the latitude with the left and right coordinates, and the longitude with the top and bottom coordinates:

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
```

```
"filter": {  
  "geo_bounding_box": {  
    "location": {  
      "top_left": {  
        "lat": 40.8,  
        "lon": -74.0  
      },  
      "bottom_right": {  
        "lat": 40.7,  
        "lon": -73.0  
      }  
    }  
  }  
}
```

Optimizing Bounding Boxes

The geo_bounding_box is the one geo-filter that doesn't require all geo-points to be loaded into memory. Because all it has to do is check whether the lat and lon values fall within the specified ranges, it can use the inverted index to do a glorified range filter.

To use this optimization, the geo_point field must be mapped to index the lat and lon values separately:

```
PUT /attractions
```

```
{  
  "mappings": {  
    "restaurant": {  
      "properties": {  
        "name": {  
          "type": "string"  
        },  
        "location": {  
          "type": "geo_point",  
          "lat_lon": true  
        }  
      }  
    }  
  }  
}
```

```
}
```

The location.lat and location.lon fields will be indexed separately. These fields can be used for searching, but their values cannot be retrieved.

Now, when we run our query, we have to tell Elasticsearch to use the indexed lat and lon values:

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "type": "indexed",
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.7,
              "lon": -73.0
            }
          }
        }
      }
    }
  }
}
```

Note: While a geo_point field can contain multiple geo-points, the lat_lon optimization can be used only on fields that contain a single geo-point.

geo_distance Filter

The geo_distance filter draws a circle around the specified location and finds all documents that have a geo-point within that circle:

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {

```

```
"distance": "1km",
"location": {
  "lat": 40.715,
  "lon": -73.988
}
}
}
}
}
}
}
```

Find all location fields within 1km of the specified point. See [Distance Units](#) for a list of the accepted units.

Faster Geo-Distance Calculations

The distance between two points can be calculated using algorithms, which trade performance for accuracy:

arc

The slowest but most accurate is the arc calculation, which treats the world as a sphere. Accuracy is still limited because the world isn't really a sphere.

plane

The plane calculation, which treats the world as if it were flat, is faster but less accurate. It is most accurate at the equator and becomes less accurate toward the poles.

sloppy_arc

So called because it uses the SloppyMath Lucene class to trade accuracy for speed, the sloppy_arc calculation uses the [Haversine formula](#) to calculate distance. It is four to five times as fast as arc, and distances are 99.9% accurate. This is the default calculation.

You can specify a different calculation as follows:

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {
          "distance": "1km",
          "distance_type": "plane",
          "location": {
            "lat": 40.715,
```

```
"lon": -73.988
}
}
}
}
}
}
```

geo_distance_range Filter

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance_range": {
          "gte": "1km",
          "lt": "2km",
          "location": {
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

Caching geo-filters

The results of geo-filters are not cached by default, for two reasons:

Geo-filters are usually used to find entities that are near to a user's current location.

The problem is that users move, and no two users are in exactly the same location. A cached filter would have little chance of being reused.

Filters are cached as bitsets that represent all documents in a **segment**. Imagine that our query excludes all documents but one in a particular segment. An uncached geo-filter just needs to check the one remaining document, but a cached geo-filter would need to check all of the documents in the segment.

That said, caching can be used to good effect with geo-filters. Imagine that your index contains restaurants from all over the United States. A user in New York is

not interested in restaurants in San Francisco. We can treat New York as a hot spot and draw a big bounding box around the city and neighboring areas. This geo_bounding_box filter can be cached and reused whenever we have a user within the city limits of New York. It will exclude all restaurants from the rest of the country. We can then use an uncached, more specific geo_bounding_box or geo_distance filter to narrow the remaining results to those that are close to the user:

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "must": [
            {
              "geo_bounding_box": {
                "type": "indexed",
                "_cache": true,
                "location": {
                  "top_left": {
                    "lat": 40.8,
                    "lon": -74.1
                  },
                  "bottom_right": {
                    "lat": 40.4,
                    "lon": -73.7
                  }
                }
              }
            },
            {
              "geo_distance": {
                "distance": "1km",
                "location": {
                  "lat": 40.715,
                  "lon": -73.988
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Reducing Memory Usage

Each lat/lon pair requires 16 bytes of memory, memory that is in short supply. It needs this much memory in order to provide very accurate results. But as we have commented before, such exacting precision is seldom required.

You can reduce the amount of memory that is used by switching to a compressed fielddata format and by specifying how precise you need your geo-points to be. Even reducing precision to 1mm reduces memory usage by a third. A more realistic setting of 3m reduces usage by 62%, and 1km saves a massive 75%!

This setting can be changed on a live index with the update-mapping API:

POST /attractions/_mapping/restaurant

```
{  
  "location": {  
    "type": "geo_point",  
    "fielddata": {  
      "format": "compressed",  
      "precision": "1km"  
    }  
  }  
}
```

Each lat/lon pair will require only 4 bytes, instead of 16.

Alternatively, you can avoid using memory for geo-points altogether, either by using the technique described in “Optimizing Bounding Boxes” on page 514, or by storing geo-points as `doc values`:

PUT /attractions

```
{  
  "mappings": {  
    "restaurant": {  
      "properties": {  
        "name": {  
          "type": "string"  
        },  
        "location": {  
          "type": "geo_point",  
        }  
      }  
    }  
  }  
}
```

```
"doc_values": true
}
}
}
}
}
```

Sorting by Distance

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "type": "indexed",
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.0
            }
          }
        }
      }
    },
    "sort": [
      {
        "_geo_distance": {
          "location": {
            "lat": 40.715,
            "lon": -73.998
          },
          "order": "asc",
          "unit": "km",
          "distance_type": "plane"
        }
      }
    ]
  }
}
```

```
]
}
```

Results:

```
...
"hits": [
{
  "_index": "attractions",
  "_type": "restaurant",
  "_id": "2",
  "_score": null,
  "_source": {
    "name": "New Malaysia",
    "location": {
      "lat": 40.715,
      "lon": -73.997
    }
  },
  "sort": [
    0.08425653647614346
  ]
},
...
...
```

Geohashes

Geohashes are a way of encoding lat/lon points as strings. The original intention was to have a URL-friendly way of specifying geolocations, but geohashes have turned out to be a useful way of indexing geo-points and geo-shapes in databases.

Geohashes divide the world into a grid of 32 cells—4 rows and 8 columns—each represented by a letter or number. The g cell covers half of Greenland, all of Iceland, and most of Great Britain. Each cell can be further divided into another 32 cells, which can be divided into another 32 cells, and so on. The gc cell covers Ireland and England, gcp covers most of London and part of Southern England, and gcpuuz94k is the entrance to Buckingham Palace, accurate to about 5 meters.

In other words, the longer the geohash string, the more accurate it is. If two geohashes share a prefix— and gcpuuz—then it implies that they are near each other. The longer the shared prefix, the closer they are.

Mapping Geohashes

The first step is to decide just how much precision you need. Although you could index all geo-points with the default full 12 levels of precision, do you really need to be accurate to within a few centimeters? You can save yourself a lot of space in the index by reducing your precision requirements to something more realistic, such as 1km:

```
PUT /attractions
{
  "mappings": {
    "restaurant": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_point",
          "geohash_prefix": true,
          "geohash_precision": "1km"
        }
      }
    }
  }
}
```

geohash_cell Filter

The geohash_cell filter simply translates a lat/lon location into a geohash with the specified precision and finds all locations that contain that geohash—a very efficient filter indeed.

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geohash_cell": {
          "location": {
            "lat": 40.718,
            "lon": -73.983
          },
          "precision": "2km"
        }
      }
    }
  }
}
```

```
}
```

```
}
```

```
}
```

To fix that, we can tell the filter to include the neighboring cells, by setting neighbours to true:

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geohash_cell": {
          "location": {
            "lat": 40.718,
            "lon": -73.983
          },
          "neighbors": true,
          "precision": "2km"
        }
      }
    }
  }
}
```

Geo-aggregations

Although filtering or scoring results by geolocation is useful, it is often more useful to be able to present information to the user on a map. A search may return way too many results to be able to display each geo-point individually, but geo-aggregations can be used to cluster geo-points into more manageable buckets.

Three aggregations work with fields of type geo_point:

geo_distance

Groups documents into concentric circles around a central point.

geohash_grid

Groups documents by geohash cell, for display on a map.

geo_bounds

Returns the lat/lon coordinates of a bounding box that would encompass all of the geo-points. This is useful for choosing the correct zoom level when displaying a map.

geo_distance Aggregation

The geo_distance agg is useful for searches such as to “find all pizza restaurants within 1km of me.” The search results should, indeed, be limited to the 1km radius specified by the user, but we can add “another result found within 2km”:

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "name": "pizza"
        }
      },
      "filter": {
        "geo_bounding_box": {
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.7
            }
          }
        }
      }
    },
    "aggs": {
      "per_ring": {
        "geo_distance": {
          "field": "location",
          "unit": "km",
          "origin": {
            "lat": 40.712,
            "lon": -73.988
          },
          "ranges": [
            { "from": 0, "to": 1 },
            { "from": 1, "to": 2 }
          ]
        }
      }
    }
  }
}
```

```
{ "from": 1, "to": 2 }
]
}
},
},
"post_filter": {
"geo_distance": {
"distance": "1km",
"location": {
"lat": 40.712,
"lon": -73.988
}
}
}
}
```

The response from the preceding request is as follows:

```
"hits": {
"total": 1,
"max_score": 0.15342641,
"hits": [
{
"_index": "attractions",
"_type": "restaurant",
"_id": "3",
"_score": 0.15342641,
"_source": {
"name": "Mini Munchies Pizza",
"location": [
-73.983,
40.719
]
}
}
],
"aggregations": {
"per_ring": {
"buckets": [
{
"key": "*-1.0",
"doc_count": 1
}
]
}
}
}
```

```

"from": 0,
"to": 1,
"doc_count": 1
},
{
"key": "1.0-2.0",
"from": 1,
"to": 2,
"doc_count": 1
}
]
}
}

```

Geo-shapes

Geo-shapes use a completely different approach than geo-points. A circle on a computer screen does not consist of a perfect continuous line. Instead it is drawn by coloring adjacent pixels as an approximation of a circle. Geo-shapes work in much the same way.

Complex shapes—such as points, lines, polygons, multipolygons, and polygons with holes,—are “painted” onto a grid of geohash cells, and the shape is converted into a list of the geohashes of all the cells that it touches.

That is the extent of what you can do with geo-shapes: determine the relationship between a query shape and a shape in the index. The relation can be one of the following:

intersects – The query shape overlaps with the indexed shape (default).

disjoint – The query shape does not overlap at all with the indexed shape.

Within – The indexed shape is entirely within the query shape.

Geo-shapes cannot be used to calculate distance, cannot be used for sorting or scoring, and cannot be used in aggregations.

Mapping geo-shapes

Like fields of type geo_point, geo-shapes have to be mapped explicitly before they can be used:

PUT /attractions

```
{
  "mappings": {
    "landmark": {
      "properties": {
        "name": {

```

```
"type": "string"
},
{
"location": {
"type": "geo_shape"
}
}
}
}
}
}
```

There are two important settings that you should consider changing precision and distance_error_pct.

For instance, we can index a polygon representing Dam Square in Amsterdam as follows:

```
PUT /attractions/landmark/dam_square
{
  "name" : "Dam Square, Amsterdam",
  "location" : {
    "type" : "polygon",
    "coordinates" : [
      [
        [
          [
            [
              [
                [
                  [
                    [
                      [
                        [
                          [
                            [
                              [
                                [
                                  [
                                    [
                                      [
                                        [
                                          [
                                            [
                                              [
                                                [
                                                  [
                                                    [
                                                      [
                                                        [
                                                          [
                                                            [
                                                              [
                                                                [
                                                                  [
                                                                    [
                                                                      [
                                                                        [
                                                                          [
                                                                            [
                                                                              [
                                                                                [
                                                                                  [
                                                                                    [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
................................................................
```

Querying geo-shapes

For instance, if our user steps out of the central train station in Amsterdam, we could find all landmarks within a 1km radius with a query like this:

```
GET /attractions/landmark/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "shape": {
```

```
"type": "circle",
"radius": "1km"
"coordinates": [
4.89994,
52.37815
]
}
}
}
}
}
}
```

Modeling Your Data

Elasticsearch is a different kind of beast, especially if you come from the world of SQL. It comes with many benefits: performance, scale, near real-time search, and analytics across massive amounts of data. And it is easy to get going! Just download and start using it.

But it is not magic. To get the most out of Elasticsearch, you need to understand how it works and how to make it work for your needs.

Handling Relationships

In the real world, relationships matter: blog posts have comments, bank accounts have transactions, customers have bank accounts, orders have order lines, and directories have files and subdirectories.

Relational databases are specifically designed—and this will not come as a surprise to you—to manage relationships:

- Each entity (or row, in the relational world) can be uniquely identified by a primary key.
- Entities are normalized. The data for a unique entity is stored only once, and related entities store just its primary key. Changing the data of an entity has to happen in only one place.
- Entities can be joined at query time, allowing for cross-entity search.
- Changes to a single entity are atomic, consistent, isolated, and durable. (See [ACID Transactions](#) for more on this subject.)
- Most relational databases support ACID transactions across multiple entities.

But relational databases do have their limitations, besides their poor support for fulltext search. Joining entities at query time is expensive—more joins that are required, the more expensive the query. Performing joins between entities that live on different hardware is so expensive that it is just not practical. This places a limit on the amount of data that can be stored on a single server.

Elasticsearch, like most NoSQL databases, treats the world as though it were flat. An index is a flat collection of independent documents. A single document should contain all of the information that is required to decide whether it matches a search request.

While changing the data of a single document in Elasticsearch is **ACIDic**, transactions involving multiple documents are not. There is no way to roll back the index to its previous state if part of a transaction fails.

This FlatWorld has its advantages:

- Indexing is fast and lock-free.
- Searching is fast and lock-free.
- Massive amounts of data can be spread across multiple nodes, because each document is independent of the others.

But relationships matter. Somehow, we need to bridge the gap between FlatWorld and the real world. Four common techniques are used to manage relational data in Elasticsearch:

- Application-side joins
- Data denormalization
- Nested objects
- Parent/child relationships

Often the final solution will require a mixture of a few of these techniques.

Designing for Scale Elasticsearch is used by some companies to index and search petabytes of data every day, but most of us start out with something a little more humble in size. Even if we aspire to be the next Facebook, it is unlikely that our bank balance matches our aspirations.

We need to build for what we have today, but in a way that will allow us to scale out flexibly and rapidly.

Elasticsearch is built to scale. It will run very happily on your laptop or in a cluster containing hundreds of nodes, and the experience is almost identical. Growing from a small cluster to a large cluster is almost entirely automatic and painless. Growing from a large cluster to a very large cluster requires a bit more planning and design, but it is still relatively painless.

Of course, it is not magic. Elasticsearch has its limitations too. If you are aware of those limitations and work with them, the growing process will be pleasant. If you treat Elasticsearch badly, you could be in for a world of pain.

The default settings in Elasticsearch will take you a long way, but to get the most bang for your buck, you need to think about how data flows through your system. We will talk about two common data flows: time-based data (such as log events or social network streams, where relevance is driven by recency), and user-based data (where a large document collection can be subdivided by user or customer).

This chapter will help you make the right decisions up front, to avoid nasty surprises later.

The Unit of Scale

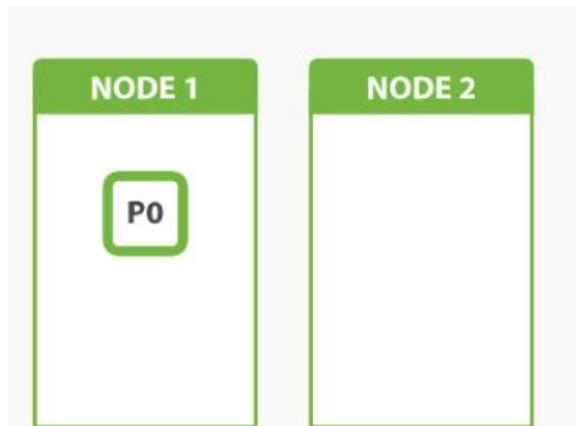
we explained that a shard is a Lucene index and that an Elasticsearch index is a collection of shards. Your application talks to an index, and Elasticsearch routes your requests to the appropriate shards.

A shard is the unit of scale. The smallest index you can have is one with a single shard.

This may be more than sufficient for your needs—a single shard can hold a lot of data—but it limits your ability to scale.

Imagine that our cluster consists of one node, and in our cluster we have one index, which has only one shard:

```
PUT /my_index
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0
  }
}
```



One glorious day, the Internet discovers us, and a single node just can't keep up with the traffic. We decide to add a second node, as per [Figure 43-1](#). What happens?

The answer is: nothing. Because we have only one shard, there is nothing to put on the second node. We can't increase the number of shards in the index, because the number of shards is an important element in the algorithm used to [route documents to shards](#):

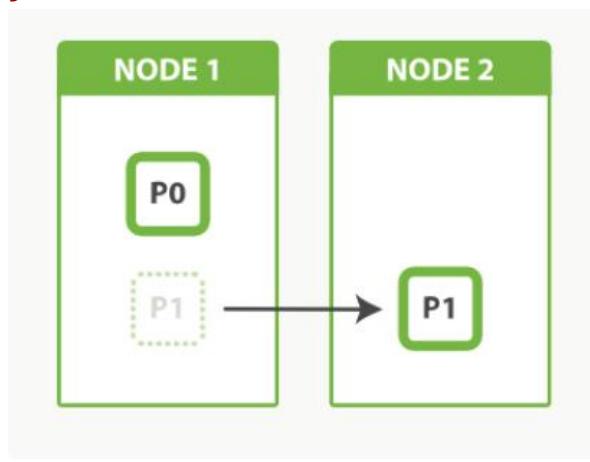
```
shard = hash(routing) % number_of_primary_shards
```

Our only option now is to reindex our data into a new, bigger index that has more shards, but that will take time that we can ill afford. By planning ahead, we could have avoided this problem completely by overallocating.

Shard Overallocation

A shard lives on a single node, but a node can hold multiple shards. Imagine that we created our index with two primary shards instead of one:

```
PUT /my_index
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 0
  }
}
```



Shard Splitting

Users often ask why Elasticsearch doesn't support shard-splitting—the ability to split each shard into two or more pieces. The reason is that shard-splitting is a bad idea:

- Splitting a shard is almost equivalent to reindexing your data. It's a much heavier process than just copying a shard from one node to another.
- Splitting is exponential. You start with one shard, then split into two, and then four, eight, sixteen, and so on. Splitting doesn't allow you to increase capacity by just 50%.
- Shard splitting requires you to have enough capacity to hold a second copy of your index. Usually, by the time you realize that you need to scale out, you don't have enough free space left to perform the split.

In a way, Elasticsearch does support shard splitting. You can always reindex your data to a new index with the appropriate number of shards (see “[Reindexing Your Data](#)”). It is still a more intensive process than moving shards around, and still requires enough free space to complete, but at least you can control the number of shards in the new index.

Kagillion Shards

The first thing that new users do when they learn about **shard overallocation** is to say to themselves:

I don't know how big this is going to be, and I can't change the index size later on, so to be on the safe side, I'll just give this index 1,000 shards...

—A new user One thousand shards—really? And you don't think that, perhaps, between now and the time you need to buy one thousand nodes, that you may need to rethink your data model once or twice and have to reindex?

A shard is not free. Remember:

- A shard is a Lucene index under the covers, which uses file handles, memory, and CPU cycles.
- Every search request needs to hit a copy of every shard in the index. That's fine if every shard is sitting on a different node, but not if many shards have to compete for the same resources.
- Term statistics, used to calculate relevance, are per shard. Having a small amount of data in many shards leads to poor relevance.

Capacity Planning:

If 1 shard is too few and 1,000 shards are too many, how do I know how many shards I need? This is a question that is impossible to answer in the general case. There are just too many variables: the hardware that you use, the size and complexity of your documents, how you index and analyze those documents, the types of queries that you run, the aggregations that you perform, how you model your data, and more.

Fortunately, it is an easy question to answer in the specific case—yours:

1. Create a cluster consisting of a single server, with the hardware that you are considering using in production.
2. Create an index with the same settings and analyzers that you plan to use in production, but with only one primary shard and no replicas.
3. Fill it with real documents (or as close to real as you can get).
4. Run real queries and aggregations (or as close to real as you can get).

Essentially, you want to replicate real-world usage and to push this single shard until it “breaks.” Even the definition of breaks depends on you: some users require that all responses return within 50ms; others are quite happy to wait for 5 seconds.

Once you define the capacity of a single shard, it is easy to extrapolate that number to your whole index. Take the total amount of data that you need to index, plus some extra for future growth, and divide by the capacity of a single shard. The result is the number of primary shards that you will need.

Note: Capacity planning should not be your first step. First look for ways to optimize how you are using Elasticsearch. Perhaps you have inefficient queries, not enough RAM, or you have left swap enabled? We have seen new users who, frustrated by initial performance, immediately start trying to tune the garbage collector or adjust the number of threads, instead of tackling the simple problems like removing wildcard queries.

Replica Shards

Up until now we have spoken only about primary shards, but we have another tool in our belt: replica shards. The main purpose of replicas is for failover, as discussed earlier, if the node holding a primary shard dies, a replica is promoted to the role of primary.

At index time, a replica shard does the same amount of work as the primary shard. New documents are first indexed on the primary and then on any replicas. Increasing the number of replicas does not change the capacity of the index.

However, replica shards can serve read requests. If, as is often the case, your index is search heavy, you can increase search performance by increasing the number of replicas, but only if you also add extra hardware.

Let's return to our example of an index with two primary shards. We increased capacity of the index by adding a second node. Adding more nodes would not help us to add indexing capacity, but we could take advantage of the extra hardware at search time by increasing the number of replicas:

```
POST /my_index/_settings
```

```
{  
  "number_of_replicas": 1  
}
```

Having two primary shards, plus a replica of each primary, would give us a total of four shards: one for each node, as shown in the below fig.

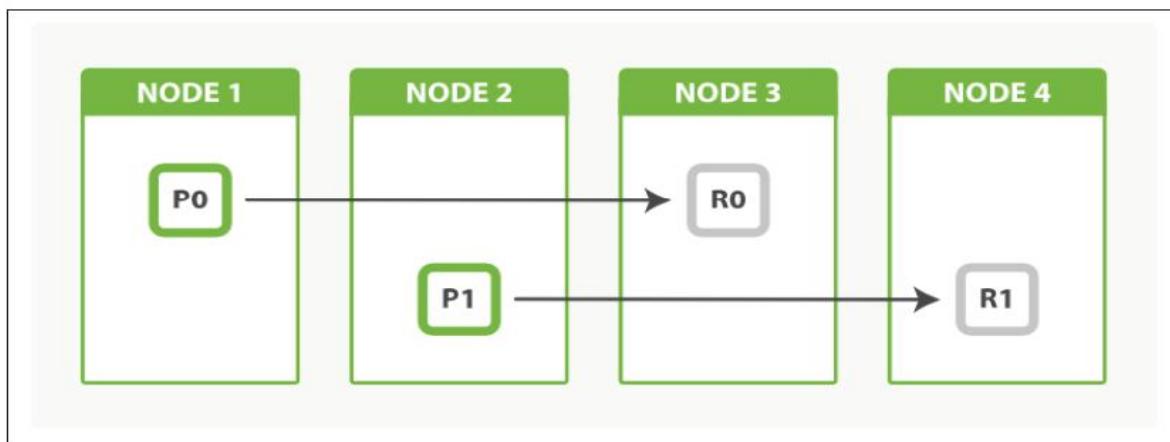


Figure 43-3. An index with two primary shards and one replica can scale out across four nodes

Balancing Load with Replicas

Search performance depends on the response times of the slowest node, so it is a good idea to try to balance out the load across all nodes. If we added just one extra node instead of two, we would end up with two nodes having one shard each, and one node doing double the work with two shards.

We can even things out by adjusting the number of replicas. By allocating two replicas instead of one, we end up with a total of six shards, which can be evenly divided between three nodes, as shown in [Figure 43-4](#):

```
POST /my_index/_settings
```

```
{  
  "number_of_replicas": 2  
}
```

As a bonus, we have also increased our availability. We can now afford to lose two nodes and still have a copy of all our data.

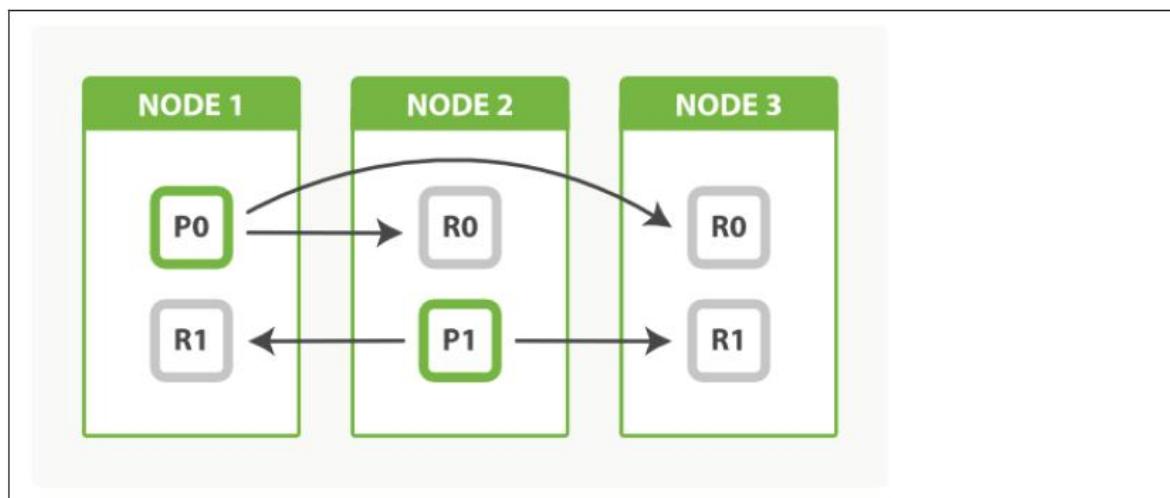


Figure 43-4. Adjust the number of replicas to balance the load between nodes

Note: The fact that node 3 holds two replicas and no primaries is not important. Replicas and primaries do the same amount of work; they just play slightly different roles. There is no need to ensure that primaries are distributed evenly across all nodes.

Multiple Indices

Finally, remember that there is no rule that limits your application to using only a single index. When we issue a search request, it is forwarded to a copy (a primary or a replica) of all the shards in an index. If we issue the same search request on multiple indices, the exact same thing happens—there are just more shards involved.

Note: Searching 1 index of 50 shards is exactly equivalent to searching 50 indices with 1 shard each: both search requests hit 50 shards.

This can be a useful fact to remember when you need to add capacity on the fly. Instead of having to reindex your data into a bigger index, you can just do the following:

- Create a new index to hold new data.
- Search across both indices to retrieve new and old data.

In fact, with a little forethought, adding a new index can be done in a completely transparent way, without your application ever knowing that anything has changed.

In “[Index Aliases and Zero Downtime](#)”, we spoke about using an index alias to point to the current version of your index. For instance, instead of naming your index tweets, name it tweets_v1. Your application would still talk to tweets, but in reality that would be an alias that points to tweets_v1. This allows you to switch the alias to point to a newer version of the index on the fly.

A similar technique can be used to expand capacity by adding a new index. It requires a bit of planning because you will need two aliases: one for searching and one for indexing:

```
PUT /tweets_1/_alias/tweets_search
```

```
PUT /tweets_1/_alias/tweets_index
```

Both the tweets_search and the tweets_index alias point to index tweets_1.

New documents should be indexed into tweets_index, and searches should be performed against tweets_search. For the moment, these two aliases point to the same index.

When we need extra capacity, we can create a new index called tweets_2 and update the aliases as follows:

```
POST /_aliases
```

```
{
  "actions": [
    { "add": { "index": "tweets_2", "alias": "tweets_search" } },
    { "remove": { "index": "tweets_1", "alias": "tweets_index" } },
    { "add": { "index": "tweets_2", "alias": "tweets_index" } }
  ]
}
```

Add index tweets_2 to the tweets_search alias.

Switch tweets_index from tweets_1 to tweets_2.

A search request can target multiple indices, so having the search alias point to tweets_1 and tweets_2 is perfectly valid. However, indexing requests can target only a single index. For this reason, we have to switch the index alias to point to only the new index.

Note: A document GET request, like an indexing request, can target only one index. This makes retrieving a document by ID a bit more complicated in this scenario. Instead, run a search request with the `ids` query, or do a `multi-get` request on `tweets_1` and `tweets_2`.

Time-Based Data

One of the most common use cases for Elasticsearch is for logging, so common in fact that Elasticsearch provides an integrated logging platform called the ELK stack—Elasticsearch, Logstash, and Kibana—to make the process easy.

`Logstash` collects, parses, and enriches logs before indexing them into Elasticsearch. Elasticsearch acts as a centralized logging server, and `Kibana` is a graphic frontend that makes it easy to query and visualize what is happening across your network in near real-time.

Most traditional use cases for search engines involve a relatively static collection of documents that grows slowly. Searches look for the most relevant documents, regardless of when they were created.

Logging—and other time-based data streams such as social-network activity—are very different in nature. The number of documents in the index grows rapidly, often accelerating with time. Documents are almost never updated, and searches mostly target the most recent documents. As documents age, they lose value.

We need to adapt our index design to function with the flow of time-based data.

Index per Time Frame

If we were to have one big index for documents of this type, we would soon run out of space. Logging events just keep on coming, without pause or interruption. We could delete the old events, with a delete-by-query:

```
DELETE /logs/event/_query
{
  "query": {
    "range": {
      "@timestamp": {
        "lt": "now-90d"
      }
    }
  }
}
```

Deletes all documents where Logstash's `@timestamp` field is older than 90 days. But this approach is very inefficient. Remember that when you delete a document, it is only marked as deleted (see “[Deletes and Updates](#)”). It won't be physically deleted until the segment containing it is merged away.

Instead, use an index per time frame. You could start out with an index per year (`logs_2014`) or per month (`logs_2014-10`). Perhaps, when your website gets really busy, you need to switch to an index per day (`logs_2014-10-24`). Purging old data is easy: just delete old indices.

This approach has the advantage of allowing you to scale as and when you need to. You don't have to make any difficult decisions up front. Every day is a new opportunity to change your indexing time frames to suit the current demand. Apply the same logic to how big you make each index. Perhaps all you need is one primary shard per week initially. Later, maybe you need five primary shards per day. It doesn't matter—you can adjust to new circumstances at any time.

Aliases can help make switching indices more transparent. For indexing, you can point `logs_current` to the index currently accepting new log events, and for searching, update `last_3_months` to point to all indices for the previous three months:

```
POST /_aliases
{
  "actions": [
    { "add": { "alias": "logs_current", "index": "logs_2014-10" } },
    { "remove": { "alias": "logs_current", "index": "logs_2014-09" } },
    { "add": { "alias": "last_3_months", "index": "logs_2014-10" } },
    { "remove": { "alias": "last_3_months", "index": "logs_2014-07" } }
  ]
}
```

Switch `logs_current` from September to October.

Add October to `last_3_months` and remove July.

Index Templates

Elasticsearch doesn't require you to create an index before using it. With logging, it is often more convenient to rely on index autorecreation than to have to create indices manually.

Logstash uses the timestamp from an event to derive the index name. By default, it indexes into a different index every day, so an event with a `@timestamp` of `2014-10-01 00:00:01` will be sent to the index `logstash-2014.10.01`. If that index doesn't already exist, it will be created for us.

Usually we want some control over the settings and mappings of the new index. Perhaps we want to limit the number of shards to 1, and we want to disable the `_all` field. Index templates can be used to control which settings should be applied to newly created indices:

```
PUT /_template/my_logs - 1
{
```

```

"template": "logstash-*", -2
"order": 1, -3
"settings": {
"number_of_shards": 1      -4
},
"mappings": {
"_default": {              -5
"_all": {
"enabled": false
}
}
},
"aliases": {
"last_3_months": {}       -6
}
}

```

1. 1 Create a template called my_logs.
2. 2 Apply this template to all indices beginning with logstash-.
3. 3 This template should override the default logstash template that has a lower order.
4. 4 Limit the number of primary shards to 1.
5. 5 Disable the _all field for all types.
6. 6 Add this index to the last_3_months alias.

Retiring Data

As time-based data ages, it becomes less relevant. It's possible that we will want to see what happened last week, last month, or even last year, but for the most part, we're interested in only the here and now.

The nice thing about an index per time frame is that it enables us to easily delete old data: just delete the indices that are no longer relevant:

DELETE /logs_2013*

Deleting a whole index is much more efficient than deleting individual documents: Elasticsearch just removes whole directories.

But deleting an index is very final. There are a number of things we can do to help data age gracefully, before we decide to delete it completely.

Migrate Old Indices

With logging data, there is likely to be one hot index—the index for today. All new documents will be added to that index, and almost all queries will target that index. It should use your best hardware.

How does Elasticsearch know which servers are your best servers? You tell it, by assigning arbitrary tags to each server. For instance, you could start a node as follows:

```
./bin/elasticsearch --node.box_type strong
```

The `box_type` parameter is completely arbitrary—you could have named it whatever you like—but you can use these arbitrary values to tell Elasticsearch where to allocate an index.

We can ensure that today's index is on our strongest boxes by creating it with the following settings:

```
PUT /logs_2014-10-01
```

```
{  
  "settings": {  
    "index.routing.allocation.include.box_type" : "strong"  
  }  
}
```

Yesterday's index no longer needs to be on our strongest boxes, so we can move it to the nodes tagged as medium by updating its index settings:

```
POST /logs_2014-09-30/_settings
```

```
{  
  "index.routing.allocation.include.box_type" : "medium"  
}
```

Optimize Indices

Yesterday's index is unlikely to change. Log events are static: what happened in the past stays in the past. If we merge each shard down to just a single segment, it'll use fewer resources and will be quicker to query. We can do this with the “[optimize API](#)”.

It would be a bad idea to optimize the index while it was still allocated to the strong boxes, as the optimization process could swamp the I/O on those nodes and impact the indexing of today's logs. But the medium boxes aren't doing very much at all, so we are safe to optimize.

Yesterday's index may have replica shards. If we issue an optimize request, it will optimize the primary shard and the replica shards, which is a waste. Instead, we can remove the replicas temporarily, optimize, and then restore the replicas:

```
POST /logs_2014-09-30/_settings
```

```
{ "number_of_replicas": 0 }
```

```
POST /logs_2014-09-30/_optimize?max_num_segments=1
```

```
POST /logs_2014-09-30/_settings
```

```
{ "number_of_replicas": 1 }
```

Of course, without replicas, we run the risk of losing data if a disk suffers catastrophic failure. You may want to back up the data first, with the [snapshot-restore API](#).

Closing Old Indices

As indices get even older, they reach a point where they are almost never accessed. We could delete them at this stage, but perhaps you want to keep them around just in case somebody asks for them in six months.

These indices can be closed. They will still exist in the cluster, but they won't consume resources other than disk space. Reopening an index is much quicker than restoring it from backup.

Before closing, it is worth flushing the index to make sure that there are no transactions left in the transaction log. An empty transaction log will make index recovery faster when it is reopened:

```
POST /logs_2014-01-*/_flush
```

```
POST /logs_2014-01-*/_close
```

```
POST /logs_2014-01-*/_open
```

Flush all indices from January to empty the transaction logs.

Close all indices from January.

When you need access to them again, reopen them with the open API.

Archiving Old Indices

Finally, very old indices can be archived off to some long-term storage like a shared disk or Amazon's S3 using the [snapshot-restore API](#), just in case you may need to access them in the future. Once a backup exists, the index can be deleted from the cluster.

Multiple forums can be queried by passing a comma-separated list of routing values, and including each forum_id in a terms filter:

```
GET /forums/post/_search?routing=baking,cooking,recipes
```

```
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "title": "ginger nuts"
        }
      }
    },
    "filter": {
      "terms": {
        "forum_id": [
          "baking", "cooking", "recipes"
        ]
      }
    }
  }
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

While this approach is technically efficient, it looks a bit clumsy because of the need to specify routing values and terms filters on every query or indexing request. Index aliases to the rescue!

Administration, Monitoring, and Deployment

The majority of this book is aimed at building applications by using Elasticsearch as the backend. This section is a little different. Here, you will learn how to manage Elasticsearch itself. Elasticsearch is a complex piece of software, with many moving parts. Many APIs are designed to help you manage your Elasticsearch deployment.

In this chapter, we cover three main topics:

- Monitoring your cluster's vital statistics, understanding which behaviors are normal and which should be cause for alarm, and interpreting various stats provided by Elasticsearch
- Deploying your cluster to production, including best practices and important configuration that should (or should not!) be changed
- Performing post-deployment logistics, such as a rolling restart or backup of your Cluster

Marvel for Monitoring - marvel sense /kibana

Cluster Health - Head plugin

OS and Process Sections

The OS and Process sections are fairly self-explanatory and won't be covered in great detail. They list basic resource statistics such as CPU and load. The OS section describes it for the entire OS, while the Process section shows just what the Elasticsearch JVM process is using.

These are obviously useful metrics, but are often being measured elsewhere in your monitoring stack. Some stats include the following:

- CPU
- Load
- Memory usage
- Swap usage
- Open file descriptors

JVM Section

The jvm section contains some critical information about the JVM process that is running Elasticsearch. Most important, it contains garbage collection details, which have a large impact on the stability of your Elasticsearch cluster.

Garbage Collection Primer

Before we describe the stats, it is useful to give a crash course in garbage collection and its impact on Elasticsearch. If you are familiar with garbage collection in the JVM, feel free to skip down.

Java is a garbage-collected language, which means that the programmer does not manually manage memory allocation and deallocation. The programmer simply writes code, and the Java Virtual Machine (JVM) manages the process of allocating memory as needed, and then later cleaning up that memory when no longer needed.

When memory is allocated to a JVM process, it is allocated in a big chunk called the heap. The JVM then breaks the heap into two groups, referred to as generations:

Young (or Eden)

The space where newly instantiated objects are allocated. The young generation space is often quite small, usually 100 MB–500 MB. The young-gen also contains two survivor spaces.

Old

The space where older objects are stored. These objects are expected to be longlived and persist for a long time. The old-gen is often much larger than the young-gen, and Elasticsearch nodes can see old-gens as large as 30 GB.

When an object is instantiated, it is placed into young-gen. When the young generation space is full, a young-gen garbage collection (GC) is started. Objects that are still “alive” are moved into one of the survivor spaces, and “dead” objects are removed. If an object has survived several young-gen GCs, it will be “tenured” into the old generation.

A similar process happens in the old generation: when the space becomes full, a garbage collection is started and dead objects are removed.

Nothing comes for free, however. Both the young- and old-generation garbage collectors have phases that “stop the world.” During this time, the JVM literally halts execution of the program so it can trace the object graph and collect dead objects. During this stop-the-world phase, nothing happens. Requests are not serviced, pings are not responded to, shards are not relocated. The world quite literally stops.

This isn’t a big deal for the young generation; its small size means GCs execute quickly. But the old-gen is quite a bit larger, and a slow GC here could mean 1s or even 15s of pausing—which is unacceptable for server software.

The garbage collectors in the JVM are very sophisticated algorithms and do a great

job minimizing pauses. And Elasticsearch tries very hard to be garbage-collection friendly, by intelligently reusing objects internally, reusing network buffers, and offering features like “[Doc Values](#)”. But ultimately, GC frequency and duration is a metric that needs to be watched by you, since it is the number one culprit for cluster instability.

A cluster that is frequently experiencing long GC will be a cluster that is under heavy load with not enough memory. These long GCs will make nodes drop off the cluster for brief periods. This instability causes shards to relocate frequently as Elasticsearch tries to keep the cluster balanced and enough replicas available. This in turn increases network traffic and disk I/O, all while your cluster is attempting to service the normal indexing and query load.

In short, long GCs are bad and need to be minimized as much as possible.

Because garbage collection is so critical to Elasticsearch, you should become intimately familiar with this section of the node-stats API:

```
"jvm": {  
  "timestamp": 1408556438203,  
  "uptime_in_millis": 14457,  
  "mem": {  
    "heap_used_in_bytes": 457252160,  
    "heap_used_percent": 44,  
    "heap_committed_in_bytes": 1038876672,  
    "heap_max_in_bytes": 1038876672,  
    "non_heap_used_in_bytes": 38680680,  
    "non_heap_committed_in_bytes": 38993920,
```

- The jvm section first lists some general stats about heap memory usage. You can see how much of the heap is being used, how much is committed (actually allocated to the process), and the max size the heap is allowed to grow to. Ideally, heap_committed_in_bytes should be identical to heap_max_in_bytes. If the committed size is smaller, the JVM will have to resize the heap eventually—and this is a very expensive process. If your numbers are not identical, see “[Heap: Sizing and Swapping](#)” for how to configure it correctly.

The heap_used_percent metric is a useful number to keep an eye on. Elasticsearch is configured to initiate GCs when the heap reaches 75% full. If your node is consistently $\geq 75\%$, your node is experiencing memory pressure. This is a warning sign that slow GCs may be in your near future.

If the heap usage is consistently $\geq 85\%$, you are in trouble. Heaps over 90–95% are in risk of horrible performance with long 10–30s GCs at best, and out-ofmemory (OOM) exceptions at worst.

```

"pools": {
  "young": {
    "used_in_bytes": 138467752,
    "max_in_bytes": 279183360,
    "peak_used_in_bytes": 279183360,
    "peak_max_in_bytes": 279183360
  },
  "survivor": {
    "used_in_bytes": 34865152,
    "max_in_bytes": 34865152,
    "peak_used_in_bytes": 34865152,
    "peak_max_in_bytes": 34865152
  },
  "old": {
    "used_in_bytes": 283919256,
    "max_in_bytes": 724828160,
    "peak_used_in_bytes": 283919256,
    "peak_max_in_bytes": 724828160
  }
}
}
},

```

- The young, survivor, and old sections will give you a breakdown of memory usage of each generation in the GC. These stats are handy for keeping an eye on relative sizes, but are often not overly important when debugging problems.

```

"gc": {
  "collectors": {
    "young": {
      "collection_count": 13,
      "collection_time_in_millis": 923
    },
    "old": {
      "collection_count": 0,
      "collection_time_in_millis": 0
    }
  }
}

```

- gc section shows the garbage collection counts and cumulative time for both young and old generations. You can safely ignore the young generation counts for the most part: this number will usually be large. That is perfectly normal.

In contrast, the old generation collection count should remain small, and have a small `collection_time_in_millis`. These are cumulative counts, so it is hard to give an exact number when you should start worrying (for example, a node with a one-year uptime will have a large count even if it is healthy). This is one of the reasons that tools such as Marvel are so helpful. GC counts over time are the important consideration.

Time spent GC'ing is also important. For example, a certain amount of garbage is generated while indexing documents. This is normal and causes a GC every now and then. These GCs are almost always fast and have little effect on the node: young generation takes a millisecond or two, and old generation takes a few hundred milliseconds. This is much different from 10-second GCs.

Threadpool Section

Elasticsearch maintains threadpools internally. These threadpools cooperate to get work done, passing work between each other as necessary. In general, you don't need to configure or tune the threadpools, but it is sometimes useful to see their stats so you can gain insight into how your cluster is behaving.

There are about a dozen threadpools, but they all share the same format:

```
"index": {  
  "threads": 1,  
  "queue": 0,  
  "active": 0,  
  "rejected": 0,  
  "largest": 1,  
  "completed": 1  
}
```

Each threadpool lists the number of threads that are configured (`threads`), how many of those threads are actively processing some work (`active`), and how many work units are sitting in a queue (`queue`).

If the queue fills up to its limit, new work units will begin to be rejected, and you will see that reflected in the `rejected` statistic. This is often a sign that your cluster is starting to bottleneck on some resources, since a full queue means your node/cluster is processing at maximum speed but unable to keep up with the influx of work.

Bulk Rejections

If you are going to encounter queue rejections, it will most likely be caused by bulk indexing requests. It is easy to send many bulk requests to Elasticsearch by using concurrent import processes. More is better, right?

In reality, each cluster has a certain limit at which it can not keep up with ingestion.

Once this threshold is crossed, the queue will quickly fill up, and new bulks will be rejected.

This is a good thing. Queue rejections are a useful form of back pressure. They let you know that your cluster is at maximum capacity, which is much better than sticking data into an in-memory queue. Increasing the queue size doesn't increase performance; it just hides the problem. If your cluster can process only 10,000 docs per second, it doesn't matter whether the queue is 100 or 10,000,000—your cluster can still process only 10,000 docs per second.

The queue simply hides the performance problem and carries a real risk of data-loss. Anything sitting in a queue is by definition not processed yet. If the node goes down, all those requests are lost forever. Furthermore, the queue eats up a lot of memory, which is not ideal.

It is much better to handle queuing in your application by gracefully handling the back pressure from a full queue. When you receive bulk rejections, you should take these steps:

1. Pause the import thread for 3–5 seconds.
2. Extract the rejected actions from the bulk response, since it is probable that many of the actions were successful. The bulk response will tell you which succeeded and which were rejected.
3. Send a new bulk request with just the rejected actions.
4. Repeat from step 1 if rejections are encountered again.

Using this procedure, your code naturally adapts to the load of your cluster and naturally backs off.

Rejections are not errors: they just mean you should try again later.

There are a dozen threadpools. Most you can safely ignore, but a few are good to keep an eye on:

Indexing – Threadpool for normal indexing requests

Bulk – Bulk requests, which are distinct from the nonbulk indexing requests

Get – Get-by-ID operations

Search – All search and query requests

Merging – Threadpool dedicated to managing Lucene merges

Cluster Stats

The cluster-stats API provides similar output to the node-stats. There is one crucial difference: Node Stats shows you statistics per node, while cluster-stats shows you the sum total of all nodes in a single metric.

This provides some useful stats to glance at. You can see for example, that your entire cluster is using 50% of the available heap or that filter cache is not evicting heavily. Its main use is to provide a quick summary that is more extensive than

the clusterhealth, but less detailed than node-stats. It is also useful for clusters that are very large, which makes node-stats output difficult to read.

The API may be invoked as follows:

```
GET _cluster/stats
```

Index Stats

So far, we have been looking at node-centric statistics: How much memory does this node have? How much CPU is being used? How many searches is this node servicing? Sometimes it is useful to look at statistics from an index-centric perspective: How many search requests is this index receiving? How much time is spent fetching docs in that index?

To do this, select the index (or indices) that you are interested in and execute an Index stats API:

```
GET my_index/_stats
```

```
GET my_index,another_index/_stats
```

```
GET _all/_stats
```

Pending Tasks

There are certain tasks that only the master can perform, such as creating a new index or moving shards around the cluster. Since a cluster can have only one master, only one node can ever process cluster-level metadata changes. For 99.9999% of the time,

this is never a problem. The queue of metadata changes remains essentially zero. In some rare clusters, the number of metadata changes occurs faster than the master can process them. This leads to a buildup of pending actions that are queued.

The pending-tasks API will show you what (if any) cluster-level metadata changes are pending in the queue:

```
GET _cluster/pending_tasks
```

Usually, the response will look like this:

```
{  
  "tasks": []  
}
```

This means there are no pending tasks. If you have one of the rare clusters that bottlenecks on the master node, your pending task list may look like this:

```
{  
  "tasks": [  
    {  
      "insert_order": 101,  
      "priority": "URGENT",  
      "source": "create-index [foo_9], cause [api]",  
      "time_in_queue_millis": 86,  
      "time_in_queue": "86ms"  
    }  
  ]  
}
```

```

},
{
  "insert_order": 46,
  "priority": "HIGH",
  "source": "shard-started ([foo_2][1], node[tMTocMvQQgGCKj7QDHl3OA], [P],
s[INITIALIZING]), reason [after recovery from gateway]",
  "time_in_queue_millis": 842,
  "time_in_queue": "842ms"
},
{
  "insert_order": 45,
  "priority": "HIGH",
  "source": "shard-started ([foo_2][0], node[tMTocMvQQgGCKj7QDHl3OA], [P],
s[INITIALIZING]), reason [after recovery from gateway]",
  "time_in_queue_millis": 858,
  "time_in_queue": "858ms"
}
]
}

```

You can see that tasks are assigned a priority (URGENT is processed before HIGH, for example), the order it was inserted, how long the action has been queued and what the action is trying to perform. In the preceding list, there is a create-index action and two shard-started actions pending.

When Should I Worry About Pending Tasks?

As mentioned, the master node is rarely the bottleneck for clusters. The only time it could bottleneck is if the cluster state is both very large and updated frequently. For example, if you allow customers to create as many dynamic fields as they wish, and have a unique index for each customer every day, your cluster state will grow very large. The cluster state includes (among other things) a list of all indices, their types, and the fields for each index.

So if you have 100,000 customers, and each customer averages 1,000 fields and 90 days of retention—that's nine billion fields to keep in the cluster state. Whenever this changes, the nodes must be notified.

The master must process these changes, which requires nontrivial CPU overhead, plus the network overhead of pushing the updated cluster state to all nodes.

It is these clusters that may begin to see cluster-state actions queuing up. There is no easy solution to this problem, however. You have three options:

- Obtain a beefier master node. Vertical scaling just delays the inevitable, unfortunately.
- Restrict the dynamic nature of the documents in some way, so as to limit the cluster-state size.

- Spin up another cluster after a certain threshold has been crossed.

cat API

If you work from the command line often, the cat APIs will be helpful to you. Named after the linux cat command, these APIs are designed to work like *nix command-line tools.

They provide statistics that are identical to all the previously discussed APIs (Health, node-stats, and so forth), but present the output in tabular form instead of JSON.

This is very convenient for a system administrator, and you just want to glance over your cluster or find nodes with high memory usage.

Executing a plain GET against the cat endpoint will show you all available APIs:

```
GET /_cat
=^.^=
/_cat/allocation
/_cat/shards
/_cat/shards/{index}
/_cat/master
/_cat/nodes
/_cat/indices
/_cat/indices/{index}
/_cat/segments
/_cat/segments/{index}
/_cat/count
/_cat/count/{index}
/_cat/recovery
/_cat/recovery/{index}
/_cat/health
/_cat/pending_tasks
/_cat/aliases
/_cat/aliases/{alias}
/_cat/thread_pool
/_cat/plugins
/_cat/fielddata
/_cat/fielddata/{fields}
```

GET /_cat/health

1408723713 12:08:33 elasticsearch_zach yellow 1 1 114 114 0 0 114

Production Deployment

If you have made it this far in the book, hopefully you've learned a thing or two about Elasticsearch and are ready to deploy your cluster to production. This chapter is not meant to be an exhaustive guide to running your cluster in production, but it covers the key things to consider before putting your cluster live.

Three main areas are covered:

- Logistical considerations, such as hardware recommendations and deployment strategies
- Configuration changes that are more suited to a production environment
- Post-deployment considerations, such as security, maximizing indexing performance, and backups

Hardware

If you've been following the normal development path, you've probably been playing with Elasticsearch on your laptop or on a small cluster of machines laying around. But when it comes time to deploy Elasticsearch to production, there are a few recommendations that you should consider. Nothing is a hard-and-fast rule; Elasticsearch is used for a wide range of tasks and on a bewildering array of machines. But these recommendations provide good starting points based on our experience with production clusters.

Memory

If there is one resource that you will run out of first, it will likely be memory. Sorting and aggregations can both be memory hungry, so enough heap space to accommodate these is important. Even when the heap is comparatively small, extra memory can be given to the OS filesystem cache. Because many data structures used by Lucene are disk-based formats, Elasticsearch leverages the OS cache to great effect.

A machine with 64 GB of RAM is the ideal sweet spot, but 32 GB and 16 GB machines are also common. Less than 8 GB tends to be counterproductive (you end up needing many, many small machines).

CPUs

Most Elasticsearch deployments tend to be rather light on CPU requirements. As such, the exact processor setup matters less than the other resources. You should choose a modern processor with multiple cores. Common clusters utilize two to eight core machines.

If you need to choose between faster CPUs or more cores, choose more cores. The extra concurrency that multiple cores offers will far outweigh a slightly faster clock speed.

Disks

Disks are important for all clusters, and doubly so for indexing-heavy clusters (such as those that ingest log data). Disks are the slowest subsystem in a server, which means that write-heavy clusters can easily saturate their disks, which in turn become the bottleneck of the cluster.

If you can afford SSDs, they are by far superior to any spinning media. SSD-backed nodes see boosts in both query and indexing performance. If you can afford it, SSDs are the way to go.

If you use spinning media, try to obtain the fastest disks possible (high-performance server disks, 15k RPM drives).

Using RAID 0 is an effective way to increase disk speed, for both spinning disks and SSD. There is no need to use mirroring or parity variants of RAID, since high availability is built into Elasticsearch via replicas.

Finally, avoid network-attached storage (NAS). People routinely claim their NAS solution is faster and more reliable than local drives. Despite these claims, we have never seen NAS live up to its hype. NAS is often slower, displays larger latencies with a wider deviation in average latency, and is a single point of failure.

Check Your I/O Scheduler

If you are using SSDs, make sure your OS I/O scheduler is configured correctly. When you write data to disk, the I/O scheduler decides when that data is actually sent to the disk. The default under most *nix distributions is a scheduler called cfq (Completely Fair Queuing).

This scheduler allocates time slices to each process, and then optimizes the delivery of these various queues to the disk. It is optimized for spinning media: the nature of rotating platters means it is more efficient to write data to disk based on physical layout.

This is inefficient for SSD, however, since there are no spinning platters involved. Instead, deadline or noop should be used instead. The deadline scheduler optimizes based on how long writes have been pending, while noop is just a simple FIFO queue.

This simple change can have dramatic impacts. We've seen a 500-fold improvement to write throughput just by using the correct scheduler.

Network

A fast and reliable network is obviously important to performance in a distributed system. Low latency helps ensure that nodes can communicate easily, while high bandwidth helps shard movement and recovery. Modern data-center networking (1 GbE, 10 GbE) is sufficient for the vast majority of clusters.

Avoid clusters that span multiple data centers, even if the data centers are colocated in close proximity. Definitely avoid clusters that span large geographic distances.

Elasticsearch clusters assume that all nodes are equal—not that half the nodes are actually 150ms distant in another data center. Larger latencies tend to exacerbate problems in distributed systems and make debugging and resolution more difficult. Similar to the NAS argument, everyone claims that their pipe between data centers is robust and low latency. This is true—until it isn’t (a network failure will happen eventually;

you can count on it). From our experience, the hassle of managing cross-data center clusters is simply not worth the cost.

General Considerations

It is possible nowadays to obtain truly enormous machines: hundreds of gigabytes of RAM with dozens of CPU cores. Conversely, it is also possible to spin up thousands of small virtual machines in cloud platforms such as EC2. Which approach is best?

In general, it is better to prefer medium-to-large boxes. Avoid small machines, because you don’t want to manage a cluster with a thousand nodes, and the overhead of simply running Elasticsearch is more apparent on such small boxes.

At the same time, avoid the truly enormous machines. They often lead to imbalanced resource usage (for example, all the memory is being used, but none of the CPU) and can add logistical complexity if you have to run multiple nodes per machine.

Java Virtual Machine

You should always run the most recent version of the Java Virtual Machine (JVM), unless otherwise stated on the Elasticsearch website. Elasticsearch, and in particular Lucene, is a demanding piece of software. The unit and integration tests from Lucene often expose bugs in the JVM itself. These bugs range from mild annoyances to serious segfaults, so it is best to use the latest version of the JVM where possible.

Java 7 is strongly preferred over Java 6. Either Oracle or OpenJDK are acceptable.

They are comparable in performance and stability.

If your application is written in Java and you are using the transport client or node client, make sure the JVM running your application is identical to the server JVM. In few locations in Elasticsearch, Java's native serialization is used (IP addresses, exceptions, and so forth). Unfortunately, Oracle has been known to change the serialization format between minor releases, leading to strange errors. This happens rarely, but it is best practice to keep the JVM versions identical between client and server.

Please Do Not Tweak JVM Settings

The JVM exposes dozens (hundreds even!) of settings, parameters, and configurations. They allow you to tweak and tune almost every aspect of the JVM. When a knob is encountered, it is human nature to want to turn it. We implore you to squash this desire and not use custom JVM settings. Elasticsearch is a complex piece of software, and the current JVM settings have been tuned over years of real-world usage.

It is easy to start turning knobs, producing opaque effects that are hard to measure, and eventually detune your cluster into a slow, unstable mess. When debugging clusters, the first step is often to remove all custom configurations. About half the time, this alone restores stability and performance.

Transport Client Versus Node Client

If you are using Java, you may wonder when to use the transport client versus the node client. As discussed at the beginning of the book, the transport client acts as a communication layer between the cluster and your application. It knows the API and can automatically round-robin between nodes, sniff the cluster for you, and more. But it is external to the cluster, similar to the REST clients.

The node client, on the other hand, is actually a node within the cluster (but does not hold data, and cannot become master). Because it is a node, it knows the entire cluster state (where all the nodes reside, which shards live in which nodes, and so forth).

This means it can execute APIs with one less network hop.

There are uses-cases for both clients:

The transport client is ideal if you want to decouple your application from the cluster. For example, if your application quickly creates and destroys connections to the cluster, a transport client is much “lighter” than a node client, since it is not part of a cluster.

Similarly, if you need to create thousands of connections, you don't want to have thousands of node clients join the cluster. The TC will be a better choice.

On the flipside, if you need only a few long-lived, persistent connection objects to the cluster, a node client can be a bit more efficient since it knows the cluster layout. But it ties your application into the cluster, so it may pose problems from a firewall perspective.

Configuration Management

If you use configuration management already (Puppet, Chef, Ansible), you can skip this tip.

If you don't use configuration management tools yet, you should! Managing a handful of servers by parallel-ssh may work now, but it will become a nightmare as you grow your cluster. It is almost impossible to edit 30 configuration files by hand without making a mistake.

Configuration management tools help make your cluster consistent by automating the process of config changes. It may take a little time to set up and learn, but it will pay itself off handsomely over time.

Important Configuration Changes

Elasticsearch ships with very good defaults, especially when it comes to performance-related settings and options. When in doubt, just leave the settings alone. We have witnessed countless dozens of clusters ruined by errant settings because the administrator thought he could turn a knob and gain 100-fold improvement.

Other databases may require tuning, but by and large, Elasticsearch does not. If you are hitting performance problems, the solution is usually better data layout or more nodes. There are very few "magic knobs" in Elasticsearch. If there were, we'd have turned them already!

With that said, there are some logistical configurations that should be changed for production. These changes are necessary either to make your life easier, or because there is no way to set a good default (because it depends on your cluster layout).

Assign Names

Elasticsearch by default starts a cluster named `elasticsearch`. It is wise to rename your production cluster to something else, simply to prevent accidents whereby someone's laptop joins the cluster. A simple change to `elasticsearch_production` can save a lot of heartache.

This can be changed in your `elasticsearch.yml` file:

`cluster.name: elasticsearch_production`

Similarly, it is wise to change the names of your nodes. As you've probably noticed by now, Elasticsearch assigns a random Marvel superhero name to your nodes at startup.

This is cute in development—but less cute when it is 3a.m. and you are trying to remember which physical machine was Tagak the Leopard Lord.

More important, since these names are generated on startup, each time you restart your node, it will get a new name. This can make logs confusing, since the names of all the nodes are constantly changing.

Boring as it might be, we recommend you give each node a name that makes sense to you—a plain, descriptive name. This is also configured in your `elasticsearch.yml`:

```
node.name: elasticsearch_005_data
```

Paths

By default, Elasticsearch will place the plug-ins, logs, and—most important—your data in the installation directory. This can lead to unfortunate accidents, whereby the installation directory is accidentally overwritten by a new installation of Elasticsearch.

If you aren't careful, you can erase all your data.

Don't laugh—we've seen it happen more than a few times.

The best thing to do is relocate your data directory outside the installation location. You can optionally move your plug-in and log directories as well.

This can be changed as follows:

```
path.data: /path/to/data1,/path/to/data2
# Path to log files:
path.logs: /path/to/logs
# Path to where plugins are installed:
path.plugins: /path/to/plugins
```

Data can be saved to multiple directories, and if each directory is mounted on a different hard drive, this is a simple and effective way to set up a software RAID 0. Elasticsearch will automatically stripe data between the different directories, boosting performance.

Minimum Master Nodes

The `minimum_master_nodes` setting is extremely important to the stability of your cluster. This setting helps prevent split brains, the existence of two masters in a single cluster.

When you have a split brain, your cluster is at danger of losing data. Because the master is considered the supreme ruler of the cluster, it decides when new

indices can be created, how shards are moved, and so forth. If you have two masters, data integrity

becomes perilous, since you have two nodes that think they are in charge. This setting tells Elasticsearch to not elect a master unless there are enough mastereligible nodes available. Only then will an election take place.

This setting should always be configured to a quorum (majority) of your mastereligible nodes. A quorum is (number of master-eligible nodes / 2) + 1. Here are some examples:

- If you have ten regular nodes (can hold data, can become master), a quorum is 6.
- If you have three dedicated master nodes and a hundred data nodes, the quorum is 2, since you need to count only nodes that are master eligible.
- If you have two regular nodes, you are in a conundrum. A quorum would be 2, but this means a loss of one node will make your cluster inoperable. A setting of 1 will allow your cluster to function, but doesn't protect against split brain. It is best to have a minimum of three nodes in situations like this.

This setting can be configured in your `elasticsearch.yml` file:

`discovery.zen.minimum_master_nodes: 2`

But because Elasticsearch clusters are dynamic, you could easily add or remove nodes that will change the quorum. It would be extremely irritating if you had to push new configurations to each node and restart your whole cluster just to change the setting.

For this reason, `minimum_master_nodes` (and other settings) can be configured via a dynamic API call. You can change the setting while your cluster is online:

```
PUT /_cluster/settings
{
  "persistent" : {
    "discovery.zen.minimum_master_nodes" : 2
  }
}
```

This will become a persistent setting that takes precedence over whatever is in the static configuration. You should modify this setting whenever you add or remove master-eligible nodes.

Recovery Settings

Several settings affect the behavior of shard recovery when your cluster restarts. First, we need to understand what happens if nothing is configured.

Imagine you have ten nodes, and each node holds a single shard—either a primary or a replica—in a 5 primary / 1 replica index. You take your entire cluster offline for maintenance (installing new drives, for example). When you

restart your cluster, it just so happens that five nodes come online before the other five. Maybe the switch to the other five is being flaky, and they didn't receive the restart command right away. Whatever the reason, you have five nodes online. These five nodes will gossip with each other, elect a master, and form a cluster. They notice that data is no longer evenly distributed, since five nodes are missing from the cluster, and immediately start replicating new shards between each other.

Finally, your other five nodes turn on and join the cluster. These nodes see that their data is being replicated to other nodes, so they delete their local data (since it is now redundant, and may be outdated). Then the cluster starts to rebalance even more, since the cluster size just went from five to ten.

During this whole process, your nodes are thrashing the disk and network, moving data around—for no good reason. For large clusters with terabytes of data, this useless shuffling of data can take a really long time. If all the nodes had simply waited for the cluster to come online, all the data would have been local and nothing would need to move.

Now that we know the problem, we can configure a few settings to alleviate it. First, we need to give Elasticsearch a hard limit:

`gateway.recover_after_nodes: 8`

This will prevent Elasticsearch from starting a recovery until at least eight nodes are present. The value for this setting is a matter of personal preference: how many nodes do you want present before you consider your cluster functional? In this case, we are setting it to 8, which means the cluster is inoperable unless there are eight nodes.

Then we tell Elasticsearch how many nodes should be in the cluster, and how long we want to wait for all those nodes:

`gateway.expected_nodes: 10`

`gateway.recover_after_time: 5m`

What this means is that Elasticsearch will do the following:

- Wait for eight nodes to be present
- Begin recovering after 5 minutes or after ten nodes have joined the cluster, whichever comes first.

These three settings allow you to avoid the excessive shard swapping that can occur on cluster restarts. It can literally make recovery take seconds instead of hours.

Prefer Unicast over Multicast

Elasticsearch is configured to use multicast discovery out of the box. Multicast works by sending UDP pings across your local network to discover nodes. Other

Elasticsearch nodes will receive these pings and respond. A cluster is formed shortly after.

Multicast is excellent for development, since you don't need to do anything. Turn a few nodes on, and they automatically find each other and form a cluster.

This ease of use is the exact reason you should disable it in production. The last thing you want is for nodes to accidentally join your production network, simply because they received an errant multicast ping. There is nothing wrong with multicast per se. Multicast simply leads to silly problems, and can be a bit more fragile (for example, a network engineer fiddles with the network without telling you—and all of a sudden nodes can't find each other anymore).

In production, it is recommended to use unicast instead of multicast. This works by providing Elasticsearch a list of nodes that it should try to contact. Once the node contacts a member of the unicast list, it will receive a full cluster state that lists all nodes in the cluster. It will then proceed to contact the master and join.

This means your unicast list does not need to hold all the nodes in your cluster. It just needs enough nodes that a new node can find someone to talk to. If you use dedicated masters, just list your three dedicated masters and call it a day. This setting is configured in your `elasticsearch.yml`:

```
discovery.zen.ping.multicast.enabled: false  
discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]
```

Make sure you disable multicast, since it can operate in parallel with unicast.

Don't Touch These Settings!

There are a few hotspots in Elasticsearch that people just can't seem to avoid tweaking.

We understand: knobs just beg to be turned. But of all the knobs to turn, these you should really leave alone. They are often abused and will contribute to terrible stability or terrible performance. Or both.

Garbage Collector

the JVM uses a garbage collector to free unused memory. This tip is really an extension of the last tip, but deserves its own section for emphasis:

Do not change the default garbage collector!

The default GC for Elasticsearch is Concurrent-Mark and Sweep (CMS). This GC runs concurrently with the execution of the application so that it can minimize pauses. It does, however, have two stop-the-world phases. It also has trouble collecting large heaps.

Despite these downsides, it is currently the best GC for low-latency server software like Elasticsearch. The official recommendation is to use CMS.

There is a newer GC called the Garbage First GC (G1GC). This newer GC is designed to minimize pausing even more than CMS, and operate on large heaps. It works by dividing the heap into regions and predicting which regions contain the most reclaimable space. By collecting those regions first (garbage first), it can minimize pauses and operate on very large heaps.

Sounds great! Unfortunately, G1GC is still new, and fresh bugs are found routinely. These bugs are usually of the segfault variety, and will cause hard crashes. The Lucene test suite is brutal on GC algorithms, and it seems that G1GC hasn't had the kinks worked out yet.

We would like to recommend G1GC someday, but for now, it is simply not stable enough to meet the demands of Elasticsearch and Lucene.

Threadpools

Everyone loves to tweak threadpools. For whatever reason, it seems people cannot resist increasing thread counts. Indexing a lot? More threads! Searching a lot? More threads! Node idling 95% of the time? More threads!

The default threadpool settings in Elasticsearch are very sensible. For all threadpools (except search) the threadcount is set to the number of CPU cores. If you have eight cores, you can be running only eight threads simultaneously. It makes sense to assign only eight threads to any particular threadpool.

Search gets a larger threadpool, and is configured to # cores * 3.

You might argue that some threads can block (such as on a disk I/O operation), which is why you need more threads. This is not a problem in Elasticsearch: much of the disk I/O is handled by threads managed by Lucene, not Elasticsearch.

Furthermore, threadpools cooperate by passing work between each other. You don't need to worry about a networking thread blocking because it is waiting on a disk write. The networking thread will have long since handed off that work unit to another threadpool and gotten back to networking.

Finally, the compute capacity of your process is finite. Having more threads just forces the processor to switch thread contexts. A processor can run only one thread at a time, so when it needs to switch to a different thread, it stores the current state (registers, and so forth) and loads another thread. If you are lucky, the switch will happen on the same core. If you are unlucky, the switch may migrate to a different core and require transport on an inter-core communication bus.

This context switching eats up cycles simply by doing administrative housekeeping; estimates can peg it as high as 30 μ s on modern CPUs. So unless the thread will be blocked for longer than 30 μ s, it is highly likely that that time would have been better spent just processing and finishing early.

People routinely set threadpools to silly values. On eight core machines, we have run across configs with 60, 100, or even 1000 threads. These settings will simply thrash the CPU more than getting real work done.

So. Next time you want to tweak a threadpool, please don't. And if you absolutely cannot resist, please keep your core count in mind and perhaps set the count to double. More than that is just a waste.

Heap: Sizing and Swapping

The default installation of Elasticsearch is configured with a 1 GB heap. For just about every deployment, this number is far too small. If you are using the default heap values, your cluster is probably configured incorrectly.

There are two ways to change the heap size in Elasticsearch. The easiest is to set an environment variable called `ES_HEAP_SIZE`. When the server process starts, it will read this environment variable and set the heap accordingly. As an example, you can set it via the command line as follows:

```
export ES_HEAP_SIZE=10g
```

Alternatively, you can pass in the heap size via a command-line argument when starting

the process, if that is easier for your setup:

```
./bin/elasticsearch -Xmx=10g -Xms=10g
```

Ensure that the min (`Xms`) and max (`Xmx`) sizes are the same to prevent the heap from resizing at runtime, a very costly process.

Generally, setting the `ES_HEAP_SIZE` environment variable is preferred over setting explicit `-Xmx` and `-Xms` values.

Give Half Your Memory to Lucene

A common problem is configuring a heap that is too large. You have a 64 GB machine —and by golly, you want to give Elasticsearch all 64 GB of memory. More is better!

Heap is definitely important to Elasticsearch. It is used by many in-memory data structures to provide fast operation. But with that said, there is another major user of memory that is off heap: Lucene.

Lucene is designed to leverage the underlying OS for caching in-memory data structures.

Lucene segments are stored in individual files. Because segments are immutable, these files never change. This makes them very cache friendly, and the underlying OS will happily keep hot segments resident in memory for faster access.

Lucene's performance relies on this interaction with the OS. But if you give all available memory to Elasticsearch's heap, there won't be any left over for Lucene. This can seriously impact the performance of full-text search.

The standard recommendation is to give 50% of the available memory to Elasticsearch heap, while leaving the other 50% free. It won't go unused; Lucene will happily gobble up whatever is left over.

Don't Cross 32 GB!

There is another reason to not allocate enormous heaps to Elasticsearch. As it turns out, the JVM uses a trick to compress object pointers when heaps are less than ~32 GB.

In Java, all objects are allocated on the heap and referenced by a pointer. Ordinary object pointers (OOP) point at these objects, and are traditionally the size of the CPU's native word: either 32 bits or 64 bits, depending on the processor. The pointer references the exact byte location of the value.

For 32-bit systems, this means the maximum heap size is 4 GB. For 64-bit systems, the heap size can get much larger, but the overhead of 64-bit pointers means there is more wasted space simply because the pointer is larger. And worse than wasted space, the larger pointers eat up more bandwidth when moving values between main memory and various caches (LLC, L1, and so forth).

Java uses a trick called **compressed oops** to get around this problem. Instead of pointing at exact byte locations in memory, the pointers reference object offsets. This means a 32-bit pointer can reference four billion objects, rather than four billion bytes. Ultimately, this means the heap can grow to around 32 GB of physical size while still using a 32-bit pointer.

Once you cross that magical ~30–32 GB boundary, the pointers switch back to ordinary object pointers. The size of each pointer grows, more CPU–memory bandwidth is used, and you effectively lose memory. In fact, it takes until around 40–50 GB of allocated heap before you have the same effective memory of a 32 GB heap using compressed oops.

The moral of the story is this: even when you have memory to spare, try to avoid crossing the 32 GB heap boundary. It wastes memory, reduces CPU performance, and makes the GC struggle with large heaps.

I Have a Machine with 1 TB RAM!

The 32 GB line is fairly important. So what do you do when your machine has a lot of memory? It is becoming increasingly common to see super-servers with 300–500 GB of RAM.

First, we would recommend avoiding such large machines (see "[Hardware](#)").

But if you already have the machines, you have two practical options:

- Are you doing mostly full-text search? Consider giving 32 GB to Elasticsearch and letting Lucene use the rest of memory via the OS filesystem cache. All that memory will cache segments and lead to blisteringly fast full-text search.

- Are you doing a lot of sorting/aggregations? You'll likely want that memory in the heap then. Instead of one node with 32 GB+ of RAM, consider running two or more nodes on a single machine. Still adhere to the 50% rule, though. So if your machine has 128 GB of RAM, run two nodes, each with 32 GB. This means 64 GB will be used for heaps, and 64 will be left over for Lucene. If you choose this option, set `cluster.routing.allocation.same_shard.host: true` in your config. This will prevent a primary and a replica shard from colocating to the same physical machine (since this would remove the benefits of replica high availability).

Revisit This List Before Production

You are likely reading this section before you go into production. The details covered in this chapter are good to be generally aware of, but it is critical to revisit this entire list right before deploying to production. Some of the topics will simply stop you cold (such as too few available file descriptors).

These are easy enough to debug because they are quickly apparent. Other issues, such as split brains and memory settings, are visible only after something bad happens. At that point, the resolution is often messy and tedious.

It is much better to proactively prevent these situations from occurring by configuring your cluster appropriately before disaster strikes. So if you are going to dog-ear (or bookmark) one section from the entire book, this chapter would be a good candidate.

The week before deploying to production, simply flip through the list presented here and check off all the recommendations.

Post-Deployment

Once you have deployed your cluster in production, there are some tools and best practices to keep your cluster running in top shape. In this short chapter, we talk about configuring settings dynamically, tweaking logging levels, improving indexing performance, and backing up your cluster.

Changing Settings Dynamically

Many settings in Elasticsearch are dynamic and can be modified through the API. Configuration changes that force a node (or cluster) restart are strenuously avoided.

And while it's possible to make the changes through the static configs, we recommend that you use the API instead.

The cluster-update API operates in two modes:

Transient

These changes are in effect until the cluster restarts. Once a full cluster restart takes place, these settings are erased.

Persistent

These changes are permanently in place unless explicitly changed. They will survive full cluster restarts and override the static configuration files.

Transient versus persistent settings are supplied in the JSON body:

```
PUT /_cluster/settings
{
  "persistent" : {
    "discovery.zen.minimum_master_nodes" : 2
  },
  "transient" : {
    "indices.store.throttle.max_bytes_per_sec" : "50mb"
  }
}
```

This persistent setting will survive full cluster restarts.

This transient setting will be removed after the first full cluster restart.

A complete list of settings that can be updated dynamically can be found in the [online reference docs](#).

Logging

Elasticsearch emits a number of logs, which are placed in ES_HOME/logs. The default logging level is INFO. It provides a moderate amount of information, but is designed to be rather light so that your logs are not enormous.

When debugging problems, particularly problems with node discovery (since this often depends on finicky network configurations), it can be helpful to bump up the logging level to DEBUG.

You could modify the logging.yml file and restart your nodes—but that is both tedious and leads to unnecessary downtime. Instead, you can update logging levels through the cluster-settings API that we just learned about.

To do so, take the logger you are interested in and prepend logger. to it. Let's turn up the discovery logging:

```
PUT /_cluster/settings
{
  "transient" : {
    "logger.discovery" : "DEBUG"
  }
}
```

While this setting is in effect, Elasticsearch will begin to emit DEBUG-level logs for the discovery module.

Note: Avoid TRACE. It is extremely verbose, to the point where the logs are no longer useful.

Slowlog

There is another log called the slowlog. The purpose of this log is to catch queries and indexing requests that take over a certain threshold of time. It is useful for hunting down user-generated queries that are particularly slow.

By default, the slowlog is not enabled. It can be enabled by defining the action (query, fetch, or index), the level that you want the event logged at (WARN, DEBUG, and so forth) and a time threshold.

This is an index-level setting, which means it is applied to individual indices:

```
PUT /my_index/_settings
{
  "index.search.slowlog.threshold.query.warn" : "10s",
  "index.search.slowlog.threshold.fetch.debug": "500ms",
  "index.indexing.slowlog.threshold.index.info": "5s"
}
```

Emit a WARN log when queries are slower than 10s.

Emit a DEBUG log when fetches are slower than 500ms.

Emit an INFO log when indexing takes longer than 5s.

You can also define these thresholds in your elasticsearch.yml file. Indices that do not have a threshold set will inherit whatever is configured in the static config.

Once the thresholds are set, you can toggle the logging level like any other logger:

```
PUT /_cluster/settings
{
  "transient" : {
    "logger.index.search.slowlog" : "DEBUG",
    "logger.index.indexing.slowlog" : "WARN"
  }
}
```

Set the search slowlog to DEBUG level.

Set the indexing slowlog to WARN level.

Indexing Performance Tips

If you are in an indexing-heavy environment, such as indexing infrastructure logs, you may be willing to sacrifice some search performance for faster indexing rates. In these scenarios, searches tend to be relatively rare and performed by people internal to your organization. They are willing to wait several seconds for a search, as opposed to a consumer facing a search that must return in milliseconds. Because of this unique position, certain trade-offs can be made that will increase your indexing performance.

Test Performance Scientifically

Performance testing is always difficult, so try to be as scientific as possible in your approach. Randomly fiddling with knobs and turning on ingestion is not a good way to tune performance. If there are too many causes, it is impossible to determine which one had the best effect. A reasonable approach to testing is as follows:

1. Test performance on a single node, with a single shard and no replicas.
2. Record performance under 100% default settings so that you have a baseline to measure against.
3. Make sure performance tests run for a long time (30+ minutes) so you can evaluate long-term performance, not short-term spikes or latencies. Some events (such as segment merging, and GCs) won't happen right away, so the performance profile can change over time.
4. Begin making single changes to the baseline defaults. Test these rigorously, and if performance improvement is acceptable, keep the setting and move on to the next one.

Using and Sizing Bulk Requests

This should be fairly obvious, but use bulk indexing requests for optimal performance. Bulk sizing is dependent on your data, analysis, and cluster configuration, but a good starting point is 5–15 MB per bulk. Note that this is physical size. Document count is not a good metric for bulk size. For example, if you are indexing 1,000 documents per bulk, keep the following in mind:

- 1,000 documents at 1 KB each is 1 MB.
- 1,000 documents at 100 KB each is 100 MB.

Those are drastically different bulk sizes. Bulks need to be loaded into memory at the coordinating node, so it is the physical size of the bulk that is more important than the document count.

Start with a bulk size around 5–15 MB and slowly increase it until you do not see performance gains anymore. Then start increasing the concurrency of your bulk ingestion (multiple threads, and so forth).

Monitor your nodes with Marvel and/or tools such as iostat, top, and ps to see when resources start to bottleneck. If you start to receive EsRejectedExecutionException, your cluster can no longer keep up: at least one resource has reached capacity. Either reduce concurrency, provide more of the limited resource (such as switching from spinning disks to SSDs), or add more nodes.

Note:

When ingesting data, make sure bulk requests are round-robined across all your data nodes. Do not send all requests to a single node, since that single node will need to store all the bulks in memory while processing.

Storage

Disk are usually the bottleneck of any modern server. Elasticsearch heavily uses disks, and the more throughput your disks can handle, the more stable your nodes will be. Here are some tips for optimizing disk I/O:

- Use SSDs. As mentioned elsewhere, they are superior to spinning media.
- Use RAID 0. Striped RAID will increase disk I/O, at the obvious expense of potential failure if a drive dies. Don't use mirrored or parity RAIDS since replicas provide that functionality.
- Alternatively, use multiple drives and allow Elasticsearch to stripe data across them via multiple path.data directories.
- Do not use remote-mounted storage, such as NFS or SMB/CIFS. The latency introduced here is antithetical to performance.
- If you are on EC2, beware of EBS. Even the SSD-backed EBS options are often slower than local instance storage.

Segments and Merging

Segment merging is computationally expensive, and can eat up a lot of disk I/O. Merges are scheduled to operate in the background because they can take a long time to finish, especially large segments. This is normally fine, because the rate of large segment merges is relatively rare.

But sometimes merging falls behind the ingestion rate. If this happens, Elasticsearch will automatically throttle indexing requests to a single thread. This prevents a segment explosion problem, in which hundreds of segments are generated before they can be merged. Elasticsearch will log INFO-level messages stating now throttling indexing when it detects merging falling behind indexing.

Elasticsearch defaults here are conservative: you don't want search performance to be impacted by background merging. But sometimes (especially on SSD, or logging scenarios), the throttle limit is too low.

The default is 20 MB/s, which is a good setting for spinning disks. If you have SSDs, you might consider increasing this to 100–200 MB/s. Test to see what works for your system:

```
PUT /_cluster/settings
{
  "persistent" : {
    "indices.store.throttle.max_bytes_per_sec" : "100mb"
  }
}
```

If you are doing a bulk import and don't care about search at all, you can disable merge throttling entirely. This will allow indexing to run as fast as your disks will allow:

```
PUT /_cluster/settings
{
  "transient" : {
    "indices.store.throttle.type" : "none"
  }
}
```

Setting the throttle type to none disables merge throttling entirely. When you are done importing, set it back to merge to reenable throttling.

If you are using spinning media instead of SSD, you need to add this to your `elasticsearch.yml`:

```
index.merge.scheduler.max_thread_count: 1
```

Spinning media has a harder time with concurrent I/O, so we need to decrease the number of threads that can concurrently access the disk per index. This setting will allow `max_thread_count + 2` threads to operate on the disk at one time, so a setting of 1 will allow three threads.

For SSDs, you can ignore this setting. The default is `Math.min(3, Runtime.getRuntime().availableProcessors() / 2)`, which works well for SSD.

Finally, you can increase `index.translog.flush_threshold_size` from the default 200 MB to something larger, such as 1 GB. This allows larger segments to accumulate in the translog before a flush occurs. By letting larger segments build, you flush less often, and the larger segments merge less often. All of this adds up to less disk I/O overhead and better indexing rates.

Other

Finally, there are some other considerations to keep in mind:

- If you don't need near real-time accuracy on your search results, consider dropping the `index.refresh_interval` of each index to 30s. If you are doing a large import, you can disable refreshes by setting this value to -1 for the duration of the import. Don't forget to reenable it when you are finished!
- If you are doing a large bulk import, consider disabling replicas by setting `index.number_of_replicas: 0`. When documents are replicated, the entire

document is sent to the replica node and the indexing process is repeated verbatim.

This means each replica will perform the analysis, indexing, and potentially merging process.

In contrast, if you index with zero replicas and then enable replicas when ingestion is finished, the recovery process is essentially a byte-for-byte network transfer.

This is much more efficient than duplicating the indexing process.

- If you don't have a natural ID for each document, use Elasticsearch's auto-ID functionality. It is optimized to avoid version lookups, since the autogenerated ID is unique.
- If you are using your own ID, try to pick an ID that is [friendly to Lucene](#)

Examples:

include zero-padded sequential IDs, UUID-1, and nanotime; these IDs have consistent, sequential patterns that compress well. In contrast, IDs such as UUID-4 are essentially random, which offer poor compression and slow down Lucene.

Rolling Restarts

There will come a time when you need to perform a rolling restart of your cluster— keeping the cluster online and operational, but taking nodes offline one at a time.

The common reason is either an Elasticsearch version upgrade, or some kind of maintenance on the server itself (such as an OS update, or hardware). Whatever the case, there is a particular method to perform a rolling restart.

By nature, Elasticsearch wants your data to be fully replicated and evenly balanced. If you shut down a single node for maintenance, the cluster will immediately recognize the loss of a node and begin rebalancing. This can be irritating if you know the node maintenance is short term, since the rebalancing of very large shards can take some time (think of trying to replicate 1TB—even on fast networks this is nontrivial).

What we want to do is tell Elasticsearch to hold off on rebalancing, because we have more knowledge about the state of the cluster due to external factors. The procedure is as follows:

1. If possible, stop indexing new data. This is not always possible, but will help speed up recovery time.
2. Disable shard allocation. This prevents Elasticsearch from rebalancing missing shards until you tell it otherwise. If you know the maintenance window will be short, this is a good idea. You can disable allocation as follows:

```
PUT /_cluster/settings
```

```
{
```

```
"transient" : {
```

```
"cluster.routing.allocation.enable" : "none"
}
}
```

3. Shut down a single node, preferably using the shutdown API on that particular machine:

```
POST /_cluster/nodes/_local/_shutdown
```

4. Perform a maintenance/upgrade.
5. Restart the node, and confirm that it joins the cluster.
6. Reenable shard allocation as follows:

```
PUT /_cluster/settings
```

```
{
"transient" : {
"cluster.routing.allocation.enable" : "all"
}
}
```

Shard rebalancing may take some time. Wait until the cluster has returned to status green before continuing.

7. Repeat steps 2 through 6 for the rest of your nodes.
8. At this point you are safe to resume indexing (if you had previously stopped), but waiting until the cluster is fully balanced before resuming indexing will help to speed up the process.

Backing Up Your Cluster

As with any software that stores data, it is important to routinely back up your data. Elasticsearch replicas provide high availability during runtime; they allow you to tolerate sporadic node loss without an interruption of service.

Replicas do not provide protection from catastrophic failure, however. For that, you need a real backup of your cluster—a complete copy in case something goes wrong.

To back up your cluster, you can use the snapshot API. This will take the current state and data in your cluster and save it to a shared repository. This backup process is “smart.” Your first snapshot will be a complete copy of data, but all subsequent snapshots will save the delta between the existing snapshots and the new data. Data is incrementally added and deleted as you snapshot data over time. This means subsequent backups will be substantially faster since they are transmitting far less data.

To use this functionality, you must first create a repository to save data. There are several repository types that you may choose from:

- Shared filesystem, such as a NAS
- Amazon S3
- HDFS (Hadoop Distributed File System)

- Azure Cloud

Creating the Repository

Let's set up a shared filesystem repository:

```
PUT _snapshot/my_backup
```

```
{
  "type": "fs",
  "settings": {
    "location": "/mount/backups/my_backup"
  }
}
```

We provide a name for our repository, in this case it is called my_backup.

We specify that the type of the repository should be a shared filesystem.

And finally, we provide a mounted drive as the destination.

Note: The shared filesystem path must be accessible from all nodes in your cluster!

This will create the repository and required metadata at the mount point. There are also some other options that you may want to configure, depending on the performance profile of your nodes, network, and repository location:

max_snapshot_bytes_per_sec

When snapshotting data into the repo, this controls the throttling of that process.

The default is 20mb per second.

max_restore_bytes_per_sec

When restoring data from the repo, this controls how much the restore is throttled so that your network is not saturated. The default is 20mb per second.

Let's assume we have a very fast network and are OK with extra traffic, so we can increase the defaults:

```
POST _snapshot/my_backup/
```

```
{
  "type": "fs",
  "settings": {
    "location": "/mount/backups/my_backup",
    "max_snapshot_bytes_per_sec" : "50mb",
    "max_restore_bytes_per_sec" : "50mb"
  }
}
```

Note: that we are using a POST instead of PUT. This will update the settings of the existing repository. Then add our new settings.

Snapshotting All Open Indices

A repository can contain multiple snapshots. Each snapshot is associated with a certain set of indices (for example, all indices, some subset, or a single index). When creating a snapshot, you specify which indices you are interested in and give the snapshot a unique name.

Let's start with the most basic snapshot command:

```
PUT _snapshot/my_backup/snapshot_1
```

This will back up all open indices into a snapshot named `snapshot_1`, under the `my_backup` repository. This call will return immediately, and the snapshot will proceed in the background.

Note: Usually you'll want your snapshots to proceed as a background process, but occasionally you may want to wait for completion in your script. This can be accomplished by adding a `wait_for_completion` flag:

```
PUT _snapshot/my_backup/snapshot_1?wait_for_completion=true
```

This will block the call until the snapshot has completed. Note that large snapshots may take a long time to return!

Snapshotting Particular Indices

The default behavior is to back up all open indices. But say you are using Marvel, and don't really want to back up all the diagnostic .marvel indices. You just don't have enough space to back up everything.

In that case, you can specify which indices to back up when snapshotting your cluster:

PUT _snapshot/my_backup/snapshot_2

{

"indices": "index_1,index_2"

}

This snapshot command will now back up only index1 and index2.

Listing Information About Snapshots

Once you start accumulating snapshots in your repository, you may forget the details relating to each—particularly when the snapshots are named based on time demarcations (for example, backup 2014 10 28).

To obtain information about a single snapshot, simply issue a GET request against the repo and snapshot name:

GET _snapshot/my_backup/snapshot_2

This will return a small response with various pieces of information regarding the snapshot:

{

"snapshots": [

{

```
"snapshot": "snapshot_1",
"indices": [
  ".marvel_2014_28_10",
  "index1",
  "index2"
],
"state": "SUCCESS",
"start_time": "2014-09-02T13:01:43.115Z",
"start_time_in_millis": 1409662903115,
"end_time": "2014-09-02T13:01:43.439Z",
"end_time_in_millis": 1409662903439,
"duration_in_millis": 324,
"failures": [],
"shards": {
  "total": 10,
  "failed": 0,
  "successful": 10
}
}
]
}
```

For a complete listing of all snapshots in a repository, use the `_all` placeholder instead of a snapshot name:

[GET `_snapshot/my_backup/_all`](#)

Deleting Snapshots

Finally, we need a command to delete old snapshots that are no longer useful. This is simply a `DELETE` HTTP call to the `repo/snapshot` name:

[DELETE `_snapshot/my_backup/snapshot_2`](#)

It is important to use the API to delete snapshots, and not some other mechanism (such as deleting by hand, or using automated cleanup tools on S3). Because snapshots are incremental, it is possible that many snapshots are relying on old segments.

The delete API understands what data is still in use by more recent snapshots, and will delete only unused segments.

If you do a manual file delete, however, you are at risk of seriously corrupting your backups because you are deleting data that is still in use.

Monitoring Snapshot Progress

The `wait_for_completion` flag provides a rudimentary form of monitoring, but really isn't sufficient when snapshotting or restoring even moderately sized clusters.

Two other APIs will give you more-detailed status about the state of the snapshotting. First you can execute a GET to the snapshot ID, just as we did earlier get information about a particular snapshot:

`GET _snapshot/my_backup/snapshot_3`

If the snapshot is still in progress when you call this, you'll see information about when it was started, how long it has been running, and so forth. Note, however, that this API uses the same threadpool as the snapshot mechanism. If you are snapshotting very large shards, the time between status updates can be quite large, since the API is competing for the same threadpool resources.

A better option is to poll the `_status` API:

`GET _snapshot/my_backup/snapshot_3/_status`

The `_status` API returns immediately and gives a much more verbose output of statistics:

```
{
  "snapshots": [
    {
      "snapshot": "snapshot_3",
      "repository": "my_backup",
      "state": "IN_PROGRESS",
      "shards_stats": {
        "initializing": 0,
        "started": 1,
        "finalizing": 0,
        "done": 4,
        "failed": 0,
        "total": 5
      },
      "stats": {
        "number_of_files": 5,
        "processed_files": 5,
        "total_size_in_bytes": 1792,
        "processed_size_in_bytes": 1792,
        "start_time_in_millis": 1409663054859,
        "time_in_millis": 64
      },
      "indices": {
        "index_3": {
          "shards_stats": {
            "initializing": 0,
            "started": 0,
            "finalizing": 0,
            "done": 0,
            "failed": 0,
            "total": 0
          }
        }
      }
    }
  ]
}
```

```

"done": 5,
"failed": 0,
"total": 5
},
"stats": {
"number_of_files": 5,
"processed_files": 5,
"total_size_in_bytes": 1792,
"processed_size_in_bytes": 1792,
"start_time_in_millis": 1409663054859,
"time_in_millis": 64
},
"shards": {
"0": {
"stage": "DONE",
"stats": {
"number_of_files": 1,
"processed_files": 1,
"total_size_in_bytes": 514,
"processed_size_in_bytes": 514,
"start_time_in_millis": 1409663054862,
"time_in_millis": 22
}
}
},
...

```

A snapshot that is currently running will show IN_PROGRESS as its status. This particular snapshot has one shard still transferring (the other four have already completed).

The response includes the overall status of the snapshot, but also drills down into perindex and per-shard statistics. This gives you an incredibly detailed view of how the snapshot is progressing. Shards can be in various states of completion:

INITIALIZING

The shard is checking with the cluster state to see whether it can be snapshotted. This is usually very fast.

STARTED

Data is being transferred to the repository.

FINALIZING

Data transfer is complete; the shard is now sending snapshot metadata.

DONE

Snapshot complete!

FAILED

An error was encountered during the snapshot process, and this shard/index/snapshot could not be completed. Check your logs for more information.

Cancelling a Snapshot

Finally, you may want to cancel a snapshot or restore. Since these are long-running processes, a typo or mistake when executing the operation could take a long time to resolve—and use up valuable resources at the same time.

To cancel a snapshot, simply delete the snapshot while it is in progress:

```
DELETE _snapshot/my_backup/snapshot_3
```

This will halt the snapshot process. Then proceed to delete the half-completed snapshot from the repository.

Restoring from a Snapshot

Once you've backed up some data, restoring it is easy: simply add _restore to the ID of the snapshot you wish to restore into your cluster:

```
POST _snapshot/my_backup/snapshot_1/_restore
```

The default behavior is to restore all indices that exist in that snapshot. If snapshot_1 contains five indices, all five will be restored into our cluster. As with the snapshot API, it is possible to select which indices we want to restore.

There are also additional options for renaming indices. This allows you to match index names with a pattern, and then provide a new name during the restore process.

This is useful if you want to restore old data to verify its contents, or perform some other processing, without replacing existing data. Let's restore a single index from the snapshot and provide a replacement name:

```
POST /_snapshot/my_backup/snapshot_1/_restore
```

```
{  
  "indices": "index_1",  
  "rename_pattern": "index_(.+)",  
  "rename_replacement": "restored_index_$1"  
}
```

Restore only the index_1 index, ignoring the rest that are present in the snapshot.

Find any indices being restored that match the provided pattern.

Then rename them with the replacement pattern.

This will restore index_1 into your cluster, but rename it to restored_index_1.

Note: Similar to snapshotting, the restore command will return immediately, and the restoration process will happen in the background.

If you would prefer your HTTP call to block until the restore is finished, simply add the wait_for_completion flag:

```
POST _snapshot/my_backup/snapshot_1/_restore?wait_for_completion=true
```

Monitoring Restore Operations

The restoration of data from a repository piggybacks on the existing recovery mechanisms already in place in Elasticsearch. Internally, recovering shards from a repository is identical to recovering from another node.

If you wish to monitor the progress of a restore, you can use the recovery API. This is a general-purpose API that shows the status of shards moving around your cluster.

The API can be invoked for the specific indices that you are recovering:

```
GET /_recovery/restored_index_3
```

Or for all indices in your cluster, which may include other shards moving around, unrelated to your restore process:

```
GET /_recovery/
```

The output will look similar to this (and note, it can become very verbose depending on the activity of your cluster!):

```
{
  "restored_index_3" : {
    "shards" : [ {
      "id" : 0,
      "type" : "snapshot",
      "stage" : "index",
      "primary" : true,
      "start_time" : "2014-02-24T12:15:59.716",
      "stop_time" : 0,
      "total_time_in_millis" : 175576,
      "source" : {
        "repository" : "my_backup",
        "snapshot" : "snapshot_3",
        "index" : "restored_index_3"
      },
      "target" : {
        "id" : "ryqJ5IO5S4-ISFbGntkEkg",
        "hostname" : "my.fqdn",
        "ip" : "10.0.1.7",
        "name" : "my_es_node"
      },
      "index" : {
        "files" : {
          "total" : 73,
          "reused" : 0,
        }
      }
    }
  }
}
```

```

    "recovered" : 69,
    "percent" : "94.5%"
},
"bytes" : {
"total" : 79063092,
"reused" : 0,
"recovered" : 68891939,
"percent" : "87.1%"
},
"total_time_in_millis" : 0
},
"translog" : {
"recovered" : 0,
"total_time_in_millis" : 0
},
"start" : {
"check_index_time" : 0,
"total_time_in_millis" : 0
}
} ]
}
}

```

The type field tells you the nature of the recovery; this shard is being recovered from a snapshot.

The source hash describes the particular snapshot and repository that is being recovered from.

The percent field gives you an idea about the status of the recovery. This particular shard has recovered 94% of the files so far; it is almost complete.

The output will list all indices currently undergoing a recovery, and then list all shards in each of those indices. Each shard will have stats about start/stop time, duration, recover percentage, bytes transferred, and more.

Canceling a Restore

To cancel a restore, you need to delete the indices being restored. Because a restore process is really just shard recovery, issuing a delete-index API alters the cluster state, which will in turn halt recovery. For example:

`DELETE /restored_index_3`

If `restored_index_3` was actively being restored, this delete command would halt the restoration as well as deleting any data that had already been restored into the cluster.

Clusters Are Living, Breathing Creatures

Once you get a cluster into production, you'll find that it takes on a life of its own. Elasticsearch works hard to make clusters self-sufficient and just work. But a cluster still requires routine care and feeding, such as routine backups and upgrades.

Elasticsearch releases new versions with bug fixes and performance enhancements at a very fast pace, and it is always a good idea to keep your cluster current. Similarly, Lucene continues to find new and exciting bugs in the JVM itself, which means you should always try to keep your JVM up-to-date.

This means it is a good idea to have a standardized, routine way to perform rolling restarts and upgrades in your cluster. Upgrading should be a routine process, rather than a once-yearly fiasco that requires countless hours of precise planning.

Similarly, it is important to have disaster recovery plans in place. Take frequent snapshots of your cluster—and periodically test those snapshots by performing a real recovery! It is all too common for organizations to make routine backups but never test their recovery strategy. Often you'll find a glaring deficiency the first time you perform a real recovery (such as users being unaware of which drive to mount). It's better to work these bugs out of your process with routine testing, rather than at 3 a.m. when there is a crisis.