
FREE eBook
LEARNING
Elasticsearch
Free unaffiliated eBook created from
Stack Overflow contributors.
#elasticsear
ch

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: elasticsearch

It is an unofficial and free Elasticsearch ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Elasticsearch.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Elasticsearch

Remarks

Elasticsearch is an advanced open source search server based on Lucene and written in Java.

It provides distributed full and partial text, query-based and geolocation-based search functionality accessible through an HTTP REST API.

Versions

Version	Release Date
5.2.1	2017-02-14
5.2.0	2017-01-31
5.1.2	2017-01-12
5.1.1	2016-12-08
5.0.2	2016-11-29
5.0.1	2016-11-15
5.0.0	2016-10-26
2.4.0	2016-08-31
2.3.0	2016-03-30
2.2.0	2016-02-02
2.1.0	2015-11-24
2.0.0	2015-10-28
1.7.0	2015-07-16
1.6.0	2015-06-09
1.5.0	2015-03-06
1.4.0	2014-11-05
1.3.0	2014-07-23
1.2.0	2014-05-22

Version	Release Date
1.1.0	2014-03-25
1.0.0	2014-02-14

Examples

Installing Elasticsearch on Ubuntu 14.04

Prerequisites

In order to run Elasticsearch, a Java Runtime Environment (JRE) is required on the machine. Elasticsearch requires Java 7 or higher and recommends `Oracle JDK version 1.8.0_73`.

Install Oracle Java 8

```
sudo add-apt-repository -y ppa:webupd8team/java
sudo apt-get update
echo "oracle-java8-installer shared/accepted-oracle-license-v1-1 select true" | sudo debconf-
set-selections
sudo apt-get install -y oracle-java8-installer
```

Check Java Version

```
java -version
```

Download and Install package

Using Binaries

1. Download the latest stable version of Elasticsearch here.
2. Unzip the file & Run

Linux:

```
$ bin/elasticsearch
```

Using apt-get

An alternative to downloading elasticsearch from the website is installing it, using `apt-get`.

```
wget -qO - https://packages.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
echo "deb https://packages.elastic.co/elasticsearch/2.x/debian stable main" | sudo tee -a
/etc/apt/sources.list.d/elasticsearch-2.x.list
sudo apt-get update && sudo apt-get install elasticsearch
```

```
sudo /etc/init.d/elasticsearch start
```

Installing elasticsearch version 5.x

```
wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
sudo apt-get install apt-transport-https
echo "deb https://artifacts.elastic.co/packages/5.x/apt stable main" | sudo tee -a
/etc/apt/sources.list.d/elastic-5.x.list
sudo apt-get update && sudo apt-get install elasticsearch
```

Running as a service on Linux:

After Installing the above doesn't start itself. so we need to start it as a service. How to start or stop Elasticsearch depends on whether your system uses SysV init or systemd. you can check it with the following command.

```
ps -p 1
```

If your distribution is using SysV init, then you will need to run:

```
sudo update-rc.d elasticsearch defaults 95 10
sudo /etc/init.d/elasticsearch start
```

Otherwise if your distribution is using systemd:

```
sudo /bin/systemctl daemon-reload
sudo /bin/systemctl enable elasticsearch.service
```

Run the `CURL` command from your browser or a REST client, to check if Elasticsearch has been installed correctly.

```
curl -X GET http://localhost:9200/
```

Installing Elasticsearch on Windows

Prerequisites

The Windows version of Elasticsearch can be obtained from this link: https://www.elastic.co/downloads/elasticsearch. The latest stable release is always at the top.

As we are installing on Windows, we need the `.ZIP` archive. Click the link in the `Downloads:` section and save the file to your computer.

This version of elastic is "portable", meaning you don't need to run an installer to use the program. Unzip the contents of the file to a location you can easily remember. For demonstration we'll

assume you unzipped everything to `C:\elasticsearch`.

Note that the archive contains a folder named `elasticsearch-<version>` by default, you can either extract that folder to `C:\` and rename it to `elasticsearch` or create `C:\elasticsearch` yourself, then unzip only the *contents* of the folder in the archive to there.

Because Elasticsearch is written in Java, it needs the Java Runtime Environment to function. So before running the server, check if Java is available by opening a command prompt and typing:

```
java -version
```

You should get a response that looks like this:

```
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) Client VM (build 25.91-b14, mixed mode)
```

If you see the following instead

> 'java' is not recognized as an internal or external command, operable program or batch file.

Java is not installed on your system or is not configured properly. You can follow this tutorial to (re)install Java. Also, make sure that these environment variables are set to similar values:

Variable	Value
JAVA_HOME	C:\Program Files\Java\jre
PATH	…;C:\Program Files\Java\jre

If you don't yet know how to inspect these variables consult this tutorial.

Run from batch file

With Java installed, open the `bin` folder. It can be found directly within the folder you unzipped everything to, so it should be under `c:\elasticsearch\bin`. Within this folder is a file called `elasticsearch.bat` which can be used to start Elasticsearch in a command window. This means that information logged by the process will be visible in the command prompt window. To stop the server, press `CTRLC` or simply close the window.

Run as a Windows service

Ideally you don't want to have an extra window you can't get rid of during development, and for this reason, Elasticsearch can be configured to run as a service.

Before we could install Elasticsearch as a service we need to add a line to the file `C:\elasticsearch\config\jvm.options`:

> The service installer requires that the thread stack size setting be configured in `jvm.options` before you install the service. On 32-bit Windows, you should add `-Xss320k` […] and on 64-bit Windows you should add `-Xss1m` to the `jvm.options` file. [source]

Once you made that change, open a command prompt and navigate to the `bin` directory by running the following command:

```
C:\Users\user> cd c:\elasticsearch\bin
```

Service management is handled by `elasticsearch-service.bat`. In older versions this file might simply be called `service.bat`. To see all available arguments, run it without any:

```
C:\elasticsearch\bin> elasticsearch-service.bat

Usage: elasticsearch-service.bat install|remove|start|stop|manager [SERVICE_ID]
```

The output also tells us that there's an optional `SERVICE_ID` argument, but we can ignore it for now. To install the service, simply run:

```
C:\elasticsearch\bin> elasticsearch-service.bat install
```

After installing the service, you can start and stop it with the respective arguments. To start the service, run

```
C:\elasticsearch\bin> elasticsearch-service.bat start
```

and to stop it, run

```
C:\elasticsearch\bin> elasticsearch-service.bat stop
```

If you prefer a GUI to manage the service instead, you can use the following command:

```
C:\elasticsearch\bin> elasticsearch-service.bat manager
```

This will open the Elastic Service Manager, which allows you to customize some service-related settings as well as stop/start the service using the buttons found at the bottom of the first tab.

Indexing and retrieving a document

Elasticsearch is accessed through a HTTP REST API, typically using the cURL library. The messages between the search server and the client (your or your application) are sent in the form of JSON strings. By default, Elasticsearch runs on port 9200.

In the examples below, `?pretty` is added to tell Elasticsearch to prettify the JSON response. When using these endpoints within an application you needn't add this query parameter.

Indexing documents

If we intend to update information within an index later, it's a good idea to assign unique IDs to the documents we index. To add a document to the index named `megacorp`, with type `employee` and ID `1` run:

```
curl -XPUT "http://localhost:9200/megacorp/employee/1?pretty" -d'
{
    "first_name" : "John",
    "last_name" :  "Smith",
    "age" :        25,
    "about" :      "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}'
```

Response:

```
{
  "_index": "megacorp",
  "_type": "employee",
  "_id": "1",
  "_version": 1,
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "created": true
}
```

The index is created if it does not exist when we send the PUT call.

Indexing without an ID

```
POST /megacorp/employee?pretty
{
    "first_name" :  "Jane",
    "last_name" :   "Smith",
    "age" :         32,
    "about" :       "I like to collect rock albums",
    "interests":  [ "music" ]
}
```

Response:

```
{
  "_index": "megacorp",
  "_type": "employee",
  "_id": "AVYg2mBJYy9ijdngfeGa",
  "_version": 1,
  "_shards": {
```

```
    "total": 2,
    "successful": 2,
    "failed": 0
  },
  "created": true
}
```

Retrieving documents

```
curl -XGET "http://localhost:9200/megacorp/employee/1?pretty"
```

Response:

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

Fetch 10 documents from the `megacorp` index with the type `employee`:

```
curl -XGET "http://localhost:9200/megacorp/employee/_search?pretty"
```

Response:

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 2,
    "max_score": 1,
    "hits": [
      {
        "_index": "megacorp",
        "_type": "employee",
```

```
        "_id": "1",
        "_score": 1,
        "_source": {
          "first_name": "John",
          "last_name": "Smith",
          "age": 25,
          "about": "I love to go rock climbing",
          "interests": [
            "sports",
            "music"
          ]
        }
      },
      {
        "_index": "megacorp",
        "_type": "employee",
        "_id": "AVYg2mBJYy9ijdngfeGa",
        "_score": 1,
        "_source": {
          "first_name": "Jane",
          "last_name": "Smith",
          "age": 32,
          "about": "I like to collect rock albums",
          "interests": [
            "music"
          ]
        }
      }
    ]
  }
}
```

Simple search using the `match` query, which looks for exact matches in the field provided:

```
curl -XGET "http://localhost:9200/megacorp/employee/_search" -d'
{
    "query" : {
        "match" : {
            "last_name" : "Smith"
        }
    }
}'
```

Response:

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0.6931472,
    "hits": [
      {
```

```
        "_index": "megacorp",
        "_type": "employee",
        "_id": "1",
        "_score": 0.6931472,
        "_source": {
          "first_name": "John",
          "last_name": "Smith",
          "age": 25,
          "about": "I love to go rock climbing",
          "interests": [
            "sports",
            "music"
          ]
        }
      }
    ]
  }
}
```

Basic Search Parameters with examples:

By default, the full indexed document is returned as part of all searches. This is referred to as the source (`_source` field in the search hits). If we don't want the entire source document returned, we have the ability to request only a few fields from within source to be returned, or we can set `_source` to false to omit the field entirely.

This example shows how to return two fields, `account_number` and `balance` (inside of `_source`), from the search:

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_all": {} },
  "_source": ["account_number", "balance"]
}'
```

Note that the above example simply reduces the information returned in the `_source` field. It will still only return one field named `_source` but only the fields `account_number` and `balance` will be included.

If you come from a SQL background, the above is somewhat similar in concept to the SQL query

```
SELECT account_number, balance FROM bank;
```

Now let's move on to the query part. Previously, we've seen how the `match_all` query is used to match all documents. Let's now introduce a new query called the match query, which can be thought of as a basic fielded search query (i.e. a search done against a specific field or set of fields).

This example returns the account with the `account_number` set to `20`:

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match": { "account_number": 20 } }
```

```
}'
```

This example returns all accounts containing the term "mill" in the `address`:

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match": { "address": "mill" } }
}'
```

This example returns all accounts containing the term "mill" or "lane" in the `address`:

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match": { "address": "mill lane" } }
}'
```

This example is a variant of `match` (`match_phrase`) that splits the query into terms and only returns documents that contain all terms in the `address` in the same positions relative to each other[1].

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_phrase": { "address": "mill lane" } }
}'
```

Let's now introduce the bool(ean) query. The bool query allows us to compose smaller queries into bigger queries using boolean logic.

This example composes two match queries and returns all accounts containing "mill" and "lane" in the address:

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": {
    "bool": {
      "must": [
        { "match": { "address": "mill" } },
        { "match": { "address": "lane" } }
      ]
    }
  }
}'
```

In the above example, the bool `must` clause specifies all the queries that must be true for a document to be considered a match.

In contrast, this example composes two match queries and returns all accounts containing "mill" or "lane" in the `address`:

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": {
    "bool": {
```

```
      "should": [
        { "match": { "address": "mill" } },
        { "match": { "address": "lane" } }
      ]
    }
  }
}'
```

In the above example, the bool `should` clause specifies a list of queries either of which must be true for a document to be considered a match.

This example composes two match queries and returns all accounts that contain neither "mill" nor "lane" in the `address`:

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": {
    "bool": {
      "must_not": [
        { "match": { "address": "mill" } },
        { "match": { "address": "lane" } }
      ]
    }
  }
}'
```

In the above example, the bool must_not clause specifies a list of queries none of which must be true for a document to be considered a match.

We can combine must, should, and must_not clauses simultaneously inside a bool query. Furthermore, we can compose bool queries inside any of these bool clauses to mimic any complex multi-level boolean logic.

This example returns all accounts that belong to people who are exactly 40 years old and don't live in Washington (`WA` for short):

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": {
    "bool": {
      "must": [
        { "match": { "age": "40" } }
      ],
      "must_not": [
        { "match": { "state": "WA" } }
      ]
    }
  }
}'
```

Installing Elasticsearch and Kibana on CentOS 7

In order to run Elasticsearch, a Java Runtime Environment (JRE) is required on the machine. Elasticsearch requires Java 7 or higher and recommends `Oracle JDK version 1.8.0_73`.

So, be sure if you have Java in your system. If not, then follow the procedure:

```
# Install wget with yum
yum -y install wget

# Download the rpm jre-8u60-linux-x64.rpm for 64 bit
wget --no-cookies --no-check-certificate --header "Cookie:
gpw_e24=http%3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-cookie"
"http://download.oracle.com/otn-pub/java/jdk/8u60-b27/jre-8u60-linux-x64.rpm"

# Download the rpm jre-8u101-linux-i586.rpm for 32 bit
wget --no-cookies --no-check-certificate --header "Cookie:
gpw_e24=http%3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-cookie"
"http://download.oracle.com/otn-pub/java/jdk/8u101-b13/jre-8u101-linux-i586.rpm"

# Install jre-.*.rpm
rpm -ivh jre-.*.rpm
```

Java should be installed by now in your centOS system. You can check it with:

```
java -version
```

Download & install elasticsearch

```
# Download elasticsearch-2.3.5.rpm
wget
https://download.elastic.co/elasticsearch/release/org/elasticsearch/distribution/rpm/elasticsearch/2.3.
2.3.5.rpm

# Install elasticsearch-.*.rpm
rpm -ivh elasticsearch-.*.rpm
```

Running elasticsearch as a systemd service on startup

```
sudo systemctl daemon-reload
sudo systemctl enable elasticsearch
sudo systemctl start elasticsearch

# check the current status to ensure everything is okay.
systemctl status elasticsearch
```

Installing Kibana

First import GPG-key on rpm

```
sudo rpm --import http://packages.elastic.co/GPG-KEY-elasticsearch
```

Then create a local repository `kibana.repo`

```
sudo vi /etc/yum.repos.d/kibana.repo
```

And Add the following content:

```
[kibana-4.4]
name=Kibana repository for 4.4.x packages
baseurl=http://packages.elastic.co/kibana/4.4/centos
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1
```

Now install the kibana by following command:

```
yum -y install kibana
```

Start it with:

```
systemctl start kibana
```

Check status with:

```
systemctl status kibana
```

You may run it as a startup service.

```
systemctl enable kibana
```

Read Getting started with Elasticsearch online:
https://riptutorial.com/elasticsearch/topic/941/getting-started-with-elasticsearch

Chapter 2: Aggregations

Syntax

- "aggregations" : { -"<aggregation_name>" : { -"<aggregation_type>" : { -<aggregation_body> -} -[,"meta" : { [<meta_data_body>] }]? -[,"aggregations" : { [<sub_aggregation>]+ }]? -} -[,"<aggregation_name_2>" : { ... }]* -}

Examples

Avg aggregation

This is a single value metrics aggregation that calculates the average of the numeric values that are extracted from the aggregated documents.

```
POST /index/_search?
{
    "aggs" : {
        "avd_value" : { "avg" : { "field" : "name_of_field" } }
    }
}
```

The above aggregation computes the average grade over all documents. The aggregation type is avg and the field setting defines the numeric field of the documents the average will be computed on. The above will return the following:

```
{
    ...
    "aggregations": {
        "avg_value": {
            "value": 75.0
        }
    }
}
```

The name of the aggregation (avg_grade above) also serves as the key by which the aggregation result can be retrieved from the returned response.

Cardinality Aggregation

A single-value metrics aggregation that calculates an approximate count of distinct values. Values can be extracted either from specific fields in the document or generated by a script.

```
POST /index/_search?size=0
{
    "aggs" : {
        "type_count" : {
            "cardinality" : {
```

```
                "field" : "type"
            }
        }
    }
}
```

Response:

```
{
    ...
    "aggregations" : {
        "type_count" : {
            "value" : 3
        }
    }
}
```

Extended Stats Aggregation

A multi-value metrics aggregation that computes stats over numeric values extracted from the aggregated documents. These values can be extracted either from specific numeric fields in the documents, or be generated by a provided script.

The extended_stats aggregations is an extended version of the stats aggregation, where additional metrics are added such as sum_of_squares, variance, std_deviation and std_deviation_bounds.

```
{
    "aggs" : {
        "stats_values" : { "extended_stats" : { "field" : "field_name" } }
    }
}
```

Sample output:

```
{
    ...

    "aggregations": {
        "stats_values": {
           "count": 9,
           "min": 72,
           "max": 99,
           "avg": 86,
           "sum": 774,
           "sum_of_squares": 67028,
           "variance": 51.55555555555556,
           "std_deviation": 7.180219742846005,
           "std_deviation_bounds": {
            "upper": 100.36043948569201,
            "lower": 71.63956051430799
           }
        }
    }
}
```

Read Aggregations online: https://riptutorial.com/elasticsearch/topic/10745/aggregations

Chapter 3: Analyzers

Remarks

Analyzers take the text from a string field and generate tokens that will be used when querying.

An Analyzer operates in a sequence:

- `CharFilters` (Zero or more)
- `Tokenizer` (One)
- `TokenFilters` (Zero or more)

The analyzer may be applied to mappings so that when fields are indexed, it is done on a per token basis rather than on the string as a whole. When querying, the input string will also be run through the Analyzer. Therefore, if you normalize text in the Analyzer, it will always match even if the query contains a non-normalized string.

Examples

Mapping

An Analyzer can be applied to a mapping by using "analyzer", by default the "standard" Analyzer is used. Alternatively, if you do not wish to have any analyzer used (because tokenizing or normalization would not be useful) you may specify "index":"not_analyzed"

```
PUT my_index
{
  "mappings": {
    "user": {
      "properties": {
        "name": {
          "type": "string"
          "analyzer":  "my_user_name_analyzer"
        },
        "id": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

Multi-fields

Sometimes it maybe useful to have multiple distinct indexes of a field with different Analyzers. You can use the multi-fields capability to do so.

```
PUT my_index
```

```
{
  "mappings": {
    "user": {
      "properties": {
        "name": {
          "type": "string"
          "analyzer": "standard",
          "fields": {
            "special": {
               "type": "string",
               "analyzer": "my_user_name_analyzer"
            },
            "unanalyzed": {
              "type": "string",
              "index": "not_analyzed"
            }
          }
        }
      }
    }
  }
}
```

When querying, instead of simply using "user.name" (which in this case would still use the stanard Analyzer) you can use "user.name.special" or "user.name.unanalyzed". Note that the document will remain unchanged, this only affects indexing.

Analyzers

Analysis in elasticsearch comes into context when you are willing to analyze the data in your index.

Analyzers allow us to perform following:

- Abbreviations
- Stemming
- Typo Handling

We will be looking at each of them now.

1. **Abbreviations**:

   Using analyzers, we can tell elasticsearch how to treat abbreviations in our data i.e. dr => Doctor so whenever we search for doctor keyword in our index, elasticsearch will also return the results which have dr mentioned in them.

2. **Stemming**:

   Using stemming in analyzers allows us to use base words for modified verbs like

Word	Modifications
require	requirement,required

3. **Typo Handling**:

   Analyzers also provide typo handling as while querying if we are searching for particular word say 'resurrection', then elasticsearch will return the results in which typos are present.i.e. it will treat typos like resurection,ressurection as same and will retun the result.

Word	Modifications
resurrection	resurection,ressurection

Analyzers in Elasticsearch

1. Standard
2. Simple
3. Whitespace
4. Stop
5. Keyword
6. Pattern
7. Language
8. Snowball

Ignore case analyzer

Sometimes, we may need to ignore the case of our query, with respect to the match in the document. An analyzer can be used in this case to ignore the case while searching. Each field will have to contain this analyzer in it's property, in order to work:

```
"settings": {
        "analysis": {
            "analyzer": {
                "case_insensitive": {
               "tokenizer": "keyword",
               "filter": ["lowercase"]
            }
            }
        }
    }
```

Read Analyzers online: https://riptutorial.com/elasticsearch/topic/6232/analyzers

Chapter 4: Cluster

Remarks

Cluster Health provides a lot of information about the cluster, such as the number of shards that are allocated ("active") as well as how many are unassigned and relocating. In addition, it provides the current number of nodes and data nodes in the cluster, which can allow you to poll for missing nodes (e.g., if you expect it to be `15`, but it only shows `14`, then you are missing a node).

For someone that knows about Elasticsearch, "assigned" and "unassigned" shards can help them to track down issues.

The most common field checked from Cluster Health is the `status`, which can be in one of three states:

- red
- yellow
- green

The colors each mean one -- and only one -- very simple thing:

1. Red indicates that you are missing *at least* one primary shard.
    - A missing primary shard means that an index cannot be used to write (index) new data in most cases.
        - Technically, you can still index to any primary shards that are available in that index, but practically it means that you cannot because you do not generally control what shard receives any given document.
        - Searching is still possible against a red cluster, but it means that you will get partial results if any index you search is missing shards.
    - In normal circumstances, it just means that the primary shard is being allocated (`initializing_shards`).
    - If a node just left the cluster (e.g., because the machine running it lost power), then it makes sense that you will be missing some primary shards *temporarily*.
        - Any replica shard for that primary shard will be promoted to be the primary shard in this scenario.
2. Yellow indicates that all primary shards are active, but *at least* one replica shard is missing.
    - A missing replica only impacts indexing if consistency settings require it to impact indexing.
        - By default, there is only one replica for any primary and indexing can happen with a single missing replica.
    - In normal circumstances, it just means that the replica shard is being allocated (`initializing_shards`).
    - A one node cluster with replicas enabled will *always* be yellow *at best*. It can be red if a primary shard is not yet assigned.
        - If you only have a single node, then it makes sense to disable replicas because you are not expecting any. Then it can be green.

3. Green indicates that all shards are active.
    - The only shard activity allowed for a green cluster is `relocating_shards`.
    - New indices, and therefore new shards, will cause the cluster to go from red to yellow to green, as each shard is allocated (primary first, making it yellow, then replicas if possible, making it green).
        - In Elasticsearch 5.x and later, new indices will **not** make your cluster red unless it takes them too long to allocate.

Examples

Human readable, tabular Cluster Health with headers

Example uses basic HTTP syntax. Any `<#>` in the example should be removed when copying it.

You can use the `_cat` APIs to get a human readable, tabular output for various reasons.

```
GET /_cat/health?v <1>
```

1. The `?v` is optional, but it implies that you want "verbose" output.

`_cat/health` has existed since Elasticsearch 1.x, but here is an example of its output from Elasticsearch 5.x:

With verbose output:

```
epoch      timestamp cluster       status node.total node.data shards pri relo init unassign
pending_tasks max_task_wait_time active_shards_percent
1469302011 15:26:51  elasticsearch yellow          1         1     45  45    0    0       44
0                   -                  50.6%
```

Human readable, tabular Cluster Health without headers

Example uses basic HTTP syntax. Any `<#>` in the example should be removed when copying it.

You can use the `_cat` APIs to get a human readable, tabular output for various reasons.

```
GET /_cat/health <1>
```

`_cat/health` has existed since Elasticsearch 1.x, but here is an example of its output from Elasticsearch 5.x:

Without verbose output:

```
1469302245 15:30:45 elasticsearch yellow 1 1 45 45 0 0 44 0 - 50.6%
```

Human readable, tabular Cluster Health with selected headers

Example uses basic HTTP syntax. Any `<#>` in the example should be removed when copying it.

Like most `_cat` APIs in Elasticsearch, the API selectively responds with a default set of fields. However, other fields exist from the API if you want them:

```
GET /_cat/health?help <1>
```

1. `?help` causes the API to return the fields (and short names) as well as a brief description.

`_cat/health` has existed since Elasticsearch 1.x, but here is an example of its output from Elasticsearch 5.x:

Fields available as-of this example's creation date:

```
epoch                | t,time                                  | seconds since 1970-01-01
00:00:00
timestamp            | ts,hms,hhmmss                           | time in HH:MM:SS

cluster              | cl                                      | cluster name

status               | st                                      | health status

node.total           | nt,nodeTotal                            | total number of nodes

node.data            | nd,nodeData                             | number of nodes that can
store data
shards               | t,sh,shards.total,shardsTotal           | total number of shards

pri                  | p,shards.primary,shardsPrimary          | number of primary shards

relo                 | r,shards.relocating,shardsRelocating    | number of relocating nodes

init                 | i,shards.initializing,shardsInitializing | number of initializing
nodes
unassign             | u,shards.unassigned,shardsUnassigned    | number of unassigned shards

pending_tasks        | pt,pendingTasks                         | number of pending tasks

max_task_wait_time   | mtwt,maxTaskWaitTime                    | wait time of longest task
pending
active_shards_percent | asp,activeShardsPercent                | active number of shards in
percent
```

You can then use this to print only those fields:

```
GET /_cat/health?h=timestamp,cl,status&v <1>
```

1. `h=...` defines the list of fields that you want returned.
2. `v` (verbose) defines that you want it to print the headers.

The output from an instance of Elasticsearch 5.x:

```
timestamp cl            status
15:38:00  elasticsearch yellow
```

JSON-based Cluster Health

Example uses basic HTTP syntax. Any `<#>` in the example should be removed when copying it.

The `_cat` APIs are often convenient for humans to get at-a-glance details about the cluster. But you frequently want consistently parseable output to use with software. In general, the JSON APIs are meant for this purpose.

```
GET /_cluster/health
```

`_cluster/health` has existed since Elasticsearch 1.x, but here is an example of its output from Elasticsearch 5.x:

```
{
  "cluster_name": "elasticsearch",
  "status": "yellow",
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 45,
  "active_shards": 45,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 44,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 50.56179775280899
}
```

Read Cluster online: https://riptutorial.com/elasticsearch/topic/2069/cluster

Chapter 5: Curl Commands

Syntax

- curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY_STRING>' -d '<BODY>'
- Where:
- 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.
- PATH: API Endpoint (for example _count will return the number of documents in the cluster). Path may contain multiple components, such as _cluster/stats or _nodes/stats/jvm
- 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.)
- Reference: Talking to Elasticsearch : Elasticsearch Docs

Examples

Curl Command for counting number of documents in the cluster

```
curl -XGET 'http://www.example.com:9200/myIndexName/_count?pretty'
```

Output:

```
{
  "count" : 90,
  "_shards" : {
    "total" : 6,
    "successful" : 6,
    "failed" : 0
  }
}
```

The index has 90 documents within it.

Reference Link: Here

Retrieve a document by Id

```
curl -XGET 'http://www.example.com:9200/myIndexName/myTypeName/1'
```

Output:

```
{
    "_index" : "myIndexName",
    "_type" : "myTypeName",
    "_id" : "1",
    "_version" : 1,
    "found": true,
    "_source" : {
        "user" : "mrunal",
        "postDate" : "2016-07-25T15:48:12",
        "message" : "This is test document!"
    }
}
```

Reference Link: Here

Create an Index

```
curl -XPUT 'www.example.com:9200/myIndexName?pretty'
```

Output:

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

Reference Link: Here

List all indices

```
curl 'www.example.com:9200/_cat/indices?v'
```

output:

```
health status index               pri rep docs.count docs.deleted store.size pri.store.size
green  open   logstash-2016.07.21   5   1       4760            0      4.8mb          2.4mb
green  open   logstash-2016.07.20   5   1       7232            0      7.5mb          3.7mb
green  open   logstash-2016.07.22   5   1      93528            0    103.6mb           52mb
green  open   logstash-2016.07.25   5   1      20683            0     41.5mb         21.1mb
```

Reference Link: Here

Delete an Index

```
curl -XDELETE 'http://www.example.com:9200/myIndexName?pretty'
```

output:

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

Reference Link: Here

List all documents in a index

```
curl -XGET http://www.example.com:9200/myIndexName/_search?pretty=true&q=*:*
```

This uses the `Search` API and will return all the entries under index `myIndexName`.

Reference Link: Here

Read Curl Commands online: https://riptutorial.com/elasticsearch/topic/3703/curl-commands

Chapter 6: Difference Between Indices and Types

Remarks

It's easy to see `type`s like a table in an SQL database, where the `index` is the SQL database. However, that is not a good way to approach `type`s.

All About Types

In fact, types are *literally* just a metadata field added to each document by Elasticsearch: `_type`. The examples above created two types: `my_type` and `my_other_type`. That means that each document associated with the types has an extra field automatically defined like `"_type": "my_type"` ; this is indexed with the document, thus making it a *searchable or filterable field*, but it does not impact the raw document itself, so your application does not need to worry about it.

All types live in the same index, and therefore in the same collective shards of the index. Even at the disk level, they live in the same files. The only separation that creating a second type provides is a logical one. Every type, whether it's unique or not, needs to exist in the mappings and all of those mappings must exist in your cluster state. This eats up memory and, if each type is being updated dynamically, it eats up performance as the mappings change.

As such, it is a best practice to define only a single type unless you actually need other types. It is common to see scenarios where multiple types are desirable. For example, imagine you had a car index. It may be useful to you to break it down with multiple types:

- bmw
- chevy
- honda
- mazda
- mercedes
- nissan
- rangerover
- toyota
- ...

This way you can search for all cars or limit it by manufacturer on demand. The difference between those two searches are as simple as:

```
GET /cars/_search
```

and

```
GET /cars/bmw/_search
```

What is not obvious to new users of Elasticsearch is that the second form is a specialization of the first form. It literally gets rewritten to:

```
GET /cars/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "term" : {
            "_type": "bmw"
          }
        }
      ]
    }
  }
}
```

It simply filters out any document that was not indexed with a `_type` field whose value was `bmw`. Since every document is indexed with its type as the `_type` field, this serves as a pretty simple filter. If an actual search had been provided in either example, then the filter would be added to the full search as appropriate.

As such, if the types are identical, it's much better to supply a single type (e.g., `manufacturer` in this example) and effectively ignore it. Then, within each document, explicitly supply a field called `make` *or whatever name you prefer* and manually filter on it whenever you want to limit to it. This will reduce the size of your mappings to `1/n` where `n` is the number of separate types. It does add another field to each document, at the benefit of an otherwise simplified mapping.

In Elasticsearch 1.x and 2.x, such a field should be defined as

```
PUT /cars
{
  "manufacturer": { <1>
    "properties": {
      "make": { <2>
        "type": "string",
        "index": "not_analyzed"
      }
    }
  }
}
```

1. The name is arbitrary.
2. The name is arbitrary *and* it could match the type name if you wanted it too.

In Elasticsearch 5.x, the above will still work (it's deprecated), but the better way is to use:

```
PUT /cars
{
  "manufacturer": { <1>
    "properties": {
      "make": { <2>
        "type": "keyword"
```

```
        }
      }
    }
}
```

1. The name is arbitrary.
2. The name is arbitrary *and* it could match the type name if you wanted it too.

Types should be used sparingly within your indices because it bloats the index mappings, usually without much benefit. You must have at least one, but there is nothing that says you must have more than one.

Common Questions

- What if I have two (or more) types that are mostly identical, but which have a few unique fields per type?

At the index level, there is no difference between one type being used with a few fields that are sparsely used *and* between multiple types that share a bunch of non-sparse fields with a few not shared (meaning the other type never even uses the field(s)).

Said differently: a sparsely used field is sparse across the index *regardless of types*. The sparsity does not benefit -- or really hurt -- the index just because it is defined in a separate type.

You should just combine these types and add a separate type field.

- Why do separate types need to define fields in the exact same way?

Because each field is really only defined once at the Lucene level, regardless of how many types there are. The fact that types exist at all is a feature of Elasticsearch and it is *only* a logical separation.

- Can I define separate types with the same field defined differently?

No. If you manage to find a way to do so in ES 2.x or later, then you should open up a bug report. As noted in the previous question, Lucene sees them all as a single field, so there is no way to make this work appropriately.

ES 1.x left this as an implicit requirement, which allowed users to create conditions where one shard's mappings in an index actually differed from another shard in the same index. This was effectively a race condition and it *could* lead to unexpected issues.

Exceptions to the Rule

- Parent/child documents **require** separate types to be used within the same index.
  - The parent lives in one type.
  - The child lives in a separate type (but each child lives in the same *shard* as its parent).
- Extremely niche use cases where creating tons of indices is undesirable and the impact of

sparse fields is preferable to the alternative.
    - For example, the Elasticsearch monitoring plugin, Marvel (1.x and 2.x) or X-Pack Monitoring (5.x+), monitors Elasticsearch itself for changes in the cluster, nodes, indices, specific indices (the index level), and even shards. It could create 5+ indices each day to isolate those documents that have unique mappings *or* it could go against best practices to reduce cluster load by sharing an index (note: the number of defined mappings is effectively the same, but the number of created indices is reduced from `n` to 1).
    - This is an advanced scenario, but you must consider the shared field definitions across types!

Examples

Explicitly creating an Index with a Type

Example uses basic HTTP, which translate easily to cURL and other HTTP applications. They also match the Sense syntax, which will be renamed to Console in Kibana 5.0.

Note: The example inserts `<#>` to help draw attention to parts. Those should be removed if you copy it!

```
PUT /my_index <1>
{
  "mappings": {
    "my_type": { <2>
      "properties": {
        "field1": {
          "type": "long"
        },
        "field2": {
          "type": "integer"
        },
        "object1": {
          "type": "object",
          "properties": {
            "field1" : {
              "type": "float"
            }
          }
        }
      }
    }
  },
  "my_other_type": {
    "properties": {
      "field1": {
        "type": "long" <3>
      },
      "field3": { <4>
        "type": "double"
      }
    }
  }
}
```

1. This is creating the `index` using the create index endpoint.
2. This is creating the `type`.
3. Shared fields in `type`s within the same `index` **must** share the same definition! ES 1.x did not strictly enforce this behavior, but it was an implicit requirement. ES 2.x and above strictly enforce this behavior.
4. Unique fields across `type`s are okay.

Indexes (or indices) *contain* types. Types are a convenient mechanism for separating documents, but they require you to define -- either dynamically/automatically or explicitly -- a mapping for each type that you use. If you define 15 types in an index, then you have 15 unique mappings.

See the remarks for more details about this concept and why you may or may not want to use types.

Dynamically creating an Index with a Type

Example uses basic HTTP, which translate easily to cURL and other HTTP applications. They also match the Sense syntax, which will be renamed to Console in Kibana 5.0.

Note: The example inserts `<#>` to help draw attention to parts. Those should be removed if you copy it!

```
DELETE /my_index <1>

PUT /my_index/my_type/abc123 <2>
{
  "field1" : 1234, <3>
  "field2" : 456,
  "object1" : {
    "field1" : 7.8 <4>
  }
}
```

1. In case it already exists (because of an earlier example), delete the index.
2. Index a document into the index, `my_index`, with the type, `my_type`, and the ID `abc123` (could be numeric, but it is always a string).
    - By default, dynamic index creation is enabled by simply indexing a document. This is great for development environments, but it is not necessarily good for production environments.
3. This field is an integer number, so the first time it is seen it must be mapped. Elasticsearch always assumes the *widest* type for any incoming type, so this would be mapped as a `long` rather than an `integer` or a `short` (both of which could contain `1234` and `456`).
4. The same is true for this field as well. It will be mapped as a `double` instead of a `float` as you might want.

This dynamically created index and type roughly match the mapping defined in the first example. However, it's critical to understand how `<3>` and `<4>` impact the automatically defined mappings.

You could follow this by adding yet another type dynamically to the same index:

```
PUT /my_index/my_other_type/abc123 <1>
{
  "field1": 91, <2>
  "field3": 4.567
}
```

1. The type is the only difference from the above document. The ID is the same and that's okay! It has no relationship to the other `abc123` other than that it *happens* to be in the same index.
2. `field1` already exists in the index, so it *must* be the same type of field as defined in the other types. Submitting a value that was a string or not an integer would fail (e.g., `"field1": "this is some text"` or `"field1": 123.0`).

This would dynamically create the mappings for `my_other_type` within the same index, `my_index`.

Note: It is *always* faster to define mappings upfront rather than having Elasticsearch dynamically perform it at index time.

The end result of indexing both documents would be similar to the first example, but the field types would be different and therefore slightly wasteful:

```
GET /my_index/_mappings <1>
{
  "mappings": {
    "my_type": { <2>
      "properties": {
        "field1": {
          "type": "long"
        },
        "field2": {
          "type": "long" <3>
        },
        "object1": {
          "type": "object",
          "properties": {
            "field1" : {
              "type": "double" <4>
            }
          }
        }
      }
    }
  },
  "my_other_type": { <5>
    "properties": {
      "field1": {
        "type": "long"
      },
      "field3": {
        "type": "double"
      }
    }
  }
}
```

1. This uses the `_mappings` endpoint to get the mappings from the index that we created.

2. We dynamically created `my_type` in the first step of this example.
3. `field2` is now a `long` instead of an `integer` because we did not define it upfront. This *may* prove to be wasteful in disk storage.
4. `object1.field1` is now a `double` for the same reason as #3 with the same ramifications as #3.
    - Technically, a `long` can be compressed in a lot of cases. However, a `double` cannot be compressed due to it being a floating point number.
5. We also dynamically created `my_other_type` in the second step of this example. Its mapping happens to be the same because we were already using `long` and `double`.
    - Remember that `field1` *must* match the definition from `my_type` (and it does).
    - `field3` is unique to this type, so it has no such restriction.

Read Difference Between Indices and Types online:
https://riptutorial.com/elasticsearch/topic/3412/difference-between-indices-and-types

Chapter 7: Difference Between Relational Databases and Elasticsearch

Introduction

This is for the readers who come from relational background and want to learn elasticsearch. This topic shows the use cases for which Relational databases are not a suitable option.

Examples

Terminology Difference

Relational Database	Elasticsearch
Database	Index
Table	Type
Row/Record	Document
Column Name	field

Above table roughly draws an analogy between basic elements of relational database and elasticsearch.

Setup

Considering Following structure in a relational database:

```
create databse test;

use test;

create table product;

create table product (name varchar, id int PRIMARY KEY);

insert into product  (id,name) VALUES (1,'Shirt');

insert into product  (id,name) VALUES (2,'Red Shirt');

select * from product;

name       | id
-----------+----
Shirt      |  1
Red Shirt  |  2
```

Elasticsearch Equivalent:

```
POST test/product
{
    "id" : 1,
    "name" : "Shirt"
}

POST test/product
{
    "id" : 2,
    "name" : "Red Shirt"
}

GET test/product/_search


"hits": [
     {                                          ==============
        "_index": "test",                       ===> index    |
        "_type": "product",                     ===>type      |
        "_id": "AVzglFomaus3G2tXc6sB",                        |
        "_score": 1,                                          |
        "_source": {                                          |===> document
           "id": 2,                           ===>field      |
           "name": "Red Shirt"                ===>field      |
        }                                                     |
     },                                         ==============
     {
        "_index": "test",
        "_type": "product",
        "_id": "AVzglD12aus3G2tXc6sA",
        "_score": 1,
        "_source": {
           "id": 1,
           "name": "Shirt"
        }
     }
  ]
```

Usecases where Relational Databases are not suitable

- Essence of searching lies in its order. Everyone wants search results to be shown in such a way that best suited results are shown on top. Relational database do not have such capability. Elasticsearch on the other hand shows results on the basis of relevancy by default.

  Setup

  Same as used in previous example.

  Problem Statement

  Suppose user wants to search for `shirts` but he is interested in `red` colored shirts. In that case, results containing `red` and `shirts` keyword should come on top. Then results for other shirts should be shown after them.

Solution Using Relational Database Query

```
select * from product where name like '%Red%' or name like '%Shirt%';
```

Output

```
name       | id
-----------+----
Shirt      |  1
Red Shirt  |  2
```

Elasticsearch Solution

```
POST test/product/_search
{
     "query": {
          "match": {
            "name": "Red Shirt"
          }
     }
}
```

Output

```
"hits": [
  {
     "_index": "test",
     "_type": "product",
     "_id": "AVzglFomaus3G2tXc6sB",
     "_score": 1.2422675,              ===> Notice this
     "_source": {
        "id": 2,
        "name": "Red Shirt"
     }
  },
  {
     "_index": "test",
     "_type": "product",
     "_id": "AVzglD12aus3G2tXc6sA",
     "_score": 0.25427115,             ===> Notice this
     "_source": {
        "id": 1,
        "name": "Shirt"
     }
  }
 ]
```

Conclusion

As we can see above Relational Database has returned results in some random order, while Elasticsearch returns results in decreasing order of `_score` which is calculated on the basis of relevancy.

- We tend to make mistakes while entering search string. There are cases when user enters

an incorrect search parameter. Relational Databases won't handle such cases. Elasticsearch to the rescue.

Setup

Same as used in previous example.

Problem Statement

Suppose user wants to search for `shirts` but he enters an incorrect word `shrt` by mistake. User still expects to see the results of shirt.

Solution Using Relational Database Query

```
select * from product where name like '%shrt%';
```

Output

```
No results found
```

Elasticsearch Solution

```
POST /test/product/_search

 {
    "query": {
      "match": {
        "name": {
          "query": "shrt",
          "fuzziness": 2,
          "prefix_length": 0
         }
      }
    }
 }
```

Output

```
 "hits": [
  {
     "_index": "test",
     "_type": "product",
     "_id": "AVzglD12aus3G2tXc6sA",
     "_score": 1,
     "_source": {
        "id": 1,
        "name": "Shirt"
     }
  },
  {
     "_index": "test",
     "_type": "product",
     "_id": "AVzglFomaus3G2tXc6sB",
     "_score": 0.8784157,
     "_source": {
```

```
        "id": 2,
        "name": "Red Shirt"
      }
  }
]
```

Conclusion

As we can see above relational database has returned no results for an incorrect word searched, while Elasticsearch using its special `fuzzy` query returns results.

Read Difference Between Relational Databases and Elasticsearch online: https://riptutorial.com/elasticsearch/topic/10632/difference-between-relational-databases-and-elasticsearch

Chapter 8: Elasticsearch Configuration

Remarks

Elasticsearch comes with a set of defaults that provide a good out of the box experience for development. The implicit statement there is that it is not necessarily great for production, which must be tailored for your own needs and therefore cannot be predicted.

The default settings make it easy to download and run multiple nodes *on the same machine* without any configuration changes.

Where are the settings?

Inside each installation of Elasticsearch is a `config/elasticsearch.yml`. That is where the following settings live:

- `cluster.name`
  - The name of the cluster that the node is joining. All nodes in the same cluster **must** share the same name.
  - Currently defaults to `elasticsearch`.
- `node.*`
  - `node.name`
    - If not supplied, a random name will be generated *each time the node starts*. This can be fun, but it is not good for production environments.
    - Names do *not* have to be unique, but they **should** be unique.
  - `node.master`
    - A boolean setting. When `true`, it means that the node is an eligible master node and it can be *the* elected master node.
    - Defaults to `true`, meaning every node is an eligible master node.
  - `node.data`
    - A boolean setting. When `true`, it means that the node stores data and handles search activity.
    - Defaults to `true`.
- `path.*`
  - `path.data`
    - The location that files are written for the node. *All nodes use this directory* to store metadata, but data nodes will also use it to store/index documents.
    - Defaults to `./data`.
      - This means that `data` will be created for you as a peer directory to `config` *inside* of the Elasticsearch directory.
  - `path.logs`
    - The location that log files are written.
    - Defaults to `./logs`.
- `network.*`
  - `network.host`
    - Defaults to `_local_`, which is effectively `localhost`.

        - This means that, by default, nodes cannot be communicated with from outside of the current machine!
    - `network.bind_host`
        - Potentially an array, this tells Elasticsearch what addresses of the current machine to bind sockets too.
            - It is this list that enables machines from outside of the machine (e.g., other nodes in the cluster) to talk to this node.
        - Defaults to `network.host`.
    - `network.publish_host`
        - A singular host that is used to advertise to other nodes how to best communicate with this node.
            - When supplying an array to `network.bind_host`, this should be the *one* host that is intended to be used for inter-node communication.
        - Defaults to network.host`.
- `discovery.zen.*`
    - `discovery.zen.minimum_master_nodes`
        - Defines quorum for master election. This **must** be set using this equation: `(M / 2) + 1` where `M` is the number of *eligible* master nodes (nodes using `node.master: true` implicitly or explicitly).
        - Defaults to `1`, which only is valid for a single node cluster!
    - `discovery.zen.ping.unicast.hosts`
        - The mechanism for joining this node to the rest of a cluster.
        - This *should* list eligible master nodes so that a node can find the rest of the cluster.
        - The value that should be used here is the `network.publish_host` of those other nodes.
        - Defaults to `localhost`, which means it only looks on the local machine for a cluster to join.

What type of settings exist?

Elasticsearch provides three different types of settings:

- Cluster-wide settings
    - These are settings that apply to everything in the cluster, such as all nodes or all indices.
- Node settings
    - These are settings that apply to just the current node.
- Index settings
    - These are settings that apply to just the index.

Depending on the setting, it can be:

- Changed dynamically at runtime
- Changed following a restart (close / open) of the index

    - Some index-level settings do not require the index to be closed and reopened, but might require the index to be forceably re-merged for the setting to apply.
        - The compression level of an index is an example of this type of setting. It can be changed dynamically, but only new *segments* take advantage of the change. So if an index will not change, then it never takes advantage of the change unless you force the index to recreate its segments.
- Changed following a restart of the node
- Changed following a restart of the cluster
- Never changed

Always check the documentation for your version of Elasticsearch for what you can or cannot do with a setting.

How can I apply settings?

You can set settings a few ways, some of which are not suggested:

- Command Line Arguments

In Elasticsearch 1.x and 2.x, you can submit most settings as Java System Properties prefixed with `es.`:

```
$ bin/elasticsearch -Des.cluster.name=my_cluster -Des.node.name=`hostname`
```

In Elasticsearch 5.x, this changes to avoid using Java System Properties, instead using a custom argument type with `-E` taking the place of `-Des.`:

```
$ bin/elasticsearch -Ecluster.name=my_cluster -Enode.name=`hostname`
```

This approach to applying settings works great when using tools like Puppet, Chef, or Ansible to start and stop the cluster. However it works very poorly when doing it manually.

- YAML settings
    - Shown in examples
- Dynamic settings
    - Shown in examples

The order that settings are applied are in the order of most dynamic:

1. Transient settings
2. Persistent settings
3. Command line settings
4. YAML (static) settings

If the setting is set twice, once at any of those levels, then the highest level takes effect.

Examples

Static Elasticsearch Settings

Elasticsearch uses a YAML (Yet Another Markup Language) configuration file that can be found inside the default Elasticsearch directory (RPM and DEB installs change this location amongst other things).

You can set basic settings in `config/elasticsearch.yml`:

```
# Change the cluster name. All nodes in the same cluster must use the same name!
cluster.name: my_cluster_name

# Set the node's name using the hostname, which is an environment variable!
# This is a convenient way to uniquely set it per machine without having to make
#  a unique configuration file per node.
node.name: ${HOSTNAME}

# ALL nodes should set this setting, regardless of node type
path.data: /path/to/store/data

# This is a both a master and data node (defaults)
node.master: true
node.data: true

# This tells Elasticsearch to bind all sockets to only be available
#  at localhost (default)
network.host: _local_
```

Persistent Dynamic Cluster Settings

If you need to apply a setting dynamically after the cluster has already started, and it can actually be set dynamically, then you can set it using `_cluster/settings` API.

Persistent settings are one of the two type of cluster-wide settings that can be applied. A persistent setting **will** survive a full cluster restart.

Note: Not all settings can be applied dynamically. For example, the cluster's name cannot be renamed dynamically. Most node-level settings cannot be set dynamically either (because they cannot be targeted individually).

This is **not** the API to use to set index-level settings. You can tell that setting is an index level setting because it should start with `index.`. Settings whose name are in the form of `indices.` *are* cluster-wide settings because they apply to all indices.

```
POST /_cluster/settings
{
  "persistent": {
    "cluster.routing.allocation.enable": "none"
  }
}
```

Warning: In Elasticsearch 1.x and 2.x, you cannot *unset* a persistent setting.

Fortunately, this has been improved in Elasticsearch 5.x and you can now remove a setting by

setting it to `null`:

```
POST /_cluster/settings
{
  "persistent": {
    "cluster.routing.allocation.enable": null
  }
}
```

An unset setting will return to its default, or any value defined at a lower priority level (e.g., command line settings).

Transient Dynamic Cluster Settings

If you need to apply a setting dynamically after the cluster has already started, and it can actually be set dynamically, then you can set it using `_cluster/settings` API.

Transient settings are one of the two type of cluster-wide settings that can be applied. A transient setting will **not** survive a full cluster restart.

Note: Not all settings can be applied dynamically. For example, the cluster's name cannot be renamed dynamically. Most node-level settings cannot be set dynamically either (because they cannot be targeted individually).

This is **not** the API to use to set index-level settings. You can tell that setting is an index level setting because it should start with `index.`. Settings whose name are in the form of `indices.` *are* cluster-wide settings because they apply to all indices.

```
POST /_cluster/settings
{
  "transient": {
    "cluster.routing.allocation.enable": "none"
  }
}
```

Warning: In Elasticsearch 1.x and 2.x, you cannot unset a transient settings without a full cluster restart.

Fortunately, this has been improved in Elasticsearch 5.x and you can now remove a setting by setting it to null:

```
POST /_cluster/settings
{
  "transient": {
    "cluster.routing.allocation.enable": null
  }
}
```

An unset setting will return to its default, or any value defined at a lower priority level (e.g., `persistent` settings).

Index Settings

Index settings are those settings that apply to a single index. Such settings will start with `index.`. The exception to that rule is `number_of_shards` and `number_of_replicas`, which also exist in the form of `index.number_of_shards` and `index.number_of_replicas`.

As the name suggests, index-level settings apply to a single index. Some settings must be applied at creation time because they cannot be changed dynamically, such as the `index.number_of_shards` setting, which controls the number of primary shards for the index.

```
PUT /my_index
{
  "settings": {
    "index.number_of_shards": 1,
    "index.number_of_replicas": 1
  }
}
```

or, in a more concise format, you can combine key prefixes at each `.`:

```
PUT /my_index
{
  "settings": {
    "index": {
      "number_of_shards": 1,
      "number_of_replicas": 1
    }
  }
}
```

The above examples will create an index with the supplied settings. You can dynamically change settings per-index by using the index `_settings` endpoint. For example, here we dynamically change the slowlog settings for *only* the warn level:

```
PUT /my_index/_settings
{
  "index": {
    "indexing.slowlog.threshold.index.warn": "1s",
    "search.slowlog.threshold": {
      "fetch.warn": "500ms",
      "query.warn": "2s"
    }
  }
}
```

Warning: Elasticsearch 1.x and 2.x did not very strictly validate index-level setting names. If you had a typo, or simply made up a setting, then it would blindly accept it, but otherwise ignore it. Elasticsearch 5.x strictly validates setting names and it will reject any attempt to apply index settings with an unknown setting(s) (due to typo or missing plugin). Both statements apply to dynamically changing index settings and at creation time.

Dynamic Index Settings for Multiple Indices at the same time

You can apply the same change shown in the `Index Settings` example to *all* existing indices with one request, or even a subset of them:

```
PUT /*/_settings
{
  "index": {
    "indexing.slowlog.threshold.index.warn": "1s",
    "search.slowlog.threshold": {
      "fetch.warn": "500ms",
      "query.warn": "2s"
    }
  }
}
```

or

```
PUT /_all/_settings
{
  "index": {
    "indexing.slowlog.threshold.index.warn": "1s",
    "search.slowlog.threshold": {
      "fetch.warn": "500ms",
      "query.warn": "2s"
    }
  }
}
```

or

```
PUT /_settings
{
  "index": {
    "indexing.slowlog.threshold.index.warn": "1s",
    "search.slowlog.threshold": {
      "fetch.warn": "500ms",
      "query.warn": "2s"
    }
  }
}
```

If you prefer to more selectively do it as well, then you can select multiple without supply all:

```
PUT /logstash-*,my_other_index,some-other-*/_settings
{
  "index": {
    "indexing.slowlog.threshold.index.warn": "1s",
    "search.slowlog.threshold": {
      "fetch.warn": "500ms",
      "query.warn": "2s"
    }
  }
}
```

Read Elasticsearch Configuration online:
https://riptutorial.com/elasticsearch/topic/3411/elasticsearch-configuration

Chapter 9: Learning Elasticsearch with kibana

Introduction

Kibana is front end data visualization tool for elasticsearch. for installing kibana refer to the kibana documentation. For running kibana on localhost go to https://localhost:5601 and go to kibana console.

Examples

Explore your Cluster using Kibana

The command syntax will be of the following type:

```
<REST Verb> /<Index>/<Type>/<ID>
```

Execute the following command to explore elasticsearch cluster through Kibana Console.

- For checking the cluster health

```
GET /_cat/health?v
```

- For listing all the indices

```
GET /_cat/indices?v
```

- For creating an index with name car

```
 PUT /car?pretty
```

- For indexing the document with name car of external type using id 1

```
PUT /car/external/1?pretty
{
  "name": "Tata Nexon"
}
```

the response of above query will be :

```
{
  "_index": "car",
  "_type": "external",
  "_id": "1",
  "_version": 1,
```

```
  "result": "created",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "created": true
}
```

- retrieving the above document can be done using:

```
GET /car/external/1?pretty
```

- For deleting an index

```
DELETE /car?pretty
```

Modify your elasticsearch data

Elasticsearch provides data manipulation & data searching capabilities in almost real time. under this example, we have update, delete & batch processing operations.

- Updating the same document. Suppose we have already indexed a document on /car/external/1 . Then running the command for indexing the data replaces the previous document.

```
PUT /car/external/1?pretty
{
  "name": "Tata Nexa"
}
```

previous car document at id 1 with name "Tata Nexon" will be updated with new name "Tata Nexa"

- indexing the data with explicit Id

```
POST /car/external?pretty
{
  "name": "Jane Doe"
}
```

  for indexing the document without an Id we use **POST** verb instead of **PUT** verb. if we don't provide an Id, elasticsearch will generate a random ID and then use it to index the document.

- Updating the previous document at an Id partially.

```
POST /car/external/1/_update?pretty
{
  "doc": { "name": "Tata Nex" }
}
```

- updating the document with additional information

```
POST /car/external/1/_update?pretty
{
  "doc": { "name": "Tata Nexon", "price": 1000000 }
}
```

- updating the document using simple scripts.

```
POST /car/external/1/_update?pretty
{
  "script" : "ctx._source.price += 50000"
}
```

ctx._source refers to the current source document that is about to be updated. Above script provides only one script to be updated at the same time.

- Deleting the document

```
DELETE /car/external/1?pretty
```

Note: deleting a whole index is more efficient than deleting all documents by using Delete by Query API

Batch Processing

Apart from indexing updating & deleting the document, elasticsearch also provides provides the ability to perform any of the above operations in batches using the _**bulk** API.

- for updating multiple documents using _**bulk** API

```
POST /car/external/_bulk?pretty
{"index":{"_id":"1"}}
{"name": "Tata Nexon" }
{"index":{"_id":"2"}}
{"name": "Tata Nano" }
```

- for updating & deleting the documents using _**bulk** API

```
POST /car/external/_bulk?pretty
{"update":{"_id":"1"}}
{"doc": { "name": "Tata Nano" } }
{"delete":{"_id":"2"}}
```

If an operation fails, bulk API doesn't stop. It executes all the operations & finally returns report for all the operations.

Read Learning Elasticsearch with kibana online:
https://riptutorial.com/elasticsearch/topic/10058/learning-elasticsearch-with-kibana

Chapter 10: Python Interface

Parameters

Parameter	Details
hosts	Array of hosts in the form of object containing keys `host` and `port`. Default `host` is 'localhost' and `port` is 9200. A sample entry looks like `[{"host": "ip of es server", "port": 9200}]`
sniff_on_start	Boolean if you want the client to sniff nodes on startup, sniffing means getting list of nodes in elasticsearch cluster
sniff_on_connection_fail	Boolean for triggering sniffing if connection fails when client is active
sniffer_timeout	time difference in seconds between each sniff
sniff_timeout	time for a single request of sniffing in seconds
retry_on_timeout	Booelan for if client should timeout trigger contacting a different elasticsearch node or just throw error
http_auth	Basic http authentication can be provided here in the form of `username:password`

Examples

Indexing a Document (ie. Adding an sample)

Install the necessary Python Library via:

```
$ pip install elasticsearch
```

Connect to Elasticsearch, Create a Document (e.g. data entry) and "Index" the document using Elasticsearch.

```
from datetime import datetime
from elasticsearch import Elasticsearch

# Connect to Elasticsearch using default options (localhost:9200)
es = Elasticsearch()

# Define a simple Dictionary object that we'll index to make a document in ES
doc = {
    'author': 'kimchy',
    'text': 'Elasticsearch: cool. bonsai cool.',
```

```
    'timestamp': datetime.now(),
}

# Write a document
res = es.index(index="test-index", doc_type='tweet', id=1, body=doc)
print(res['created'])

# Fetch the document
res = es.get(index="test-index", doc_type='tweet', id=1)
print(res['_source'])

# Refresh the specified index (or indices) to guarantee that the document
#  is searchable (avoid race conditions with near realtime search)
es.indices.refresh(index="test-index")

# Search for the document
res = es.search(index="test-index", body={"query": {"match_all": {}}})
print("Got %d Hits:" % res['hits']['total'])

# Show each "hit" or search response (max of 10 by default)
for hit in res['hits']['hits']:
    print("%(timestamp)s %(author)s: %(text)s" % hit["_source"])
```

Connection to a cluster

```
es = Elasticsearch(hosts=hosts, sniff_on_start=True, sniff_on_connection_fail=True,
sniffer_timeout=60, sniff_timeout=10, retry_on_timeout=True)
```

Creating an empty index and setting the mapping

In this example, we create an empty index (we index no documents in it) by defining its mapping.

First, we create an `ElasticSearch` instance and we then define the mapping of our choice. Next, we check if the index exists and if not, we create it by specifying the `index` and `body` parameters that contain the index name and the body of the mapping, respectively.

```
from elasticsearch import Elasticsearch

# create an ElasticSearch instance
es = Elasticsearch()
# name the index
index_name = "my_index"
# define the mapping
mapping = {
    "mappings": {
        "my_type": {
                "properties": {
                    "foo": {'type': 'text'},
                    "bar": {'type': 'keyword'}
                }
            }
        }
    }

# create an empty index with the defined mapping - no documents added
if not es.indices.exists(index_name):
```

```
    res = es.indices.create(
        index=index_name,
        body=mapping
    )
    # check the response of the request
    print(res)
    # check the result of the mapping on the index
    print(es.indices.get_mapping(index_name))
```

Partial Update and Update by query

Partial Update: Used when a partial document update is needed to be done, i.e. in the following example the field `name` of the document with id `doc_id` is going to be updated to 'John'. Note that if the field is missing, it will just be added to the document.

```
doc = {
    "doc": {
        "name": "John"
    }
}
es.update(index='index_name',
          doc_type='doc_name',
          id='doc_id',
          body=doc)
```

Update by query: Used when is needed to update documents that satisfy a condition, i.e. in the following example we update the age of the documents whose `name` field matches 'John'.

```
q = {
  "script": {
    "inline": "ctx._source.age=23",
    "lang": "painless"
  },
  "query": {
    "match": {
        "name": "John"
    }
  }
}

es.update_by_query(body=q,
                   doc_type='doc_name',
                   index='index_name')
```

Read Python Interface online: https://riptutorial.com/elasticsearch/topic/2068/python-interface

Chapter 11: Search API

Introduction

The search API allows you to execute a search query and get back search hits that match the query. The query can either be provided using a simple query string as a parameter, or using a request body.

Examples

Routing

When executing a search, it will be broadcast to all the index/indices shards (round robin between replicas). Which shards will be searched on can be controlled by providing the routing parameter. For example, when indexing tweets, the routing value can be the user name:

```
curl -XPOST 'localhost:9200/twitter/tweet?routing=kimchy&pretty' -d'
{
    "user" : "kimchy",
    "postDate" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}'
```

Search using request body

Searches can also be done on elasticsearch using a search DSL.The query element within the search request body allows to define a query using the Query DSL.

```
GET /my_index/type/_search
{
    "query" : {
        "term" : { "field_to_search" : "search_item" }
    }
}
```

Multi search

The multi_search option allows us to search for a query in multiple fields at once.

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query":    "text to search",
      "fields": [ "field_1", "field_2" ]
    }
  }
}
```

We can also boost the score of certain fields using the boost operator(^), and use wild cards in the field name (*)

```
GET /_search
    {
      "query": {
        "multi_match" : {
          "query":    "text to search",
          "fields": [ "field_1^2", "field_2*" ]
        }
      }
    }
```

URI search, and Highlighting

A search request can be executed purely using a URI by providing request parameters. Not all search options are exposed when executing a search using this mode, but it can be handy for quick "curl tests".

```
GET Index/type/_search?q=field:value
```

Another useful feature provided is highlighting the match hits in the documents.

```
GET /_search
{
    "query" : {
        "match": { "field": "value" }
    },
    "highlight" : {
        "fields" : {
            "content" : {}
        }
    }
}
```

In the above case, the particular field will be highlighted for each search hit

Read Search API online: https://riptutorial.com/elasticsearch/topic/8625/search-api

Credits

S. No	Chapters	Contributors
1	Getting started with Elasticsearch	Ahsanul Haque, Berto, Community, DJanssens, Dulguun, igo, KartikKannapur, manishrw, mightyteja, noscreenname, Onur, rafa.ferreira, RustyBuckets, sarvajeetsuman, SeinopSys, Shivkumar Mallesappa, Stephan-v, Suhas K, Sumit Kumar, Trilarion
2	Aggregations	Sid1199
3	Analyzers	Bhushan Gadekar, Sid1199, Thomas
4	Cluster	Gerardo Rochín, pickypg
5	Curl Commands	Fawix, Mrunal Pagnis, Mrunal Pagnis
6	Difference Between Indices and Types	pickypg
7	Difference Between Relational Databases and Elasticsearch	Richa
8	Elasticsearch Configuration	pickypg
9	Learning Elasticsearch with kibana	sarvajeetsuman
10	Python Interface	aidan.plenert.macdonald, christinabo, KartikKannapur, pickypg, Sumit Kumar
11	Search API	aerokite, Sid1199