Elasticsearch - Quick Guide

Elasticsearch - Basic Concepts

Elasticsearch is an Apache Lucene-based search server. It was developed by Shay Ba and published in 2010. It is now maintained by Elasticsearch BV. Its latest version is 2.1.0.

Elasticsearch is a real-time distributed and open source full-text search and analytics engine. It is accessible from RESTful web service interface and uses schema less JSON (JavaScript Object Notation) documents to store data. It is built on Java programming language, which enables Elasticsearch to run on different platforms. It enables users to explore very large amount of data at very high speed.

Elasticsearch – General Features

The general features of Elasticsearch are as follows −

- Elasticsearch is scalable up to petabytes of structured and unstructured data.
- Elasticsearch can be used as a replacement of document stores like MongoDB and RavenDB.
- Elasticsearch uses denormalization to improve the search performance.
- Elasticsearch is one of the popular enterprise search engines, which is currently being used by many big organizations like Wikipedia, The Guardian, StackOverflow, GitHub etc.
- Elasticsearch is open source and available under the Apache license version 2.0.

Elasticsearch – Key Concepts

The key concepts of Elasticsearch are as follows −

Node – It refers to a single running instance of Elasticsearch. Single physical and virtual server accommodates multiple nodes depending upon the capabilities of their physical resources like RAM, storage and processing power.

Cluster – It is a collection of one or more nodes. Cluster provides collective indexing and search capabilities across all the nodes for entire data.

Index – It is a collection of different type of documents and document properties. Index also uses the concept of shards to improve the performance. For example, a set of document contains data of a social networking application.

Type/Mapping – It is a collection of documents sharing a set of common fields present in the same index. For example, an Index contains data of a social networking application, and then there can be a specific type for user profile data, another type for messaging data and another for comments data.

Document – It is a collection of fields in a specific manner defined in JSON format. Every document belongs to a type and resides inside an index. Every document is associated with a unique identifier, called the UID.

Shard – Indexes are horizontally subdivided into shards. This means each shard contains all the properties of document, but contains less number of JSON objects than index. The horizontal separation makes shard an independent node, which can be store in any node. Primary shard is the original horizontal part of an index and then these primary shards are replicated into replica shards.

Replicas – Elasticsearch allows a user to create replicas of their indexes and shards. Replication not only helps in increasing the availability of data in case of failure, but also improves the performance of searching by carrying out a parallel search operation in these replicas.

Elasticsearch – Advantages

Elasticsearch is developed on Java, which makes it compatible on almost every platform.

Elasticsearch is real time, in other words after one second the added document is searchable in this engine.

Elasticsearch is distributed, which makes it easy to scale and integrate in any big organization.

Creating full backups are easy by using the concept of gateway, which is present in Elasticsearch.

Handling multi-tenancy is very easy in Elasticsearch when compared to Apache Solr.

Elasticsearch uses JSON objects as responses, which makes it possible to invoke the Elasticsearch server with a large number of different programming languages.

Elasticsearch supports almost every document type except those that do not support text rendering.

Elasticsearch – Disadvantages

Elasticsearch does not have multi-language support in terms of handling request and response data (only possible in JSON) unlike in Apache Solr, where it is possible in CSV, XML and JSON formats.

Elasticsearch also have a problem of Split brain situations, but in rare cases.

Comparison between Elasticsearch and RDBMS

In Elasticsearch, index is a collection of type just as database is a collection of tables in RDBMS (Relation Database Management System). Every table is a collection of rows just as every mapping is a collection of JSON objects Elasticsearch.

Elasticsearch	RDBMS
Index	Database
Shard	Shard
Mapping	Table
Field	Field
JSON Object	Tuple

Elasticsearch - Installation

The steps for installation of Elasticsearch are as follows −

Step 1 − Check the minimum version of your java in installed your computer, it should be java 7 or more updated version. You can check by doing the following −

In Windows Operating System (OS) (using command prompt) −

```
> java -version
```

In UNIX OS (Using Terminal) −

```
$ echo $JAVA_HOME
```

Step 2 − Download Elasticsearch from www.elastic.co

For windows OS download ZIP file.

For UNIX OS download TAR file.

For Debian OS download DEB file.

For Red Hat and other Linux distributions download RPN file.

APT and Yum utilities can also be used to install Elasticsearch in many Linux distributions.

Step 3 – Installation process for Elasticsearch is very easy and described below for different OS –

Windows OS – Unzip the zip package and the Elasticsearch is installed.

UNIX OS – Extract tar file in any location and the Elasticsearch is installed.

```
$tar –xvf elasticsearch-2.1.0.tar.gz
```

Using APT utility for Linux OS –

Download and install the Public Signing Key –

```
$ wget -qO - https://packages.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
```

Save the repository definition –

```
$ echo "deb http://packages.elastic.co/elasticsearch/2.x/debian stable main" | sudo tee -a /etc
   /apt/sources.list.d/elasticsearch-2.x.list
```

Run update –

```
$ sudo apt-get update
```

Now you can install by using the following command –

```
$ sudo apt-get install elasticsearch
```

Using YUM utility for Debian Linux OS –

Download and install the Public Signing Key –

```
$ rpm --import https://packages.elastic.co/GPG-KEY-elasticsearch
```

ADD the below text in the file with .repo suffix in your "/etc/yum.repos.d/" directory. For example, **elasticsearch.repo**

```
[elasticsearch-2.x]
name = Elasticsearch repository for2.x packages
```

```
baseurl = http://packages.elastic.co/elasticsearch/2.x/centos
gpgcheck = 1
gpgkey = http://packages.elastic.co/GPG-KEY-elasticsearch
enabled = 1
```

You can now install Elasticsearch by using the following command –

```
$ yum install elasticsearch
```

Step 4 – Go to the Elasticsearch home directory and inside the bin folder. Run the elasticsearch.bat file in case of windows or you can do the same using command prompt and through terminal in case of UNIX rum Elasticsearch file.

In Windows –

```
> cd elasticsearch-2.1.0/bin
> elasticsearch
```

In Linux –

```
$ cd elasticsearch-2.1.0/bin
$ ./elasticsearch
```

Note – in case of windows, you might get error stating JAVA_HOME is not set, please set it in environment variables to "C:\Program Files\Java\jre1.8.0_31" or the location where you installed java.

Step 5 – Default port for Elasticsearch web interface is 9200 or you can change it by changing http.port inside elasticsearch.yml file present in bin directory. You can check if the server is up and running by browsing **http://localhost:9200**. It will return a JSON object, which contains the information about the installed Elasticsearch in the following way –

```
{
   "name" : "Brain-Child",
   "cluster_name" : "elasticsearch", "version" : {
      "number" : "2.1.0",
      "build_hash" : "72cd1f1a3eee09505e036106146dc1949dc5dc87",
      "build_timestamp" : "2015-11-18T22:40:03Z",
      "build_snapshot" : false,
      "lucene_version" : "5.3.1"
   },
   "tagline" : "You Know, for Search"
}
```

Step 6 – You can install fiddler2 from www.telerik.com as a front end for your Elasticsearch.

In the configure window of fiddler2, you can hit the address of Elasticsearch adding an index and if you want, then the type/mapping also using HTTP POST method, for example –

Address bar

```
http://localhost:9200/schools/school
```

Request body – You can add JSON object, which will get store into that index.

You can use the same for searching anything by just adding “_search” keyword at the end of URL and sent a query in request body for example –

Address bar

```
POST http://localhost:9200/city/schools/_search
```

Request body

{ "query":{ "match_all":{} } }

This query will return everything from that index, which belongs to that particular type.

You can delete a particular index or type by just putting the URL of the same in address bar and hit it with HTTP DELETE method.

Elasticsearch - Populate

In this section, we will add some index, mapping and data to Elasticsearch. This data will be used in the examples explained in this tutorial.

Create Index

```
PUT http://localhost:9200/schools
```

Request Body

It can contain index specific settings, but for now, it is empty for default settings.

Response

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

(This means index is created)

Create Mapping and Add data

Elasticsearch will auto-create the mapping according to the data provided in request body, we will use its bulk functionality to add more than one JSON object in this index.

```
POST http://localhost:9200/schools/_bulk
```

Request Body

```
{
   "index":{
      "_index":"schools", "_type":"school", "_id":"1"
   }
}
{
   "name":"Central School", "description":"CBSE Affiliation", "street":"Nagan",
   "city":"paprola", "state":"HP", "zip":"176115", "location":[31.8955385, 76.8380405],
   "fees":2000, "tags":["Senior Secondary", "beautiful campus"], "rating":"3.5"
}
{
   "index":{
      "_index":"schools", "_type":"school", "_id":"2"
   }
}
{
   "name":"Saint Paul School", "description":"ICSE
   Afiliation", "street":"Dawarka", "city":"Delhi", "state":"Delhi", "zip":"110075",
   "location":[28.5733056, 77.0122136], "fees":5000,
   "tags":["Good Faculty", "Great Sports"], "rating":"4.5"
}
{
   "index":{"_index":"schools", "_type":"school", "_id":"3"}
}
{
   "name":"Crescent School", "description":"State Board Affiliation", "street":"Tonk Road",
   "city":"Jaipur", "state":"RJ", "zip":"176114","location":[26.8535922, 75.7923988],
   "fees":2500, "tags":["Well equipped labs"], "rating":"4.5"
}
```

Response

```
{
   "took":328, "errors":false,"items":[
      {
         "index":{
            "_index":"schools", "_type":"school", "_id":"1", "_version":1, "_shards":{
               "total":2, "successful":1, "failed":0
            }, "status":201
         }
      },

      {
         "index":{
            "_index":"schools", "_type":"school", "_id":"2", "_version":1, "_shards":{
               "total":2, "successful":1, "failed":0
            }, "status":201
         }
      },

      {
         "index":{
            "_index":"schools", "_type":"school", "_id":"3", "_version":1, "_shards":{
               "total":2, "successful":1, "failed":0
            }, "status":201
         }
      }
   ]
}
```

Add another Index

Create Index

```
POST http://localhost:9200/schools_gov
```

Request Body

It can contain index specific settings, but for now it’s empty for default settings.

Response

```
{"acknowledged": true} (This means index is created)
```

Create Mapping and Add Data

```
POST http://localhost:9200/schools_gov/_bulk
```

Request Body

```
{
   "index":{
      "_index":"schools_gov", "_type":"school", "_id":"1"
   }
}
{
   "name":"Model School", "description":"CBSE Affiliation", "street":"silk city",
   "city":"Hyderabad", "state":"AP", "zip":"500030", "location":[17.3903703, 78.4752129],
   "fees":200, "tags":["Senior Secondary", "beautiful campus"], "rating":"3"
}
{
   "index":{
      "_index":"schools_gov", "_type":"school", "_id":"2"
   }
}
```

```
{
   "name":"Government School", "description":"State Board Affiliation",
   "street":"Hinjewadi", "city":"Pune", "state":"MH", "zip":"411057",
   "location": [18.599752, 73.6821995], "fees":500, "tags":["Great Sports"], "rating":"4"
}
```

Response

```
{
   "took":179, "errors":false, "items":[
      {
        "index":{
           "_index":"schools_gov", "_type":"school", "_id":"1", "_version":1, "_shards":{
              "total":2, "successful":1, "failed":0
           }, "status":201
         }
      },

      {
         "index":{
            "_index":"schools_gov", "_type":"school", "_id":"2", "_version":1, "_shards":{
               "total":2, "successful":1, "failed":0
           }, "status":201
         }
      }
   ]
}
```

Migration between Versions

In any system or software, when we are upgrading to newer version, we need to follow a few steps to maintain the application settings, configurations, data and other things. These steps are required to make the application stable in new system or to maintain the integrity of data (prevent data from getting corrupt).

The following are the steps to upgrade Elasticsearch −

- Read breaking changes docs from https://www.elastic.co/
- Test the upgraded version in your non production environments like in UAT, E2E, SIT or DEV environment.
- Rollback to previous Elasticsearch version is not possible without data backup. A data backup is recommended before upgrading to a higher version.
- We can upgrade using full cluster restart or rolling upgrade. Rolling upgrade is for new versions (for 2.x and newer). There is no service outage, when you are using rolling upgrade method for migration.

Old Version	New Version	Upgrading Method
0.90.x	2.x	Full cluster restart
1.x	2.x	Full cluster restart
2.x	2.y	Rolling upgrade (y > x)

Take data backup before migration and follow the instructions to carry out the backup process. The snapshot and restore module can be used to take backup. This module can be used to take a snapshot of index or full cluster and can be stored in remote repository.

Snapshot and Restore Module

Before starting the backup process, a snapshot repository needs to be registered in Elasticsearch.

```
PUT /_snapshot/backup1
{
   "type": "fs", "settings": {
      ... repository settings ...
   }
}
```

Note – The above text is a HTTP PUT request to **http://localhost:9200/_snapshot/backup1** (there can be an IP address of the remote server instead of the localhost). Rest of the text is request body. You can do this easily using fiddler2 and other web tools in Windows.

We use shared file system (type: fs) for backup; it needs to be registered in every master and data nodes. We just need to add the path.repo variable having path of backup repository as a value.

After we add the repository path, we need to restart the nodes and then registration can be carried out by executing the following command –

```
PUT http://localhost:9200/_snapshot/backup1
{
   "type": "fs", "settings": {
      "location": "/mount/backups/backup1", "compress": true
   }
}
```

Full Cluster Restart

This upgrade process includes the following steps –

Step 1 – Disable shard allocation process and turn off the node.

```
PUT http://localhost:9200/_cluster/settings
{
   "persistent": {
      "cluster.routing.allocation.enable": "none"
   }
}
```

In case of upgrading 0.90.x to 1.x use the following request –

```
PUT http://localhost:9200/_cluster/settings
{
   "persistent": {
      "cluster.routing.allocation.disable_allocation": false,
      "cluster.routing.allocation.enable": "none"
   }
}
```

Step 2 – Make a synched flush to Elasticsearch –

```
POST http://localhost:9200/_flush/synced
```

Step 3 – On all nodes, kill all the elastic services.

Step 4 – Do the following on every node –

> **In Debian or Red Hat Node** – rmp or dpkg can be used to upgrade the node by installing new packages. Do not overwrite config files.
>
> **In Windows (zip file) or UNIX (tar file)** – Extract the new version without overwriting the config directory. You can copy the files from old installation or can change path.conf or path.data.

Step 5 – Initiate the nodes again starting with the master node (nodes with node.master set to true and node.data set to false) in the cluster. Wait for some time to establish a cluster. You can check by monitoring the logs or using the following request –

```
GET _cat/health or http://localhost:9200/_cat/health
GET _cat/nodes or http://localhost:9200/_cat/health
```

Step 6 – Monitor the progress of formation of cluster by using **GET _cat/health** request and wait for the yellow in response, the response will be something like this –

```
1451295971 17:46:11 elasticsearch yellow 1 1 5 5 0 0 5 0 - 50.0%
```

Step 7 – Enable the shard allocation process, which was disabled in Step 1, by using the following request –

```
PUT http://localhost:9200/_cluster/settings
{
   "persistent": {
      "cluster.routing.allocation.enable": "all"
   }
}
```

In case of upgrading 0.90.x to 1.x, use the following request −

```
PUT http://localhost:9200/_cluster/settings
{
   "persistent": {
      "cluster.routing.allocation.disable_allocation": true,
      "cluster.routing.allocation.enable": "all"
   }
}
```

Rolling Upgrades

It is same like Full cluster restart, except Step 3. Here, you stop one node and upgrade. After upgrading, restart the node and repeat these for the all nodes. After enabling the shard allocation process, it can be monitored by the following request −

```
GET http://localhost:9200/_cat/recovery
```

Elasticsearch - API Conventions

Application Programming Interface (API) in web is a group of function calls or other programming instructions to access the software component in that particular web application. For example, Facebook API helps a developer to create applications by accessing data or other functionalities from Facebook; it can be date of birth or status update.

Elasticsearch provides a REST API, which is accessed by JSON over HTTP. Elasticsearch uses the following conventions −

Multiple Indices

Most of the operations, mainly searching and other operations, in APIs are for one or more than one indices. This helps the user to search in multiple places or all the available data by just executing a query once. Many different notations are used to perform operations in multiple indices. We will discuss a few of them here in this section.

Comma Separated Notation

```
POST http://localhost:9200/index1,index2,index3/_search
```

Request Body

```
{
   "query":{
      "query_string":{
         "query":"any_string"
      }
   }
}
```

Response

JSON objects from index1, index2, index3 having any_string in it.

_all keyword for all indices

```
POST http://localhost:9200/_all/_search
```

Request Body

```
{
   "query":{
      "query_string":{
         "query":"any_string"
      }
   }
}
```

Response

JSON objects from all indices and having any_string in it.

Wildcards (* , + , –)

```
POST http://localhost:9200/school*/_search
```

Request Body

```
{
   "query":{
      "query_string":{
         "query":"CBSE"
      }
   }
}
```

Response

JSON objects from all indices which start with school having CBSE in it.

Alternatively, you can use the following code as well –

```
POST http://localhost:9200/school*,-schools_gov /_search
```

Request Body

```
{
   "query":{
      "query_string":{
         "query":"CBSE"
      }
   }
}
```

Response

JSON objects from all indices which start with “school” but not from schools_gov and having CBSE in it.

There are also some **URL query string parameters** –

> **ignore_unavailable** – No error will occur or operation will not be stopped, if the one or more index present in URL does not exist. For example, schools index exist but book_shops does not exist –

```
POST http://localhost:9200/school*,book_shops/_search
```

Request Body

```
{
   "query":{
      "query_string":{
         "query":"CBSE"
      }
   }
}
```

Response

```
{
   "error":{
      "root_cause":[{
         "type":"index_not_found_exception", "reason":"no such index",
         "resource.type":"index_or_alias", "resource.id":"book_shops",
         "index":"book_shops"
      }],

      "type":"index_not_found_exception", "reason":"no such index",
      "resource.type":"index_or_alias", "resource.id":"book_shops",
      "index":"book_shops"

   },"status":404
}
```

Take a look at the following code –

```
POST http://localhost:9200/school*,book_shops/_search?ignore_unavailable = true
```

Request Body

```
{
   "query":{
      "query_string":{
         "query":"CBSE"
      }
   }
}
```

Response (no error)

JSON objects from all indices which start with school having CBSE in it.

allow_no_indices

true value of this parameter will prevent error, if a URL with wildcard results in no indices.

For example, there is no index that starts with schools_pri –

```
POST

http://localhost:9200/schools_pri*/_search?allow_no_indices = true
```

Request Body

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

Response (No errors)

```
{
   "took":1,"timed_out": false, "_shards":{"total":0, "successful":0, "failed":0},
   "hits":{"total":0, "max_score":0.0, "hits":[]}
}
```

expand_wildcards

This parameter decides whether the wildcards need to be expanded to open indices or closed indices or both. The value of this parameter can be open and closed or none and all.

For example, close index schools –

```
POST http://localhost:9200/schools/_close
```

Response

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

Take a look at the following code −

```
POST http://localhost:9200/school*/_search?expand_wildcards = closed
```

Request Body

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

Response

```
{
   "error":{
      "root_cause":[{
         "type":"index_closed_exception", "reason":"closed", "index":"schools"
      }],

      "type":"index_closed_exception", "reason":"closed", "index":"schools"
   }, "status":403
}
```

Date Math Support in Index Names

Elasticsearch offers a functionality to search indices according to date and time. We need to specify date and time in a specific format. For example, accountdetail-2015.12.30, index will store the bank account details of 30th December 2015. Mathematic operations can be performed to get details for a particular date or a range of date and time.

Format for date math index name −

```
<static_name{date_math_expr{date_format|time_zone}}>
http://localhost:9200/<accountdetail-{now-2d{YYYY.MM.dd|utc}}>/_search
```

static_name is a part of expression which remains same in every date math index like account detail. date_math_expr contains the mathematical expression that determines the date and time dynamically like now-2d. date_format contains the format in which the date is written in index like YYYY.MM.dd. If today's date is 30th December 2015, then <accountdetail-{now-2d{YYYY.MM.dd}}> will return accountdetail-2015.12.28.

Expression	Resolves to

<accountdetail-{now-d}>	accountdetail-2015.12.29
<accountdetail-{now-M}>	accountdetail-2015.11.30
<accountdetail-{now{YYYY.MM}}>	accountdetail-2015.12

We will now see some of the common options available in Elasticsearch that can be used to get the response in a specified format.

Pretty Results

We can get response in a well-formatted JSON object by just appending a URL query parameter, i.e., pretty = true.

```
POST http://localhost:9200/schools/_search?pretty = true
```

Request Body

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

Response

```
……………………..
{
   "_index" : "schools", "_type" : "school", "_id" : "1", "_score" : 1.0,
   "_source":{
      "name":"Central School", "description":"CBSE Affiliation",
      "street":"Nagan", "city":"paprola", "state":"HP", "zip":"176115",
      "location": [31.8955385, 76.8380405], "fees":2000,
      "tags":["Senior Secondary", "beautiful campus"], "rating":"3.5"
   }

}
………………….
```

Human Readable Output

This option can change the statistical responses either into human readable form (If human = true) or computer readable form (if human = false). For example, if human = true then distance_kilometer = 20KM and if human = false then distance_meter = 20000, when response needs to be used by another computer program.

Response Filtering

We can filter the response to less fields by adding them in the field_path parameter. For example,

```
POST http://localhost:9200/schools/_search?filter_path = hits.total
```

Request Body

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

Response

```
{"hits":{"total":3}}
```

Elasticsearch - Document APIs

Elasticsearch provides single document APIs and multi-document APIs, where the API call is targeting single document and multiple documents respectively.

Index API

It helps to add or updates the JSON document in an index when a request is made to that respective index with specific mapping. For example, the below request will add the JSON object to index schools and under school mapping.

```
POST http://localhost:9200/schools/school/4
```

Request Body

```
{
   "name":"City School", "description":"ICSE", "street":"West End", "city":"Meerut",
   "state":"UP", "zip":"250002", "location":[28.9926174, 77.692485], "fees":3500,
   "tags":["fully computerized"], "rating":"4.5"
}
```

Response

```
{

   "_index":"schools", "_type":"school", "_id":"4", "_version":1,

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

}
```

Automatic Index Creation

When a request is made to add JSON object to a particular index and if that index does not exist then this API automatically creates that index and also the underlying mapping for that particular JSON object. This functionality can be disabled by changing the values of following parameters to false, which are present in elasticsearch.yml file.

```
action.auto_create_index:false
index.mapper.dynamic:false
```

You can also restrict the auto creation of index, where only index name with specific patterns are allowed by changing the value of the following parameter –

```
action.auto_create_index:+acc*,-bank*
```

(where + indicates **allowed** and – indicates **not allowed**)

Versioning

Elasticsearch also provides version control facility. We can use a version query parameter to specify the version of a particular document. For example,

```
POST http://localhost:9200/schools/school/1?version = 1
```

Request Body

```
{
   "name":"Central School", "description":"CBSE Affiliation", "street":"Nagan",
   "city":"paprola", "state":"HP", "zip":"176115", "location":[31.8955385, 76.8380405],
   "fees":2200, "tags":["Senior Secondary", "beautiful campus"], "rating":"3.3"
}
```

Response

```
{
   "_index":"schools", "_type":"school", "_id":"1", "_version":2,
   "_shards":{"total":2, "successful":1,"failed":0}, "created":false
}
```

There are two most important types of versioning; internal versioning is the default version that starts with 1 and increments with each update, deletes included. The version number can be set externally. To enable this functionality, we need to set version_type to external.

Versioning is a real-time process and it is not affected by the real time search operations.

Operation Type

The operation type is used to force a create operation, this helps to avoid the overwriting of existing document.

```
POST http://localhost:9200/tutorials/chapter/1?op_type = create
```

Request Body

```
{
   "Text":"this is chapter one"
}
```

Response

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

Automatic ID generation

When ID is not specified in index operation, then Elasticsearch automatically generates id for that document.

Parents and Children

You can define the parent of any document by passing the id of parent document in parent URL query parameter.

```
POST http://localhost:9200/tutorials/article/1?parent = 1
```

Request Body

```
{
   "Text":"This is article 1 of chapter 1"
}
```

Note − If you get exception while executing this example, please recreate the index by adding the following in the index.

```
{
   "mappings": {
      "chapter": {},
      "article": {
         "_parent": {
            "type": "chapter"
         }
      }
   }
}
```

Timeout

By default, the index operation will wait on the primary shard to become available for up to 1 minute before failing and responding with an error. This timeout value can be changed explicitly by passing a value to timeout parameter.

```
POST http://localhost:9200/tutorials/chapter/2?timeout = 3m
```

Request Body

```
{
   "Text":"This is chapter 2 waiting for primary shard for 3 minutes"
}
```

Get API

API helps to extract type JSON object by performing a get request for a particular document. For example,

```
GET http://localhost:9200/schools/school/1
```

Response

```
{
   "_index":"schools", "_type":"school", "_id":"1", "_version":2,
   "found":true, "_source":{
      "name":"Central School", "description":"CBSE Affiliation",
      "street":"Nagan", "city":"paprola", "state":"HP", "zip":"176115",
      "location":[31.8955385,76.8380405], "fees":2200,
      "tags":["Senior Secondary", "beautiful campus"], "rating":"3.3"
   }
}
```

- This operation is real time and does not get affected by the refresh rate of Index.
- You can also specify the version, then Elasticsearch will fetch that version of document only.
- You can also specify the _all in the request, so that the Elasticsearch can search for that document id in every type and it will return the first matched document.
- You can also specify the fields you want in your result from that particular document.

```
GET http://localhost:9200/schools/school/1?fields = name,fees
```

Response

```
……………………..
"fields":{
   "name":["Central School"], "fees":[2200]
}
……………………..
```

You can also fetch the source part in your result by just adding _source part in your get request.

```
GET http://localhost:9200/schools/school/1/_source
```

Response

```
{
   "name":"Central School", "description":"CBSE Afiliation", "street":"Nagan",
   "city":"paprola", "state":"HP", "zip":"176115", "location":[31.8955385, 76.8380405],
   "fees":2200, "tags":["Senior Secondary", "beatiful campus"], "rating":"3.3"
}
```

You can also refresh the shard before doing get operation by set refresh parameter to true.

Delete API

You can delete a particular index, mapping or a document by sending a HTTP DELETE request to Elasticsearch. For example,

```
DELETE http://localhost:9200/schools/school/4
```

Response

```
{
   "found":true, "_index":"schools", "_type":"school", "_id":"4", "_version":2,
   "_shards":{"total":2, "successful":1, "failed":0}
}
```

Version of the document can be specified to delete that particular version.

Routing parameter can be specified to delete the document from a particular user and the operation fails if the document does not belong to that particular user.

In this operation, you can specify refresh and timeout option same like GET API.

Update API

Script is used for performing this operation and versioning is used to make sure that no updates have happened during the get and re-index. For example, update the fees of school using script –

```
POST http://localhost:9200/schools_gov/school/1/_update
```

Request Body

```
{
   "script":{
      "inline": "ctx._source.fees+ = inc", "params":{
         "inc": 500
      }
   }
}
```

Response

```
{
   "_index":"schools_gov", "_type":"school", "_id":"1", "_version":2,
   "_shards":{"total":2, "successful":1, "failed":0}
}
```

Note – If you get script exception, it is recommended to add the following lines in elastcisearch.yml

```
script.inline: on
script.indexed: on
```

You can check the update by sending get request to the updated document.

```
GET http://localhost:9200/schools_gov/school/1
```

Multi Get API

It possesses same functionality like GET API, but this get request can return more than one document. We use a doc array to specify the index, type and id of all the documents that need to be extracted.

```
POST http://localhost:9200/_mget
```

Request Body

```
{
   "docs":[
      {
         "_index": "schools", "_type": "school", "_id": "1"
      },

      {
         "_index":"schools_gev", "_type":"school", "_id": "2"
      }
   ]
}
```

Response

```
{
   "docs":[
      {
         "_index":"schools", "_type":"school", "_id":"1",
         "_version":1, "found":true, "_source":{
            "name":"Central School", "description":"CBSE Afiliation",
            "street":"Nagan", "city":"paprola", "state":"HP", "zip":"176115",
            "location":[31.8955385,76.8380405], "fees":2000,
            "tags":["Senior Secondary", "beatiful campus"], "rating":"3.5"
         }
      },

      {
         "_index":"schools_gev", "_type":"school", "_id":"2", "error":{

            "root_cause":[{
               "type":"index_not_found_exception", "reason":"no such index",
               "index":"schools_gev"
            }],

            "type":"index_not_found_exception", "reason":"no such index",
            "index":"schools_gev"
         }
      }
   ]
}
```

Bulk API

This API is used to upload or delete the JSON objects in bulk by making multiple index/delete operations in a single request. We need to add “_bulk” keyword to call this API. The example of this API is already performed in populate Elasticsearch article. All other functionalities are same as of GET API.

Elasticsearch - Search APIs

This API is used to search content in Elasticsearch. Either a user can search by sending a get request with query string as a parameter or a query in the message body of post request. Mainly all the search APIS are multi-index, multi-type.

Multi-Index

Elasticsearch allows us to search for the documents present in all the indices or in some specific indices. For example, if we need to search all the documents with a name that contains central.

```
GET http://localhost:9200/_search?q = name:central
```

Response

```
{
   "took":78, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{
      "total":1, "max_score":0.19178301, "hits":[{
         "_index":"schools", "_type":"school", "_id":"1", "_score":0.19178301,
         "_source":{
            "name":"Central School", "description":"CBSE Affiliation",
            "street":"Nagan", "city":"paprola", "state":"HP", "zip":"176115",
            "location":[31.8955385, 76.8380405], "fees":2000,
            "tags":["Senior Secondary", "beautiful campus"], "rating":"3.5"
         }
      }]
   }
}
```

Or, we can search for the same in schools, schools_gov indices –

```
GET http://localhost:9200/schools,schools_gov/_search?q = name:model
```

Multi-Type

We can also search all the documents in an index across all types or in some specified type. For example,

```
Get http://localhost:9200/schools/_search?q = tags:sports
```

Response

```
{
   "took":16, "timed_out":false, "_shards":{"total":5, "successful":5, "failed":0},
   "hits":{
      "total":1, "max_score":0.5, "hits":[{
         "_index":"schools", "_type":"school", "_id":"2", "_score":0.5,
         "_source":{
            "name":"Saint Paul School", "description":"ICSE Afiliation",
            "street":"Dawarka", "city":"Delhi", "state":"Delhi", "zip":"110075",
            "location":[28.5733056, 77.0122136], "fees":5000,
            "tags":["Good Faculty", "Great Sports"], "rating":"4.5"
         }
      }]
   }
}
```

URI Search

Many parameters can be passed in a search operation using Uniform Resource Identifier −

Sr.No	Parameter & Description
1	**Q** This parameter is used to specify query string.
2	**lenient** Format based errors can be ignored by just setting this parameter to true. It is false by default.
3	**fields** This parameter helps us to get response from selective fields.
4	**sort** We can get sorted result by using this parameter, the possible values for this parameter is fieldName, fieldName:asc/fieldname:desc
5	**timeout** We can restrict the search time by using this parameter and response only contains the hits in that specified time. By default, there is no timeout.
6	**terminate_after** We can restrict the response to a specified number of documents for each shard, upon reaching which the query will terminate early. By default, there is no terminate_after.
7	**from**

	The starting from index of the hits to return. Defaults to 0.
8	**size** It denotes the number of hits to return. Defaults to 10.

Request Body Search

We can also specify query using query DSL in request body and there are many examples already given in previous chapters like –

```
POST http://localhost:9200/schools/_search
```

Request Body

```
{
   "query":{
      "query_string":{
         "query":"up"
      }
   }
}
```

Response

```
……………………………………………….

{
   "_source":{
      "name":"City School", "description":"ICSE", "street":"West End",
      "city":"Meerut", "state":"UP", "zip":"250002", "location":[28.9926174, 77.692485],
      "fees":3500, "tags":["Well equipped labs"],"rating":"4.5"
   }
}
……………………………………………….
```

Elasticsearch - Aggregations

This framework collects all the data selected by the search query. This framework consists of many building blocks, which help in building complex summaries of the data. The basic structure of aggregation is presented below –

```
"aggregations" : {
   "<aggregation_name>" : {
      "<aggregation_type>" : {
         <aggregation_body>
      }
      [,"meta" : { [<meta_data_body>] } ]?
      [,"aggregations" : { [<sub_aggregation>]+ } ]?
   }
}
```

There are different types of aggregations, each with its own purpose –

Metrics Aggregations

These aggregations help in computing matrices from the field’s values of the aggregated documents and sometime some values can be generated from scripts.

Numeric matrices are either **single-valued** like **average aggregation** or **multi-valued** like **stats**.

Avg Aggregation

This aggregation is used to get the average of any numeric field present in the aggregated documents. For example,

```
POST http://localhost:9200/schools/_search
```

Request Body

```
{
   "aggs":{
      "avg_fees":{"avg":{"field":"fees"}}
   }
}
```

Response

```
{
   "took":44, "timed_out":false, "_shards":{"total":5, "successful":5, "failed":0},
   "hits":{
      "total":3, "max_score":1.0, "hits":[
         {
            "_index":"schools", "_type":"school", "_id":"2", "_score":1.0,
            "_source":{
               "name":"Saint Paul School", "description":"ICSE Affiliation",
               "street":"Dawarka", "city":"Delhi", "state":"Delhi",
               "zip":"110075", "location":[28.5733056, 77.0122136], "fees":5000,
               "tags":["Good Faculty", "Great Sports"], "rating":"4.5"
            }
         },

         {
            "_index":"schools", "_type":"school", "_id":"1", "_score":1.0,
            "_source":{
               "name":"Central School", "description":"CBSE Affiliation",
               "street":"Nagan", "city":"paprola", "state":"HP", "zip":"176115",
               "location":[31.8955385, 76.8380405], "fees":2200,
               "tags":["Senior Secondary", "beautiful campus"], "rating":"3.3"
            }
         },

         {
            "_index":"schools", "_type":"school", "_id":"3", "_score":1.0,
            "_source":{
               "name":"Crescent School", "description":"State Board Affiliation",
               "street":"Tonk Road", "city":"Jaipur", "state":"RJ",
               "zip":"176114", "location":[26.8535922, 75.7923988], "fees":2500,
               "tags":["Well equipped labs"], "rating":"4.5"
            }
         }
      ]
   }, "aggregations":{"avg_fees":{"value":3233.3333333333335}}
}
```

If the value is not present in one or more aggregated documents, it gets ignored by default. You can add a missing field in the aggregation for treating missing value as default.

```
{
   "aggs":{
      "avg_fees":{
         "avg":{
            "field":"fees"
            "missing":0
         }
      }
   }
}
```

Cardinality Aggregation

This aggregation gives the count of distinct values of a particular field. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "aggs":{
      "distinct_name_count":{"cardinality":{"field":"name"}}
   }
}
```

Response

```
………………………………………………
{
   "name":"Government School", "description":"State Board Afiliation",
   "street":"Hinjewadi", "city":"Pune", "state":"MH", "zip":"411057",
   "location":[18.599752, 73.6821995], "fees":500, "tags":["Great Sports"],
   "rating":"4"
},

{
   "_index":"schools_gov", "_type": "school", "_id":"1", "_score":1.0,
   "_source":{
      "name":"Model School", "description":"CBSE Affiliation", "street":"silk city",
      "city":"Hyderabad", "state":"AP", "zip":"500030",
      "location":[17.3903703, 78.4752129], "fees":700,
      "tags":["Senior Secondary", "beautiful campus"], "rating":"3"
   }
}, "aggregations":{"disticnt_name_count":{"value":3}}
………………………………………………
```

Note – The value of cardinality is 3 because there are three distinct values in name — Government, School and Model.

Extended Stats Aggregation

This aggregation generates all the statistics about a specific numerical field in aggregated documents. For example,

```
POST http://localhost:9200/schools/school/_search
```

Request Body

```
{
   "aggs" : {
      "fees_stats" : { "extended_stats" : { "field" : "fees" } }
   }
}
```

Response

```
………………………………………………
{
   "aggregations":{
      "fees_stats":{
         "count":3, "min":2200.0, "max":5000.0,
         "avg":3233.3333333333335, "sum":9700.0,
         "sum_of_squares":3.609E7, "variance":1575555.555555556,
         "std_deviation":1255.2113589175156,

         "std_deviation_bounds":{
            "upper":5743.756051168364, "lower":722.9106154983024
         }
      }
   }
}
………………………………………………
```

Max Aggregation

This aggregation finds the max value of a specific numeric field in aggregated documents. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "aggs" : {
      "max_fees" : { "max" : { "field" : "fees" } }
   }
}
```

Response

```
………………………………………………
{
   aggregations":{"max_fees":{"value":5000.0}}
}
………………………………………………
```

Min Aggregation

This aggregation finds the max value of a specific numeric field in aggregated documents. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "aggs" : {
      "min_fees" : { "min" : { "field" : "fees" } }
   }
}
```

Response

```
……………………………………………………
"aggregations":{"min_fees":{"value":500.0}}
……………………………………………………
```

Sum Aggregation

This aggregation calculates the sum of a specific numeric field in aggregated documents. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "aggs" : {
      "total_fees" : { "sum" : { "field" : "fees" } }
   }
}
```

Response

```
……………………………………………………
"aggregations":{"total_fees":{"value":10900.0}}
……………………………………………………
```

There are some other metrics aggregations which are used in special cases like geo bounds aggregation and geo centroid aggregation for the purpose of geo location.

Bucket Aggregations

These aggregations contain many buckets for different types of aggregations having a criterion, which determines whether a document belongs to that bucket or not. The bucket aggregations have been described below –

Children Aggregation

This bucket aggregation makes a collection of documents, which are mapped to parent bucket. A type parameter is used to define the parent index. For example, we have a brand and its different models, and then the model type will have the following _parent field –

```
{
   "model" : {
      "_parent" : {
         "type" : "brand"
      }
   }
}
```

There are many other special bucket aggregations, which are useful in many other cases, those are −

- Date Histogram Aggregation
- Date Range Aggregation
- Filter Aggregation
- Filters Aggregation
- Geo Distance Aggregation
- GeoHash grid Aggregation
- Global Aggregation
- Histogram Aggregation
- IPv4 Range Aggregation
- Missing Aggregation
- Nested Aggregation
- Range Aggregation
- Reverse nested Aggregation
- Sampler Aggregation
- Significant Terms Aggregation
- Terms Aggregation

Aggregation Metadata

You can add some data about the aggregation at the time of request by using meta tag and can get that in response. For example,

```
POST http://localhost:9200/school*/report/_search
```

Request Body

```
{
   "aggs" : {
      "min_fees" : { "avg" : { "field" : "fees" } ,
         "meta" :{
            "dsc" :"Lowest Fees"
         }
      }
   }
}
```

Response

```
……………………………………………………
{
   "aggregations":{"min_fees":{"meta":{"dsc":"Lowest Fees"}, "value":2180.0}}
}
……………………………………………………
```

Elasticsearch - Index APIs

These APIs are responsible for managing all the aspects of index like settings, aliases, mappings, index templates.

Create Index

This API helps you to create index. Index can be created automatically when a user is passing JSON objects to any index or it can be created before that. To create an index, you just need to send a post request with settings, mappings and aliases or just a simple request without body. For example,

```
POST http://localhost:9200/colleges
```

Response

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

Or, with some settings –

```
POST http://localhost:9200/colleges
```

Request Body

```
{
   "settings" : {
      "index" : {
         "number_of_shards" : 5, "number_of_replicas" : 3
      }
   }
}
```

Response

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

Or with mapping –

```
POST http://localhost:9200/colleges
```

Request Body

```
{
   "settings" : {
      "number_of_shards" : 3
   },
   "mappings" : {
      "type1" : {
         "_source" : { "enabled" : false }, "properties" : {
            "college_name" : { "type" : "string" }, "college type" : {"type":"string"}
         }
      }
   }
}
```

Response

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

Or, with alias –

```
POST http://localhost:9200/colleges
```

Request Body

```
{
   "aliases" : {
      "alias_1" : {}, "alias_2" : {
         "filter" : {
            "term" : {"user" : "manu" }
         },
         "routing" : "manu"
      }
   }
}
```

Response

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

Delete Index

This API helps you to delete any index. You just need to pass a delete request with the URL of that particular Index. For example,

```
DELETE http://localhost:9200/colleges
```

You can delete all indices by just using _all,*.

Get Index

This API can be called by just sending get request to one or more than one indices. This returns the information about index.

```
GET http://localhost:9200/schools
```

Response

```
{
   "schools":{
      "aliases":{}, "mappings":{
         "school":{
            "properties":{
               "city":{"type":"string"}, "description":{"type":"string"},
                  "fees":{"type":"long"}, "location":{"type":"double"},
                  "name":{"type":"string"}, "rating":{"type":"string"},
                  "state":{"type":"string"}, "street":{"type":"string"},
                  "tags":{"type":"string"}, "zip":{"type":"string"}
            }
         }
      },
      "settings":{
         "index":{
            "creation_date":"1454409831535", "number_of_shards":"5",
               "number_of_replicas":"1", "uuid":"iKdjTtXQSMCW4xZMhpsOVA",
               "version":{"created":"2010199"}
         }
      },
      "warmers":{}
   }
}
```

You can get the information of all the indices by using _all or *.

Index Exist

Existence of an index can be determined by just sending a get request to that index. If the HTTP response is 200, it exists; if it is 404, it does not exist.

Open / Close Index API

It’s very easy to close or open one or more index by just adding _close or _open in post to request to that index. For example,

```
POST http://localhost:9200/schools/_close
```

Or

```
POST http://localhost:9200/schools/_open
```

Index Aliases

This API helps to give an alias to any index by using _aliases keyword. Single alias can be mapped to more than one and alias cannot have the same name as index. For example,

```
POST http://localhost:9200/_aliases
```

Request Body

```
{
   "actions" : [
      { "add" : { "index" : "schools", "alias" : "schools_pri" } }
   ]
}
```

Response

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

Then,

```
GET http://localhost:9200/schools_pri
```

Response

```
………………………………………………
{"schools":{"aliases":{"schools_pri":{}},"}}
………………………………………………
```

Index Settings

You can get the index settings by just appending _settings keyword at the end of URL. For example,

```
GET http://localhost:9200/schools/_settings
```

Response

```
{
   "schools":{
      "settings":{
         "index":{
            "creation_date":"1454409831535", "number_of_shards":"5",
               "number_of_replicas":"1", "uuid":"iKdjTtXQSMCW4xZMhpsOVA",
               "version":{"created":"2010199"}
         }
      }
   }
}
```

Analyze

This API helps to analyze the text and send the tokens with offset value and data type. For example,

```
POST http://localhost:9200/_analyze
```

Request Body

```
{
   "analyzer" : "standard",
   "text" : "you are reading this at tutorials point"
}
```

Response

```
{
   "tokens":[
      {"token":"you", "start_offset":0, "end_offset":3, "type":"<ALPHANUM>", "position":0},
         {"token":"are", "start_offset":4, "end_offset":7, "type":"<ALPHANUM>", "position":1},
         {"token":"reading", "start_offset":8, "end_offset":15, "type":"<ALPHANUM>", "position":2},
         {"token":"this", "start_offset":16, "end_offset":20, "type":"<ALPHANUM>", "position":3},
         {"token":"at", "start_offset":21, "end_offset":23, "type":"<ALPHANUM>", "position":4},
         {"token":"tutorials", "start_offset":24, "end_offset":33, "type":"<ALPHANUM>", "position":5},
         {"token":"point", "start_offset":34, "end_offset":39, "type":"<ALPHANUM>", "position":6}
   ]
}
```

You can also analyze a text with any index, and then the text will be analyzed according to the analyzer associated with that index.

Index Templates

You can also create index templates with mappings, which can be applied to new indices. For example,

```
POST http://localhost:9200/_template/template_a
```

Request Body

```
{
   "template" : "tu*",
      "settings" : {
         "number_of_shards" : 3
   },
   "mappings" : {
      "chapter" : {
         "_source" : { "enabled" : false }
      }
   }
}
```

Any index that starts with “tu” will have the same settings as template_a.

Index Stats

This API helps you to extract statistics about a particular index. You just need to send a get request with the index URL and _stats keyword at the end.

```
GET http://localhost:9200/schools/_stats
```

Response

```
………………………………………………

{"_shards":{"total":10, "successful":5, "failed":0}, "_all":{"primaries":{"docs":{
   "count":3, "deleted":0}}}, "store":{"size_in_bytes":16653, "throttle_time_in_millis":0},

………………………………………………
```

Flush

This API helps to clean the data from index memory and migrate it to index storage and also cleans internal transaction log. For example,

```
GET http://localhost:9200/schools/_flush
```

Response

```
{"_shards":{"total":10, "successful":5, "failed":0}}
```

Refresh

Refresh is scheduled by default in Elasticsearch, but you can refresh one or more indices explicitly by using _refresh. For example,

```
GET http://localhost:9200/schools/_refresh
```

Response

```
{"_shards":{"total":10, "successful":5, "failed":0}}
```

Elasticsearch - Cluster APIs

This API is used for getting information about cluster and its nodes and making changes in them. For calling this API, we need to specify the node name, address or _local. For example,

```
GET http://localhost:9200/_nodes/_local
```

Response

```
……………………………………………………
{
   "cluster_name":"elasticsearch", "nodes":{
      "Vy3KxqcHQdm4cIM22U1ewA":{
         "name":"Red Guardian", "transport_address":"127.0.0.1:9300",
         "host":"127.0.0.1", "ip":"127.0.0.1", "version":"2.1.1",
         "build":"40e2c53", "http_address":"127.0.0.1:9200",
      }
   }
}
……………………………………………………
```

Or

```
Get http://localhost:9200/_nodes/127.0.0.1
```

Response

Same as in the above example.

Cluster Health

This API is used to get the status on the health of the cluster by appending health keyword. For example,

```
GET http://localhost:9200/_cluster/health
```

Response

```
{
   "cluster_name":"elasticsearch", "status":"yellow", "timed_out":false,
   "number_of_nodes":1, "number_of_data_nodes":1, "active_primary_shards":23,
   "active_shards":23, "relocating_shards":0, "initializing_shards":0,
   "unassigned_shards":23, "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":50.0
}
```

Cluster State

This API is used to get state information about a cluster by appending 'state' keyword URL. The state information contains version, master node, other nodes, routing table, metadata and blocks. For example,

```
GET http://localhost:9200/_cluster/state 10. Elasticsearch – Cluster APIs
```

Response

```
………………………………………………

{
   "cluster_name":"elasticsearch", "version":27, "state_uuid":"B3P7uHGKQUGsSsiX2rGYUQ",
   "master_node":"Vy3KxqcHQdm4cIM22U1ewA",

}
………………………………………………
```

Cluster Stats

This API helps to retrieve statistics about cluster by using 'stats' keyword. This API returns shard number, store size, memory usage, number of nodes, roles, OS, and file system. For example,

```
GET http://localhost:9200/_cluster/stats
```

Response

```
………………………………………………

{
   "timestamp":1454496710020, "cluster_name":"elasticsearch", "status":"yellow",
   "indices":{
      "count":5, "shards":{
         "total":23, "primaries":23, "replication":0.0,"
      }
   }

}
………………………………………………
```

Pending Cluster Tasks

This API is used for monitoring pending tasks in any cluster. Tasks are like create index, update mapping, allocate shard, fail shard etc. For example,

```
GET http://localhost:9200/_cluster/pending_tasks
```

Cluster Reroute

This API is used for moving shard from one node to another or to cancel any allocation or allocate any unassigned shard. For example,

```
POST http://localhost:9200/_cluster/reroute
```

Request Body

```
{
   "commands" : [
      {
         "move" :
         {
            "index" : "schools", "shard" : 2,
            "from_node" : "nodea", "to_node" : "nodeb"
         }
      },

      {
         "allocate" : {
            "index" : "test", "shard" : 1, "node" : "nodec"
         }
      }
   ]
}
```

Cluster Update Settings

This API allows you to update the settings of a cluster by using settings keyword. There are two types of settings — persistent (applied across restarts) and transient (do not survive a full cluster restart).

Node Stats

This API is used to retrieve the statistics of one more nodes of the cluster. Node stats are almost the same as cluster. For example,

```
GET http://localhost:9200/_nodes/stats
```

Response

```
………………………………………………

{
   "cluster_name":"elasticsearch", "nodes":{
      "Vy3KxqcHQdm4cIM22U1ewA":{
         "timestamp":1454497097572, "name":"Red Guardian",
         "transport_address":"127.0.0.1:9300", "host":"127.0.0.1", "ip":["127.0.0.1:9300",
      }
   }

}
………………………………………………
```

Nodes hot_threads

This API helps you to retrieve information about the current hot threads on each node in cluster. For example,

```
GET http://localhost:9200/_nodes/hot_threads
```

Response

```
………………………………………………
::: {Red Guardian} {Vy3KxqcHQdm4cIM22U1ewA} {127.0.0.1}{127.0.0.1:9300}Hot threads at
   2016-02-03T10:59:48.856Z, interval = 500ms, busiestThreads = 3,
   ignoreIdleThreads = true:0.0% (0s out of 500ms) cpu usage by thread 'Attach Listener'
      unique snapshot
      unique snapshot
………………………………………………
```

Elasticsearch - Query DSL

In Elasticsearch, searching is carried out by using query based on JSON. Query is made up of two clauses –

- **Leaf Query Clauses** – These clauses are match, term or range, which look for a specific value in specific field.
- **Compound Query Clauses** – These queries are a combination of leaf query clauses and other compound queries to extract the desired information.

Elasticsearch supports a large number of queries. A query starts with a query key word and then has conditions and filters inside in the form of JSON object. The different types of queries have been described below –

Match All Query

This is the most basic query; it returns all the content and with the score of 1.0 for every object. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

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

Response

```
{
   "took":1, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{
      "total":5, "max_score":1.0, "hits":[
         {
            "_index":"schools", "_type":"school", "_id":"2", "_score":1.0,
            "_source":{
               "name":"Saint Paul School", "description":"ICSE Affiliation",
               "street":"Dawarka", "city":"Delhi", "state":"Delhi",
               "zip":"110075", "location":[28.5733056, 77.0122136], "fees":5000,
               "tags":["Good Faculty", "Great Sports"], "rating":"4.5"
            }
         },

         {
            "_index":"schools_gov", "_type":"school", "_id":"2", "_score":1.0,
            "_source":{
               "name":"Government School", "description":"State Board Affiliation",
               "street":"Hinjewadi", "city":"Pune", "state":"MH", "zip":"411057",
               "location":[18.599752, 73.6821995], "fees":500, "tags":["Great Sports"],
               "rating":"4"
            }
         },

         {
            "_index":"schools", "_type":"school", "_id":"1", "_score":1.0,
            "_source":{
               "name":"Central School", "description":"CBSE Affiliation",
               "street":"Nagan", "city":"paprola", "state":"HP",
               "zip":"176115", "location":[31.8955385, 76.8380405],
               "fees":2200, "tags":["Senior Secondary", "beautiful campus"],
               "rating":"3.3"
            }
         },
         {
            "_index":"schools_gov", "_type":"school", "_id":"1", "_score":1.0,
            "_source":{
               "name":"Model School", "description":"CBSE Affiliation",
               "street":"silk city", "city":"Hyderabad", "state":"AP",
               "zip":"500030", "location":[17.3903703, 78.4752129], "fees":700,
               "tags":["Senior Secondary", "beautiful campus"], "rating":"3"
            }
         },

         {
            "_index":"schools", "_type":"school", "_id":"3", "_score":1.0,
            "_source":{
               "name":"Crescent School", "description":"State Board Affiliation",
               "street":"Tonk Road", "city":"Jaipur", "state":"RJ", "zip":"176114",
               "location":[26.8535922, 75.7923988], "fees":2500,
               "tags":["Well equipped labs"], "rating":"4.5"
            }
         }
      ]
   }
}
```

Full Text Queries

These queries are used to search a full body of text like a chapter or a news article. This query works according to the analyzer associated with that particular index or document. In this section, we will discuss the different types of full text queries.

Match query

This query matches a text or phrase with the values of one or more fields. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "query":{
      "match" : {
         "city":"pune"
      }
   }
}
```

Response

```
{
   "took":1, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{
      "total":1, "max_score":0.30685282, "hits":[{
         "_index":"schools_gov", "_type":"school", "_id":"2", "_score":0.30685282,
         "_source":{
            "name":"Government School", "description":"State Board Afiliation",
            "street":"Hinjewadi", "city":"Pune", "state":"MH", "zip":"411057",
            "location":[18.599752, 73.6821995], "fees":500,
            "tags":["Great Sports"], "rating":"4"
         }
      }]
   }
}
```

multi_match query

This query matches a text or phrase with more than one field. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "query":{
      "multi_match" : {
         "query": "hyderabad",
         "fields": [ "city", "state" ]
      }
   }
}
```

Response

```
{
   "took":16, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{
      "total":1, "max_score":0.09415865, "hits":[{
         "_index":"schools_gov", "_type":"school", "_id":"1", "_score":0.09415865,
         "_source":{
            "name":"Model School", " description":"CBSE Affiliation",
            "street":"silk city", "city":"Hyderabad", "state":"AP",
            "zip":"500030", "location":[17.3903703, 78.4752129], "fees":700,
            "tags":["Senior Secondary", "beautiful campus"], "rating":"3"
         }
      }]
   }
}
```

Query String Query

This query uses query parser and query_string keyword. For example,

```
POST http://localhost:9200/schools/_search
```

Request Body

```
{
   "query":{
      "query_string":{
         "query":"good faculty"
      }
   }
}
```

Response

```
{
   "took":16, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{
      "total":1, "max_score":0.09492774, "hits":[{
         "_index":"schools", "_type":"school", "_id":"2", "_score":0.09492774,
         "_source":{
            "name":"Saint Paul School", "description":"ICSE Affiliation",
            "street":"Dawarka", "city":"Delhi", "state":"Delhi",
            "zip":"110075", "location":[28.5733056, 77.0122136],
            "fees":5000, "tags":["Good Faculty", "Great Sports"],
            "rating":"4.5"
         }
      }]
   }
}
```

Term Level Queries

These queries mainly deal with structured data like numbers, dates and emuns. For example,

```
POST http://localhost:9200/schools/_search
```

Request Body

```
{
   "query":{
      "term":{"zip":"176115"}
   }
}
```

Response

```
{
   "took":1, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{
      "total":1, "max_score":0.30685282, "hits":[{
         "_index":"schools", "_type":"school", "_id":"1", "_score":0.30685282,
         "_source":{
            "name":"Central School", "description":"CBSE Affiliation",
            "street":"Nagan", "city":"paprola", "state":"HP", "zip":"176115",
            "location":[31.8955385, 76.8380405], "fees":2200,
            "tags":["Senior Secondary", "beautiful campus"], "rating":"3.3"
         }
      }]
   }
}
```

Range Query

This query is used to find the objects having values between the ranges of values. For this, we need to use operators like –

- **gte** – greater than equal to
- **gt** – greater-than
- **lte** – less-than equal to
- **lt** – less-than

For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "query":{
      "range":{
         "rating":{
            "gte":3.5
         }
      }
   }
}
```

Response

```
{
   "took":31, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{
      "total":3, "max_score":1.0, "hits":[
         {
            "_index":"schools", "_type":"school", "_id":"2", "_score":1.0,
            "_source":{
               "name":"Saint Paul School", "description":"ICSE Affiliation",
               "street":"Dawarka", "city":"Delhi", "state":"Delhi",
               "zip":"110075", "location":[28.5733056, 77.0122136], "fees":5000,
               "tags":["Good Faculty", "Great Sports"], "rating":"4.5"
            }
         },

         {
            "_index":"schools_gov", "_type":"school", "_id":"2", "_score":1.0,
            "_source":{
               "name":"Government School", "description":"State Board Affiliation",
               "street":"Hinjewadi", "city":"Pune", "state":"MH", "zip":"411057",
               "location":[18.599752, 73.6821995] "fees":500,
               "tags":["Great Sports"], "rating":"4"
            }
         },

         {
            "_index":"schools", "_type":"school", "_id":"3", "_score":1.0,
            "_source":{
               "name":"Crescent School", "description":"State Board Affiliation",
               "street":"Tonk Road", "city":"Jaipur", "state":"RJ", "zip":"176114",
               "location":[26.8535922, 75.7923988], "fees":2500,
               "tags":["Well equipped labs"], "rating":"4.5"
            }
         }
      ]
   }
}
```

Other types of term level queries are −

- **Exists query** − If a certain field has non null value.

Missing query – This is completely opposite to exists query, this query searches for objects without specific fields or fields having null value.

Wildcard or regexp query – This query uses regular expressions to find patterns in the objects.

Type query – documents with specific type. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "query":{
      "type" : {
         "value" : "school"
      }
   }
}
```

Response

All the school JSON objects present in the specified indices.

Compound Queries

These queries are a collection of different queries merged with each other by using Boolean operators like and, or, not or for different indices or having function calls etc. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "query":{
      "filtered":{
         "query":{
            "match":{
               "state":"UP"
            }
         },

         "filter":{
            "range":{
               "rating":{
                  "gte":4.0
               }
            }
         }
      }
   }
}
```

Response

```
{
   "took":16, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{"total":0, "max_score":null, "hits":[]}
}
```

Joining Queries

These queries are used where more than one mapping or document is included. There are two types of joining queries –

Nested Query

These queries deal with nested mapping (you will read more about it in the next chapter).

has_child and has_parent queries

These queries are used to retrieve child or parent of the document, which got match in the query. For example,

```
POST http://localhost:9200/tutorials/_search
```

Request Body

```
{
   "query":
   {
      "has_child" : {
         "type" : "article", "query" : {
            "match" : {
               "Text" : "This is article 1 of chapter 1"
            }
         }
      }
   }
}
```

Response

```
{
   "took":21, "timed_out":false, "_shards":{"total":5, "successful":5, "failed":0},
   "hits":{
      "total":1, "max_score":1.0, "hits":[{
         "_index":"tutorials", "_type":"chapter", "_id":"1", "_score":1.0,
         "_source":{
            "Text":"this is chapter one"
         }
      }]
   }
}
```

Geo Queries

These queries deal with geo locations and geo points. These queries help to find out schools or any other geographical object near to any location. You need to use geo point data type. For example,

```
POST http://localhost:9200/schools*/_search
```

Request Body

```
{
   "query":{
      "filtered":{
         "filter":{
            "geo_distance":{
               "distance":"100km",
               "location":[32.052098, 76.649294]
            }
         }
      }
   }
}
```

Response

```
{
   "took":6, "timed_out":false, "_shards":{"total":10, "successful":10, "failed":0},
   "hits":{
      "total":2, "max_score":1.0, "hits":[
         {
            "_index":"schools", "_type":"school", "_id":"2", "_score":1.0,
            "_source":{
               "name":"Saint Paul School", "description":"ICSE Affiliation",
               "street":"Dawarka", "city":"Delhi", "state":"Delhi", "zip":"110075",
               "location":[28.5733056, 77.0122136], "fees":5000,
               "tags":["Good Faculty", "Great Sports"], "rating":"4.5"
            }
         },

         {
            "_index":"schools", "_type":"school", "_id":"1", "_score":1.0,
            "_source":{
               "name":"Central School", "description":"CBSE Affiliation",
               "street":"Nagan", "city":"paprola", "state":"HP", "zip":"176115",
               "location":[31.8955385, 76.8380405], "fees":2000,
               "tags":["Senior Secondary", "beautiful campus"], "rating":"3.5"
            }
         }
      ]
   }
}
```

Note – If you get an exception while performing the above example, please add the following mapping to your index.

```
{
   "mappings":{
      "school":{
         "_all":{
            "enabled":true
         },

         "properties":{
            "location":{
               "type":"geo_point"
            }
         }
      }
   }
}
```

Elasticsearch - Mapping

Mapping is the outline of the documents stored in an index. It defines the data type like geo_point or string and format of the fields present in the documents and rules to control the mapping of dynamically added fields. For example,

```
POST http://localhost:9200/bankaccountdetails
```

Request Body

```
{
   "mappings":{
      "report":{
         "_all":{
            "enabled":true
         },

         "properties":{
            "name":{ "type":"string"}, "date":{ "type":"date"},
            "balance":{ "type":"double"}, "liability":{ "type":"double"}
         }
      }
   }
}
```

Response

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

Field Data Types

Elasticsearch supports a number of different datatypes for the fields in a document. The following data types are used to store fields in Elasticsearch –

Core Data Types

These are the basic data types supported by almost all the systems like integer, long, double, short, byte, double, float, string, date, Boolean and binary.

Complex Data Types

These data types are a combination of core data types. Like array, JSON object and nested data type. Following is the example of nested data type –

```
POST http://localhost:9200/tabletennis/team/1
```

Request Body

```
{
   "group" : "players",
   "user" : [
      {
         "first" : "dave", "last" : "jones"
      },

      {
         "first" : "kevin", "last" : "morris"
      }

   ]
}
```

Response

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

Geo Data Types

These data types are used for defining geographic properties. For instance, geo_point is used for defining longitude and latitude, and geo_shape for defining different geometric shapes like rectangle.

Specialized Data Types

These data types are used for special purposes like IPv4 (“ip”) accepts IP address, completion data type is used to support auto-complete suggestions and token_count for counting the number of tokens in a string.

Mapping Types

Each index has one or more mapping types, which are used to divide the documents of an index into logical groups. Mapping can be different from each other on the basis of the following parameters –

Meta-Fields

These fields provide information about the mappings and the other objects associated with it. Like _index, _type, _id, and _source fields.

Fields

Different mapping contains different number of fields and fields with different data types.

Dynamic Mapping

Elasticsearch provides a user-friendly mechanism for the automatic creation of mapping. A user can post the data directly to any undefined mapping and Elasticsearch will automatically create the mapping, which is called dynamic mapping. For example,

```
POST http://localhost:9200/accountdetails/tansferreport
```

Request Body

```
{
   "from_acc":"7056443341", "to_acc":"7032460534",
   "date":"11/1/2016", "amount":10000
}
```

Response

```
{
   "_index":"accountdetails", "_type":"tansferreport",
   "_id":"AVI3FeH0icjGpNBI4ake", "_version":1,
   "_shards":{"total":2, "successful":1, "failed":0},
   "created":true
}
```

Mapping Parameters

The mapping parameters define the structure of mapping, information about fields and about storage and how the mapped data will be analyzed at the time of searching. These are the following mapping parameters −

- analyzer
- boost
- coerce
- copy_to
- doc_values
- dynamic
- enabled
- fielddata
- geohash
- geohash_precision
- geohash_prefix
- format
- ignore_above
- ignore_malformed
- include_in_all
- index_options
- lat_lon
- index
- fields
- norms
- null_value
- position_increment_gap
- properties
- search_analyzer
- similarity
- store
- term_vector

Elasticsearch - Analysis

When a query is processed during a search operation , the content in any index is analyzed by analysis module. This module consists of analyzer, tokenizer, tokenfilters and charfilters. If no analyzer is defined, then by default the built in analyzers, token, filters and tokenizers get registered with analysis module. For example.

```
POST http://localhost:9200/pictures
```

Request Body

```
{
   "settings": {
      "analysis": {
         "analyzer": {
            "index_analyzer": {
               "tokenizer": "standard", "filter": [
                  "standard", "my_delimiter", "lowercase", "stop",
                     "asciifolding", "porter_stem"
               ]
            },

            "search_analyzer": {
               "tokenizer": "standard", "filter": [
                  "standard", "lowercase", "stop", "asciifolding", "porter_stem"
               ]
            }
         },

         "filter": {
            "my_delimiter": {
               "type": "word_delimiter",
               "generate_word_parts": true,
               "catenate_words": true,
               "catenate_numbers": true,
               "catenate_all": true,
               "split_on_case_change": true,
               "preserve_original": true,
               "split_on_numerics": true,
               "stem_english_possessive": true
            }
         }
      }
   }
}
```

Analyzers

An analyzer consists of a tokenizer and optional token filters. These analyzers are registered in analysis module with logical names, which can be referenced either in mapping definitions or in some APIs. There are a number of default analyzers as follows –

Sr.No	Analyzer & Description
1	**Standard analyzer (standard)** stopwords and max_token_length setting can be set for this analyzer. By default, stopwords list is empty and max_token_length is 255.
2	**Simple analyzer (simple)**

	This analyzer is composed of lowercase tokenizer.
3	**Whitespace analyzer (whitespace)** This analyzer is composed of whitespace tokenizer.
4	**Stop analyzer (stop)** stopwords and stopwords_path can be configured. By default stopwords initialized to English stop words and stopwords_path contains path to a text file with stop words.
5	**Keyword analyzer (keyword)** This analyzer tokenizes an entire stream as a single token. It can be used for zip code.
6	**Pattern analyzer (pattern)** This analyzer mainly deals with regular expressions. Settings like lowercase, pattern, flags, stopwords can be set in this analyzer.
7	**Language analyzer** This analyzer deals with languages like hindi, arabic, ducth etc.
8	**Snowball analyzer (snowball)** This analyzer uses the standard tokenizer, with standard filter, lowercase filter, stop filter, and snowball filter.
9	**Custom analyzer (custom)** This analyzer is used to create customized analyzer with a tokenizer with optional token filters and char filters. Settings like tokenizer, filter, char_filter and position_increment_gap can be configured in this analyzer.

Tokenizers

Tokenizers are used for generating tokens from a text in Elasticsearch. Text can be broken down into tokens by taking whitespace or other punctuations into account. Elasticsearch has plenty of built-in tokenizers, which can be used in custom analyzer.

Sr.No	Tokenizer & Description

1	**Standard tokenizer (standard)** This is built on grammar based tokenizer and max_token_length can be configured for this tokenizer.
2	**Edge NGram tokenizer (edgeNGram)** Settings like min_gram, max_gram, token_chars can be set for this tokenizer.
3	**Keyword tokenizer (keyword)** This generates entire input as an output and buffer_size can be set for this.
4	**Letter tokenizer (letter)** This captures the whole word until a non-letter is encountered.
5	**Lowercase tokenizer (lowercase)** This works the same as the letter tokenizer, but after creating tokens, it changes them to lower case.
6	**NGram Tokenizer (nGram)** Settings like min_gram (default value is 1), max_gram (default value is 2) and token_chars can be set for this tokenizer.
7	**Whitespace tokenizer (whitespace)** This divides text on the basis of whitespaces.
8	**Pattern tokenizer (pattern)** This uses regular expressions as a token separator. Pattern, flags and group settings can be set for this tokenizer.
9	**UAX Email URL Tokenizer (uax_url_email)** This works same lie standard tokenizer but it treats email and URL as single token.
10	**Path hierarchy tokenizer (path_hierarchy)** This tokenizer generated all the possible paths present in the input directory path. Settings available for this tokenizer are delimiter (defaults to /),

	replacement, buffer_size (defaults to 1024), reverse (defaults to false) and skip (defaults to 0).
11	**Classic tokenizer (classic)** This works on the basis of grammar based tokens. max_token_length can be set for this tokenizer.
12	**Thai tokenizer (thai)** This tokenizer is used for Thai language and uses built-in Thai segmentation algorithm.

Token Filters

Token filters receive input from tokenizers and then these filters can modify, delete or add text in that input. Elasticsearch offers plenty of built-in token filters. Most of them have already been explained in previous sections.

Character Filters

These filters process the text before tokenizers. Character filters look for special characters or html tags or specified pattern and then either delete then or change them to appropriate words like ‘&’ to and, delete html markup tags. Here is an example of analyzer with synonym specified in synonym.txt –

```
{
   "settings":{
      "index":{
         "analysis":{
            "analyzer":{
               "synonym":{
                  "tokenizer":"whitespace", "filter":["synonym"]
               }
            },

            "filter":{
               "synonym":{
                  "type":"synonym", "synonyms_path":"synonym.txt", "ignore_case":"true"
               }
            }
         }
      }
   }
}
```

Elasticsearch - Modules

Elasticsearch is composed of a number of modules, which are responsible for its functionality. These modules have the following two types of settings −

- **Static Settings** − These settings need to be configured in config (elasticsearch.yml) file before starting Elasticsearch. You need to update all the concern nodes in the cluster to reflect the changes by these settings.
- **Dynamic Settings** − These settings can be set on live Elasticsearch.

We will discuss the different modules of Elasticsearch in the following sections of this chapter.

Cluster-Level Routing and Shard Allocation

Cluster level settings decide the allocation of shards to different nodes and reallocation of shards to rebalance cluster. These are the following settings to control shard allocation −

Cluster-Level Shard Allocation

Setting	Possible value	Description
cluster.routing.allocation.enable	all	This default value allows shard allocation for all kinds of shards.
	primaries	This allows shard allocation only for primary shards.
	new_primaries	This allows shard allocation only for primary shards for new indices.
	none	This does not allow any shard allocations.
cluster.routing.allocation .node_concurrent_recoveries	Numeric value (by default 2)	This restricts the number of concurrent shard recovery.
cluster.routing.allocation .node_initial_primaries_recoveries	Numeric value (by default 4)	This restricts the number of parallel initial primary recoveries.
cluster.routing.allocation .same_shard.host	Boolean value (by default false)	This restricts the allocation of more than one replica of the same shard in the same physical node.
indices.recovery.concurrent _streams	Numeric value (by default 3)	This controls the number of open network streams per node at the time of shard recovery from peer shards.
indices.recovery.concurrent _small_file_streams	Numeric value (by default 2)	This controls the number of open streams per node for small files having

		size less than 5mb at the time of shard recovery.
cluster.routing.rebalance.enable	all	This default value allows balancing for all kinds of shards.
	primaries	This allows shard balancing only for primary shards.
	replicas	This allows shard balancing only for replica shards.
	none	This does not allow any kind of shard balancing.
cluster.routing.allocation .allow_rebalance	always	This default value always allows rebalancing.
	indices_primaries _active	This allows rebalancing when all primary shards in cluster are allocated.
	Indices_all_active	This allows rebalancing when all the primary and replica shards are allocated.
cluster.routing.allocation.cluster _concurrent_rebalance	Numeric value (by default 2)	This restricts the number of concurrent shard balancing in cluster.
cluster.routing.allocation .balance.shard	Float value (by default 0.45f)	This defines the weight factor for shards allocated on every node.
cluster.routing.allocation .balance.index	Float value (by default 0.55f)	This defines the ratio of the number of shards per index allocated on a specific node.
cluster.routing.allocation .balance.threshold	Non negative float value (by default 1.0f)	This is the minimum optimization value of operations that should be performed.

Disk-based Shard Allocation

Setting	Possible value	Description
cluster.routing.allocation .disk.threshold_enabled	Boolean value (by default true)	This enables and disables disk allocation decider.
cluster.routing.allocation .disk.watermark.low	String value (by default 85%)	This denotes maximum usage of disk; after this point, no other shard can be allocated to that disk.

cluster.routing.allocation .disk.watermark.high	String value (by default 90%)	This denotes the maximum usage at the time of allocation; if this point is reached at the time of allocation, then Elasticsearch will allocate that shard to another disk.
cluster.info.update.interval	String value (by default 30s)	This is the interval between disk usages checkups.
cluster.routing.allocation .disk.include_relocations	Boolean value (by default true)	This decides whether to consider the shards currently being allocated, while calculating disk usage.

Discovery

This module helps a cluster to discover and maintain the state of all the nodes in it. The state of cluster changes when a node is added or deleted from a cluster. The cluster name setting is used to create logical difference between different clusters. There are some modules which help you to use the APIs provided by cloud vendors and those are −

- Azure discovery
- EC2 discovery
- Google compute engine discovery
- Zen discovery

Gateway

This module maintains the cluster state and the shard data across full cluster restarts. Following are the static settings of this module −

Setting	Possible value	Description
gateway.expected_ nodes	numeric value (by default 0)	The number of nodes that are expected to be in the cluster for the recovery of local shards.
gateway.expected_ master_nodes	numeric value (by default 0)	The number of master nodes that are expected to be in the cluster before start recovery.
gateway.expected_ data_nodes	numeric value (by default 0)	The number of data nodes expected in the cluster before start recovery.
gateway.recover_	String value	This specifies the time for which the recovery process will

after_time	(by default 5m)	wait to start regardless of the number of nodes joined in the cluster. gateway.recover_ after_nodes gateway.recover_after_ master_nodes gateway.recover_after_ data_nodes

HTTP

This module manages the communication between HTTP client and Elasticsearch APIs. This module can be disabled by changing the value of http.enabled to false. The following are the settings (configured in elasticsearch.yml) to control this module −

Sr.No	Setting & Description
1	**http.port** This is a port to access Elasticsearch and it ranges from 9200-9300.
2	**http.publish_port** This port is for http clients and is also useful in case of firewall.
3	**http.bind_host** This is a host address for http service.
4	**http.publish_host** This is a host address for http client.
5	**http.max_content_length** This is the maximum size of content in an http request. Its default value is 100mb.
6	**http.max_initial_line_length** This is the maximum size of URL and its default value is 4kb.
7	**http.max_header_size** This is the maximum http header size and its default value is 8kb.

8	**http.compression** This enables or disables support for compression and its default value is false.
9	**http.pipelinig** This enables or disables HTTP pipelining.
10	**http.pipelining.max_events** This restricts the number of events to be queued before closing an HTTP request.

Indices

This module maintains the settings, which are set globally for every index. The following settings are mainly related to memory usage −

Circuit Breaker

This is used for preventing operation from causing an OutOfMemroyError. The setting mainly restricts the JVM heap size. For example, indices.breaker.total.limit setting, which defaults to 70% of JVM heap.

Fielddata Cache

This is used mainly when aggregating on a field. It is recommended to have enough memory to allocate it. The amount of memory used for the field data cache can be controlled using indices.fielddata.cache.size setting.

Node Query Cache

This memory is used for caching the query results. This cache uses Least Recently Used (LRU) eviction policy. Indices.queries.cahce.size setting controls the memory size of this cache.

Indexing Buffer

This buffer stores the newly created documents in the index and flushes them when the buffer is full. Setting like indices.memory.index_buffer_size control the amount of heap allocated for this buffer.

Shard Request Cache

This cache is used to store local search data for every shard. Cache can be enabled during the creation of index or can be disabled by sending URL parameter.

```
Disable cache - ?request_cache = true
Enable cache "index.requests.cache.enable": true
```

Indices Recovery

It controls the resources during recovery process. The following are the settings −

Setting	Default value
indices.recovery.concurrent_streams	3
indices.recovery.concurrent_small_file_streams	2
indices.recovery.file_chunk_size	512kb
indices.recovery.translog_ops	1000
indices.recovery.translog_size	512kb
indices.recovery.compress	true
indices.recovery.max_bytes_per_sec	40mb

TTL Interval

Time to Live (TTL) interval defines the time of a document, after which the document gets deleted. The following are the dynamic settings for controlling this process −

Setting	Default value
indices.ttl.interval	60s
indices.ttl.bulk_size	1000

Node

Each node has an option to be data node or not. You can change this property by changing **node.data** setting. Setting the value as **false** defines that the node is not a data node.

Elasticsearch - Testing

Elasticsearch provides a jar file, which can be added to any java IDE and can be used to test the code which is related to Elasticsearch. A range of tests can be performed by using the framework provided by Elasticsearch −

- Unit testing
- Integration testing
- Randomized testing

To start with testing, you need to add the Elasticsearch testing dependency to your program. You can use maven for this purpose and can add the following in pom.xml.

```
<dependency>
   <groupId>org.elasticsearch</groupId>
   <artifactId>elasticsearch</artifactId>
   <version>2.1.0</version>
</dependency>
```

EsSetup has been initialized to start and stop Elasticsearch node and also to create indices.

EsSetup esSetup = new EsSetup();

esSetup.execute() function with createIndex will create the indices, you need to specify the settings, type and data.

Unit Testing

Unit test is carried out by using JUnit and Elasticsearch test framework. Node and indices can be created using Elasticsearch classes and in test method can be used to perform the testing. ESTestCase and ESTokenStreamTestCase classes are used for this testing.

Integration Testing

Integration testing uses multiple nodes in a cluster. ESIntegTestCase class is used for this testing. There are various methods which make the job of preparing a test case easier.

Sr.No	Method & Description
1	**refresh()** All the indices in a cluster are refreshed
2	**ensureGreen()** Ensures a green health cluster state
3	**ensureYellow()** Ensures a yellow health cluster state
4	**createIndex(name)** Create index with the name passed to this method
5	**flush()** All indices in cluster are flushed

6	**flushAndRefresh()**
	
	flush() and refresh()

7	**indexExists(name)** Verifies the existence of specified index
8	**clusterService()** Returns the cluster service java class
9	**cluster()** Returns the test cluster class

Test Cluster Methods

Sr.No	Method & Description
1	**ensureAtLeastNumNodes(n)** Ensures minimum number of nodes up in a cluster is more than or equal to specified number.
2	**ensureAtMostNumNodes(n)** Ensures maximum number of nodes up in a cluster is less than or equal to specified number.
3	**stopRandomNode()** To stop a random node in a cluster
4	**stopCurrentMasterNode()** To stop the master node
5	**stopRandomNonMaster()** To stop a random node in a cluster, which is not a master node
6	**buildNode()** Create a new node

7	**startNode(settings)** Start a new node
8	**nodeSettings()** Override this method for changing node settings

Accessing Clients

Client is used to access different nodes in a cluster and carry out some action. ESIntegTestCase.client() method is used for getting a random client. Elasticsearch offers other methods also to access client and those methods can be accessed using ESIntegTestCase.internalCluster() method.

Sr.No	Method & Description
1	**iterator()** This helps you to access all the available clients.
2	**masterClient()** This returns a client, which is communicating with master node.
3	**nonMasterClient()** This returns a client, which is not communicating with master node.
4	**clientNodeClient()** This returns a client currently up on client node.

Randomized Testing

This testing is used to test the user's code with every possible data, so that there will be no failure in future with any type of data. Random data is the best option to carry out this testing.

Generating Random Data

In this testing, the Random class is instantiated by the instance provided by RandomizedTest and offers many methods for getting different types of data.

Method	Return value

getRandom()	Instance of random class
randomBoolean()	Random boolean
randomByte()	Random byte
randomShort()	Random short
randomInt()	Random integer
randomLong()	Random long
randomFloat()	Random float
randomDouble()	Random double
randomLocale()	Random locale
randomTimeZone()	Random time zone
randomFrom()	Random element from array

Assertions

ElasticsearchAssertions and ElasticsearchGeoAssertions classes contain assertions, which are used for performing some common checks at the time of testing. For example,

```
SearchResponse seearchResponse = client().prepareSearch();
assertHitCount(searchResponse, 6);
assertFirstHit(searchResponse, hasId("6"));
assertSearchHits(searchResponse, "1", "2", "3", "4",”5”,”6”);
```