# Elasticsearch

- Elasticsearch is the distributed search and analytics engine.
- Open source developed in java
- Elasticsearch is based on lucene engine.
- Logstash facilitate collecting your data and storing it in Elasticsearch.
- Kibana enables you to visualize your data.
- Elasticsearch is where the indexing, search, and analysis magic happen.

## Features :

- Elasticsearch provides real-time search and analytics for all types of data. Whether you have structured or unstructured text, numerical data, or geospatial data.
- Scalable, Fast, Multilingual
- Elasticsearch can efficiently store and index it in a way that supports fast searches
- Add a search box to an app or website
- Store and analyze logs, metrics data
- Use machine learning to automatically model the behavior of your data in real time
- Elasticsearch uses a data structure called an inverted index that supports very fast full-text searches. An inverted index lists every unique word that appears in any document and identifies all of the documents each word occurs in.
- Text fields are stored in inverted indices, and numeric and geo fields are stored in BKD trees.

# Scalability

Elasticsearch is built to be always available and to scale with your needs. It does this by being distributed by nature. You can add nodes to a cluster to increase capacity and Elasticsearch automatically distributes your data and query load across all of the available nodes.

- Culster => Combination of nodes
- Indxes => Combination of multiple shards
- Shards => Combination of multiple nodes
- Where each shard is actually a self-contained index
- By distributing the documents in an index across multiple shards, and distributing those shards across multiple nodes, Elasticsearch can ensure redundancy, which both protects against hardware failures
- There are two types of shards:
    1. Primary or Master => fixed in number

2. Replicas or Slave => Number can be changed later

Elastic Search API's

- Document API
  - Single Document API
    - Index API
    - GET API
    - UPDATE API
    - DELETE API
  - Multi-Document API
    - Multi GET API
    - Bulk API
    - Update ByQuery
    - Delete By Query
    - Reindex API
- Search API
  - Multi Index
  - Multi Type
  - URI Search

    q, timeout, lenient , terminate_after, fields, from, sort, size
- Aggregation
  - Bucket
  - Metric
  - Matrix
  - Pipeline
- Index API

  Responsible for managing all aspect of index like settings, aliases, mapping, index template.

  Create Index, Delete Index, GET Index, Index Exists, Open/Close Index API, Index Alies, Index Settings, Analyze, Index Template, Index Stats, Flush, Refresh
- Cluster API

  Cluster API is used for getting information about it's cluster and it's nodes and making changes in them.

  Health, State, Stats, Pending Task, Reroute, Nodes hot_threads

## Query DSL

## Index some documents

```
PUT /customer/_doc/1
{
  "name": "John Doe"
}
```

This request automatically creates the customer index if it doesn't already exist, adds a new document that has an ID of 1, and stores and indexes the name field.

Since this is a new document, the response shows that the result of the operation was that version 1 of the document was created:

```
{
  "_index" : "customer",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 2,
    "failed" : 0
  },
  "_seq_no" : 26,
  "_primary_term" : 4
}
```

```
GET /customer/_doc/1

{

  "_index" : "customer",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "_seq_no" : 26,
  "_primary_term" : 4,
  "found" : true,
  "_source" : {
    "name": "John Doe"
  }
}
```

**** If you have a lot of documents to index, you can submit them in batches with the bulk api.

## Searching

```
GET /bank/_search
{
  "query": { "match_all": {} },
  "sort": [
    { "account_number": "asc" }
  ]
}
```

By default, the hits section of the response includes the first 10 documents that match the search criteria:

- took – how long it took Elasticsearch to run the query, in milliseconds
- timed_out – whether or not the search request timed out
- _shards – how many shards were searched and a breakdown of how many shards succeeded, failed, or were skipped.
- max_score – the score of the most relevant document found
- hits.total.value - how many matching documents were found
- hits.sort - the document's sort position (when not sorting by relevance score)
- hits._score - the document's relevance score (not applicable when using match_all)

Each search request is self-contained: Elasticsearch does not maintain any state information across requests. To page through the search hits, specify the from and size parameters in your request.

```
GET /bank/_search
{
  "query": { "match_all": {} },
  "from": 10,
  "size": 10
}
```

```
{
  "took" : 63,
  "timed_out" : false,
```

```
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 1000,
    "max_score": null,
    "hits" : [ {
      "_index" : "bank",
      "_type" : "_doc",
      "_id" : "0",
      "sort": [0],
      "_score" : null,
      "_source" :
{"account_number":0,"balance":16623,"firstname":"Bradshaw","lastname":"Mckenzie","age
":29,"gender":"F","address":"244 Columbus
Place","employer":"Euron","email":"bradshawmckenzie@euron.com","city":"Hobucken","st
ate":"CO"}
    }, ...
    ]
  }
}
```


To search for specific terms within a field, you can use a match query. For example, the following request searches the address field to find customers whose addresses contain mill or lane:

GET /bank/_search

```
{
  "query": { "match": { "address": "mill lane" } }
}
```

For example, the following request only matches addresses that contain the phrase mill lane:

```
GET /bank/_search
{
  "query": { "match_phrase": { "address": "mill lane" } }
}
```

To construct more complex queries, you can use a bool query to combine multiple query criteria. You can designate criteria as required (must match), desirable (should match), or undesirable (must not match).

For example, the following request searches the bank index for accounts that belong to customers who are 40 years old, but excludes anyone who lives in Idaho (ID):

```
GET /bank/_search

{
  "query": {
    "bool": {
      "must": [
        { "match": { "age": "40" } }
      ],
      "must_not": [
        { "match": { "state": "ID" } }
      ]
    }
  }
}
```

Each must, should, and must_not element in a Boolean query is referred to as a query clause. How well a document meets the criteria in each must or should clause contributes to the document's *relevance score*. The higher the score, the better the document matches your search criteria. By default, Elasticsearch returns documents ranked by these relevance scores.

The criteria in a must_not clause is treated as a *filter*. It affects whether or not the document is included in the results, but does not contribute to how documents are scored. You can also explicitly specify arbitrary filters to include or exclude documents based on structured data.

For example, the following request uses a range filter to limit the results to accounts with a balance between $20,000 and $30,000 (inclusive).

```
GET /bank/_search

{
  "query": {
    "bool": {
      "must": { "match_all": {} },
      "filter": {
        "range": {
          "balance": {
            "gte": 20000,
            "lte": 30000
          }
        }
      }
    }
  }
}
```

## Aggregations

For example, the following request uses a terms aggregation to group all of the accounts in the bank index by state, and returns the ten states with the most accounts in descending order:

```
GET /bank/_search
{
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": {
        "field": "state.keyword"
      }
    }
  }
}
```

The buckets in the response are the values of the state field. The doc_count shows the number of accounts in each state. For example, you can see that there are 27 accounts in ID (Idaho). Because the request set size=0, the response only contains the aggregation results.


```
{

  "took": 29,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped" : 0,
    "failed": 0
  },
  "hits" : {
     "total" : 1000,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "group_by_state" : {
      "doc_count_error_upper_bound": 20,
      "sum_other_doc_count": 770,
      "buckets" : [ {
        "key" : "ID",
        "doc_count" : 27
      }, {
        "key" : "TX",
        "doc_count" : 27
      }, {
        "key" : "AL",
        "doc_count" : 25
      }, {
        "key" : "NC",
        "doc_count" : 21
      }, {
        "key" : "ND",
        "doc_count" : 21
      }, {
        "key" : "ME",
        "doc_count" : 20
      }, {
        "key" : "MO",
        "doc_count" : 20
      } ]
    }
  }
}
```

### Multiple Indices

Most APIs that refer to an `index` parameter support execution across multiple indices, using simple `test1,test2,test3` notation (or `_all` for all indices). It also supports wildcards, for example: `test*` or `*test` or `te*t` or `*test*`, and the ability to "exclude" (`-`), for example: `test*,-test3`.

All multi indices APIs support the following url query string parameters:

- `ignore_unavailable`
- `allow_no_indices`
- `expand_wildcards`

## Common options

- `?pretty=true`
- `?human=false`
- `filter_path`

```
GET /_search?
q=elasticsearch&filter_path=took,hits.hits._id,hits.hits._s
core

{

   "took" : 3,
   "hits" : {
         "hits" : [
               {
                     "_id" : "0",
                     "_score" : 1.6375021
               }
         ]
   }
}
```

- `Flat Settings`

```
GET twitter/_settings?flat_settings=true

{

      "twitter" : {
            "settings": {
                  "index.number_of_replicas": "1",
                  "index.number_of_shards": "1",
                  "index.creation_date": "1474389951325",
                  "index.uuid": "n6gzFZTgS664GUfx0Xrpjw",
```

```
                        "index.version.created": ...,
                        "index.provided_name" : "twitter"
                    }
                }
            }
```

# Single document APIs

### *Index API*

The index API adds or updates a typed JSON document in a specific index.

**PUT twitter/_doc/1**

```
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

**When create is used, the index operation will fail if a document by that id already exists in the index**

**PUT twitter/_doc/1?op_type=create**

```
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

### *Get API*

**GET twitter/_doc/0**

**{**

```
    "_index" : "twitter",
    "_type" : "_doc",
    "_id" : "0",
    "_version" : 1,
    "_seq_no" : 10,
    "_primary_term" : 1,
    "found": true,
    "_source" : {
        "user" : "kimchy",
        "date" : "2009-11-15T14:12:12",
        "likes": 0,
        "message" : "trying out Elasticsearch"
    }
}
```

****The API also check for the existence of a document using `HEAD`, for example:**

**HEAD twitter/_doc/0**

*****Use the `/{index}/{type}/{id}/_source` endpoint to get just the `_source` field of the document, without any additional content around it. For example:**

```
GET twitter/_doc/1/_source
```

### *Delete API*

```
DELETE /twitter/_doc/1
```

Delete By Query API

The simplest usage of `_delete_by_query` just performs a deletion on every document that matches a query. Here is the API:

```
POST twitter/_delete_by_query
{
  "query": {
    "match": {
      "message": "some message"
    }
  }
}
```

### *Update API*

```
PUT test/_doc/1

{
    "counter" : 1,
    "tags" : ["red"]
}
```

## Scripted updates

Now, we can execute a script that would increment the counter:

```
POST test/_doc/1/_update
{
    "script" : {
        "source": "ctx._source.counter += params.count",
        "lang": "painless",
        "params" : {
            "count" : 4
        }
    }
}
```

# Multi-document APIs

- *Multi Get API*
- *Bulk API*
- *Delete By Query API*
- *Update By Query API*

```
POST twitter/_update_by_query?conflicts=proceed
```

- *Reindex API*

## Optimistic concurrency control

Elasticsearch is distributed. When documents are created, updated, or deleted, the new version of the document has to be replicated to other nodes in the cluster. Elasticsearch needs a way of ensuring that an older version of a document never overwrites a newer version.

To ensure an older version of a document doesn’t overwrite a newer version, every operation performed to a document is assigned a sequence number by the primary shard.

## Search API

### `Routing`