INDEX

Elasticsearch Documentation

1. Elasticsearch

- Elasticsearch is a **document** oriented search engine.
- It is built on a search software 'Lucene'.
- Underlying data structure in Lucene is known as **Inverted Index.**

2. Inverted Index

- Maps words to the actual document locations of where they occur.
- eg:

Doc 1 : A brown fox jumps.
Doc 2 : The fox quickly jumps over dog.

Tokens/Terms	Doc 1	Doc 2
A	X	
brown	X	
fox	X	X
jumps	X	X
The		X
quickly		X
over		X
dog		X

.

3. How data is stored inside elasticsearch?

3.1. Document

- It is smallest unit of data that can be indexed(stored) in elasticsearch.
- It is expressed as a JSON file.
- A JSON file consists of Key(field)-Value pairs.
- Structure of JSON file :-
  eg:

```
{
        “total” : 2,
        “Employees” : [
                {
                        “name” : “David”,
                        “id” : 1101,
                        “Hiredate” : “October 3, 1998”
                },
                {
                        “name” : “Tom”,
                        “id” : 1121,
                        “Hiredate” : “May  8, 2004”
                }
        ]
}
```

3.2. Index

- It is a collection of different types of documents and their properties.

3.3. Shards

- 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 a fully-functional and independent "index" that can be hosted on any node in the cluster.
- Shards are of two types - Primary and Replica.

3.3.1. Primary Shards

- Primary shard is the original horizontal part of an index and then these primary shards are replicated into replica shards.

3.3.2. Replica Shards

- Elasticsearch allows a user to create replicas of their primary 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.
- Replicas and their corresponding primary shards are not stored in the same node. This helps in recovery of data in case of node failure.

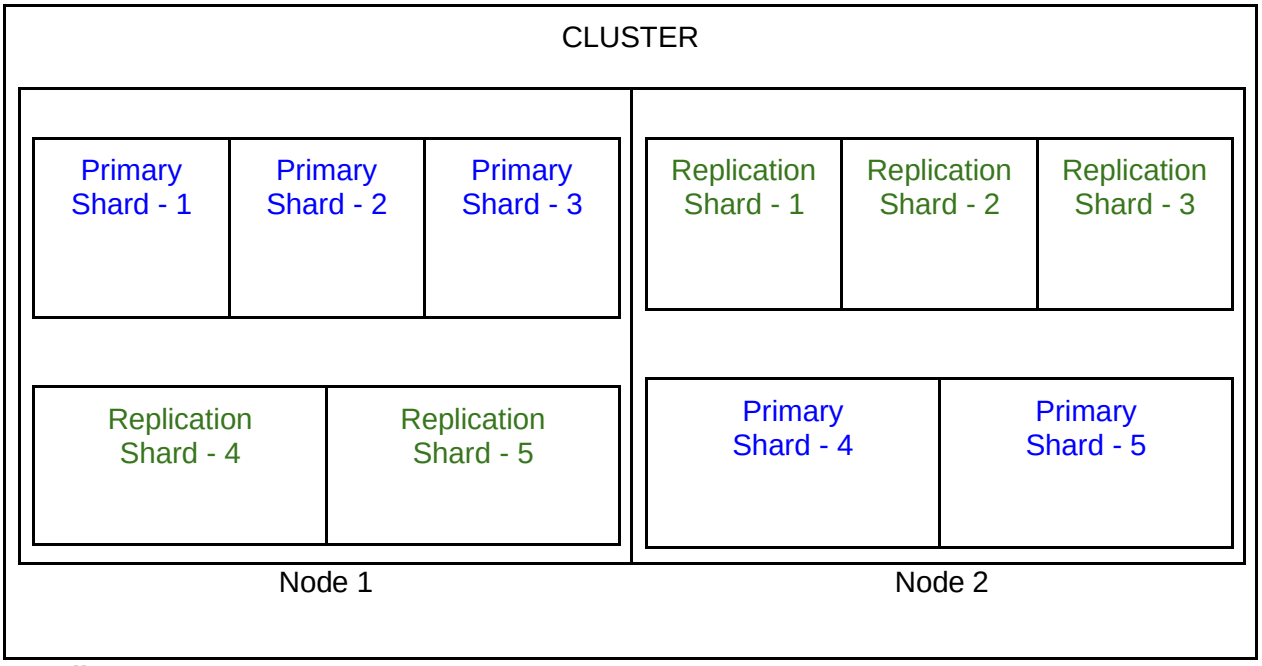
3.4. Nodes

- It refers to a single running instance of Elasticsearch.
- A node is a single server that is part of your cluster, stores your data, and participates in the cluster's indexing and search capabilities.
- A node can have multiple primary shards and replica shards.
- By default, each index in Elasticsearch is allocated 5 primary shards and 1 replica which means that if you have at least two nodes in your cluster, your index will have 5 primary shards and another 5 replica shards (1 complete replica) for a total of 10 shards per index.
- if there are no other Elasticsearch nodes currently running on your network, starting a single node will by default form a new single-node cluster named *elasticsearch*.

3.5. Cluster

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

Fig : Architecture of Elasticsearch



4. Indices API

4.1. Create Index

- The following command creates the index named "customer" using the PUT verb.

```
curl -X PUT "localhost:9200/customer"
```

4.2. Delete Index

- The following command deletes the index named "customer" using the DELETE verb.

```
curl -X DELETE "localhost:9200/customer"
```

4.3. Get Index

- The following command fetched the index details such as mappings, settings and aliases.

```
curl -XGET "http://localhost:9200/customer"
```

4.4. Index exist

- Used to check if the index (indices) exists or not. For example :

```
curl -XHEAD "http://localhost:9200/customer"
```

- The HTTP status code indicates if the index exists or not. A *404* means it does not exist, and *200* means it does.

4.5. Index Settings

- Each index created can have specific settings associated with it, defined in the body.

```
curl -X PUT "localhost:9200/customer" -H 'Content-Type: application/json' -d'
{    "settings" : {
        "index" : {
            "number_of_shards" : 3,
            "number_of_replicas" : 2
        }
    }
}'
```

    - Default for number_of_shards is 5.
    - Default for number_of_replicas is 1 (ie one replica for each primary shard)

4.5. Get Settings

- The get settings API allows to retrieve settings of index/indices:

```
curl -X GET "localhost:9200/customer/_settings"
```

4.6. Index Mappings

- The PUT mapping API allows you to add fields to an existing index or to change search only settings of existing fields.

```
curl -X PUT "localhost:9200/customer/_mapping/_doc" -H 'Content-Type: application/json' -d'
{
  "properties": {
    "email": {
      "type": "keyword"
    },
    "age": {
      "type": "integer"
    }
  }
}'
```

5. Mapping

5.1. Field Data-types

5.1.1 Numeric Data-types

- *integer*
- *long*
- *short*
- *byte*
- *double*
- *float*
- *half_float*
- *scaled_float*

5.1.2 String Data-types

- *text*
- *keyword*

5.1.3 Range Data-types

- *integer_range*
- *float_range*
- *long_range*
- *double_range*
- *date_range*

5.1.3 Other Data-types

- *date*
- *boolean*
- *binary*
- Array : Array support does not require a dedicated type.
- *object* : for single JSON object.
- *nested* : for arrays of JSON objects
- *geo_point* : for lat/lon points
- *geo_shape* : for complex shapes like polygons
- *ip* : for IPv4 and IPv6 addresses
- *completion :* to provide auto-complete suggestions
- *token_count*
- *murmur3*
- *mapper-murmur3*
- *join*
- *alias*
- *percolator*

5.2. 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.
- Format :

```
PUT index_name/_doc/_id
{
    "field_name": “value”
}
```

- eg :

```
PUT data/_doc/1
{
        "firstname": "Peter",
        "lastname": "Parkar",
        "age": 21,
        "encryption_key": "ecaescEWDEecwWEreW531",
        "experience": {
          "gte": 3,
          "lte": 5
        },
        "hire_date": "2018-08-29 12:20:57",
        "ip_address": "127.0.0.0",
        "isRegular": true,
        "location": [
          42.5687,
          84.4511
        ],
        "manager": {
          "age": 31,
          "name": {
            "first": "Tony",
            "last": "Stark"
          }
        },
        "position": "Trainee",
        "salary": 15000.5,
        "tags": [
          "Genius",
          "Lazy"
        ]
 }
```

6. Query DSL (Domain Specific Language)

- In Elasticsearch, searching is carried out by using query based on JSON.

6.1. Match all query

- matches all documents, giving them all a _score of 1.0.
- To fetch all documents in all indices:

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

- To fetch all documents in particular index:

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

6.2. Match query

- matches a text or phrase with the values of one field.
- Eg:

```
GET /_search
{
   "query":{
      "match" : {
         "city":"pune"
      }
   }
}
```

6.3. Multi Match query

- matches a text or phrase with the values of one or more fields.
- Eg:

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

6.4. Match None query

- matches no documents.
- Eg:

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

6.5. Match Phrase query

- analyzes the text and creates a phrase query out of the analyzed text.
- Eg:

```
GET /_search
{
    "query": {
        "match_phrase" : {
            "message" : "this is a test"
        }
    }
}
```

6.6. Term query (Deprecated in 7.3.0)

- The term query finds documents that contain the exact term (without analyzing), specified in the inverted index.
- Eg:

```
POST _search
{
  "query": {
    "term" : { "user" : "Kimchy" }
  }
}
```

6.7. Range query

- Matches documents with fields that have terms within a certain range..
- Eg:

```
GET _search
{
    "query": {
        "range" : {
            "age" : {
                "gte" : 10,
                "lte" : 20
            }
        }
    }
}
```

6.8. Fuzzy query

- Autocomplete and autocorrect facility.
- Eg:

```
GET /_search
{
    "query": {
        "fuzzy" : { "user" : "ki" }
    }
}
```

6.9. Wildcard query

- Matches documents that have fields matching a wildcard expression.
- Supported wildcards are :
    - * : matches any character sequence (including the empty one),
    - ? : matches any single character.
- Eg:

```
GET /_search
{
   "query": {
      "wildcard" : { "user" : "ki*y" }
   }
}
```

6.10. Bool query

- matches documents using boolean combinations.
    - And : must
    - Or : should
    - Not : must_not
- Eg:

```
GET /bank/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "address": "mill" } },
        { "match": { "address": "lane" } }
      ]
    }
  }
}
```

```
GET /bank/_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "address": "mill" } },
        { "match": { "address": "lane" } }
      ]
    }
  }}
```

6.11. Nested query

- allows to query nested documents .
- Eg:

```
PUT /my_index
{
   "mappings": {
      "type1" : {
         "properties" : {
            "obj1" : {
               "type" : "nested"
            }
         }
      }
   }
}
```

6.12. Has Child query

- The has_child filter accepts a query and the child type to run against, and results in parent documents that have child docs matching the query
- Eg:

```
GET /_search
{
   "query": {
      "has_child" : {
         "type" : "blog_tag",
         "query" : {
            "term" : {
               "tag" : "something"
            }
         }
      }
   }
}
```

6.13. Has Parent query

- This query returns child documents which associated parents have matched.
- Eg:

```
GET /_search
{
    "query": {
        "has_parent" : {
            "parent_type" : "blog",
            "query" : {
                "term" : {
                    "tag" : "something"
                }
            }
        }
    }
}
```

6.14. Geo Shape query

- Filter documents indexed using the geo_shape type.
- Eg:

```
GET /example/_search
{
    "query":{
        "bool": {
            "must": {
                "match_all": {}
            },
            "filter": {
                "geo_shape": {
                    "location": {
                        "shape": {
                            "type": "envelope",
                            "coordinates" : [[13.0, 53.0], [14.0, 52.0]]
                        },
                        "relation": "within"
                    }
                }
            }
        }
    }
}
```

6.15. Geo Bounding box query

- A query allowing to filter hits based on a point location using a bounding box.
- Eg:

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "pin.location" : {
                        "top_left" : {
                            "lat" : 40.73,
                            "lon" : -74.1
                        },
                        "bottom_right" : {
                            "lat" : 40.01,
                            "lon" : -71.12
                        }
                    }
                }
            }
        }
    }
}
```

6.16. Geo Distance query

- These queries help to find out schools or any other geographical object near to any location.
- Eg:

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

6.17. Geo Polygon query

- A query allowing to include hits that only fall within a polygon of points.

- Eg:

```
GET /_search
{
   "query": {
      "bool" : {
         "must" : {
            "match_all" : {}
         },
         "filter" : {
            "geo_polygon" : {
               "person.location" : {
                  "points" : [
                  {"lat" : 40, "lon" : -70},
                  {"lat" : 30, "lon" : -80},
                  {"lat" : 20, "lon" : -90}
                  ]
               }
            }
         }
      }
   }
}
```

6.18. Query String query

- A query that uses a query parser in order to parse its content.
- Eg:

```
GET /_search
{
   "query": {
      "query_string" : {
         "default_field" : "content",
         "query" : "this AND that OR thus"
      }
   }
}
```

- It supports field name prefixes, wildcard characters, or other "advanced" features.

7. Aggregations

- The aggregations framework helps provide aggregated data based on a search query.

7.1. Metric Aggregations

- Avg Aggregation

```
POST /exams/_search?size=0
{
   "aggs" : {
      "avg_grade" : {
          "avg" : {
            "field" : "grade"
          }
      }
   }
}
```

- Weighted Avg Aggregation

```
POST /exams/_search
{
   "size": 0,
   "aggs" : {
      "weighted_grade": {
         "weighted_avg": {
            "value": {
               "field": "grade"
            },
            "weight": {
               "field": "weight"
            }
         }
      }
   }
}
```

- Cardinality Aggregation

```
POST                                                    /sales/_search?size=0
{
   "aggs"                                  :                                {
      "type_count"                              :                           {
         "cardinality"                           :                          {
            "field"                          :                        "type"
         }
      }
   }
}
```

- Extended Stats Aggregation

```
GET                                                     /exams/_search
{
   "size":                                                              0,
   "aggs"                                  :                            {
      "grades_stats"   :   {   "extended_stats"   :   {   "field"   :   "grade"   }   }
   }
}
```

- Geo Bounds Aggregation

```
POST                                                    /museums/_search?size=0
{
   "query"                                 :                            {
      "match"          :          {          "name"          :          "musée"          }
   },
   "aggs"                                  :                            {
      "viewport"                                :                       {
         "geo_bounds"                              :                    {
            "field"                          :                  "location",
            "wrap_longitude"                          :                 true
         }
      }
   }
}
```

- Geo Centroid Aggregation

```
PUT /museums
{
  "mappings": {
    "doc": {
      "properties": {
        "location": {
          "type": "geo_point"
        }
      }
    }
  }
}
```

- Max Aggregation

```
POST /sales/_search?size=0
{
  "aggs" : {
    "max_price" : {
      "max" : {
        "field" : "price"
      }
    }
  }
}
```

- Min Aggregation

```
POST /sales/_search?size=0
{
  "aggs" : {
    "max_price" : {
      "max" : {
        "field" : "price"
      }
    }
  }
}
```

- Percentiles Aggregation

```
GET latency/_search
{
  "size": 0,
  "aggs" : {
    "load_time_outlier" : {
      "percentiles" : {
        "field" : "load_time"
      }
    }
  }
}
```

- Percentile Ranks Aggregation

```
GET latency/_search
{
  "size": 0,
  "aggs" : {
    "load_time_ranks" : {
      "percentile_ranks" : {
        "field" : "load_time",
        "values" : [500, 600]
      }
    }
  }
}
```

- Stats Aggregation

```
POST /exams/_search?size=0
{
  "aggs" : {
    "grades_stats" : { "stats" : { "field" : "grade" } }
  }
}
```

- Sum Aggregation

```
POST /sales/_search?size=0
{
  "query" : {
    "constant_score" : {
      "filter" : {
        "match" : { "type" : "hat" }
      }
    }
  },
  "aggs" : {
    "hat_prices" : { "sum" : { "field" : "price" } }
  }
}
```

- Value Count Aggregation

```
POST /sales/_search?size=0
{
  "aggs" : {
    "types_count" : { "value_count" : { "field" : "type" } }
  }
}
```

8. Index Aliases API

- The index aliases API allows aliasing an index with a name, with all APIs automatically converting the alias name to the actual index name.

- *Eg. Associating the alias alias1 with index test1:*

```
POST                                                                    /_aliases
{
   "actions"                                                :                    [
      { "add" : {
          "index" : "test1",
          "alias" : "alias1"
          }
       }
   ]
}
```

- *Eg. Removing the alias alias1 with index test1:*

```
POST                                                                    /_aliases
{
   "actions"                                                :                    [
      { "remove" : {
          "index" : "test1",
         "alias" : "alias1"
          }
      }
   ]
}
```

- *Eg. Rename an alias by doing 'remove' then 'add' operation within the same API :*

```
POST                                                                    /_aliases
{
   "actions"                                                :                    [
      { "remove" :
         { "index" : "test1", "alias" : "alias1" }
      },
      { "add" :
         { "index" : "test2", "alias" : "alias1" }
      }
   ]
}
```

- Eg. Associating an alias to more than one index :

```
POST /_aliases
{
  "actions" : [
    { "add" : { "index" : "test1", "alias" : "alias1" } },
    { "add" : { "index" : "test2", "alias" : "alias1" } }
  ]
}
```

OR

```
POST /_aliases
{
  "actions" : [
    { "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } }
  ]
}
```

- Eg. Associating an alias to more than one index that shares a common name:

```
POST /_aliases
{
  "actions" : [
    { "add" : { "index" : "test*", "alias" : "all_test_indices" } }
  ]
}
```

- Eg. Swap an index/ rename Index:

```
PUT test_2
POST /_aliases
{
  "actions" : [
    { "add": { "index": "test_2", "alias": "test" } },
    { "remove_index": { "index": "test" } }
  ]
}
```