ElasticSearch Query:

if we need to search all the documents with a name that contains central

```
GET /_all/_search?q=city:paprola
```

Search query using POST:

```
POST /schools/_search
{
  "query":{
    "query_string":{
      "query":"up"
    }
  }
}
```

Avg Aggregation

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

```
POST /schools/_search
{
  "aggs":{
    "avg_fees":{"avg":{"field":"fees"}}
  }
}
```

Create Index:

```
PUT colleges
{
  "settings" : {
     "index" : {
        "number_of_shards" : 3,
        "number_of_replicas" : 2
     }
  }
}
```

```
Delete index:
DELETE /colleges
```

Index Settings:

```
GET /colleges/_settings
```

Count

The count parameter provides the count of total number of documents in the entire cluster.

```
GET /_cat/count?v
```

Cluster State

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

```
GET /_cluster/state
```

Cluster Stats

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

```
GET /_cluster/stats
```

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.

```
GET /_nodes/stats
```

Match All Query

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

```
POST /schools/_search
{
  "query":{
    "match_all":{}
  }
}
```

Match query

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

```
POST /schools*/_search
{
  "query":{
    "match" : {
      "rating":"4.5"
    }
  }
}
```

Multi Match Query

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

```
POST /schools*/_search
{
  "query":{
    "multi_match" : {
      "query": "paprola",
      "fields": [ "city", "state" ]
    }
  }
}
```

Query String Query

This query uses query parser and query_string keyword.

```
POST /schools*/_search
{
  "query":{
    "query_string":{
      "query":"beautiful"
    }
  }
}
```

Term Level Queries

These queries mainly deal with structured data like numbers, dates and enums.

```
POST /schools*/_search
{
  "query":{
    "term":{"zip":"176115"}
  }
}
```

Range Query

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

gte – greater than equal to

gt – greater-than

lte – less-than equal to

lt – less-than

For example, observe the code given below –

```
POST /schools*/_search
{
  "query":{
    "range":{
      "rating":{
        "gte":3.5
      }
    }
  }
}
```

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.

```
POST /schools/_search
{
  "query": {
    "bool" : {
      "must" : {
        "term" : { "state" : "UP" }
      },
      "filter": {
        "term" : { "fees" : "2200" }
      },
      "minimum_should_match" : 1,
      "boost" : 1.0
    }
  }
}
```

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.

```
PUT /geo_example
{
  "mappings": {
    "properties": {
      "location": {
        "type": "geo_shape"
      }
    }
  }
}
```

On running the above code, we get the response as shown below –

```
{  "acknowledged" : true,
   "shards_acknowledged" : true,
   "index" : "geo_example"
}
```

Now we post the data in the index created above.

```
POST /geo_example/_doc?refresh
{
   "name": "Chapter One, London, UK",
   "location": {
      "type": "point",
      "coordinates": [11.660544, 57.800286]
   }
}
```

Alias:

Let's add a new alias in our index, let's use the name

"movies_idx_alias".

```
POST _aliases

{
 "actions": [
   {
     "add": {
       "index": "idx_movies",
       "alias": "movies_idx_alias"
     }
   }
 ]
}
```

```
GET idx_movies/_alias
```

Note that the alias was created:

```
{
 "idx_movies" : {
   "aliases" : {
     "movies_idx_alias" : { }
   }
 }
}
```

If I want to add another alias for the same index, just run the “add” command again and the result will be as follows:

```
{
 "idx_movies" : {
   "aliases" : {
     "movies_idx_2020_alias" : { },

     "movies_idx_alias" : { }

   }

 }

}
```

We can add alias and also remove. To remove an alias, run the command below:

```
POST _aliases

{

 "actions": [

   {

     "remove": {
       "index": "idx_movies",
       "alias": "movies_idx_2020_alias"
     }
   }
 ]
}
```

If you want to perform more than one Alias operation, you can run the command as follows:

```
POST _aliases
{
 "actions": [
   {
     "remove": {
       "index": "idx_cars",
       "alias": "cars_latest_alias"
     }
   },
   {
     "add": {
       "index": "idx_movies",
       "alias": "movies_idx_2020_alias"
     }
   }
 ]
```

```
}
```

Filtered Alias

Filtered Alias is a way to add a filter to the alias creating a kind of "view". When this alias is queried, it will run the filter that was created.

```
POST _aliases

{

 "actions": [

   {

       "add": {

         "index": "idx_movies",

         "alias": "get_all_movies_action",

         "filter": {
          "term": {
            "genre": "action"
          }
        }
      }
    }
  ]
}
```

To test your filtered alias, run the command below:

```
GET get_all_movies_action/_search
```

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.

```
PUT bankaccountdetails
{
  "mappings":{
    "properties":{
      "name": { "type":"text"}, "date":{ "type":"date"},
      "balance":{ "type":"double"}, "liability":{ "type":"double"}
    }
  }
 }
```

Ingest Node:

Sometimes we need to transform a document before we index it. For instance, we want to remove a field from the document or rename a field and then index it. This is handled by Ingest node.

Every node in the cluster has the ability to ingest but it can also be customized to be processed only by specific nodes.

Steps Involved

There are two steps involved in the working of the ingest node –

Creating a pipeline

Creating a doc

Create a Pipeline

First creating a pipeline which contains the processors and then executing the pipeline, as shown below –

```
PUT _ingest/pipeline/int-converter
{
  "description": "converts the content of the seq field to an integer",
  "processors" : [
    {
      "convert" : {
        "field" : "seq",
        "type": "integer"
      }
    }
  ]
}
```

Next We Create a Doc

Next we create a document using the pipeline converter.

```
PUT /logs/_doc/1?pipeline=int-converter
{
  "seq":"21",
  "name":"Tutorialspoint",
  "Addrs":"Hyderabad"
}
```

Next we search for the doc created above by using the GET command as shown below

```
GET /logs/_doc/1
```

Index API - Document

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

```
PUT schools/_doc/5
{
  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"
}
```

Managing the index lifecycle involves performing management actions based on factors like shard size and performance requirements. The index lifecycle management (ILM) APIs enable you to automate how you want to manage your indices over time.

This chapter gives a list of ILM APIs and their usage.

Policy Management APIs

API Name	Purpose	Example
Create lifecycle policy.	Creates a lifecycle policy. If the specified policy exists, the policy is replaced and the policy version is incremented.	PUT_ilm/policy/policy_id
Get lifecycle policy.	Returns the specified policy definition. Includes the policy version and last modified date. If no policy is specified, returns all defined policies.	GET_ilm/policy/policy_id
Delete lifecycle policy	Deletes the specified lifecycle policy definition. You cannot delete policies that are currently in use. If the policy is being used to manage any indices, the request fails and returns an error.	DELETE_ilm/policy/policy_id

Index Management APIs

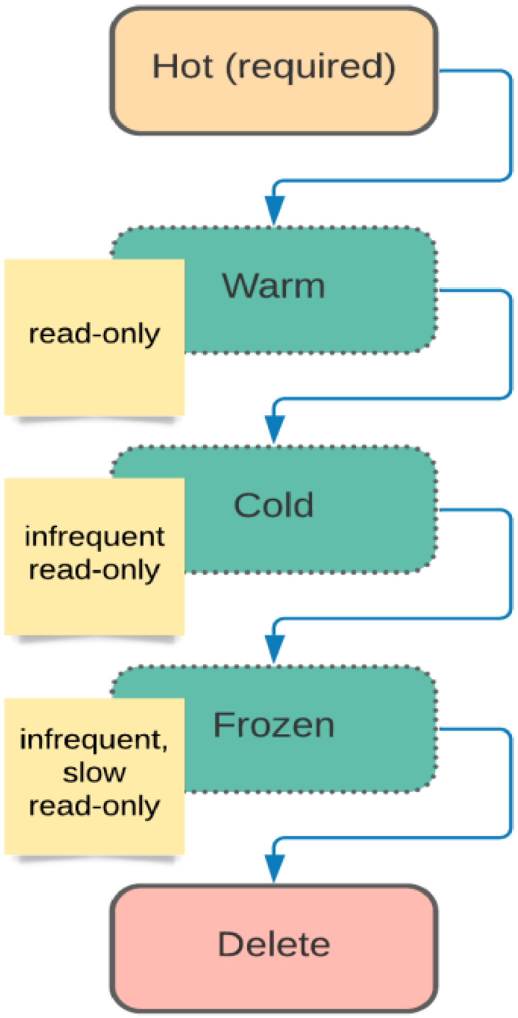
API Name	Purpose	Example
Move to lifecycle step API.	Manually moves an index into the specified step and executes that step.	POST_ilm/move/index

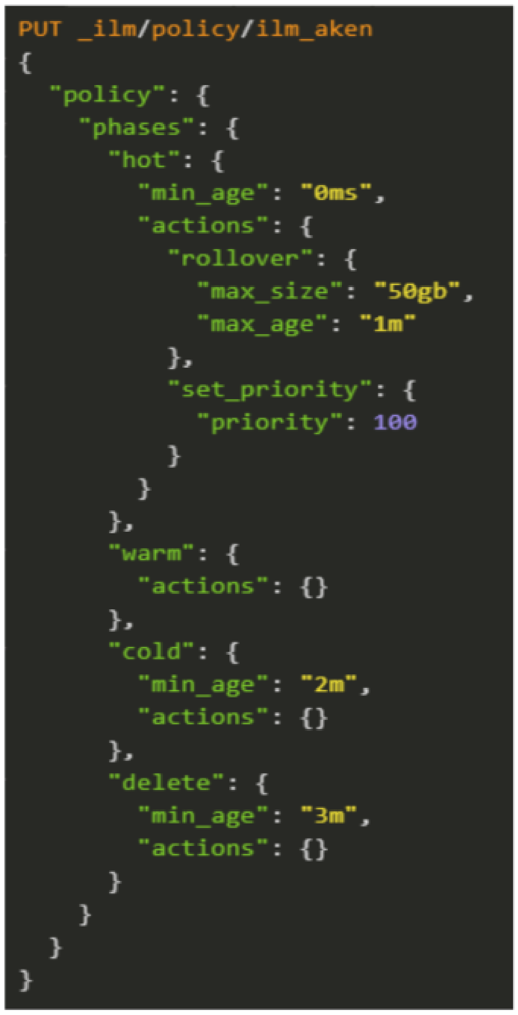
Retry policy.	Sets the policy back to the step where the error occurred and executes the step.	POST index/_ilm/retry
Remove policy from index API edit.	Removes the assigned lifecycle policy and stops managing the specified index. If an index pattern is specified, removes the assigned policies from all matching indices.	POST index/_ilm/remove

Operation Management APIs

API Name	Purpose	Example
Get index lifecycle management status API.	Returns the status of the ILM plugin. The operation_mode field in the response shows one of three states: STARTED, STOPPING, or STOPPED.	GET /_ilm/status
Start index lifecycle management API.	Starts the ILM plugin if it is currently stopped. ILM is started automatically when the cluster is formed.	POST /_ilm/start
Stop index lifecycle management API.	Halts all lifecycle management operations and stops the ILM plugin. This is useful when you are performing maintenance on the cluster and need to prevent ILM from performing any actions on your indices.	POST /_ilm/stop

Explain lifecycle API.	Retrieves information about the index’s current lifecycle state, such as the currently executing phase, action, and step. Shows when the index entered each one, the definition of the running phase, and information about any failures.

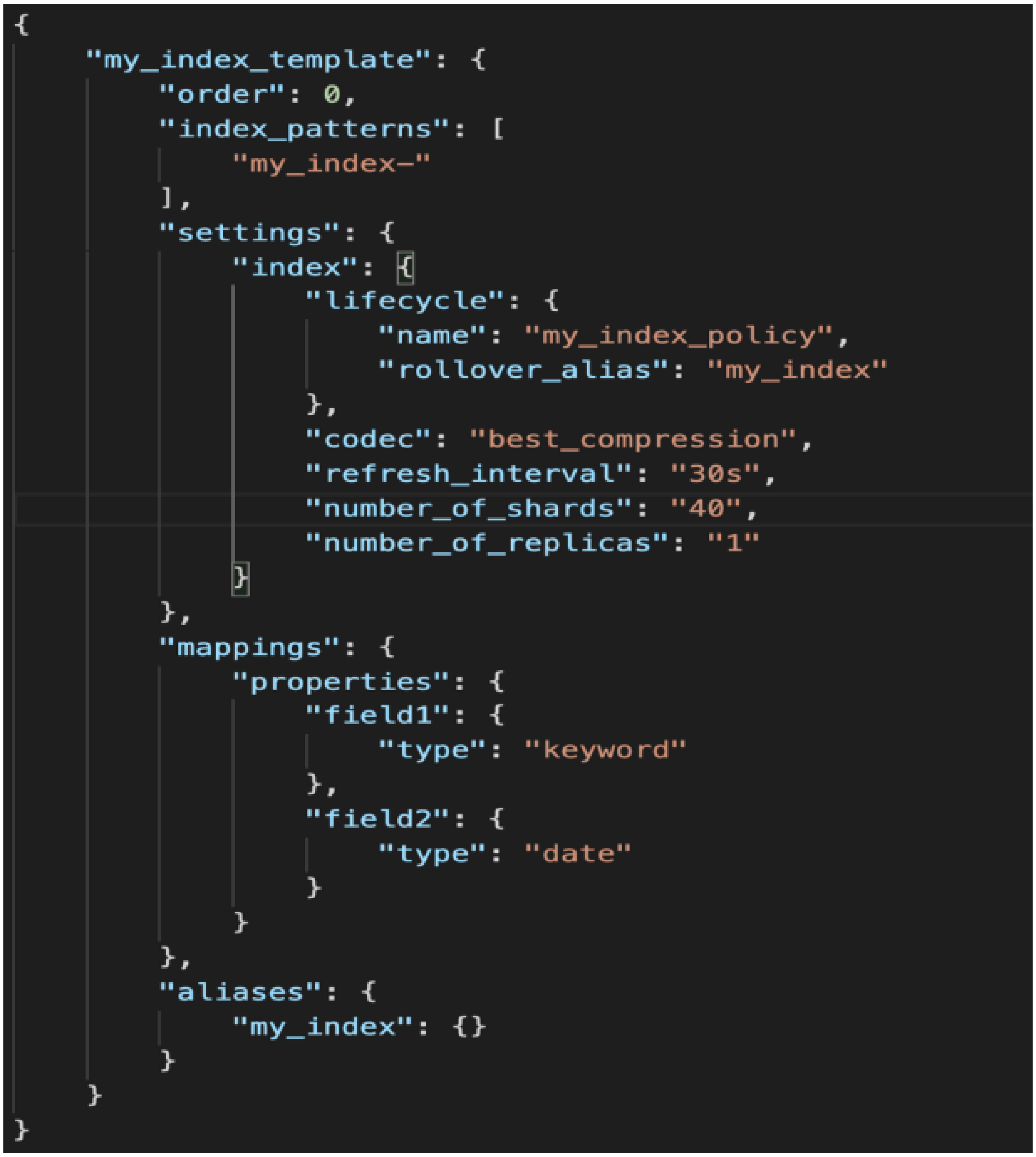




```
PUT _ilm/policy/ilm_aken
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "0ms",
        "actions": {
          "rollover": {
            "max_size": "50gb",
            "max_age": "1m"
          },
          "set_priority": {
            "priority": 100
          }
        }
      },
      "warm": {
        "actions": {}
      },
      "cold": {
        "min_age": "2m",
        "actions": {}
      },
      "delete": {
        "min_age": "3m",
        "actions": {}
      }
    }
  }
}
```

What is the role of an index template?

Index template lets you initialize new indices with predefined mappings and settings. An index template is a way to tell Elasticsearch how to configure an index when it is created manually or by inserting a document in the index. For example, if time-series data is indexed, we can define an index template so that all of these indexes have the same number of shards and replicas.



```
{
    "my_index_template": {
        "order": 0,
        "index_patterns": [
            "my_index-"
        ],
        "settings": {
            "index": {
                "lifecycle": {
                    "name": "my_index_policy",
                    "rollover_alias": "my_index"
                },
                "codec": "best_compression",
                "refresh_interval": "30s",
                "number_of_shards": "40",
                "number_of_replicas": "1"
            }
        },
        "mappings": {
            "properties": {
                "field1": {
                    "type": "keyword"
                },
                "field2": {
                    "type": "date"
                }
            }
        },
        "aliases": {
            "my_index": {}
        }
    }
}
```

This index template will be applied to all the indexes which start with **my_index** and all indexes will follow the same **my_index_policy** lifecycle policy.

In the above template definition, **my_index** is the alias that can be used as a **CONSTANT** name to search/index documents in the indexes. For example, **my_index-000001**, **my_index-000002** are two indexes that will follow the above index template and **my_index_policy** lifecycle policy.

//API to fetch all the templates defined in a cluster

GET /_template API to get template

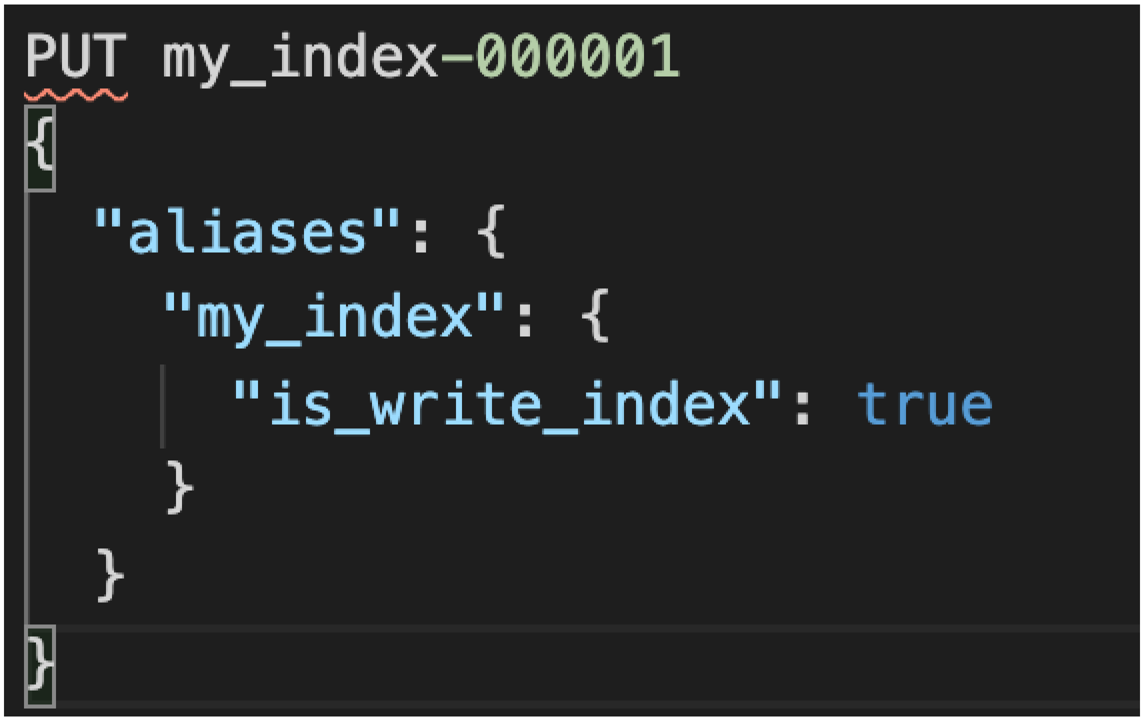
GET /_template/my_index_template // to create an index template

PUT /_template/my_index_template //create template with mapping and setting request body

Steps to create a time-series index using ILM

Now that we have a basic understanding of ILM, let’s move forward towards some practical implementation.

1. Create an index policy to specify the actions to be performed in each phase of the lifecycle.
2. Create a template for the index that will automatically initialize new indices with predefined mappings and settings on the rollover of an existing index to the other phases from the hot phase.
3. To get started, first you have to initialize an index with specified mapping and settings as specified in the template. Next, write privileges to the index with an alias name. The index name should match the template pattern and must end with a number. On rollover, this value with the index name gets incremented to generate a name for the new index.



```
PUT my_index-000001
{
  "aliases": {
    "my_index": {
      "is_write_index": true
    }
  }
}
```

To check ILM status

The *explain* API shows information about the actions performed on the index in each phase.

GET my_index/_ilm/explain

Checking ILM roll over

After ingesting docs and running refresh, when the condition in the policy matches the ILM, it will run and rollover the index from hot tier to warm tier. If we don't ingest any data, ILM will still keep rolling over the active index when it meets the specified condition mentioned in the policy.

```
GET _cat/indices/my-index*?v
```