Elasticsearch.yml : Elasticsearch.yml is the main configuration file in which all the configuration related changes need to be done on the server. (**ElasticSearch version 7.3**)

Kibana: Kibana is a web interface to communicate with the elasticsearch like curl,postman does it but more efficiently as it is particularly designed for the elasticsearch only. (**Kibana version 7.3**)
There are 2 key aspects to build any search application **Indexing & Searching process**.

Sharding:
-> Sharding is the way of dividing the indices into smaller pieces.
-> Each piece is referred to as shard.
-> Sharding is done at index level.
-> Sharding is done before the creation of the index(collection or table).
-> After creation of index & some document is inserted then if you try to add one more shard then its fine for the new doc but it will create the problem with the existing doc. So always create shard first.
NOTE: In the end of doc i have added the replication notes please refer to that after sharding then go ahead further or you can learn that at last as well but better if you can start now.

Create Index (collection name)

POST /products/_doc
{
Document data in json format
}
While writing data, data is written on primary shard but while writing if primary shard fails then elastic search immediately selects the secondary to primary shard & there is **Global Checkpoints & Local Checkpoint.**

Retrieving Doc
-> GET /products/doc_id
Elastic search shard while retrieving data uses the technique ARS(Adaptive Replica Section). It selects from which replica set data it needs to retrieve i.e. primary or secondary data set.

Updating document
-> POST /products/_doc/doc_id
-> **Update_by_query**

```
POST /products/_update_by_query
{
  "script":{
    "source":"ctx._source.instock--"
  },
 "query": {
```

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

Upsert (Update the doc if present else insert it)

```
POST /products/_doc
{
  "upsert":{
    "name":"bat",
    "tournament":"tennis",
    "willow_type":["kashmiri"],
    "Instock":10 } }
```

Routing:

Through routing elastic search gets to know where to store the doc & from which shard doc need to retrieve.

Versioning :
In elastic Search there is versioning of the document to identify how many times the document is modified.
So whenever an update is performed the _version is increased by 1.
"_version" : 1,

Batch Processing:
-> In batch processing Content-Type: application/x-ndjson. application/json will also work but the correct type is application/x-ndjson.
-> Batch Processing is used for processing data in bulk:

-> Bulk Insert:

POST /_bulk **_bulk is the keyword.**
{"index":{"_index":"products", "_id":100}} **_index is for specifying the index(collection/table) name & _id need to specify.**
{"name":"Ball","type":"season","price":500, "instock":20} **data that needs to be inserted.**
{"create":{"_index":"products", "_id":101}} **Before inserting the next data you need to specify the index name. If it's not there then it will create it.**
{"name":"Ball","type":"tennis","price":60, "instock":20} **data that needs to be inserted.**

-> Bulk Update/Delete:

```
POST /products/_bulk
{"update":{"_id":100}}
{"doc":{"price":750}}
{"delete":{"_id":101}}
```

-> Importing Data with Bulk API

-> curl -H "Content-Type: application/x-ndjson" -XPOST http://localhost:9200/products/_bulk --data-binary "@elastic_search_dummy_data.json"

Field DataTypes

1) Core Data type
2) Complex data type
3) Geo data type
4) Specialized data type

Core Data type :

-> Text data type : Used to Index full text value such as description.
-> Keyword data type : Used for structured data (tags, categories, email address)
-> Numeric data type : Basic numeric data like int ,float, byte, long, short, etc
-> Date Data type
-> Binary data type
-> Range data type : Used for the range value search of date & numeric value

Complex Data Type:

-> Object data type : json object stored in key & value pair
-> Array data type
-> Arrays of object : Association between object values are lost

```
{
        “Persons” :[{“name”:”test”,”value”:”10”},
                {“name”:”test1”,”value”:”20”},]
}
```

Which will internally becomes

```
{
        “Persons.name” :[“test”,test1],
        “Persons.value”: [“10”,”20”]
}
```

So which becomes difficult in search to resolve this there is Nested data type.
-> Nested Data type : Specialized version of the object.
Enables arrays of objects to be query independent of each other.

Geo Data type:

Goe data type as name suggest used to store the geographical lat & lang point
-> Geo point data type: Accepts lat & lang

Specialized Data type: Used to store data like IP address.

-> IP data type: Used to store ipv4 & ipv6 addresses.
-> Completion data type : Used for the auto suggest
-> Attachment data type

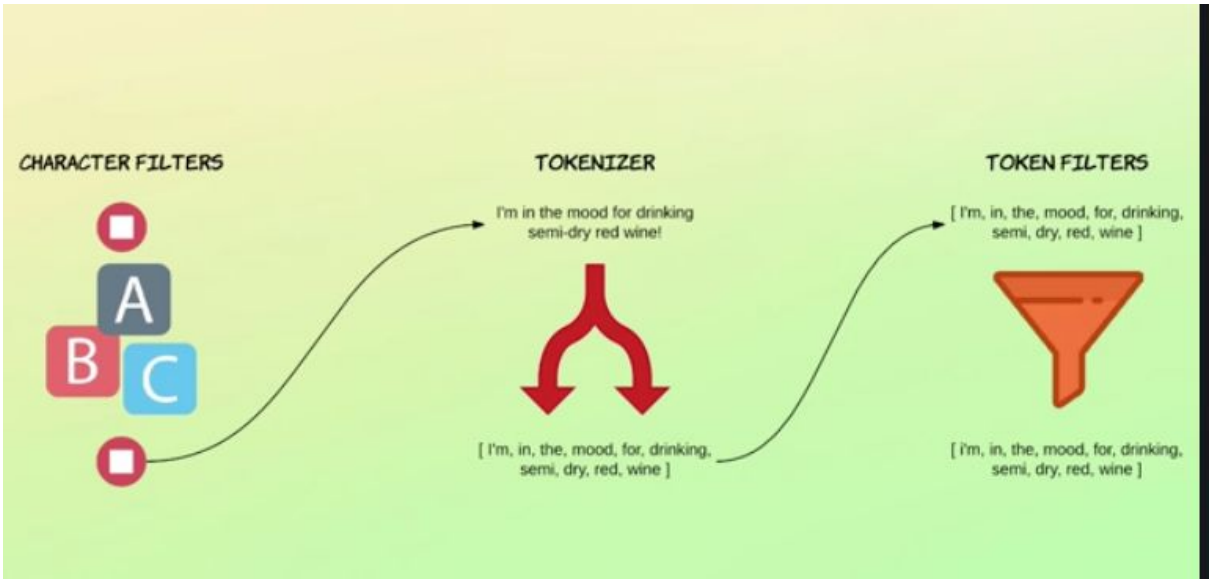
Analyzer & Tokenizer

-> Analyzer consists of 3 thing character filters, token filter & tokenizer filter.

Working of analyzer & tokenizer :

Standard tokenizer example: As shown below the text is divided into array with the help of tokenizer.

```
POST _analyze
{
  "tokenizer": "standard",
  "text" : "I'm in mood to drink wine"
}
```



Token filter with lowercase :

```
POST _analyze
{
  "filter": ["lowercase"],
  "text" : "I'm in mood to drink wine"
}

POST _analyze
{
  "analyzer": "standard",
  "text" : "I'm in mood to drink wine"
}



POST _analyze
{
  "char_filter": [],
  "tokenizer": "standard",
  "text" : "I'm in mood to drink wine",
  "filter": ["lowercase"]
}
```

Character Filter :

-> Html stripe character filter: Strips out html elements & decode the html entities like &

-> **Mapping character filter** : Replaces values based on the key & values

-> **Pattern Replace (pattern_replace) :** uses regular expressions to match the character & replace them.

Tokenizers:

-> Word oriented tokenizers : Typically used tokenizing full text into individual words.

#standard tokenizer : Divides text into terms on word boundaries & removes most symbols.(Best choice or default choice)

#Letter tokenizer : Divides text into terms when encountering characters that are not text e.g: I'm semi-dry= [i,m,semi,dry].

#Lowercase tokenizer : Works like letter tokenizer, but also lowercase letter.

#White Space tokenizer : Dividers text into term when encounters white space & white space tokenizer does not remove special symbols or characters, it is preserved as it is.

-> Partial word tokenizers

-> Structured text tokenizers

Note: While Learning from udemy on the mid way MAPPING & Analysis section is updated so contain might be repeated somewhere as before this mapping & analysis section was different.

Inverted indices:

-> Mapping between the terms & which documents contain them.

-> In the below example as data is stored in terms & whichever documents contain that terms cross(X) is marked which is inverted index so when the search is performed with terms it simply looks for the documents that contain the terms.

 # DataTypes:

-> Datatype as you know int,long, string , boolean, etc are there but there are some complex data types as well.
You can refer to this link for the datatypes:
https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-types.html

 # Object datatype : As elastic search is based on the Apache Lucen so while storing the object it converts it to json using dot notation as Apache lucen does not support object data type.
{manufacturer:{name:"test","age": 30}} stored as ->
{"manufacturer.name":"test","manufacturer.age":30}

 # Keyword datatype: ->Keyword data types are used to search for the exact match, aggregation & sorting.
-> Keyword fields are analyzed using the **keyword analyzer.**
-> As you can see in the below example, In case of keyword analyzer text are stored as it is in token while in case of text search text are stored in terms form in token. That's why keyword search is used for the exact match.

```
POST _analyze
{
  "text" : "text are stored as it is in keyword analyzer",
  "analyzer": "keyword"
```

```
}
o/p: {
  "tokens" : [
    {
      "token" : "text are stored as it is in keyword analyzer",
      "start_offset" : 0,
      "end_offset" : 44,
      "type" : "word",
      "position" : 0
    }
  ]
}
```

-> Another keyword example of storing the email address which is case sensitive & needs an exact match for the email so over here we can use the keyword search.
Shown in the below screenshot.

 # Understanding the type coercion :

-> Coercion helps to dynamically correct the datatype while storing the data.

So suppose while storing the data price = 7.4 which is float data type & stored as float.

-> And while inserting the next document we set the price = “7.4” which is string so elastic search coercion will internally convert the price to float.

-> What will happen if we pass the text in place of number in string like price= “7.4ssss”

So over here it will throw the error **“failed to parse field[price] of float type”.**

 # Adding Explicit mapping

-> Index created with the mapping

```
PUT /reviews {
  "mappings": {
    "properties": {
      "product_id":{"type": "integer"},
      "content" :{"type": "text"},
      "ratings":{"type": "float"},
      "authors" :{
        "properties": {
          "first_name":{"type":"text"},
          "last_name":{"type":"text"},
          "email_id" :{"type":"keyword"}
       }
      }
    }
  }
}
```

o/p: {

```
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "reviews" }
```

 # Using dotnotaion for mapping

-> Will result in the same output & same way it will store the data as above one but its shortcut for the above query.

```
PUT /reviews_dotnotaion
{
  "mappings": {
    "properties": {
      "product_id":{"type": "integer"},
      "content" :{"type": "text"},
      "ratings":{"type": "float"},
      "authors.first_name":{"type":"text"},
      "authors.last_name":{"type":"text"},
      "authors.email_id" :{"type":"keyword"}
    }
  }
}
```

 # retrieving mapping for specific fields

```
GET /reviews/_mapping/field/authors.email_id
```

o/p:

```
{
  "reviews" : {
```

```
  "mappings" : {
    "authors.email_id" : {
      "full_name" : "authors.email_id",
      "mapping" : {
        "email_id" : {
          "type" : "keyword"
        }
      }
    }
  }
}
```

 # Adding mapping to existing indices

```
PUT /reviews/_mapping
{
  "properties":{
    "created_at":{"type":"date"}
  }
}
```

 # Updating existing mapping(updating data type)

-> Generally ,Elasticsearch existing field mapping cannot be changed.

-> We can add new field mapping.

-> The reason we can't update the datatype is that the value is already assigned & indexed.

-> The only solution is reindexing the whole document to a new index.

 # ReIndexing document with Reindex API:

-> In reindexing basically we do create a new index & copy all the data, but if data is huge then copying huge data will be more difficult.

-> To resolve this issue there is a reindex API in which we need to define the source & destination path & it will copy all the data.

-> If for some field we want to change the datatype that also we can do it while reindexing.

-> As in the example below we have used script we can use match query to fetch specific docs only or _source:{"content","product_id"} to select specific fields only & reindex it.

E.g.: Created new index **reviews_new** & in that product_id data type is set to **keyword.**

So while moving the data we are converting the product_id data which is int to string.

```
POST /_reindex {
  "source":{
   "index":"reviews"
  },
  "dest": {
    "index":"reviews_new"
```

```
  },
  "script": {
    "source": """
      if(ctx._source.product_id != null){
        ctx._source.product_id = ctx._source.product_id.toString();
      }
    """
  }}
```

o/p:

```
"_source" : {
"authors.first_name" : "swapnil",
"product_id" : "1",
"rating" : 4.2,
"authors.email_id" : "swapnil@gmail.com",
"authors.last_name" : "pal",
"content" : "text content"}
```

Just pasted the source data of the whole result just to show product_id data is string now.
For more information on reindex
https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-reindex.html

Defining field alias:

-> As we have seen that updating the field is not possible the only way is reindexing which is not a better solution when you want to rename only the field name.

-> So to resolve this we can create an alias of the field & use the same alias name in the query as well.

-> As shown in the below query we have created a comments alias for the content field & further will query using the comments alias only.

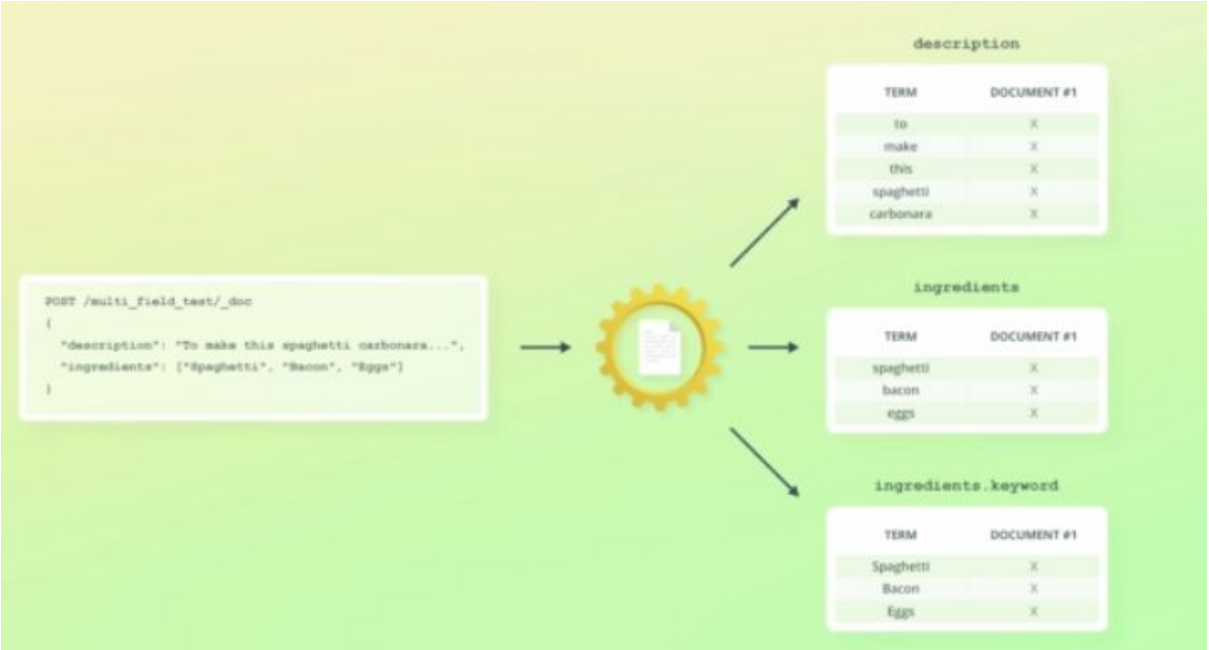
```
PUT /reviews/_mapping
{
  "properties":{
    "comments":{
      "type":"alias",
      "path":"content"
    }
  }
}
```

Multi-field Mapping:

-> Multi-field mapping is useful when you want one field to be used in keyword search & text search both.

-> As keyword search is used for the exact match & text search is for normal search.

-> As shown below screenshot multi-field data are stored in this form.



-> you can see while creating a multi-field mapping keyword term is used & we need to use keyword only.

```
PUT /reviews/_mapping
{
   "properties":{
     "source_of_origin":{
       "type":"text",
       "fields":{
         "keyword":{
           "type" :"keyword"
         }
       }
     }
  }
}

GET /reviews/_search
{
  "query": {
    "match": {
      "source_of_origin": "google"
    }
  }
}

GET /reviews/_search
{
```

```
  "query": {
    "term": {
      "source_of_origin.keyword": "Google"
    }
  }
}
```

Dynamic Mapping:

-> Dynamic mapping is basically default mapping if you do not specify the explicit mapping then default mapping for the field will be assigned.

-> Default basically consists of both text data type & keyword data type so the search can be performed on both the cases.

-> But adding dynamic mapping on description fields won't be worth it as on description you are always going to perform full text search query & not keyword search so if you use dynamic mapping for description then it will create index for text search & keyword search both which will just consume the space of your disk.

-> You can disable the dynamic mapping while adding the explicit mapping for the index.

```
mapping :{
        “Dynamic”: false,
        “Properties”:{
                Add your field mapping here
        }
}
```

-> So after setting the dynamic mapping to false if you add another field which is not in mapping then for that field index won’t be created & we can’t use that field in search query but that field will always present in _source data.

-> So if you want to restrict to add other field apart from the mapping field the set the dynamic to strict

```
mapping :{
        “Dynamic”: “strict”,
        “Properties”:{
                Add your field mapping here
        }
}
```

-> so it will throw an error if some other field is added in the POST method JSON body.

Stemming & stop words:

-> In **Stem analyzer** when the sentence is entered it converts all the words into their root form like **loved & drinking** will be stored as **love & drink** & while searching as well if **stem analyzer** is used then it will convert it to its root form & perform the search.

Built-in Analyzer :
https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-analyzers.html

Creating Custom Analyzer:

-> By Using built-in analyzer, filter & token analyzer you can create your own analyzer and use it accordingly for text manipulation.

Custom analyzer query

```
PUT /custom_analyzer_reviews
{
  "settings": {
    "analysis": {
      "analyzer": {
        "My_custom_analyzer":{ //our custom analyzer name
          "type":"custom",
          "char_filter":[
            "html_strip" //to strip out html character
            ],
          "tokenizer":"standard", // standard tokenizer to store the data in array format
          "filter":[ //filters to be used , mostly name will indicate its use
            "lowercase",
            "asciifolding", //This will remove non ascii character like in Ñ from word NÑot
            "stop" // This remove stop words like be,a,an,will
            ]
        }
      }
    }
  }
}

POST /custom_analyzer_reviews/_analyze
{
  "analyzer":"my_custom_analyzer",
  "text":"<b>Checking Custom analyzer</b> will be working or NÑot"
```

}
As you can see in the output data is filtered ,stop word is removed, asciifolding has been done & stored in lowercase.
o/p:


```
{
  "tokens" : [
    {
      "token" : "checking",
      "start_offset" : 3,
      "end_offset" : 11,
      "type" : "<ALPHANUM>",
      "position" : 0
    },
    {
      "token" : "custom",
      "start_offset" : 12,
      "end_offset" : 18,
      "type" : "<ALPHANUM>",
      "position" : 1
    },
    {
      "token" : "analyzer",
      "start_offset" : 19,
      "end_offset" : 31,
      "type" : "<ALPHANUM>",
      "position" : 2
    },
    {
      "token" : "working",
      "start_offset" : 40,
      "end_offset" : 47,
      "type" : "<ALPHANUM>",
      "position" : 5
    },
    {
      "token" : "nnot",
      "start_offset" : 51,
      "end_offset" : 55,
      "type" : "<ALPHANUM>",
      "position" : 7
    }
  ]
}
```


Example 2:
PUT /custom_analyzer_reviews

```
{
  "settings": {
    "analysis": {
      "filter": {
      "danish_stop":{ //created a custom filter to remove danish words & not english
words.
          "type":"stop",
          "stopwords":"_danish_"
        }},
      "analyzer": {   Reset all are same above just in filter added the danish_stop
instead of stop
"filter":["danish_stop" ]
```

Adding analyzer to existing indices

```
PUT /custom_analyzer_reviews/_settings
{
  "analysis":{
    "analyzer": {
      "second_custom_analyzer":{
        "type":"custom",
        "char_filter":[
          "html_strip"
          ],
        "tokenizer":"standard",
        "filter":[
          "lowercase",
          "asciifolding",
          "stop"
          ] }
      }
  }
}
{
  "error": {
    "root_cause": [ { "type": "illegal_argument_exception",
       "reason": "Can't update non dynamic settings
[[index.analysis.analyzer.second_custom_analyzer.tokenizer,
index.analysis.analyzer.second_custom_analyzer.type,
index.analysis.analyzer.second_custom_analyzer.filter,
index.analysis.analyzer.second_custom_analyzer.char_filter]] for open indices
[[custom_analyzer_reviews/rXCNDREnT8udMJt0Cmhl0g]]"
```

This has thrown the error of a non dynamic setting which means our index is static, so to add any analyzer to the existing index we need to first close the index.
Which means no read, write or search operation will be performed on the index & after adding the index we need to open the index again to perform the read, write & search

operations.
NOTE: But closing & opening the index is not a solution if the index is critical where a lot of search is performed. So over situation like this we need to reindex & create alias & use it

POST /custom_analyzer_reviews/_close
POST /custom_analyzer_reviews/_open
#To check the custom analyzer added successfully
GET /custom_analyzer_reviews/_settings

Updating the analyzer:

-> To update the analyzer we need to perform the same step that is close the index & update the analyzer & open it again.

-> But if data is already present in the index & after the modification of the analyzer so it will not reindex the previous data & search will use the updated analyzer .

-> So our search will be inconsistent, so we need to reindex so that updated index will be applicable.

-> To do that we need to use the update_by_query which will reindex the data & search will be consistent.

Search Methods

```
GET /products/_search
{
"query": {
   "match": {"description": "Lorem"}

 }
}
```

 # Query DSL (basically query string which we pass in the **GET URL)**

```
GET /products/_search
{
"query": {
   "query_string": {"query": "description:Lorem"}

 }
}
```

 # Searching with request uri

NOTE: As i am running the below query in **Kibana** which automatically adds the %20 in place of space or whatever is required for space. So if you are running the query in **postman or curl** you need to add the %20 for space.

```
GET /products/_search?q=name=Sprouts
```

Searching in array. Over Here tags is the array which consists of vegetables in it.

```
"tags" : [
          "Vegetable"
        ],
```

GET /products/_search?q=tags:vegetable

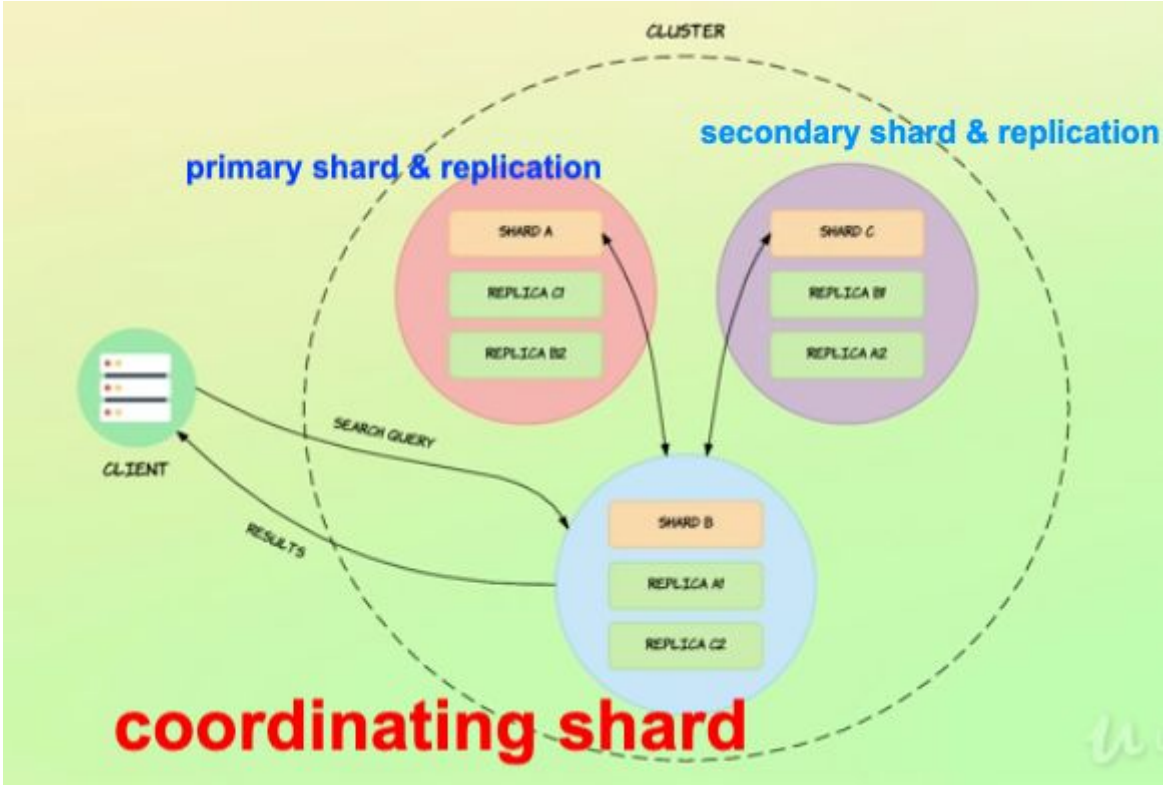
And further if you want to add the AND condition as well then query will be like this:

GET /products/_search?q=tags:vegetable and name = onions

 # Search query working

-> When client requests for the search query it first goes to coordinating shard which decides on which shard it needs to send & according to that result is processed & returned back to coordinating server to client.

Image source : udemy video



 # Understanding the query results

```
{
  "took" : 8, -> took keyword shows that time(milliseconds) taken to execute the query
  "timed_out" : false, -> As name suggest to check whether query is timeout or not
  "_shards" : { -> this array shows how many shards were created while creating
index
    "total" : 1, -> On how many shards it performed the search
    "successful" : 1, -> On this many shards it performed the search successfully
    "skipped" : 0, -> skipped & failed as name suggest performed on the shards
    "failed" : 0
  },
  "hits" : { -> Hits contain the result objects
    "total" : {
      "value" : 43, -> total number of doc it searched for the query
      "relation" : "eq"
    },
    "max_score" : 8.470718,
    "hits" : [
      {
        "_index" : "products", -> index(table/collection) name
        "_type" : "_doc",
        "_id" : "9",
        "_score" : 8.470718, -> _Score indicates how well the doc matches with the
search query, further explained about the score.
        "_source" : {
          "name" : "Onions - Green",
          "price" : 105,
          "in_stock" : 22,
          "sold" : 199,
          "tags" : [
            "Vegetable"
          ],
```

#Understanding of relevance score

-> In elasticsearch only a match of a document is not enough it also checks how well the doc is matched with the search query & it is measured by the _score field.

-> To calculate the _score elasticsearch uses the algorithm

TF/IDF(Term Frequency/ Inverse document frequency), Okapi BM25.

-> OKapi BM25: The relevance scoring algorithm currently used by elasticsearch.

-> **Term Frequency(TF) : The more number of times the term appears in the query the higher the number of relevance scores would be.**

-> Inverse Document Frequency: For each query term, the total number of documents divided by the total number of documents containing that term.

-> **Field Length norms: The longer the field the lesser its appearance in the doc frequency.**

#Debugging the query

_explain : explain will list the description of why the data is not matched or matched

```
GET /products/_doc/895/_explain
{
"query": {
   "query_string": {"query": "description:Lorem"}

 }
}
O/P:   {
          "_index" : "products",
          "_type" : "_doc",
          "_id" : "898",
          "matched" : false,
          "explanation" : {
            "value" : 0.0,
            "description" : "no matching term",
            "details" : [ ]
          }
}
```

#TERM LEVEL Query : Term level query matches with exact match of word i.e case sensitive
Full text search : Full text search is case insensitive. It matches regardless of case.

TERM LEVEL Query:

```
GET /products/_search
{
  "query":{
    "term": {
      "is_active": true
    }
  }
}
```

#Terms:

-> Terms are basically like a query in mysql or mongodb.

-> For single check we use keyword **TERM** & for in queries **TERMS**

```
GET /products/_search
{
  "query":{
    "terms": {
      "tags.keyword": [
        "Meat",
        "Soup"
        ]
    }
  }
}
```

-> If you want to search with ids field of elasticsearch we can do it like this

```
GET /products/_search
{
  "query": {
    "ids": {
      "values": ["1","2","3"]
    }
  }
}
```

#Range Query:

```
GET /products/_search
{
  "query": {
```

```
      "range": {
        "in_stock": {
          "gte": 1,
          "lte": 10
        }
      }
    }
  }
```

-> Range query with dates

-> In Date range query we can also specify date format as well elastic search will match the query in any format.

```
GET /products/_search
{
  "query": {
    "range": {
      "created": {
        "gte": "02/08/2005",
        "lte": "02/05/2008",
        "format": "dd/MM/yyyy"
      }
    }
  }
}
```

#DATE MATH:

-> Date math is basically adding or subtracting 1 year or 1 month or 1day or any numbers in the search. So to perform that we can do this following way:
If you want you can specify date format as well.
-> below query will add 1 year 1month & 1 day in gte query means 03/09/2006 & subtracting 1 year in less than so it will search between 03/09/2006 >= && <= 02/05/2007

```
 "range": {
    "created": {
      "gte": "02/08/2005||+1y+1d+1m",
      "lte": "02/05/2008||-1y",
      "format": "dd/MM/yyyy"
    }
```

https://www.elastic.co/guide/en/elasticsearch/reference/current/common-options.html#date-math

#Exists

-> For some fields we add checksum of not null, So in elastic search exists will check data for not null which means an empty array is also considered as null.

```
GET /products/_search{
  "query": {
    "exists": {
      "field": "tags"
```

```
    }
  }
}
```

#Prefix Match

-> Matches with the prefix it's like a **LIKE** query of mysql

```
GET /products/_search
{
  "query": {
    "prefix": {
      "tags.keyword":"Veg"
    }
  }
}
```

#WildCard Search

-> * will search with the start & end letter & gives the results whatever satisfies the query.

```
GET /products/_search
{
  "query": {
    "wildcard": {
      "tags.keyword":"Veg*le"
    }
  }
}
```

-> ? will search for the 1 word only missing

```
        "tags.keyword":"Vege?able"
```

-> As we all know wildcard queries are slow as compared to other queries as it has to perform all permutations & combinations for the results especially when there are millions of records present on our cluster.

#RegularExpression:

```
GET /products/_search
{
  "query": {
    "regexp": {
      "tags.keyword":"Vege[a-zA-Z]+able"
}}}
```

https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-regexp-query.html#regexp-syntax

Full Text Searching:

-> In Full text search it matches the word in the text & if it matches any words it returns the result.

-> So while performing the search if a word appeared more than one time in the text then its relevance score would be.

-> By default in search an OR query is applied so if you search for more than one word then it will share all the matching docs with words. And you manually need to specify the AND operator.

1) By default OR is applied

```
GET /products/_search
{
  "query": {
    "match": {
      "name": "pasta banana"
    }
  }
}
```

2) And operator query, So in AND operator if PASTRY & BANANA both present in the name field then only it will share the results.

```
GET /products/_search
{
  "query": {
    "match": {
      "name": {
        "query": "Pastry banana",
        "operator": "and"
      }
    }
  }
}
```

#Matching Phrase: It will match data in sequence of given phrase & if you switch the phrase then no result will be displayed.

```
GET /products/_search
{
  "query": {
    "match_phrase": {
      "name": "Pastry banana"
    }
  }
}
```

#Multi Match:

-> In multi match words are searched in multiple fields & share the results.

-> So if the same words if matched in all the given fields then its score will be more relevant than words matching in the one or two fields so more number of matches in the given field then better chance of having a high number of relevance scores.

```
GET /products/_search
{
  "query": {
    "multi_match": {
      "query": "Chocolate",
      "fields": ["name","description"]
    }
  }
}

#Boolean Query
GET /reciep/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "ingredients.name": "dry pasta"
          }
        },
        {
          "range": {
            "preparation_time_minutes": {
              "lte": 15
            }
          }
        }
      ]
    }
  }
}
```

-> Adding the filter in query will change the relevance score calculation

```
GET /reciep/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "ingredients.name": "dry pasta"
```

```
        }
       }
      ],
      "must_not": [
       {
        "match": {
         "ingredients.name": "Cherry tomatoes"
        }
       }
      ],
      "filter": [
       {
        "range": {
         "preparation_time_minutes": {
          "lte": 15
         }
        }
       }
      ]
     }
    }
}
```

-> When we use the **should** term in query with must or filter then its basically used to boost the relevance score but if **should term** is alone in the query then it should match the condition in the given query & boost the relevance score as well

```
GET /reciep/_search
{
  "query": {
    "bool": {
      "must": [
       {
        "match": {
         "ingredients.name": "pasta"
        }
       }
      ],
      "must_not": [
       {
        "match": {
         "ingredients.name": "Cherry tomatoes"
        }
```

```
      }
    ],
    "should": [
      {
        "match": {
          "ingredients.name": "Parmesan"
        }
      }
    ],
    "filter": [
      {
        "range": {
          "preparation_time_minutes": {
            "lte": 15
          }
        }
      }
    ]
  }
 }
}
```

#Debugging Boolean query

-> In the above query if you want to check which conditions are applied so you can add the _name field in the object. Shown below

```
      "must": [
      {
        "match": {
          "ingredients.name": {
            "query": "pasta",
            "_name" :"pasta_must"
          }
        }
      }
    ],
    "should": [
      {
        "match": {
          "ingredients.name": {
            "query": "Parmesan",
            "_name": "Parmesan_should"
          }
        }
```

```
      }
    ],
        o/p: this will be added in the output which shows all this filtered are applied.
         "matched_queries" : [
                "prep_time_filter",
                "Parmesan_should",
                "pasta_must"
              ]
```

How Match Query works:
-> whenever a match query is performed internally it runs the boolean query for performing the search.
-> Normal match query is performed for single or multiple words then it basically performs the boolean **SHOULD** query & when the And operator is added then it performs the **MUST** query.

Querying Nested Objects:

```
GET /vendor_index2020-06-02-21-03-55/_search
{
  "query": {
    "nested": {
      "path": "finderArray",
      "query": {
        "bool": {
          "must": [
            {
              "match": {
                "finderArray.finder_name": "sutherland"
              }
            }
          ],
          "filter": {
            "term": {
              "finderArray.finder_location": "malad"
            }
          }
        }
      }
    }
  }
```

}

Nested inner hits:
-> Nested inner hits will return the exact match of array from the nested object instead of returning the whole set of objects which is the case in nested query.
-> Just need to add **inner_hits.**
-> Apart from giving specific match objects it returns the index key of the array as well that on which position the matched data was. As shown in the **_nested object.**

```
"nested": {
      "path": "finderArray",
      "inner_hits": {},
      "query": {
```

O/P

```
"inner_hits" : {
         "finderArray" : {
           "hits" : {
             "total" : {
               "value" : 1,
               "relation" : "eq"
             },
             "max_score" : 2.8626328,
             "hits" : [
               {
                 "_index" : "vendor_index2020-06-02-21-03-55",
                 "_type" : "_doc",
                 "_id" : "zCKSc3IBafDnXOtRehcC",
                 "_nested" : {
                   "field" : "finderArray",
                   "offset" : 0
                 },
                 "_score" : 2.8626328,
                 "_source" : {
                   "finder_name" : "Sutherland Global Services - Magarpatta",
                   "finder_slug" : "sutherland-global-services---magarpatta-hadapsar",
                   "finder_location" : "Hadapsar"
                 }
               }
             ]
```

Join Queries & mapping documents Relationships :

-> In elasticsearch there is no parent child relationship between indexes(table), we can create a relationship between the same index & that we need to define the relationship in mapping.

-> Mapping documents relationships are basically creating parent child relationships within the document.
-> "relations": {
          "department": "employee"
       } in this department the parent & employee is a child.

```
PUT /department
{
  "mappings": {
    "properties": {
      "join_field": {
        "type": "join",
        "relations": {
          "department": "employee"
        }
      }
    }
  }
}
```

-> Department doc with _id as 1 which is the primary key

```
PUT /department/_doc/1
{
  "name": "Development",
  "join_field": "department"
}
```

-> Now we are adding data for employee which have the foreign key of the department doc. So here the foreign key is 1 i.e specified as parent:1.
-> So this doc has a parent child relationship between department & employee doc.

```
PUT /department/_doc/8?routing=1
{
  "name": "Christina Parker",
  "age": 29,
  "gender": "F",
  "join_field": {
    "name": "employee",
    "parent": 1
  }
}

GET /department/_search
{
  "query": {
    "parent_id": {
      "type": "employee",
```

```
      "id": 1
    }
  }
}

GET /department/_search
{
  "query": {
    "has_parent": {
      "parent_type": "department",
      "score": true,
      "query": {
        "match": {
          "name": "development"
        }
      }
    }
  }
}

GET /department/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "query": {
        "bool": {
          "must": [
            {
              "range": {
                "age": {
                  "gte": 40
                }
              }
            }
          ],
          "should": [
            {
              "term": {
                "gender.keyword":"M"
              }
            }
          ]
        }
      }
    }
```

```
  }
}
```

Multi Level Relationship:

-> So in a multi level relationship it's the same as we defined above the relationship but if multiple are at the same level then define in array.

```
PUT /company
{
  "mappings": {
    "properties": {
      "join_field": {
        "type": "join",
        "relations": {
          "company":["department","supplier"],
          "department": "employee"
        }
      }
    }
  }
}
GET /company/_mapping
PUT /company/_doc/1
{
  "name" :"abc corp",
  "join_field":"company"
}
```

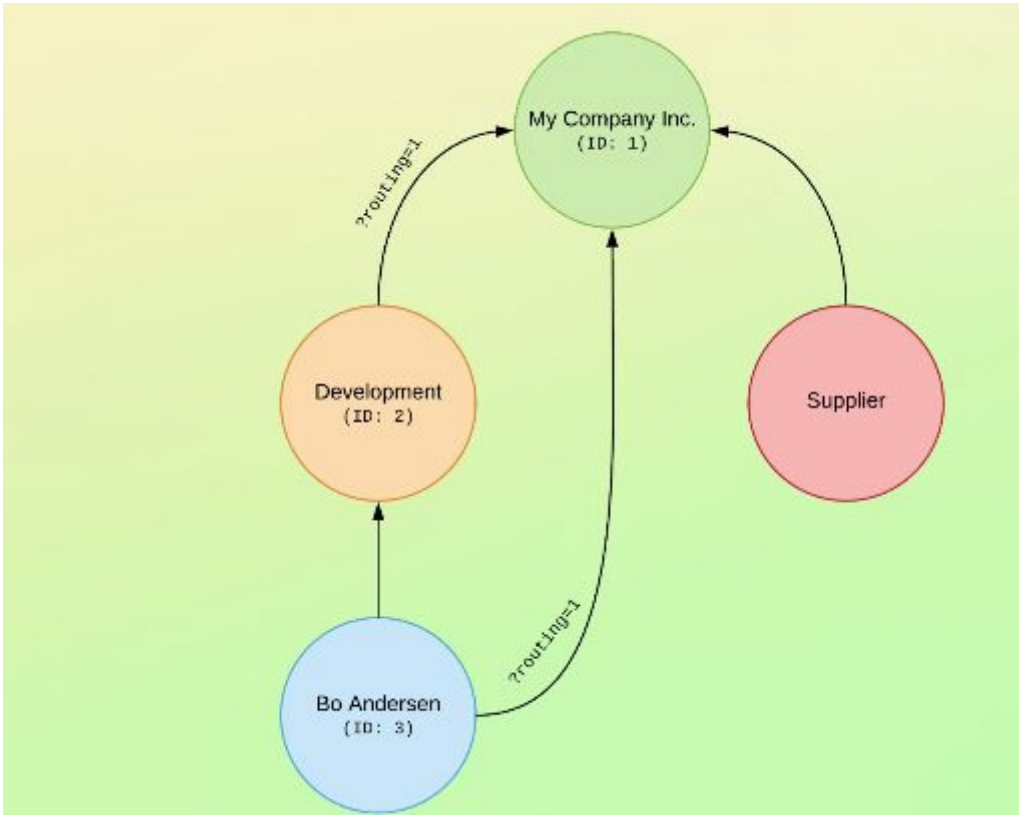
-> As you can see the company is top level which has a department under it so parent:1 is set.

```
PUT /company/_doc/2?routing=1
{
  "name": "development",
  "join_field": {
    "name": "department",
    "parent": 1
  }
}
```

-> Employee is under department so we set the parent:2 but routing is 1 because we want all the docs from the same routing to be on the same shared.

```
PUT /company/_doc/3?routing=1
{
  "name":"swa",
  "join_field":{
    "name":"employee",
    "parent" :2
  }
}
```

-> Relationship is explained in below image:



-> multi level search query.

```
GET /company/_search
{
  "query": {
    "has_child": {
      "type": "department",
      "query": {
        "has_child": {
          "type": "employee",
          "query": {
            "match": {
              "name": "swa"
            }
          }
        }
```

```
    }
   }
  }
}
```

Formatting Results:
-> _search?pretty, _search?format=json,_search?format=yaml. Specify the format to output the data.
Source Filtering:
-> Source filtering is basically select query. You can specify which field you want to include in results.

```
GET /recipe/_search
{
  "_source": "created",
  "query": {
    "match": { "title": "pasta" }
  }
}
```

```
GET /recipe/_search
{
  "_source": {
    "includes": "ingredients.*",
    "excludes": "ingredients.name"
  },
  "query": {
    "match": { "title": "pasta" }
  }
}
```

Limit query(pagination):

```
 "from": 0,
  "size": 20,
```

Aggregation :

 # METRIC Aggregations:

1) **Single value numeric metric aggregation(SVNMA):**

-> SVNMA which returns results in single values like sum, avg,min,max etc.

-> When we don't specify the match or match_all query then by default match_all is in where condition. And we have set the size zero as we don’t need doc result just aggregations result we needed.

```
GET /order/_search
{
  "size": 0,
  "aggs": {
    "sum_amount": {
      "sum": {
        "field": "total_amount"
      }
    },
    "avg_amount" :{
      "avg": {
        "field": "total_amount"
      }
    },
    "min_amount":{
      "min": {
        "field": "total_amount"
      }
    },
    "max_amount" :{
      "max": {
        "field": "total_amount"
      }
    }
  }
}
o/p:
"aggregations" : {
    "sum_amount" : {
      "value" : 109209.61
    },
    "min_amount" : {
      "value" : 10.27
    },
    "avg_amount" : {
      "value" : 109.20961
```

```
    },
    "max_amount" : {
      "value" : 281.77
    }
  }
```

- **Cardinality Aggregations:**

-> A single-value metrics aggregation that calculates an approximate count of distinct values. You can say to it

-> To get the accurate result we need to perform a lot of analysis which will require disk space as well & it will take the time to perform.

-> The precision_threshold option allows to trade memory for accuracy, and defines a unique count below which counts are expected to be close to accurate.

-> The maximum number of precision_threshold we can set is 40000 & precision_threshold set above 40000 will give you the same results.

-> if you set the precision_threshold to 1000 or 100 so the result won't differ that much when you set the precision_threshold to 40000 its approx 1-6% of difference only.

```
GET /order/_search
{
  "size": 0,
  "aggs": {
    "total_sales_person": {
      "cardinality": {
        "field": "salesman.id",
        "precision_threshold": 100
      }
    }
  }
}
```

- **value_count Aggregation:**

-> To get the count of the index we can use value_count or to perform the average from the index.

```
GET /order/_search
{
  "size": 0,
  "aggs": {
    "total_sales_person": {
      "value_count": {
        "field": "total_amount"}}}}
```

2) Multi value numeric metric aggregation :

-> Multi value aggregation is nothing but the combination of all the count, max,min,avg,sum aggregation into one as **stats.**

```
GET /order/_search
{
```

```
  "size": 0,
  "aggs": {
    "multilevel_aggs_stats": {
      "stats": {
        "field": "total_amount"
      }
    }
  }
}
```

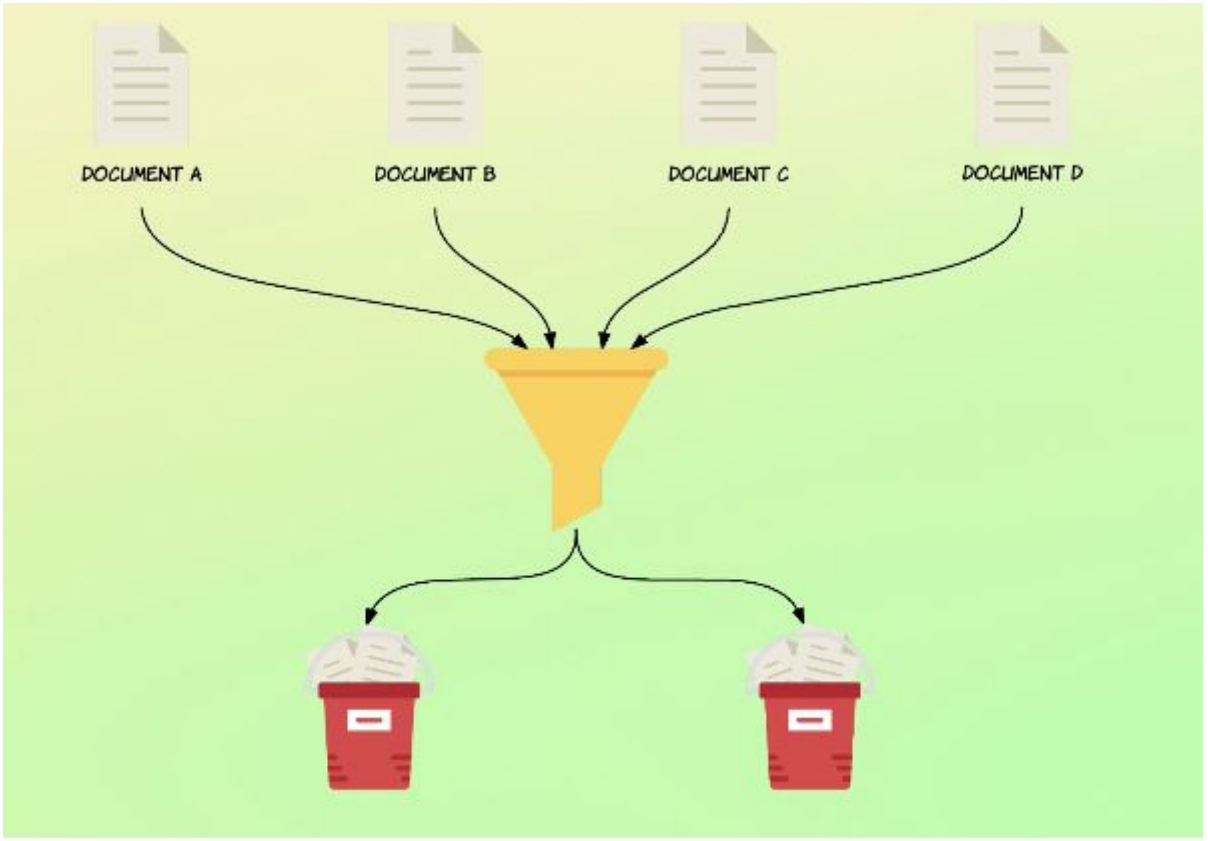
-> Output is the combination of all the above single level numeric metric aggregation results.

o/p:

```
"aggregations" : {
    "multilevel_aggs_stats" : {
      "count" : 1000,
      "min" : 10.27,
      "max" : 281.77,
      "avg" : 109.20961,
      "sum" : 109209.61
    }
  }
```

#Bucket Aggregation:

-> Bucket level aggregation basically after filtering the data creates one or more dynamic buckets & which has the unique data according to the criteria. And criteria is based on the aggregation bucket used like terms , etc



Term Aggregation:

```
GET /order/_search
{
  "size": 0,
  "aggs": {
    "bucket_level_aggs": {
      "terms": {
        "field": "status"
      }
    }
  }
}
```

o/p snippet :

As you can see a bucket has been created which has the count of different different status.

```
"aggregations" : {
    "bucket_level_aggs" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [
        {
          "key" : "processed",
          "doc_count" : 209
        },
```

```
      {
        "key" : "completed",
        "doc_count" : 204
      },
      {
        "key" : "pending",
        "doc_count" : 199
      },
      {
        "key" : "cancelled",
        "doc_count" : 196
      },
      {
        "key" : "confirmed",
        "doc_count" : 192
      }
    ]
  }
 }
```

-> **sum_other_doc_count** implies that how many documents have not been considered in the buckets. As we have created a bucket on status field which has 5 types only if it would have been 50 or 100 then it would not consider each & every one and would have shown the count of the lefted doc.

-> I have changed the status field to total_amount & see the results below, it's just a snippet.

```
"sum_other_doc_count" : 980,
    "buckets" : [
      {
        "key" : 23.77,
        "doc_count" : 2
      },
```

-> min_doc_count to set the filter on the bucket that min doc should be matched.

```
"bucket_level_aggs": {
    "terms": {
      "field": "status",
      "min_doc_count": 0} }
```

-> _term or _key used for order by bucket data according to the field.

```
"aggs": {
   "bucket_level_aggs": {
     "terms": {
       "field": "status",
       "min_doc_count": 0,
       "order": {
         "_key": "asc"
       }
     }
```

}

Note : Count shared by the term bucket aggregation is not 100% correct as data is stored on different shared & on some shared the score can be lower and suppose if you have added the size to 3 in the query then that doc won't be there in top 3. As shown below.

And this issue is mainly if there are multiple shared but if there is one shared only or your data fits on one shared only then the data will be accurate.

-> So usually keep the size higher by default it's 10. The higher the value the more accurate the result.

	SHARD A	SHARD B	SHARD C
1	PRODUCT A (50)	PRODUCT A (50)	PRODUCT A (50)
2	PRODUCT B (40)	PRODUCT B (40)	PRODUCT E (40)
3	PRODUCT C (30)	PRODUCT F (30)	PRODUCT F (30)
4	PRODUCT F (20)	PRODUCT C (20)	PRODUCT B (20)
5	PRODUCT D (10)	PRODUCT E (10)	PRODUCT C (10)

	ACTUAL RESULT	CORRECT RESULT
1	PRODUCT A (150)	PRODUCT A (150)
2	PRODUCT B (80)	PRODUCT B (100)
3	PRODUCT F (60)	PRODUCT F (80)

-> **doc_count_error_upper_bound :** shows the count of the data that has missed in above scenario.

Nested Aggregations:

-> Nested aggregation are basically output of previous aggregation and act as input of further aggregation like in mongo db we do.

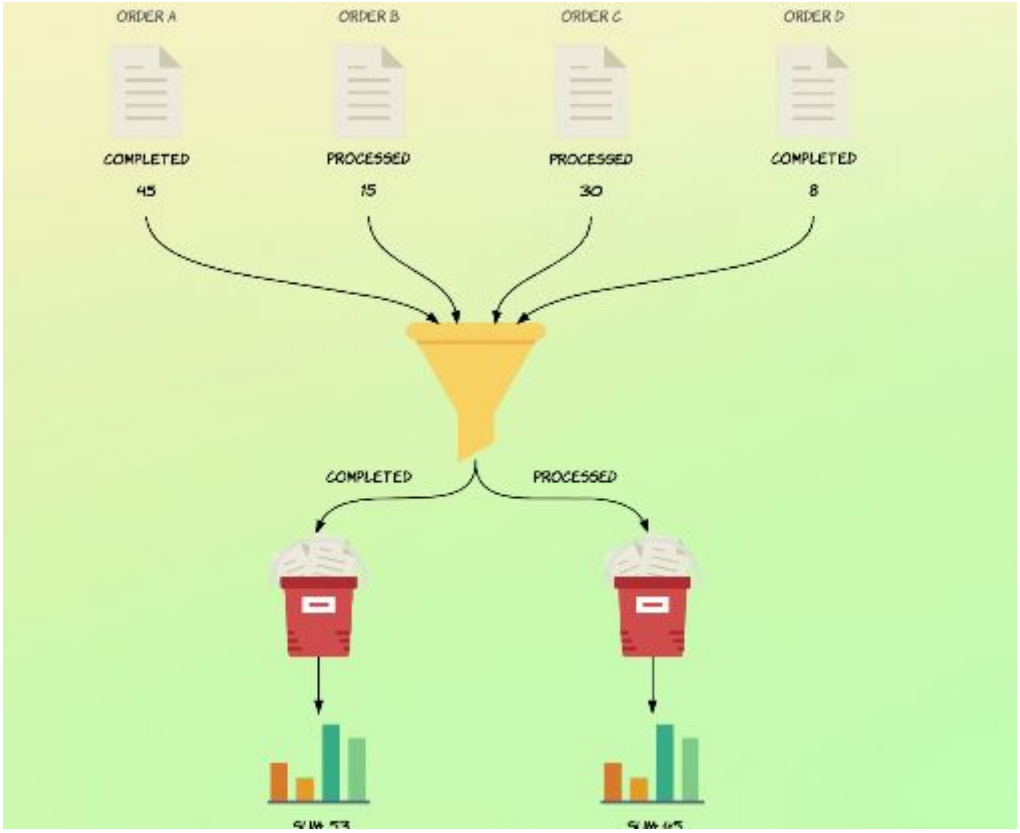
-> So in the below aggregation output of terms aggregation is further used as input for the multi value metric aggregation.

```
GET /order/_search
{
  "size": 0,
  "aggs": {
    "status_stats": {
      "terms": {
        "field": "status"
```

```
    },
    "aggs": {
      "total_cal": {
        "stats": {
          "field": "total_amount"
        }
      }
    }
  }
 }
}
```

Output snippet :

```
"buckets" : [
      {
        "key" : "processed",
        "doc_count" : 209,
        "total_cal" : {
          "count" : 209,
          "min" : 10.27,
          "max" : 281.77,
          "avg" : 109.30703349282295,
          "sum" : 22845.17
        }
      },
```



Filter Aggregation:

-> Filter Aggregation is top level aggregation to filter out the data like match query.

```
"aggs": {
       "min_amount_filter":{
         "filter": {
           "range": {
             "total_amount": {
               "gte": 100
             }
           }
         },
         "aggs": {
           "total_cal": {
           "stats": {
             "field": "total_amount"
           }
         }
         }
       }
     }
```

Defining Bucket rules filter/ creating bucket with filters:

```
GET vendor_index2020-06-02-21-03-55/_search
{
  "size": 0,
  "aggs": {
    "location_bucket": {
      "filters": {
        "filters": {
          "mumbai_bucket": {
              "match": {
                "city_name": "mumbai"
              }
          },
          "pune_bukcet":{
              "match": {
                "city_name": "pune"
              }
          }
        }
      }
    }
  }
}
```

Range Aggregation:
-> Range aggregation basically to get the data between given criteria.
-> there are range aggregation, date aggregation,etc.
Note: In range aggregation **from** is included & **to** is excluded from results.

```
GET order/_search
{
  "size": 0,
  "aggs": {
    "range_aggregation": {
      "date_range": {
        "field": "purchased_at",
        "format": "yyy-MM-dd",
        "keyed": true,  // This is for setting the key name for the output
        "ranges": [
          {
            "from": "2016-01-01",
            "to": "2016-01-01||+6m",
            "key": "first_half" //This is the key name for output
          },
          {
            "from": "2016-01-01||+6m",
            "to": "2016-01-01||+1y",
            "key": "second_half" //This is the key name for output
          }
        ]
      }
    }
  }
}
```

Output snippet :

```
"aggregations" : {
    "range_aggregation" : {
      "buckets" : {
        "first_half" : {
          "from" : 1.4516064E12,
          "from_as_string" : "2016-01-01",
          "to" : 1.45160676E12,
          "to_as_string" : "2016-01-01",
          "doc_count" : 0
        },
        "second_half" : {
          "from" : 1.45160676E12,
          "from_as_string" : "2016-01-01",
          "to" : 1.4832288E12,
```

```
        "to_as_string" : "2017-01-01",
        "doc_count" : 1000
      }
    }
  }
}
```

Histogram:
-> Histogram are used to create a dynamic bucket at some interval.
-> So in histogram if you have set the interval to 25 then on every 25th interval it will create the bucket & document falling between that bucket will set the doc count of that bucket.
-> **extended_bounds** is used to create a bucket between min & max range.

```
GET /order/_search
{
  "size": 0,
  "aggs": {
    "histogram_agg": {
      "histogram": {
        "field": "total_amount",
        "interval": 25,
        "extended_bounds": {
          "min": 0,
          "max": 400
        }
      }
    }
  }
}
```

Output:

```
"aggregations" : {
    "histogram_agg" : {
      "buckets" : [
        {
          "key" : 0.0,
          "doc_count" : 42
```

```
        },
        {
          "key" : 25.0,
          "doc_count" : 122
        },
        {
          "key" : 50.0,
          "doc_count" : 153
        },
        {
          "key" : 75.0,
          "doc_count" : 194
        },
```

-> in **date_histogram** dynamic bucket are created based on week,month,quarter,year,day,hour,etc

```
GET /order/_search
{
  "size": 0,
  "aggs": {
    "histogram_agg": {
      "date_histogram": {
        "field": "purchased_at",
        "interval": "quarter"
      }
    }
  }
}
```

Output:

```
"aggregations" : {
    "histogram_agg" : {
      "buckets" : [
        {
          "key_as_string" : "2016-01-01T00:00:00.000Z",
          "key" : 1451606400000,
          "doc_count" : 232
        },
        {
          "key_as_string" : "2016-04-01T00:00:00.000Z",
          "key" : 1459468800000,
          "doc_count" : 249
        },
```

Global Aggregation:

-> In global aggregation it calculates the data from all the docs in an index even if you have specified the match query it will ignore & get the results on all.

```
GET /order/_search
{
  "size": 0,
  "query": {
    "range": {
      "total_amount": {
        "gte": 100
      }
    }
  },
  "aggs": {
    "global_aggregation": {
      "global": {}
    }
  }
}
```

Output :

-> As you can see in hits, total.value is 489 doc but in global aggregation you can doc_count to 1000.

-> So apart from global aggregation if other aggregation would be there then data would be calculated at 489 docs only.

```
"hits" : {
    "total" : {
      "value" : 489,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [ ]
  },
  "aggregations" : {
    "global_aggregation" : {
      "doc_count" : 1000
    }
```

```
 }
```

Missing Aggregation:

-> Missing aggregation is used to find the doc_count with missing field or null value.

```
GET /vendor_index2020-06-02-21-03-55/_search
{
  "size": 0,
  "aggs": {
    "missing_field": {
      "missing": {
        "field": "created_date"
      }
    }
  }
}
"aggregations" : {
    "missing_field" : {
      "doc_count" : 100
    }
  }
```

Nested Aggregation:

```
GET /vendor_index2020-06-02-21-03-55/_search
{
  "size": 0,
  "aggs": {
    "nested_aggregations": {
      "nested": {
        "path": "finderArray"
      },
      "aggs": {
        "finder_location_aggs": {
          "filters": {
            "filters": {
              "loc": {
                "match":{
                  "finderArray.finder_location":"mumbai"
                }
              }
            }
          }
```

```
        }
       }
     }
   }}}
```

Improving Search results:

 # Proximity Searches:

-> In a match query for the text search we do **match_phrase** in that the order of the text should be in line but if we need to find some text for which you know the start & end text but nothing in middle so here comes the proximity search.

-> **slop** keyword is used so if you set the slop to 1 then it will shift the end word to place forward & gives the results of matches with one step forward as shown in below query.

```
GET /vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title",
  "query": {
    "match_phrase": {
      "title": {
        "query": "Sutherland Services",
        "slop": 1
      }
    }
  }
}
```

Output snippet :

-> As you can see I have searched for Sutherland Services and don't know the middle word so I have done this with a slop **keyword.**

Note: So while performing a search try to put a high number of slop values to get the maximum match results.

```
"hits" : [
     {
       "_index" : "vendor_index2020-06-02-21-03-55",
       "_type" : "_doc",
       "_id" : "yyKSc3IBafDnXOtRehcC",
       "_score" : 3.3900094,
       "_source" : {
         "title" : "Sutherland Global Services - BCIT"
       }
     },{
       "_index" : "vendor_index2020-06-02-21-03-55",
       "_type" : "_doc",
       "_id" : "zCKSc3IBafDnXOtRehcC",
       "_score" : 3.3900094,
       "_source" : {
         "title" : "Sutherland Global Services - Magarpatta"}
```

-> As shown in the below image what happens when slop =1 set & when slop =2 is set.

	Position 1	Position 2	Position 3
Document	spicy	tomato	sauce
Query	spicy	sauce	
Slop 1	spicy	→	sauce

	Position 1	Position 2	Position 3
Document	tomato	sauce	spicy
Query	spicy	sauce	
Slop 1	→	spicy|sauce	
Slop 2		sauce	→ spicy

Affecting Relevance scoring by proximity search:

-> Proximity search affect the scoring if you set the slop to 1 then matching doc result scoring would be higher than the setting the slop to 2.

-> So the higher the number of slop the lower the scoring of results.

-> This basically happens but it's no guarantee that lower the slop will higher the scoring as scoring is calculated by other relevant factors as well.

-> below query is basically OR on pasta & and so it has given the results that contains any of them.

```
GET /reciep/_search
{
  "_source": "title",
  "query": {
    "bool": {
      "must": [
        {
          "match": {
```

```
          "title": {
            "query": "pasta and"
          }
        }
      }
    ]
  }
 }
}
```

Output :

-> Just look at the score of the matching doc.

```
   "hits" : [
     {
       "_index" : "reciep",
       "_type" : "_doc",
       "_id" : "13",
       "_score" : 1.3769004,
       "_source" : {
         "title" : "Pesto Pasta With Potatoes and Green Beans"
       }
     },
     {
       "_index" : "reciep",
       "_type" : "_doc",
       "_id" : "19",
       "_score" : 1.3769004,
       "_source" : {
         "title" : "Pasta With Mushrooms, Brussels Sprouts, and Parmesan"
       }
     },
```

-> Now to boost the scoring we will use the proximity search.

```
GET /reciep/_search
{
  "_source": "title",
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "title": {
              "query": "pasta and"
            }
          }
        }
      ],
```

```
      "should": [
        {
          "match_phrase": {
            "title": {
              "query": "pasta and",
              "slop": 5
            }
          }
        }
      ]
    }
  }
}
```

Output :

-> Look at the scores now it is boosted.

```
"hits" : [
      {
        "_index" : "reciep",
        "_type" : "_doc",
        "_id" : "13",
        "_score" : 2.0297287,
        "_source" : {
          "title" : "Pesto Pasta With Potatoes and Green Beans"
        }
      },
      {
        "_index" : "reciep",
        "_type" : "_doc",
        "_id" : "19",
        "_score" : 1.8047401,
        "_source" : {
          "title" : "Pasta With Mushrooms, Brussels Sprouts, and Parmesan"
        }
      },
```

 # Fuzzy Query:

-> Fuzzy query is useful basically when the user makes a spelling mistake while searching.

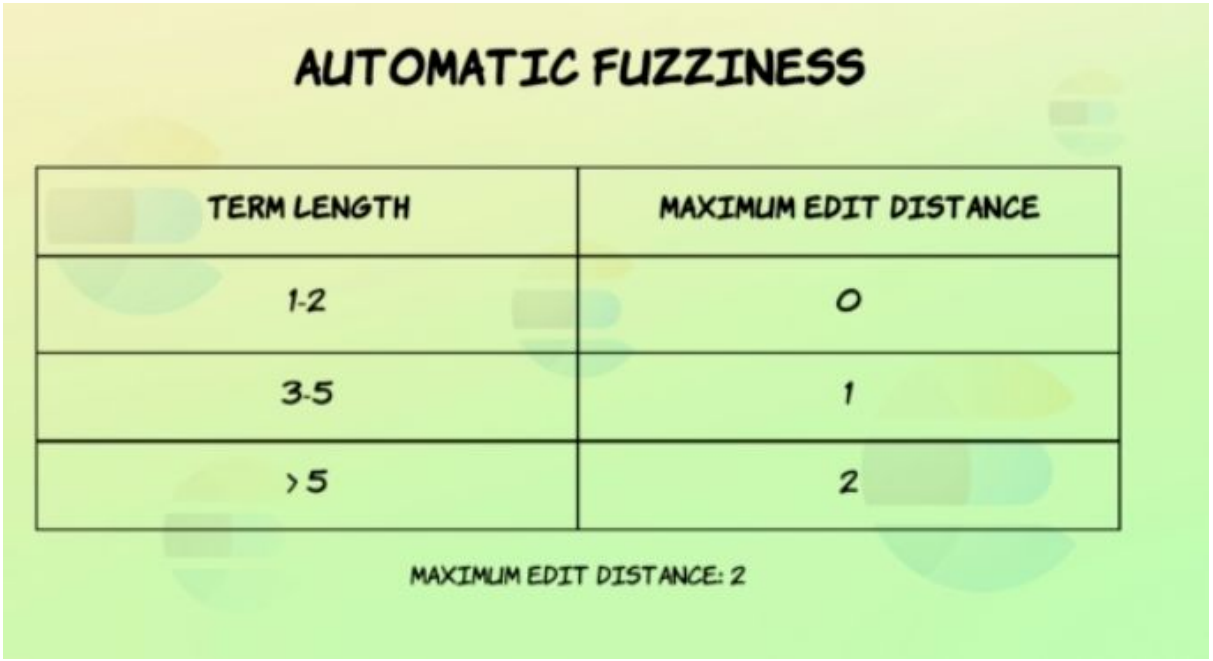
-> Running below query without fuzziness will return null output as there is a spelling error.

-> But by using fuzziness it resolves the problem and returns the result.

```
GET vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": {
```

```
      "query": "sutherlnd",
      "fuzziness": "auto"
    }
  }
 }
}
```

-> Fuzziness works, the edit is required to do the changes, that changes could be addition or deletion of words.
-> If you set the Fuzziness to auto & character is greater than 5 then max 2 edits are allowed.
-> Edits that require more than 2 changes will not work & result will be null.
-> If in your query you have used Fuzziness and the user enters the right value as well but if some is there in which just 1 or 2 changes & doc could match it will show in the results.
Like Lobster & oyster
-> Fuzziness is applied to each & every term of doc.



-> Transposition is by default enabled while doing Fuzziness query.
-> transposition is basically if while searching if change the word synchronization like AB to BA then also it will work.
-> As shown in the below query i have misspelled the word but it still worked because of transposition.
-> by default transposition is enabled you can disable it as well but then misspelled word will not work. "fuzzy_transpositions": "false"

```
GET vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": {
```

```
      "query": "suthelrnad",
      "fuzziness": "auto"
    }
   }
  }
}
```

Fuzzy Query:
-> IF the user has by mistake on the caps lock & we have used the fuzzy query then the result will be null as fuzzy query works like term query.
-> So it's better to use a match query with fuzziness.

```
GET vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title",
  "query": {
    "fuzzy": {
      "title": {
        "value":"SUTHERLAND",
        "fuzziness": "auto"
      }
    }
  }
}
```

 # synonyms:
-> Synonyms as name suggest exactly or nearly the same meaning of words.
-> so while creating settings & mapping of index, for some words we can set the synonyms of that word as well so when a user searches that word or synonyms of that word it will give the results.
-> So in the below query created the custom synonym filter & custom synonym analyzer.

```
PUT /synonyms
{
  "settings": {
    "analysis": {
      "filter": {
        "synonyms_filter":{
          "type":"synonym",
          "synonyms":[
            "awful => terrible",
            "awesome => great, super",
```

```
          "elasticsearch, logstash, kibana => elk",
          "weird, strange"
          ]
       }
     },
     "analyzer": {
       "synonyms_custom_analyzer":{
         "tokenizer":"standard",
         "filter":[
          "lowercase",
          "synonyms_filter"
          ]
       }
     }
   }
 },
 "mappings": {
   "properties": {
     "description": {
       "type": "text",
       "analyzer": "synonyms_custom_analyzer"
     }
   }
 }
}
```

```
POST /synonyms/_analyze
{ "analyzer": "synonyms_custom_analyzer",
  "text": "Elasticsearch is awesome, but can also seem weird sometimes."
}
```

Output :

-> As you can see **Elasticsearch is replaced with elk.**

-> So you can add elk or elasticsearch both will work in search.

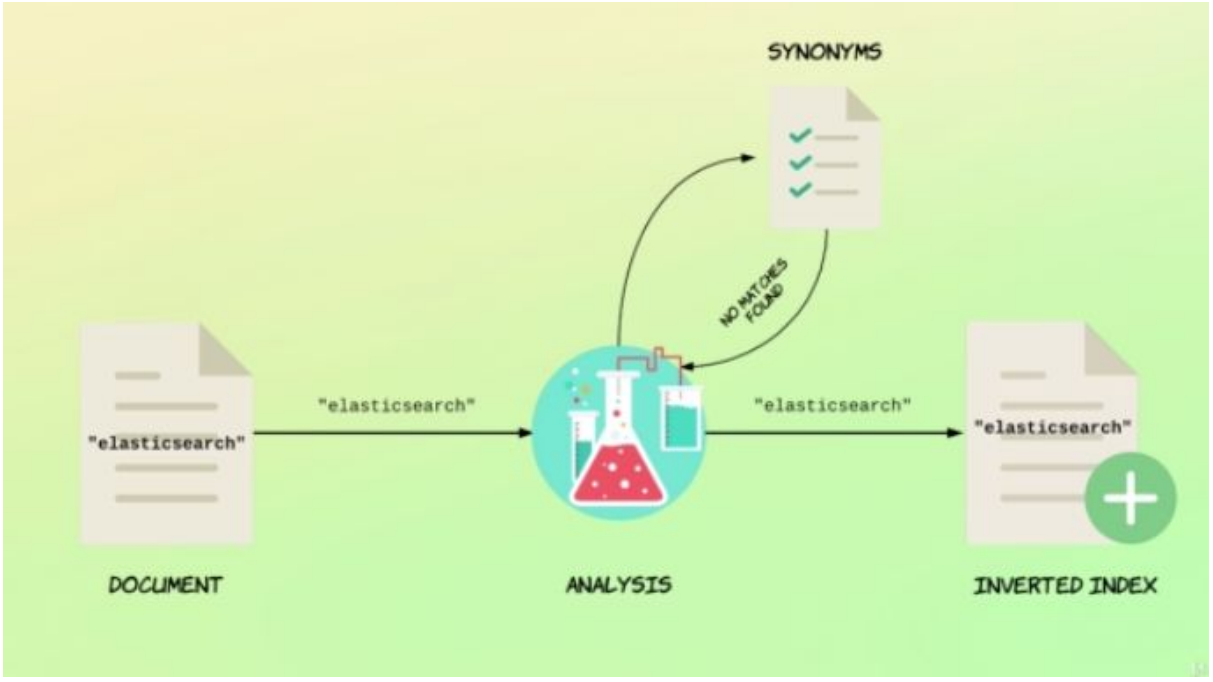
```
"tokens" : [{ "token" : "elk",
     "start_offset" : 0,
     "end_offset" : 13,
     "type" : "SYNONYM",
     "position" : 0},
```

Adding synonyms from file

-> So there might be a case when there are lots of synonyms which need to be added so instead of adding that in the query we can give the path of that synonyms file.

```
"filter": {"synonym_test": {
         "type": "synonym",
         "synonyms_path": "analysis/synonyms.txt" }}
```

-> If you add the synonyms to the existing index so documents which are previously present won't be part of synonyms. So this could be problematic. So always create a new index if you need to add the synonyms or for an existing index do it via **_update_by_query as doc is reindexed**.



 # Highlighting matches in fields:
-> Like most of the search engines or google search does highlight the search field while showing that same thing we can also achieve using highlight.
-> Elastic search highlights the synonyms of word as well.

```
GET  vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": "fitness"
    }
  },
  "highlight": {
    "pre_tags": ["<b>"],
    "post_tags": ["</b>"],
    "fields": {
      "title": {}
    }
  }
}
```

Output:
-> As you can see fitness is wrapped with a <b> tag, this something we defined in pre_tags & post_tags.

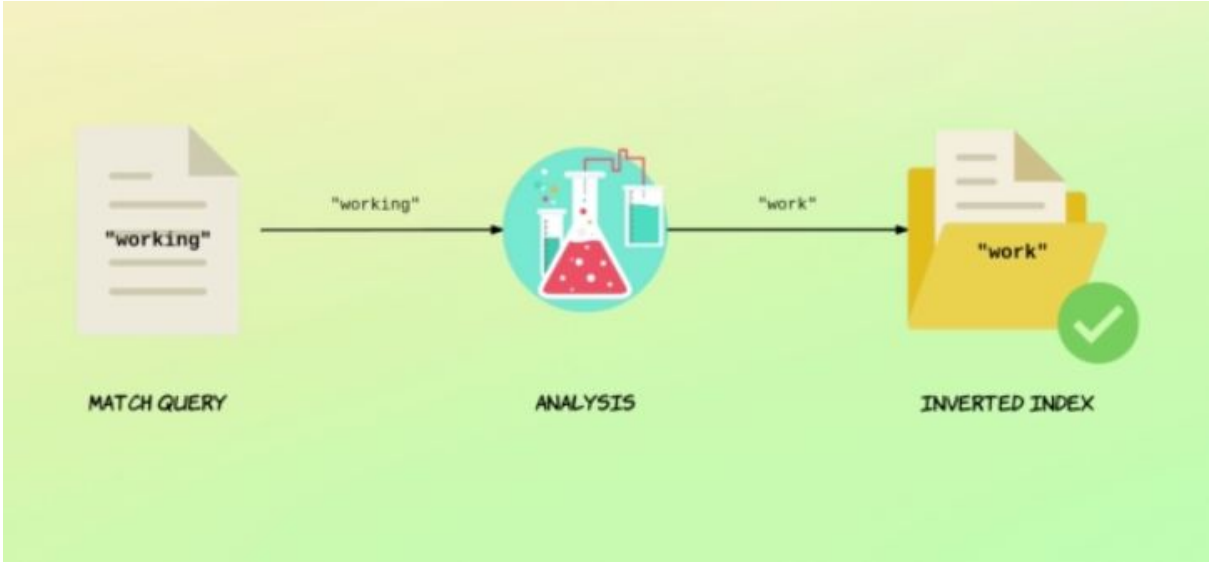
```
"_source" : {
        "title" : "Dynamite Fitness"
      },
      "highlight" : {
        "title" : [
          "Dynamite <b>Fitness</b>"
        ]
      }
```

Note: Highlighting is a deep topic so please refer below url for deep learning.
https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-body.html#request-body-search-highlighting

 # Stemming:

-> In **Stem analyzer** when the sentence is entered it converts all the words into their root form like **loved & drinking** will be stored as **love & drink** & while searching as well if **stem analyzer** is used then it will convert it to its root form & perform the search.

-> When the stemming is applied the root form is stored in the inverted index & original word is stored as offset. So searching with **loved or love** will give the same results.



-> In below query we have created the settings with a combination of synonym & stemming.

```
PUT /stemming_test
{
  "settings": {
```

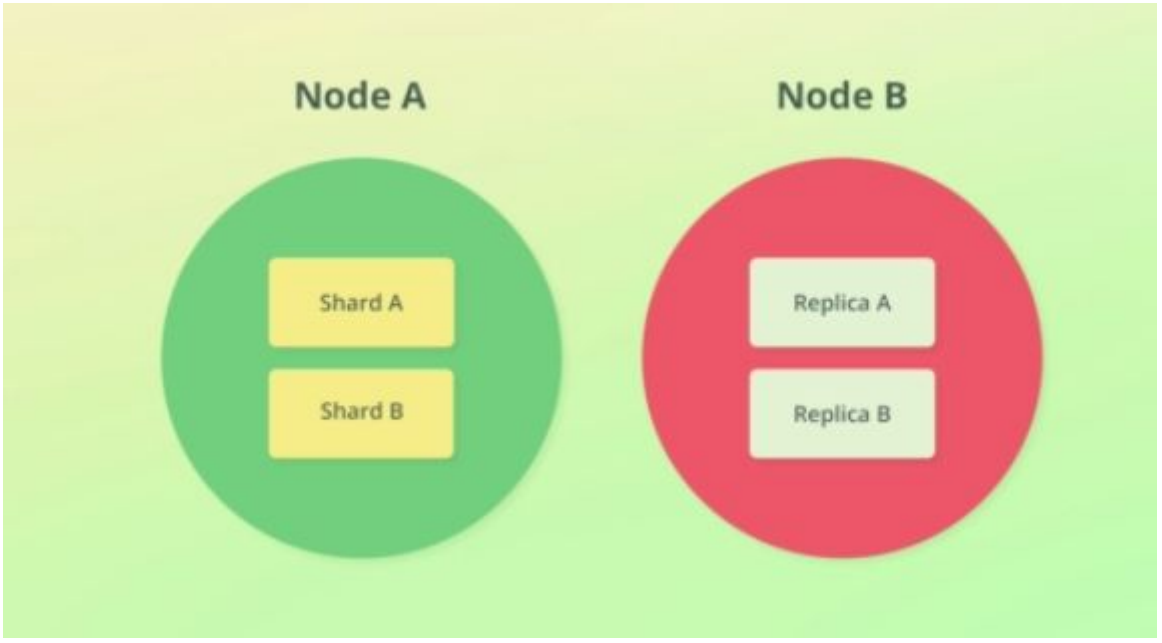
```
    "analysis": {
      "filter": {
        "synonym_test": {
          "type": "synonym",
          "synonyms": [
            "firm => company",
            "love, enjoy"
          ]
        },
        "stemmer_test" : {
          "type" : "stemmer",
          "name" : "english"
        }
      },
      "analyzer": {
        "my_analyzer": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "synonym_test",
            "stemmer_test"
          ]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "description": {
        "type": "text",
        "analyzer": "my_analyzer"
      }
    }
  }
}
PUT /stemming_test/_doc/1
{
"description": "I love working for my firm!"
}

GET /stemming_test/_search
{
  "query": {
    "match": {
      "description": "enjoy work"
    }
```

```
  }
}
```

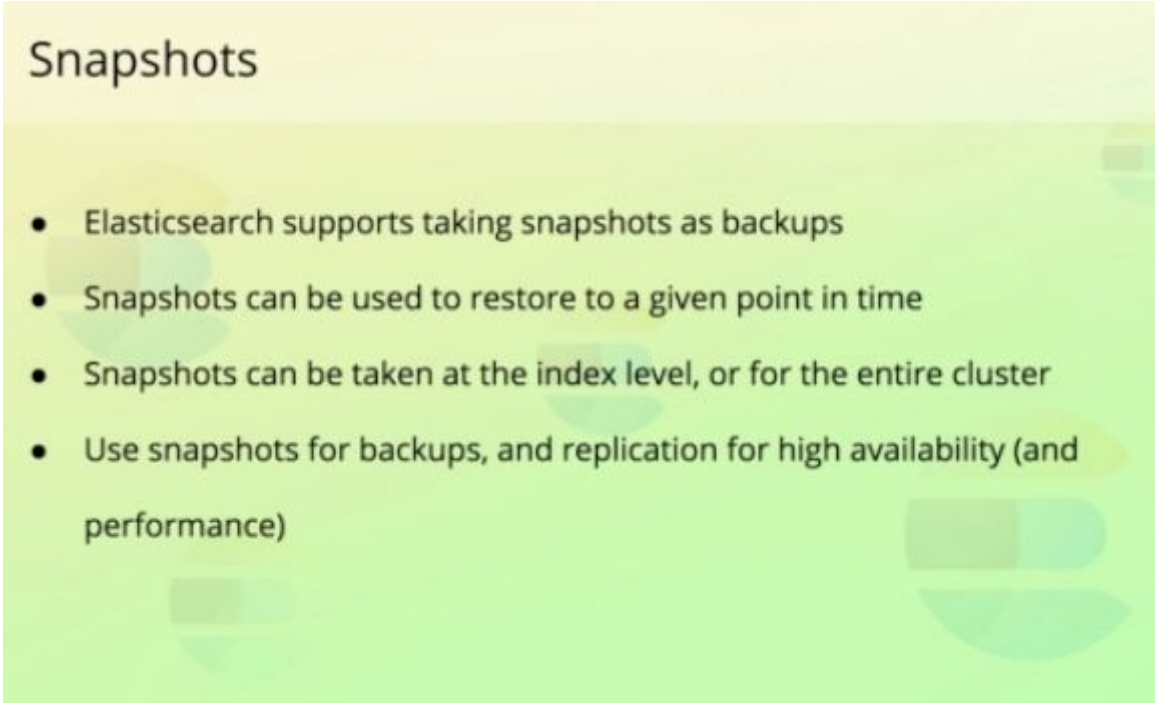
Replication:

-> Replication is always stored on different nodes & not on the primary node ,as if the primary node gets down the replication node will be available.



Snapshot:

-> Snapshots are basically used to restore the data at any given point in time.

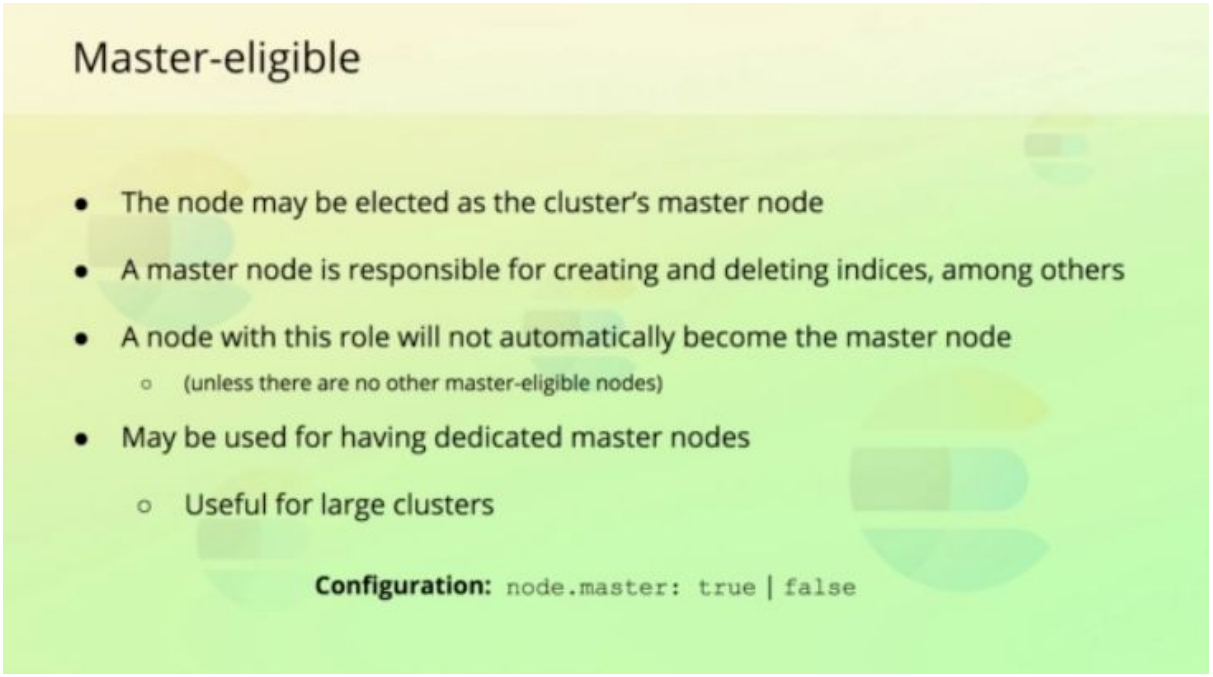


NOTE:
-> Status **yellow** means that the shard has a replica which is unassigned. So you need to assign the node for that replica to get them used.
-> Status **Green** means either there is no replica created or a replica is assigned.
GET _cluster/health
{

```
  "cluster_name" : "elasticsearch",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 15,
  "active_shards" : 15,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 12,
  "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" : 55.55555555555556
}
```


Nodes & Adding Node to cluster:
-> Data is stored on shards & shards are stored on nodes.
-> To add the node to the cluster is quite simple, open the elasticsearch.yml file & in that there is node parameter add it over there and then start the elasticsearch again.
-> So after adding the additional node you can the health status it will be green now. An additional node is assigned to the replica shared.

Node Roles:
-> Node roles which we can see in _cat/indies?v API is DIM(data, ingest,master-eligible)
-> Master-eligible
-> Data
-> ingest(https://www.elastic.co/guide/en/elasticsearch/reference/7.3/ingest.html)
-> coordination
-> voting-only



Queries :
https://github.com/codingexplained/complete-guide-to-elasticsearch

#query for bulk insert from json file
POST /reciep/_bulk -binary @test-data.json

GET /products/_doc/895/_explain
{
"query": {
"query_string": {"query": "description:Lorem"}

}
}

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

GET /products/_search?q=tags:vegetable and name = onions

POST /reviews/_doc
{
"upsert":{
"name":"bat",

```
    "tournament":"tennis",
    "willow_type":["kashmiri"],
    "instock":10
  }
}


POST /products/_update_by_query
{
  "script":{
    "source":"ctx._source.instock--"
  },
 "query": {
    "match_all": {}
  }
}

# Bulk operation & creating index if not exists
POST /_bulk
{"index":{"_index":"products", "_id":100}}
{"name":"Ball","type":"season","price":500, "instock":20}
{"create":{"_index":"products", "_id":101}}
{"name":"Ball","type":"tennis","price":60, "instock":20}

# Bulk Operation on specific index
POST /products/_bulk
{"update":{"_id":100}}
{"doc":{"price":750}}
{"delete":{"_id":101}}

# retreive Shards
GET /_cat/shards?v

# Creating mapping
PUT /products/_mapping
{
  "properties":{
    "discount":{
      "type":"double"
    }
  }
}
# to retreive mapping
GET /products/_mapping

# Standard tokenizer
```

```
POST _analyze
{
  "tokenizer": "standard",
  "text" : "I'm in mood to drink wine"
}

#lowercase filter
POST _analyze
{
  "filter": ["lowercase"],
  "text" : "I'm in mood to drink wine"
}

# Standard Analyzer example
POST _analyze
{
  "analyzer": "standard",
  "text" : "I'm in mood to drink wine"
}

# Standard tokenizer with lowercase filter
POST _analyze
{
  "char_filter": [],
  "tokenizer": "standard",
  "text" : "I'm in mood to drink wine",
  "filter": ["lowercase"]
}

# Keyword Example
POST _analyze
{
  "text" : "text are stored as it is in keyword analyzer",
  "analyzer": "keyword"

}

#Storing array
POST _analyze
{
  "text" : ["text are stored as" ,"it is in keyword analyzer"],
  "analyzer": "standard"

}

#Adding Explicts mapping
```

```
PUT /reviews
{
  "mappings": {
    "properties": {
      "product_id":{"type": "integer"},
      "content" :{"type": "text"},
      "ratings":{"type": "float"},
      "authors" :{
        "properties": {
          "first_name":{"type":"text"},
          "last_name":{"type":"text"},
          "email_id" :{"type":"keyword"}
       }
      }
    }
  }
}

#inserting doc in reviews table
POST /reviews/_doc
{
  "product_id":4,
  "content":"text content",
  "rating": 4.2,
  "authors.first_name":"swapnil",
  "authors.last_name":"pal",
  "authors.email_id":"swapnil@gmail.com",
  "review_source_new":["Google","FaceBook","Youtube"],
  "source_of_origin":["Google","FaceBook","Youtube"]
}

# Using dotnotaion for mapping
PUT /reviews_dotnotaion
{
  "mappings": {
    "properties": {
      "product_id":{"type": "integer"},
      "content" :{"type": "text"},
      "ratings":{"type": "float"},
      "authors.first_name":{"type":"text"},
      "authors.last_name":{"type":"text"},
      "authors.email_id" :{"type":"keyword"}
    }
  }
}
```

```
#retreving mapping for specific fields
GET /reviews/_mapping/field/authors.email_id
GET /reviews/_mapping

#Adding mapping to existing indices
PUT /reviews/_mapping
{
  "properties":{
    "created_at":{"type":"date"}
  }
}

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

POST /_reindex
{
  "source":{
   "index":"reviews"
  },
  "dest": {
    "index":"reviews_new"
  },
  "script": {
    "source": """
      if(ctx._source.product_id != null){
        ctx._source.product_id = ctx._source.product_id.toString();
      }
    """
  }
}

PUT /reviews/_mapping
{
    "properties":{
      "source_of_origin":{
        "type":"text",
        "fields":{
          "keyword":{
            "type" :"keyword"
          }
        }
```

```
    }
  }
}

#multi-field mapping query
GET /reviews/_search
{
  "query": {
    "match": {
      "source_of_origin": "google"
    }
  }
}

GET /reviews/_search
{
  "query": {
    "term": {
      "source_of_origin.keyword": "Google"
    }
  }
}

#creating custom analyzer
POST /vendor_index2020-05-24-21-03-55/_close
PUT /vendor_index2020-05-24-21-03-55
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer":{
          "type":"custom",
          "char_filter":[
            "html_strip"
            ],
          "tokenizer":"standard",
          "filter":[
            "lowercase",
            "asciifolding",
            "stop"
            ]
        }
      }
    }
  }
}
```

```
POST /custom_analyzer_reviews/_analyze
{
  "analyzer":"my_custom_analyzer",
  "text":"<b>Checking Custom analyzer</b> will be working or NÑot"
}

#Adding analyzer to existing indices
PUT /vendor_index2020-05-24-21-03-55/_settings
{
  "analysis":{
    "analyzer": {
       "second_custom_analyzer":{
         "type":"custom",
         "char_filter":[
           "html_strip"
           ],
         "tokenizer":"standard",
         "filter":[
           "lowercase",
           "asciifolding",
           "stop"
           ]
       }
     }
  }
}
POST /custom_analyzer_reviews/_close
POST /custom_analyzer_reviews/_open
#To check the custom analyzer added successfully
GET /custom_analyzer_reviews/_settings

DELETE /vendor_index2020-05-10-21-10-18

#Updating the analyzer
PUT /custom_analyzer_reviews/_mapping
{
  "properties":{
    "descrption":{
      "type":"text",
      "analyzer":"my_custom_analyzer"
    }
  }
}

POST /custom_analyzer_reviews/_doc
```

```
{
  "description":"<b>Checking Custom analyzer</b> will be working or NÑot"
}
GET /custom_analyzer_reviews/_search
{
  "query": {
    "match": {
      "description": {
        "query": "will",
        "analyzer": "keyword"
      }
    }
  }
}
#terms(IN query basically) Query
GET /products/_search
{
  "query":{
    "terms": {
      "tags.keyword": [
        "Meat",
        "Soup"
        ]
    }
  }
}
#Range Query
GET /products/_search
{
  "query": {
    "range": {
      "sold": {
        "lte": 10
      }
    }
  }
}

GET /products/_search
{
  "query": {
    "range": {
      "created": {
        "gte": "02/08/2005",
        "lte": "02/05/2008",
        "format": "dd/MM/yyyy"
```

```
    }
   }
  }
}

GET /products/_search
{
  "query": {
    "range": {
      "created": {
        "gte": "02/08/2005||+1y+1d+1m",
        "lte": "02/05/2008||-1y",
        "format": "dd/MM/yyyy"
      }
    }
  }
}
#exists true which also checks not-null values
GET /products/_search
{
  "query": {
    "exists": {
      "field": "tags"
    }
  }
}

GET /products/_search
{
  "query": {
    "wildcard": {
      "name":"past*"
    }
  }
}
GET /products/_search
{
  "query": {
    "match": {
      "name": "pasta"
    }
  }
}

GET /products/_search
{
```

```
  "query": {
    "regexp": {
      "tags.keyword":"Vege[a-zA-Z]+able"
    }
  }
}
#Full text Search
GET /reciep/_search
{
  "query": {
    "match": {
      "title": "pasta with parmesan and spinach"
    }
  }
}

GET /products/_search
{
  "query": {
    "match": {
      "name": {
        "query": "Pastry banana",
        "operator": "and"
      }
    }
  }
}
#matching pharse
GET /reciep/_search
{
  "query": {
    "match_phrase": {
      "title": "pasta carbonara"
    }
  }
}
#multi match
GET /reciep/_search
{
  "query": {
    "multi_match": {
      "query": "pasta pesto",
      "fields": ["title","description"]
    }
  }
}
```

```
#boolean Query
GET /reciep/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "ingredients.name": "pasta"
          }
        },
        {
          "range": {
            "preparation_time_minutes": {
              "lte": 15
            }
          }
        }
      ]
    }
  }
}

GET /reciep/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "ingredients.name": {
              "query": "pasta",
              "_name" :"pasta_must"
            }
          }
        }
      ],
      "should": [
        {
          "match": {
            "ingredients.name": {
              "query": "Parmesan",
              "_name": "Parmesan_should"
            }
          }
        }
```

```
      ],
      "filter": [
        {
          "range": {
            "preparation_time_minutes": {
              "lte": 15,
              "_name" :"prep_time_filter"
            }
          }
        }
      ]
    }
  }
}
#Adding analyzer to existing indices
PUT /vendor_index2020-06-02-21-03-55/
POST /vendor_index2020-06-02-21-03-55/_close
PUT /vendor_index2020-06-02-21-03-55/_settings
{
  "analysis":{
    "analyzer": {
       "my_custom_analyzer":{
         "type":"custom",
         "char_filter":[
           "html_strip"
           ],
         "tokenizer":"standard",
         "filter":[
           "lowercase",
           "asciifolding",
           "stop"
           ]
       }
     }
  }
}
POST /vendor_index2020-06-02-21-03-55/_open
PUT /vendor_index2020-06-02-21-03-55/_mapping
{
    "properties": {
      "finder_id":{
        "type": "integer"
      },
      "title":{
        "type": "text",
        "fields":{
```

```
    "keyword":{
      "type" :"keyword"
    }
  },
  "analyzer":"my_custom_analyzer"
},
"slug":{
  "type": "text",
  "fields": {
    "keyword":{
      "type":"keyword"
    }
  },
  "analyzer":"my_custom_analyzer"
},
"country_name":{
  "type": "text",
  "fields": {
    "keyword":{
      "type":"keyword"
    }
  },
  "analyzer":"my_custom_analyzer"
},
"city_name":{
  "type": "text",
  "fields": {
    "keyword":{
      "type" :"keyword"
    }
  },
  "analyzer":"my_custom_analyzer"
},
"category_meta_title":{
  "type": "text",
  "fields": {
    "keyword":{
      "type" :"keyword"
    }
  },
  "analyzer":"my_custom_analyzer"
},
"category_meta_description":{
  "type": "text",
  "analyzer":"my_custom_analyzer"
},
```

```
      "category_meta_keywords":{
        "type": "keyword"
      },
      "location_name":{
        "type": "text",
        "analyzer":"my_custom_analyzer"
      },
      "finderArray":{
        "type":"nested"
      }
    }

}
GET /vendor_index2020-05-10-20-58-03/_mapping

GET /vendor_index2020-06-02-21-03-55/_search?format=json
{
  "_source": "false",
  "from": 0,
  "size": 20,
  "query": {
    "nested": {
      "path": "finderArray",
      "inner_hits": {},
      "query": {
        "bool": {
          "must": [
            {
              "match": {
                "finderArray.finder_name": "sutherland"
              }
            }
          ],
          "filter": {
            "terms": {
              "finderArray.finder_location": ["malad","hadapsar"]
            }
          }
        }
      }
    }
  },
  "sort": [
    {
      "slug": "desc"
    }
```

```
 ]

}
DELETE /department
PUT /company
{
  "mappings": {
    "properties": {
      "join_field": {
        "type": "join",
        "relations": {
          "company":["department","supplier"],
          "department": "employee"
        }
      }
    }
  }
}
GET /company/_mapping
PUT /company/_doc/1
{
  "name" :"abc corp",
  "join_field":"company"
}

PUT /company/_doc/2?routing=1
{
  "name": "development",
  "join_field": {
    "name": "department",
    "parent": 1
  }
}

PUT /company/_doc/3?routing=1
{
  "name":"swa",
  "join_field":{
    "name":"employee",
    "parent" :2
  }
}

GET /company/_search
{
  "query": {
```

```
    "match_all": {}
  }
}
GET /company/_search
{
  "query": {
    "has_child": {
      "type": "department",
      "query": {
        "has_child": {
          "type": "employee",
          "query": {
            "match": {
              "name": "swa"
            }
          }
        }
      }
    }
  }
}

GET /department/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "score_mode": "sum",
      "query": {
        "bool": {
          "must": [
            {
              "range": {
                "age": {
                  "gte": 40
                }
              }
            }
          ],
          "should": [
            {
              "term": {
                "gender.keyword":"M"
              }
            }
          ]
```

```
        }
      }
    }
  }
}

PUT /order
{
  "mappings": {
    "properties": {
      "purchased_at": {
        "type": "date"
      },
      "lines": {
        "type": "nested",
        "properties": {
          "product_id": {
            "type": "integer"
          },
          "amount": {
            "type": "double"
          },
          "quantity": {
            "type": "short"
          }
        }
      },
      "total_amount": {
        "type": "double"
      },
      "status": {
        "type": "keyword"
      },
      "sales_channel": {
        "type": "keyword"
      },
      "salesman": {
        "type": "object",
        "properties": {
          "id": {
            "type": "integer"
          },
          "name": {
            "type": "text"
          }
        }
```

```
      }
    }
  }
}

GET /order/_search
{
  "size": 0,
  "aggs": {
    "sum_amount": {
      "sum": {
        "field": "total_amount"
      }
    },
    "avg_amount" :{
      "avg": {
        "field": "total_amount"
      }
    },
    "min_amount":{
      "min": {
        "field": "total_amount"
      }
    },
    "max_amount" :{
      "max": {
        "field": "total_amount"
      }
    }
  }
}
#cardinality metrics
GET /order/_search
{
  "size": 0,
  "aggs": {
    "total_sales_person": {
      "cardinality": {
        "field": "salesman.id",
        "precision_threshold": 100
      }
    }
  }
}

GET /order/_search
```

```
{
  "size": 0,
  "aggs": {
    "total_sales_person": {
      "value_count": {
        "field": "total_amount"
      }
    }
  }
}
#multilevel metric aggs
GET /order/_search
{
  "size": 0,
  "aggs": {
    "multilevel_aggs_stats": {
      "stats": {
        "field": "total_amount"
      }
    }
  }
}

GET /order/_search
{
  "size": 0,
  "aggs": {
    "bucket_level_aggs": {
      "terms": {
        "field": "status",
        "min_doc_count": 0,
        "order": {
          "_key": "asc"
        }
      }
    }
  }
}
#Nested Aggregation
GET /order/_search
{
  "size": 0,
  "aggs": {
    "status_stats": {
      "terms": {
        "field": "status"
```

```
    },
    "aggs": {
      "min_amount_filter":{
        "filter": {
          "range": {
            "total_amount": {
              "gte": 100
            }
          }
        },
        "aggs": {
          "total_cal": {
          "stats": {
            "field": "total_amount"
          }
        }
        }
      }
    }
  }
}
GET vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title"
}
GET vendor_index2020-06-02-21-03-55/_search
{
  "size": 0,
  "aggs": {
    "location_bucket": {
      "filters": {
        "filters": {
          "mumbai_bucket": {
              "match": {
                "city_name": "mumbai"
              }
          },
          "pune_bukcet":{
              "match": {
                "city_name": "pune"
              }
          }
        }
      }
    }
```

```
  }
}
GET order/_search
GET order/_search
{
  "size": 0,
  "aggs": {
    "range_aggregation": {
      "date_range": {
        "field": "purchased_at",
        "format": "yyy-MM-dd",
        "keyed": true,
        "ranges": [
          {
            "from": "2016-01-01",
            "to": "2016-01-01||+6m",
            "key": "first_half"
          },
          {
            "from": "2016-01-01||+6m",
            "to": "2016-01-01||+1y",
            "key": "second_half"
          }
        ]
      }
    }
  }
}

GET /order/_search
{
  "size": 0,
  "aggs": {
    "histogram_agg": {
      "histogram": {
        "field": "total_amount",
        "interval": 25,
        "extended_bounds": {
          "min": 0,
          "max": 400
        }
      }
    }
  }
}
```

```
GET /order/_search
{
  "size": 0,
  "aggs": {
    "histogram_agg": {
      "date_histogram": {
        "field": "purchased_at",
        "interval": "quarter"
      }
    }
  }
}

GET /order/_search
{
  "size": 0,
  "query": {
    "range": {
      "total_amount": {
        "gte": 100
      }
    }
  },
  "aggs": {
    "global_aggregation": {
      "global": {}
    }
  }
}

GET /vendor_index2020-06-02-21-03-55/_search
{
  "size": 0,
  "aggs": {
    "missing_field": {
      "missing": {
        "field": "created_date"
      }
    }
  }
}

GET /vendor_index2020-06-02-21-03-55/_search
{
  "size": 0,
  "aggs": {
```

```
    "nested_aggregations": {
      "nested": {
        "path": "finderArray"
      },
      "aggs": {
        "finder_location_aggs": {
          "filters": {
            "filters": {
              "loc": {
                "match":{
                  "finderArray.finder_location":"mumbai"
                }
              }
            }
          }
        }
      }
    }
  }
}

GET /reciep/_search
{
  "_source": "title",
  "query": {
    "match_phrase": {
      "title": {
        "query": "pasta and",
        "slop": 2
      }
    }
  }
}

GET /reciep/_search
{
  "_source": "title",
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "title": {
              "query": "pasta and"
            }
          }
```

```
      }
     ],
     "should": [
      {
       "match_phrase": {
        "title": {
         "query": "pasta and",
         "slop": 5
        }
       }
      }
     ]
    }
   }
}

GET vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": {
        "query": "suthelrnad",
        "fuzziness": "auto"
      }
    }
  }
}

GET vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": {
        "query":"SUTHERLAND",
        "fuzziness": "auto"
      }
    }
  }
}

PUT /synonyms
{
  "settings": {
    "analysis": {
```

```
      "filter": {
        "synonyms_filter":{
          "type":"synonym",
          "synonyms":[
            "awful => terrible",
            "awesome => great, super",
            "elasticsearch, logstash, kibana => elk",
            "weird, strange"
            ]
        }
      },
      "analyzer": {
        "synonyms_custom_analyzer":{
          "tokenizer":"standard",
          "filter":[
            "lowercase",
            "synonyms_filter"
            ]
        }
      }
    },
    "mappings": {
      "properties": {
        "description": {
          "type": "text",
          "analyzer": "synonyms_custom_analyzer"
        }
      }
    }
}

POST /synonyms/_analyze
{
  "analyzer": "synonyms_custom_analyzer",
  "text": "Elasticsearch is awesome, but can also seem weird sometimes."
}

GET  vendor_index2020-06-02-21-03-55/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": "fitness"
    }
  },
```

```
  "highlight": {
    "pre_tags": ["<b>"],
    "post_tags": ["</b>"],
    "fields": {
      "title": {}
    }
  }
}

GET _cluster/health
GET _cat/indices?v
```