```
#####################################################
# 2018/12
# Elasticsearch/Kibana: Version: 6.5.2
# reference: https://www.elastic.co/
# reference: Udemy: Complete Guide to Elasticsearch
#####################################################

# node: a server that stores data and is part of a cluster; default name is UUID
(Universally Unique Identifier)
# cluster: a collection of nodes (servers); default name is Elasticsearch
# document: each data item stored within a cluster is called document, being a basic
unit of information that can be indexed; documents are stored within indices;
documents are JSON objects
# index: an index is a collection of documents
# sharding divides indices into smaller pieces named shards; sharding enables you to
distribute data across multiple nodes within a cluster; sharding also increses
performance in cases where shards are distributed on multiple nodes because search
queries can then be parallelized
# Elasticsearch natively support replication of your shards, meaning that shards are
copied

# HTTP request methods
# GET: The GET method requests a representation of the specified resource. Requests
using GET should only retrieve data.
# POST: The POST method is used to submit an entity to the specified resource, often
causing a change in state or side effects on the server.
# PUT: The PUT method replaces all current representations of the target resource
with the request payload.
# HTTP verb: GET, POST, PUT, DELETE

# <REST verb>/<index>/<type>/<API>     # <Type> will be removed in later/newer
version of Elasticsearch

# create an index
PUT /examples_index_name/

# add/replace a document to an index
POST /examples_index_name/_doc
{
    "my_field_1": "my_value",
    "my_field_2": {
        "my_field_3": true
    },
    "my_field_4": "2018-01-23T13:24:35+0800",
    "my_field_5": null,
    "my_field_6": 12.34
}
# _id would be generated automatically, for example - "_id" : "_JLDuGoBjiKx_hYq32wY"

# add/replace a document with _id to an index
```

```
PUT /examples_index_name/_doc/<_id>
{
    "my_field_1": "my_value_1",
    "my_field_2": {
        "my_field_3": false
    },
    "my_field_6": 56.78
}

# retrive a document by _id
GET /examples_index_name/_doc/<_id>

# using update API to update a document
POST /examples_index_name/_doc/<_id>/_update
{
    "doc": {"my_field_1": "my_value_update", "my_field_new": ["my_value_new"]}
}

# scripted update; use "script" property instead of "doc" property
POST /examples_index_name/_doc/<_id>/_update
{
    "script": "ctx._source.my_field_6 += 10"
}

# documents in Elasticsearch are immutable and cannot be changed

# upsert; if the document exist, then my_field_1 will be added by 5; if not exist,
my_field_1 will be 100
POST /examples_index_name/_doc/<_id>/_update
{
    "script": "ctx._source.my_field_1 += 5",
    "upsert": {
        "my_field_1": 100
    }
}

# delete a document
DELETE /examples_index_name/_doc/<_id>

# use _delete_by_query API to delete multiple documents
POST /examples_index_name/_delete_by_query
{
    "query": {
        "match": {
            "my_field_1": "my_value_1"
        }
    }
}

# bulk API can be used for add, update, and delete documents
```

```
# batch processing to add documents
POST /examples_index_name/_doc/_bulk
{"index": {"_id": "100"}}
{"my_field_1": "my_value_1"}
{"index": {"_id": "101"}}
{"my_field_1": "my_value_2"}

# batch processing to update/delete documents
POST /examples_index_name/_doc/_bulk
{"update": {"_id": "100"}}
{"doc": {"my_field_1": 1000}}
{"delete": {"_id": "101"}}

# import data with cURL
$ curl -H "Content-Type: application/json" -XPOST
"http://localhost:9200/examples_index_name/_doc/_bulk?pretty" --data-binary
"@my_data_file_1.jsonl" && curl -H "Content-Type: application/json" -XPOST
"http://localhost:9200/examples_index_name/_doc/_bulk?pretty" --data-binary
"@my_data_file_2.jsonl"

$ java -version

# use cat API for information about cluster, node, index, etc..
GET /_cat/health?v    # v for verbose
GET /_cat/nodes?v
GET /_cat/indices?v
GET /_cat/allocation?v    # how many shards are bing allocated
GET /_cat/shards?v

# in Elasticsearch, mappings are used to define how documents and their fields
should be stored and indexed

GET /examples_index_name/_doc/_mapping

# each document has meta-data associated with it and they're called meta fields
# meta fields: _index, _id, _source, _field_names, _routing, _version, _meta
# field data types: core, complex, geo, specialized data types
# core: text, keyword (for filter/sort/aggregate), numeric, date, boolean
(true/false), binary, range
# complex: object, array, nested    # Lucene has no concept of inner objects
# geo: geo-point, geo-shape
# specialize: ip, completion, attchment

# create an index
PUT /examples
{
  "settings": {
    "index.number_of_shards": 1,
    "index.number_of_replicas": 0
  }
```

```
}

# create an index with mappings
PUT /examples_index_name/
{
    "mappings": {
        "my_default": {
            "dynamic": false,
            "properties": {
                "my_field_1": {
                    "type": "integer"
                },
                "my_field_2": {
                    "type": "boolean"
                },
                "latitude": {
                    "type":  "geo_point"
                }
            }
        }
    }
}

# mapping parameters: coerce (automatically cleaning up values), copy_to, dynamic,
properties, norms (used for relevance scores), format (for date fields), null_value,
fields

# add a mapping for new fields
PUT /examples_index_name/_doc/_mapping
{
    "properties: {
        "my_field_new_1": {
            "type": "date",
            "format": "yyyy/MM/dd HH:mm:ss||yyyy/MM/dd"
        },
        "my_field_new_2": {
            "type": "text",
            "fields": {
                "keyword": {
                    "type": "keyword"
                }
            }
        }
    }"
}

# existing mappings for fields cannot be updated

# analysis process involves tokenizing and normalizing a block of text
# text is tokenized into terms and terms are converted into lower case letters
```

```
(default behavior)
# results of analysis process are stored in inverted index
# inverted index is a mapping of a field's terms and which documents contain each
term

# analyzer contains 1. character filter, 2. tokenizer, and 3. token filter
# character filter: manipulate text before tokenization; HTML strip character
filter, mapping character filter, pattern replace
# tokenizer: split text into terms; word oriented tokenizers (standard, letter,
lowercase, whitespace, UAX URL Email), partial work tokenizers (N-Gram, Edge
N-Gram), structured text tokenizers (keyword, pattern, path)
# token filter: manipulate terms before adding them to an inverted index; standard,
lowercase, uppercase, NGram, Edge NGram, stop, word delimiter, stemmer (going ->
go), keyword marker, snowball, synonym, ASCII folding (résumé -> resume))
# built-in analyzer: standard, simple, stop, language, keyword, pattern, whitespace

# analyze API
POST _analyze
{
    "tokenizer": "standard",
    "filter": ["lowercase", "unique", "asciifolding"],
    "text": "My@email.com mY.email my_email my-email résumé"
}

GET _analyze
{
  "tokenizer": "letter",
  "filter": ["lowercase"],
  "text": "Brown.brown $brown fox $ brown @ dog https://www.google.com my@gmail.com"
}

GET _analyze
{
  "tokenizer": "uax_url_email",
  "filter": ["lowercase"],
  "text": "Brown.brown $brown fox $ brown @ dog https://www.google.com my@gmail.com"
}

# define a custome analyzer which uses standard analyzer (configure built-in
analyzers and token filters)
PUT /examples_custom_analyzer
{
    "settings": {
        "analysis": {
            "analyzer": {
                "my_english_stop": {
                    "type": "standard",
                    "stopwords": "_english_"
                }
            },
```

```
            "filter": {
                "my_stemmer": {
                    "type": "stemmer",
                    "name": "english"
                }
            }
        }
    }
}

POST /examples_custom_analyzer/_analyze
{
  "analyzer": "my_english_stop",
  "text": "I'm in the mood of drinking semi-dry red wine!"
}

POST /examples_custom_analyzer/_analyze
{
  "tokenizer": "standard",
  "filter": ["my_stemmer"],
  "text": "I'm in the mood of drinking semi-dry red wine!"
}

PUT /examples_custom_analyzer_2
{
    "settings": {
        "analysis": {
            "analyzer": {
                "my_english_stop": {
                    "type": "standard",
                    "stopwords": "_english_"
                },
                "my_custom_analyzer": {
                    "type": "custom",
                    "tokenizer": "standard",
                    "char_filter": [
                        "html_strip"
                    ],
                    "filter": [
                        "standard",
                        "lowercase",
                        "trim",
                        "my_stemmer"
                    ]
                }
            },
            "filter": {
                "my_stemmer": {
                    "type": "stemmer",
                    "name": "english"
```

```
                }
            }
        }
    }
}

POST /examples_custom_analyzer_2/_analyze
{
    "analyzer": "my_custom_analyzer",
    "text": "I'm in the mood for drinking <strong> semi-dry </strong> red wine! go
went fly flew"
}

# use analyzer in mapping
PUT /examples_custom_analyzer_2/_doc/_mapping
{
    "properties": {
        "my_field_1": {
            "type": "text",
            "analyzer": "my_custom_analyzer"
        },
        "my_field_2": {
            "type": "text",
            "analyzer": "standard"
        },
        "my_fields_3": {
            "type": "text"
        }
    }
}

POST /examples_custom_analyzer_2/_doc/1
{
    "my_field_1": "drinking",
    "my_field_2": "drinking"
}

# query below will not hit any documents, since my_field_1 is stemmed to drink
GET /examples_custom_analyzer_2/_search
{
  "query": {
    "term": {
      "my_field_1": "drinking"
    }
  }
}

# add analyzers to existing indices (close index and then add analyzers, and then
open index)
```

```
POST /examples_custom_analyzer_2/_close

PUT /examples_custom_analyzer_2/_settings
{
    "analysis": {
        "analyzer": {
            "my_french_stop": {
                "type": "standard",
                "stopwords": "_french_"
            }
        }
    }
}

POST /examples_custom_analyzer_2/_open

# the relevance scoring algorithm, OkAPI BM25, is currently used by Elasticsearch
(it considers term frequency, inverse document frequency, field-length norm)

GET /examples_index_name/_search
{
    "explain": true,
    "query": {
        "match": {
            "my_field_1": "my_value"
        }
    }
}

GET /examples_index_name/_doc/<_id>/_explain
{
  "query": {
    "match": {
      "my_field_1": "my_value"
    }
  }
}

# query context affects relevance; filter context doesn't

# full-text queries
# term queries (enum, numbers, dates, etc..)



############### term queries ###############
GET /examples_index_name/_doc/_search
{
    "query": {
        "term": {
```

```
            "my_field_1.keyword": "my_value"
        }
    }
}

GET /examples_index_name/_doc/_search
{
    "query": {
        "term": {
            "my_field_7": {
                "value": false
            }
        }
    }
}

# search for multiple terms
GET /examples_index_name/_doc/_search
{
    "query": {
        "terms": {
            "my_field_7": [
                    true,
                    false
                ]
        }
    }
}

# note: term/terms & my_field_7 is boolean type

# retrieve documents based on IDs
GET /examples_index_name/_doc/_search
{
    "query": {
        "terms": {
            "_id": ["1", "2", "3"]
        }
    }
}

# use range query for number or date
# example: date math & round time
#     2 year and one day before and round date to month
#     round current time to month and substract a year
GET /examples_index_name/_doc/_search
{
    "query": {
        "range": {
            "my_field_4": {
```

```
                "gte": "01-01-2013||-2y-1d/M",
                "lte": "now/M-1y",
                "format": "dd-MM-yyyy"
            }
        }
    }
}

# match document with non-null value
GET /examples_index_name/_doc/_search
{
    "query": {
        "exists": {
            "field": "my_field"
        }
    }
}

# match based on prefix
GET /examples_index_name/_doc/_search
{
    "query": {
        "prefix": {
            "my_field.keyword": "veget"
        }
    }
}

# search with wildcards (performance is slow)
# *: match any characters sequence including no characters
#    "vege*able": start with vege and end with able
# ?: match any single character
#    "vege?able"
GET /examples_index_name/_doc/_search
{
    "query": {
        "wildcard": {
            "my_field.keyword": "vege*able"
        }

    }
}

# search with regular expressions
# Elasticsearch uses Lucene regular expression engine
GET /examples_index_name/_doc/_search
{
    "query": {
        "regexp": {
            "my_field.keyword": "veget[a-zA-Z]+ble"
```

```
        }

    }
}
############### term queries ###############



########## full text queries ##########
# match query operates on terms

GET /examples_index_name/_doc/_search
{
    "query": {
        "match": {
            "my_field_1": {
                "query": "my_value",
                "operator": "and"
            }
        }
    }
}

# terms in specific order (exact phrase)
GET /examples_index_name/_doc/_search
{
    "query": {
        "match_phrase": {
            "my_field_1": "my_value"
        }
    }
}

# search multiple fields
GET /examples_index_name/_doc/_search
{
    "query": {
        "multi_match": {
            "query": "my_value",
            "fields": ["my_field_1", "my_field_2"]
        }
    }
}
########## full text queries ##########



########## compound queries ##########
GET /examples_index_name/_doc/_search
{
```

```
    "query": {
        "bool": {
            "must": [
                {
                    "match": {
                        "my_field_1": "my_value"
                    }
                },
                {
                    "range": {
                        "my_field_6": {
                            "lte": 15
                        }
                    }
                }
            ]
        }
    }
}

# move range query to filter object to improve efficiency
# if there is a match for "should", score will be boosted
# if there is a "must", "should" is optional and having a match for "should" will
boost score.
# if only has "should" for example no "must", then records have to match "should"
and "should" becomes required.

GET /examples_index_name/_doc/_search
{
    "query": {
        "bool": {
            "must": [
                {
                    "match": {
                        "my_field_1": "my_value"
                    }
                }
            ],
            "must_not": [
                {
                    "match": {
                        "my_field_1": "my_value_2"
                    }
                }
            ],
            "should": [
                {
                    "match": {
                        "my_field_1": "my_value_3"
                    }
```

```
                }
            ],
            "filter": [
                {
                    "range": {
                        "my_field_6": {
                            "lte": 15
                        }
                    }
                }
            ]
        }
    }
}

# "matched_queries" will show which my_identifier has a match in results
# "explain" can help debug ES DSL queries
GET /examples_index_name/_doc/_search
{
    "query": {
        "bool": {
            "must": [
                {
                    "match": {
                        "my_field_1": {
                            "query": "my_value",
                            "_name": "my_identifier_1"
                        }
                    }
                }
            ],
            "must_not": [
                {
                    "match": {
                        "my_field_1": {
                            "query": "my_value_2",
                            "_name": "my_identifier_2"
                        }
                    }
                }
            ],
            "should": [
                {
                    "match": {
                        "my_field_1": {
                            "query": "my_value_3",
                            "_name": "my_identifier_3"
                        }
                    }
                }
```

```
            ],
            "filter": [
                {
                    "range": {
                        "my_field_6": {
                            "lte": 15,
                            "_name": "my_identifier_4"
                        }
                    }
                },
                {
                    "term": {
                        "my_field_1.keyword": "my_value"
                    }
                }
            ]
        }
    }
}

# the two queries example 1 and 2 are the same
# one of them should match

# example 1:
GET /examples_index_name/_doc/_search
{
    "query": {
        "match": {
            "my_field_1": "my_value_1 my_value"
        }
    }
}

# example 2:
GET /examples_index_name/_doc/_search
{
    "query": {
        "bool": {
            "should": [
                {
                    "term": {
                        "my_field_1": "my_value_1"
                    }
                },
                {
                    "term": {
                        "my_field_1": "my_value"
                    }
                }
            ]
```

```
        }
    }
}

# in elasticsearch, you generally denormalize data for best performance instead of
mapping document relationships
# don't use Elasticsearch as a primary data store
# Elasticsearch only supports simple joins (joins are expensive)
########## compound queries ##########



########## nested ##########
# query type: nested

PUT /examples_nested
{
    "mappings": {
        "_doc": {
            "properties": {
                "name": {
                    "type": "text"
                },
                "employees": {
                    "type": "nested"
                }
            }
        }
    }
}

# if employees aren't defined as nested datatype, association among Eric, 39, M
would loss
POST /examples_nested/_doc/1
{
    "name": "Development",
    "employees": [
        {
            "name": "Eric",
            "age": 39,
            "gender": "M"
        },
        {
            "name": "John",
            "age": 20,
            "gender": "M"
        }
    ]
}
```

```
POST /examples_nested/_doc/2
{
    "name": "HHR",
    "employees": [
        {
            "name": "Maria",
            "age": 30,
            "gender": "F"
        },
        {
            "name": "Ryan",
            "age": 25,
            "gender": "M"
        }
    ]
}

GET /examples_nested/_search
{
    "query": {
        "nested": {
            "path": "employees",
            "query": {
                "term": {
                    "employees.gender.keyword": "M"
                }
            }
        }
    }
}
# note that result is 2 instead of 3; might need to loop through each object to get
total counts of 3
# number of department has M is 2
# number of employees with gender = M is 3

# nested inner hit
GET /examples_nested/_search?filter_path=hits.hits.inner_hits.employees.hits.total
{
    "_source": false,
    "query": {
        "nested": {
            "path": "employees",
            "inner_hits": {},
            "query": {
                "term": {
                    "employees.gender.keyword": "M"
                }
            }
        }
    }
```

```
}
########## nested ##########



########## join ##########
# define parent/child relationship for my_department/my_employee
PUT /examples_join
{
    "mappings": {
        "_doc": {
            "properties": {
                "my_join_field": {
                    "type": "join",
                    "relations": {
                        "my_parent_department": "my_child_employee"
                    }
                }
            }
        }
    }
}

PUT /examples_join/_doc/2
{
    "my_name": "R&D",
    "my_join_field": "my_parent_department"
}

PUT /examples_join/_doc/3
{
    "my_name": "HHR",
    "my_join_field": "my_parent_department"
}

# parent/child documents must be stored on the same shard
# routing: in which shard a document with a given ID is stored
# using parent routing id = 2 in an example
PUT /examples_join/_doc/4?routing=2
{
    "my_name": "Bo R&D",
    "my_age": 10,
    "my_gender": "F",
    "my_join_field": {
        "name": "my_child_employee",
        "parent": 2
    }
}

PUT /examples_join/_doc/5?routing=2
```

```
{
    "my_name": "John R&D",
    "my_age": 40,
    "my_gender": "F",
    "my_join_field": {
        "name": "my_child_employee",
        "parent": 2
    }
}

PUT /examples_join/_doc/6?routing=3
{
    "my_name": "Bill HHR",
    "my_age": 45,
    "my_gender": "F",
    "my_join_field": {
        "name": "my_child_employee",
        "parent": 3
    }
}

PUT /examples_join/_doc/7?routing=3
{
    "my_name": "Joe HHR",
    "my_age": 48,
    "my_gender": "M",
    "my_join_field": {
        "name": "my_child_employee",
        "parent": 3
    }
}

# query based on parent id
GET /examples_join/_search
{
    "query": {
        "parent_id": {
            "type": "my_child_employee",
            "id": 2
        }
    }
}

# query child document by parent
GET /examples_join/_search
{
    "query": {
        "has_parent": {
            "parent_type": "my_parent_department",
            "score": true,
```

```
            "query": {
                "term": {
                    "my_name.keyword": "R&D"
                }
            }
        }
    }
}

# query parent document by child
GET /examples_join/_search
{
    "query": {
        "has_child": {
            "type": "my_child_employee",
            "score_mode": "sum",
            "min_children": 2,
            "max_children": 10,
            "query": {
                "bool": {
                    "must": [
                        {
                            "range": {
                                "my_age": {
                                    "gte": 10,
                                    "lte": 46
                                }
                            }
                        }
                    ],
                    "should": [
                        {
                            "term": {
                                "my_gender.keyword": "M"
                            }
                        }
                    ]
                }
            }
        }
    }
}

# define multiple-level relations
PUT /examples_join_multi_level
{
    "mappings": {
        "_doc": {
            "properties": {
                "my_join_field": {
```

```
                    "type": "join",
                    "relations": {
                        "my_grandparent_company": ["my_parent_department",
"my_parent_supplier"],
                        "my_parent_department": "my_child_employee"
                    }
                }
            }
        }
    }
}

PUT /examples_join_multi_level/_doc/1
{
    "my_name": "my grandparent company R&D",
    "my_join_field": "my_grandparent_company"
}

PUT /examples_join_multi_level/_doc/2?routing=1
{
    "my_name": "R&D",
    "my_join_field": {
        "name": "my_parent_department",
        "parent": 1
    }
}

# routing: specify ID of grandparent (highest level of hierarchy)
PUT /examples_join_multi_level/_doc/3?routing=1
{
    "my_name": "Bo R&D",
    "my_join_field": {
        "name": "my_child_employee",
        "parent": 2
    }
}

PUT /examples_join_multi_level/_doc/4
{
    "my_name": "my second grandparent company HHR",
    "my_join_field": "my_grandparent_company"
}

PUT /examples_join_multi_level/_doc/5?routing=4
{
    "my_name": "HHR",
    "my_join_field": {
        "name": "my_parent_department",
        "parent": 4
    }
```

```
}

PUT /examples_join_multi_level/_doc/6?routing=4
{
    "my_name": "Bill HHR",
    "my_join_field": {
        "name": "my_child_employee",
        "parent": 5
    }
}

GET /examples_join_multi_level/_search
{
    "query": {
        "has_child": {
            "type": "my_parent_department",
            "query": {
                "has_child": {
                    "type": "my_child_employee",
                    "query": {
                        "term": {
                            "my_name.keyword": "Bill HHR"
                        }
                    }
                }
            }
        }
    }
}



# parent/child inner hits
GET /examples_join/_search
{
    "query": {
        "has_parent": {
            "parent_type": "my_parent_department",
            "score": true,
            "inner_hits": {},
            "query": {
                "term": {
                    "my_name.keyword": "R&D"
                }
            }
        }
    }
}

GET /examples_join/_search
```

```
{
    "query": {
        "has_child": {
            "type": "my_child_employee",
            "score_mode": "sum",
            "min_children": 2,
            "max_children": 10,
            "inner_hits": {},
            "query": {
                "bool": {
                    "must": [
                        {
                            "range": {
                                "my_age": {
                                    "gte": 10,
                                    "lte": 46
                                }
                            }
                        }
                    ],
                    "should": [
                        {
                            "term": {
                                "my_gender.keyword": "M"
                            }
                        }
                    ]
                }
            }
        }
    }
}

# terms lookup mechanism (fetch a term from documents)

# join limitations:
#     documents must be stored within the same index
#     parent/child documents must be on the same shard
#     only one join field per index; a join field can have as many relations as you
want; new relations can be added after creating the index
#     child relations can only be added to existing parents
#     document can only have one parent (employee only belongs to one department)
########## join ##########



# specify result formats (yaml/json)
GET /examples_index_name/_doc/_search?format=yaml
{
    "query": {
```

```
        "match_all": {}
    }
}

# filter source
GET /examples_index_name/_doc/_search
{
    "_source": false,
    "query": {
        "match_all": {}
    }
}

# only my_fields under _source is returned
GET /examples_index_name/_doc/_search
{
    "_source": ["my_field_1", "my_field_2"],
    "query": {
        "match_all": {}
    }
}

GET /examples_index_name/_doc/_search
{
    "_source": {
        "includes": ["my_field_1", "my_field_2"],
        "excludes": "my_field_2"
    },
    "query": {
        "match_all": {}
    }
}

# limit result size; hits.total still has correct counts of results
# default size is 10
GET /examples_index_name/_doc/_search?size=2
{
    "query": {
        "match_all": {}
    }
}

GET /examples_index_name/_doc/_search
{
    "size": 2,
    "query": {
        "match_all": {}
    }
}
```

```
# use offset to specify the number of matches to skip before returning the matches
GET /examples_index_name/_doc/_search
{
    "from": 2,
    "query": {
        "match_all": {}
    }
}

# can use "from" and "size" for pagination

# sort
GET /examples_index_name/_search
{
    "_source": false,
    "query": {
        "match_all": {}
    },
    "sort": [
        "my_field_6"
    ]
}

# sort key includes a number, date will be # of millisecond since the epoch (unix
timestamp * 1000)
GET /examples_index_name/_search
{
    "query": {
        "match_all": {}
    },
    "sort": [
        {"my_field_4": "desc"},
        {"my_field_6": "asc"}
    ]
}

# sort by multi-value fields
GET /examples_index_name/_search
{
    "query": {
        "match_all": {}
    },
    "sort": [
        {
            "my_field_6": {
                "order": "desc",
                "mode": "avg"
            }
        }
    ]
```

```
}



########## filter ##########
# filter is more efficient
# typically use filter for number & date ranges and term query for keyword
GET /examples_index_name/_search
{
    "query": {
        "bool": {
            "must": [
                {
                    "match_all": {}
                }
            ],
            "filter": [
                {
                    "range": {
                        "my_field_6": {
                            "lte": 2.5
                        }
                    }
                }
            ]
        }
    }
}
########## filter ##########



########## aggregations ##########
# metric aggregations
# do aggregation on all documents
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_avg": {
            "avg": {
                "field": "my_field_6"
            }
        },
        "my_sum": {
            "sum": {
                "field": "my_field_6"
            }
        }
    }
```

```
}

# count unique values
# use my_field.keyword because Fielddata is disabled on text fields by default or
enable fielddata on a text field ("fielddata": true)
# index a field as both "text" and "keyword"
# cardinality aggregation produces an approximate number
# the precision_threshold options allows to trade memory for accuracy
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_unique_field": {
            "cardinality": {
                "field": "my_field_1.keyword"
            }
        }
    }
}

GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_unique_field": {
            "terms": {
                "field": "my_field_1.keyword"
            }
        }
    }
}

# value_count aggregation: count number of values that the aggregation is based on
(# of values that are extracted from the aggregation documents)
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_value_count": {
            "value_count": {
                "field": "my_field_6"
            }
        }
    }
}

# multi-value aggregation: stats
GET /examples_index_name/_search
{
    "size": 0,
```

```
    "aggs": {
        "my_stats": {
            "stats": {
                "field": "my_field_6"
            }
        }
    }
}

# bucket aggregations
# terms aggregation: build buckets for each unique value
# elasticsearch only returns top unique terms so some terms won't appear in results;
"sum_other_doc_count" key is the sum of document counts which aren't part of the
results
# "missing" is for documents contain null values or don't have my_field at all; user
can name the bucket "my N/A" for "missing"
# "min_doc_count" key specifies minimum number of documents a bucket needs to
contain to be present in results (default is 1)
# "doc_count" is approximate and not always accurate
# "_term" is a special key which allows users to refer to buckets' keys
# "doc_count_error_upper_bound" means the max possible document counts for a term
isn't part of the final results
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_terms": {
            "terms": {
                "field": "my_field_1.keyword",
                "missing": "my N/A",
                "min_doc_count": 0,
                "order": {
                    "_term": "asc"
                }
            }
        }
    }
}

# nested aggregation (sub-aggregation)
GET /examples_index_name/_search
{
    "size": 0,
    "query": {
        "range": {
            "my_field_6": {
                "lte": 100
            }
        }
    },
```

```
    "aggs": {
        "my_terms": {
            "terms": {
                "field": "my_field_1.keyword"
            },
            "aggs": {
                "my_terms_sub": {
                    "stats": {
                        "field": "my_field_6"
                    }
                }
            }
        }
    }
}

# filter out document in aggregation
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_range": {
            "filter": {
                "range": {
                    "my_field_6": {
                        "lte": 5000
                    }
                }
            },
            "aggs": {
                "my_avg": {
                    "avg": {
                        "field": "my_field_6"
                    }
                }
            }
        }
    }
}

# define bucket rules with filters
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_filter": {
            "filters": {
                "filters": {
                    "my_bucket_rule_1": {
                        "match": {
```

```
                        "my_field_1": "my_value"
                    }
                },
                "my_bucket_rule_2": {
                    "match": {
                        "my_field_2": "my_value_2"
                    }
                }
            }
        },
        "aggs": {
            "my_avg": {
                "avg": {
                    "field": "my_field_6"
                }
            }
        }
    }
}

# range aggregation (range/date range)
# "from" key is included and "to" key is excluded
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_range": {
            "range": {
                "field": "my_field_6",
                "ranges": [
                    {
                        "to": 50
                    },
                    {
                        "from": 50,
                        "to": 100
                    },
                    {
                        "from": 100
                    }
                ]
            }
        }
    }
}

# use double pipes for date math
GET /examples_index_name/_search
{
```

```
    "size": 0,
    "aggs": {
        "my_range": {
            "date_range": {
                "field": "my_field_4",
                "format": "yyyy-MM-dd",
                "keyed": true,
                "ranges": [
                    {
                        "from": "2017-07-07",
                        "to": "2017-07-07||+6M",
                        "key": "my_bucket_name_1"
                    },
                    {
                        "from": "2017-07-07||+6M",
                        "to": "2017-07-07||+1y",
                        "key": "my_bucket_name_2"
                    }
                ]
            },
            "aggs": {
                "my_stats": {
                    "stats": {
                        "field": "my_field_6"
                    }
                }
            }
        }
    }
}

# histogram - bucket results using values (min/max/specified interval)
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_histogram": {
            "histogram": {
                "field": "my_field_6",
                "interval": 10,
                "min_doc_count": 0,
                "extended_bounds": {
                    "min": 0,
                    "max": 100
                }
            }
        }
    }
}
```

```
# date_histogram
GET /examples_index_name/_search
{
    "size": 0,
    "aggs": {
        "my_date_histogram": {
            "date_histogram": {
                "field": "my_field_4",
                "interval": "month"
            }
        }
    }
}

# global aggregation isn't influenced by the search query
GET /examples_index_name/_search
{
    "size": 0,
    "query": {
        "range": {
            "my_field_6": {
                "gte": 20
            }
        }
    },
    "aggs": {
        "my_aggs_range": {
            "stats": {
                "field": "my_field_6"
            }
        },
        "my_aggs_all": {
            "global": {},
            "aggs": {
                "my_global_stats": {
                    "stats": {
                        "field": "my_field_6"
                    }
                }
            }
        }
    }
}

# missing field values
POST /examples_missing/_doc/1
{
    "my_field_1": 100,
    "my_field_2": "text"
}
```

```
POST /examples_missing/_doc/2
{
    "my_field_1": 200,
    "my_field_2": null
}

GET /examples_missing/_search
{
    "size": 0,
    "aggs": {
        "my_missing_aggs": {
            "missing": {
                "field": "my_field_2.keyword"
            },
            "aggs": {
                "my_missing_sum": {
                    "sum": {
                        "field": "my_field_1"
                    }
                }
            }
        }
    }
}

# aggregating nested objects
# it is important to include the path of the object while working on nested query
GET /examples_nested/_search
{
    "size": 0,
    "aggs": {
        "my_nested_min": {
            "nested": {
                "path": "employees"
            },
            "aggs": {
                "my_min": {
                    "min": {
                        "field": "employees.age"
                    }
                }
            }
        }
    }
}
########## aggregations ##########
```

```
########## improve search results ##########
# add more data
POST /examples_index_name/_doc/_bulk
{"index": {"_id": "100"}}
{"my_field_1": "dog brown", "my_field_8": "vegetable"}
{"index": {"_id": "101"}}
{"my_field_1": "brown dog", "my_field_8": "veget"}
{"index": {"_id": "102"}}
{"my_field_1": "brown a dog", "my_field_8": "vegetables"}
{"index": {"_id": "103"}}
{"my_field_1": "doggy", "my_field_8": "vegetable"}

# proximity searches
# slop 2: phrase can be reversed
GET /examples_index_name/_search
{
    "query": {
        "match_phrase": {
            "my_field": {
                "query": "my_value",
                "slop": 1
            }
        }
    }
}

# match query will match documents with my_value_0 or my_value_1 ("vegetable" or
"veget")
# "match_phrase" under "should" is optional and it boosts scores
GET /examples_index_name/_search
{
    "query": {
        "bool": {
            "must": [
                {
                    "match": {
                        "my_field_8": {
                            "query": "vegetable veget"
                        }
                    }
                }
            ],
            "should": [
                {
                    "match_phrase": {
                        "my_field_1": {
                            "query": "brown dog",
                            "slop": 2
                        }
                    }
```

```
                }
            ]
        }
    }
}

# fuzzy match query for typo; levenshtein distance (edit distance); maximum edit
distance: 2
# automatic fuzziness:
# term length 1-2 uses maximum edit distance 0
# term length 3-5 uses maximum edit distance 1
# term length >5 uses maximum edit distance 2
# lobster & oyster are two levenshtein distance apart; majority of human typos is
edit distance 1
GET /examples_index_name/_search
{
    "query": {
        "match": {
            "my_field_8": {
                "query": "vegetable",
                "fuzziness": "auto"
            }
        }
    }
}

# transposition (live v.s. lvie) is 1 edit distance apart
# or to set fuzzy_transpositions to be false and edit distance is 2
GET /examples_index_name/_search
{
    "query": {
        "match": {
            "my_field_8": {
                "query": "live",
                "fuzziness": 1,
                "fuzzy_transpositions": true
            }
        }
    }
}

# fuzzy query (term-level query) isn't analyzed
# match query with fuzziness option (full-text query): analyzed; preferred method
than fuzzy query
GET /examples_index_name/_search
{
    "query": {
        "fuzzy": {
            "my_field_8": {
                "value": "VEGETABLE",
```

```
                "fuzziness": "auto"
            }
        }
    }
}

GET /examples_index_name/_search
{
    "query": {
        "match": {
            "my_field_8": {
                "query": "VEGETABLE",
                "fuzziness": "auto"
            }
        }
    }
}

# synonyms
# "lowercase" filter is loaded before "my_synonym_test" filter
PUT /examples_synonyms_index
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonym_test": {
          "type": "synonym",
          "synonyms_path": "analysis/my_synonym_test.txt"
        }
      },
      "analyzer": {
        "my_analyzer": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "my_synonym_test"
          ]
        }
      }
    }
  },
  "mappings": {
    "_doc": {
      "properties": {
        "my_description": {
          "type": "text",
          "analyzer": "my_analyzer"
        }
      }
    }
```

```
  }
}

##### my_synonym_test.txt (config/analysis/my_synonym_test.txt) ##### all nodes have
to have this file

# This is a comment
awful => terrible    # awful will be replaced by terrible (note: term query isn't
analyzed so won't work)
awesome => great, super
elasticsearch, logstash, kibana => elk    # all lower cases because "lowercase"
filter is loaded before "my_synonym_test" filter
weird, strange    # both two terms are placed in the same position

##### my_synonym_test.txt #####

# use analyzer to test synonyms
POST /examples_synonyms_index/_analyze
{
    "analyzer": "my_analyzer",
    "text": "awesome"
}

POST /examples_synonyms_index/_analyze
{
    "analyzer": "my_analyzer",
    "text": "ELASTICSEARCH"
}

POST /examples_synonyms_index/_analyze
{
    "analyzer": "my_analyzer",
    "text": "ELASTICSEARCH is weird"
}

POST /examples_synonyms_index/_doc
{
    "my_description": "Elasticsearch is awesome, but can also seem weird sometimes."
}

# there is a match even though there is no "great" in my_description;
"my_synonym_test" works
GET /examples_synonyms_index/_doc/_search
{
    "query": {
        "match": {
            "my_description": "great"
        }
    }
}
```

```
GET /examples_synonyms_index/_doc/_search
{
    "query": {
        "match": {
            "my_description": "awesome"
        }
    }
}

# note: if there are documents indexed before adding synonyms, update by query API
needs to be executed so all documents will be re-index
POST /examples_synonyms_index/_update_by_query

# highlight matches in fields
POST /examples_highlight_index/_doc/1
{
    "my_description": "Elasticsearch is a search engine based on the Lucene
library."
}

GET /examples_highlight_index/_doc/_search
{
    "query": {
        "match": {
            "my_description": "Lucene library"
        }
    },
    "highlight": {
        "pre_tags": ["<my_tag>"],
        "post_tags": ["</my_tag>"],
        "fields": {
            "my_description": {}
        }
    }
}

# stemming
PUT /examples_stemming_index
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonym_test": {
          "type": "synonym",
          "synonyms": [
            "firm => company",
            "love, enjoy"
          ]
        },
```

```
        "my_stemmer_test" : {
          "type" : "stemmer",
          "name" : "english"
        }
      },
      "analyzer": {
        "my_analyzer": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "my_synonym_test",
            "my_stemmer_test"
          ]
        }
      }
    }
  },
  "mappings": {
    "_doc": {
      "properties": {
        "my_description": {
          "type": "text",
          "analyzer": "my_analyzer"
        }
      }
    }
  }
}

POST /examples_stemming_index/_doc/1
{
    "my_description": "I love working for my firm!"
}

# search enjoy, love, work, or working
GET /examples_stemming_index/_doc/_search
{
    "query": {
        "match": {
            "my_description": "enjoy work love working"
        }
    },
    "highlight": {
        "fields": {
            "my_description": {}
        }
    }
}
########## improve search results ##########
```

```
########## other notes ##########
# document contains fields
# text search in Elasticsearch
# score is based on relevance (_score)
# find "field" contains "text or phrase"
# "must": all conditions have to be true for results to return
# "should" - can weight criteria differently using boost
# match, match with fuzziness, match_phrase, match_phrase_prefix, match_phrase with
slop

# boost score
GET /examples_index_name/_search
{
    "query": {
        "bool": {
            "should": [
                {
                    "match_phrase": {
                        "my_field_1": "brown dog"
                    }
                },
                {
                    "match_phrase": {
                        "my_field_1": {
                            "query": "a brown dog",
                            "boost": 3
                        }
                    }
                }
            ]
        }
    }
}

"""
GET _search
{
    "min_score": 0,
    "size": 250,
    "query": {
        "match_all": {}

    },
    "highlight": {
        "fields": {
            "my_field": {}
        }
    },
```

```
    "sort": [
        {"my_field" : "desc"}
    ],
    "aggs": {
        "my_statistics": {
            "stats": {
                "field" : "my_field.keyword",
                "size": 10
            }
        },
        "my_unique_count": {
            "terms": {
                "field":  "my_field"
            }
        },
        "my_doc_count": {
            "range": {
                "field": "my_field",
                "ranges": [
                    {
                        "key" : "0-10000",
                        "from" : 0,
                        "to" : 10
                    }
                ]
            }
        }
    }
}
"""

# SQL in Elasticsearch
POST /_xpack/sql?format=txt
{
    "query": "SELECT * FROM examples_index_name WHERE my_field_1 = 'brown dog'"
}

# list all stored scripts
GET _cluster/state/metadata?pretty&filter_path=**.stored_scripts

# delete a stored script
DELETE _scripts/my_stored_script

# register a stored script
POST _scripts/my_stored_script
{
    "script": {
        "lang": "mustache",
        "source": {
            "query": {
```

```
                "match": {
                    "my_field_1": "{{var_my_field_1}}"
                }
            }
        }
    }
}

# retrieve a stored script
GET _scripts/my_stored_script

# use a stored script
GET /examples_index_name/_search/template
{
    "id": "my_stored_script",
    "params": {
        "var_my_field_1": "my_value"
    }
}

GET /_msearch
{ "index": "examples_index_name"}
{ "query": { "match": {"my_field_1":"my_value"}}}
{ "index": "examples_index_name"}
{ "query": { "match": {"my_field_1":"dog"}}}

$ bin/logstash -e "input { stdin { } } output { stdout {} }"
$ cd \logstash\bin
$ logstash -f my_logstash.conf
##### my_logstash.conf #####
input {
    file {
        path => "my_input_data.csv"
        id => "my_plugin_id"
        start_position => "beginning"
        sincedb_path => "/dev/null"
    }
}
filter {
    csv {
        separator => ","
        columns => [ "my_column_1", "my_column_2", "my_column_3" ]
    }
    mutate {convert => ["my_column_2","integer"] }
    mutate {convert => ["my_column_3","float"] }
}
output {
    elasticsearch {
        hosts => "localhost"
        index => "examples_logstash_index"
```

```
    }
    stdout {codec => dots}
}
##### my_logstash.conf #####

# examples of dates which Elasticsearch can recognize by default
"2019-05-01" or "2019/05/01 13:15:30"

# geo-distance
PUT /examples_lat_long
{
    "mappings": {
        "_doc": {
            "properties": {
                "my_address": {
                    "properties": {
                        "latlong": {
                            "type": "geo_point"
                        }
                    }
                }
            }
        }
    }
}

PUT /examples_lat_long/_doc/1
{
    "my_address" : {
        "latlong" : {
            "lat" : 32.936461,
            "lon" : -117.234071
        }
    }
}
PUT /examples_lat_long/_doc/2
{
    "my_address" : {
        "latlong" : {
            "lat" : 32.936529,
            "lon" : -117.233178
        }
    }
}

GET /examples_lat_long/_search
{
    "query": {
        "bool" : {
            "must" : {
```

```
                "match_all" : {}
            },
            "filter" : {
                "geo_distance" : {
                    "distance" : "270ft",
                    "my_address.latlong" : {
                      "lat" : 32.936461,
                      "lon" : -117.234071
                    }
                }
            }
        }
    }
}

########## other notes ##########



########## Python ##########
# register elasticsearch stored scripts from a folder using python
import elasticsearch
import glob
import os

my_es_client = elasticsearch.Elasticsearch(["localhost:9200"])
my_file_path = "./my_es_search_templates/*.txt"
files = glob.glob(my_file_path)

if len(files) == 0:
    print("No files of elasticsearch stored scripts found")
    exit(1)
else:
    for file_name in files:
        with open(file_name) as file_search_template:
            es_search_template_body = file_search_template.read()
            es_search_template_id = os.path.basename(file_name).split(".")[0]
            # register elasticsearch search templates
            my_es_client.put_script(id=es_search_template_id,
body=es_search_template_body)
            es_search_template_id = ''
            es_search_template_body = ''



my_es_client.search(
    index="examples_index_name",
    body={
        "query": {
            "match_all": {}
```

```
        }
    }
)

my_es_client.count(
    index="examples_index_name",
    body={
        "query": {
            "match_all": {}
        }
    }
)

my_es_client.search_template(
    index="examples_index_name",
    body={
        "id": "my_stored_script",
        "params": {
            "var_my_field_1": "my_value"
        }
    }
)
########## Python ##########
```