Elasticsearch

4장 데이터 다루기

4.1 단건문서 API

색인 API
PUT [인덱스이름]/_doc/[_id값]

라우팅
PUT [인덱스이름]/_doc/[_id값]?routing=myid2
{ 내용}

조회 API
GET [인덱스이름]/_doc/[_id값], GET [인덱스이름]/_source/[_id값] (문서본문)

필드 필터링
GET [인덱스이름]/_doc/[_id값]?**_source_includes**=p*,views&**_source_excludes**=public

라우팅조회
GET [인덱스이름]/_doc/[_id값]?routing=myid2

업데이트
POST [인덱스이름]/_update/[_id값]
```
{
 “doc” : {
   [업데이트할 내용]
  }
}
```

Detect_noop
변경된점이 있는지 체크하고 없으면 안함 (기본값 true)

Doc_as_upsert
update시 없으면 insert (기본값 false)

4.1 단건문서 API

Script를 이용하여 업데이트

POST [인덱스이름]/_update/[id]

```
{
 “script” : {
   “source” : “ctx._source.views += params.amount”,
   “lang” : “painless”
   “params” : {
     “amount” : 1
    }
  }
  “scripted_upsert” : false
}
```

삭제

DELETE [인덱스이름]/_doc/[id]

4.2 복수문서 API

Bulk API
Application/x-ndjson 으로 요청해야함

```
1  POST _bulk
2  {"index":{"_index":"test","_id":"1"}}
3  {"field":"value one"}
4  {"index":{"_index":"test","_id":"2"}}
5  {"field":"value two"}
6  {"delete":{"_index":"test","_id":"2"}}
7  {"create":{"_index":"test","_id":"3"}}
8  {"field":"value three"}
9  {"update":{"_index":"test","_id":"1"}}
10 {"doc":{"field":"value two"}}
```

test/_doc/1 에
{ "field" : "value one" } 입력

test/_doc/2 에
{ "field" : "value two" } 입력

test/_doc/2 도큐먼트 삭제

test/_doc/3 에
{ "field" : "value three" } 입력

test/_doc/1 도큐먼트를
{"field" : "value two"} 로 수정

```
1  POST test/_bulk
2  {"index":{"_id":"1"}}
3  {"field":"value one"}
4  {"index":{"_id":"2"}}
5  {"field":"value two"}
6  {"delete":{"_id":"2"}}
7  {"create":{"_id":"3"}}
8  {"field":"value three"}
9  {"update":{"_id":"1"}}
10 {"doc":{"field":"value two"}}
```

인덱스 단위로 _bulk 사용

4.2 복수문서 API

Multi Get API
단건씩 여러 번 조회 하는것보다 성능 좋음

```
GET /_mget
{
  "docs": [
    {
      "_index": "my-index-000001",
      "_id": "1"
    },
    {
      "_index": "my-index-000001",
      "_id": "2"
    }
  ]
}
```

4.2 복수문서 API

Update_by_query, delete_by_query

conflicts 매개변수를 proceed를 지정하면 다음작업으로 넘어가고 abort는 작업을 중단한다. (기본값 abort)
Scroll_size, scroll, requests_per_seconds로 작업 속도를 조정해서 클러스터 부하와 서비스 영향을 최소화 할수 있다.

특정 쿼리에 해당하는 데이터들을 한번에 update 하는 쿼리

```
index_name/_update_by_query [POST]

{
    "script":{
        "source":"ctx._source.USER_LOCATION = params.location",
        "params":{
            "location":"경기도"
        }
    },
    "query":{
        "term":{
            "USER_LOCATION":"서울특별시"
        }
    }
}
```

delete by query의 search 요청으로 term query를 전달하여 데이터 삭제요청을 한다. term query 조건에 만족되는 모든 데이터에 대하여 삭제를 요청하는 것이다.

```
POST /es_data_delete_test_index/_delete_by_query
{
    "query": {
        "term" : { "db_ref" : 1 }
    }
}
```

4.3 검색 API

URI 검색으로 검색어 "value AND three" 검색

```
GET test/_search?q=value AND three
```

데이터 본문(data body) 검색은 검색 쿼리를 데이터 본문으로 입력하는 방식입니다. Elasticsearch의 QueryDSL을 사용하며 쿼리 또한 Json 형식으로 되어 있습니다. 처음 익힐때는 다소 복잡 해 보일 수 있으나 주로 사용하는 쿼리 몇가지들 부터 차근 차근 익혀나가면 크게 어렵지 않게 사용이 가능합니다.

가장 쉽고 많이 사용되는 것은 **match** 쿼리입니다. 여기서는 문법만 살펴보고 다음 검색 장에서 더 많은 쿼리들에 대해 자세히 다뤄보도록 하겠습니다. 데이터 본문 검색으로 field 필드값이 value 인 도큐먼트를 검색하기 위해서는 다음 명령을 실행합니다.

request | response

데이터 본문 검색으로 "field" 필드에서 검색어 "value" 검색

```
GET test/_search
{
  "query": {
    "match": {
      "field": "value"
    }
  }
}
```

쿼리 입력은 항상 `query` 지정자로 시작합니다. 그 다음 레벨에서 쿼리의 종류를 지정하는데 위에서는 **match** 쿼리를 지정했습니다. 그 다음은 사용할 쿼리 별로 문법이 상이할 수 있는데 match 쿼리는 `<필드명>:<검색어>` 방식으로 입력합니다.

🔍 Term Query

📝 term

형태소 분석이 적용된 컬럼의 값들은 형태소 분석기에 따라 토큰으로 분리되는데 이것을 **텀(term)** 이라 한다.
모든 대문자는 소문자로 변형되고, 중복된 단어는 삭제된다.
Term query는 주어진 질의문이 저장된 텀과 완전히 일치한 내용만 찾는다.

```
{
  "query": {
    "term": {
      "title": "awesome"
    }
  }
}
```

📝 terms

2개 이상의 term을 같이 검색하려면 terms 쿼리를 이용한다.
필드의 값은 항상 배열로 전달해야 한다.

```
{
  "query": {
    "terms": {
      "title": ["awsome", "elastic", "jonnung"],
      "minium_should_match": 2
    }
  }
}
```

- **minium_should_match**: 몇 개 이상의 term과 일치해야 검색 결과에 시킬지 설정

4.3 검색 API

📝 prefix

term 쿼리와 마찬가지로 질의어에 형태소 분석이 적용되지 않기 때문에 정확한 term값으로 검색해야 한다.
주어진 질의어로 term의 접두어를 검색하므로 term의 일부만으로도 검색할 수 있다.

```
{
  "query": {
    "prefix": {
      "title": "awe"
    }
  }
}
```

- gte(greater than or equal): 주어진 값보다 크거나 같다.
- ge(greater than): 주어진 값보다 크다.
- lte(less than or equal): 주어진 값보다 작거나 같다.
- lt(less than): 주어진 값보다 작다.

비교할 수 있는 필드는 숫자 또는 날짜/시간 형식이어야 한다.

```
{
  "query": {
    "range": {
      "pages": {"gte": 50, "lt": 150}
    }
  }
}
```

🔍 **Bool Query**

여러 쿼리를 boolean 조건으로 결합해서 도큐먼트를 검색한다.

- `must` : 반드시 매칭되는 조건, score에 영향을 준다.
- `filter` : `must`와 동일한 동작하지만, score에 영향을 주지 않는다.
- `should` : bool 쿼리가 query context에 있고 `must` 또는 `filter` 절이 있다면, `should` 쿼리와 일치하는 결과가 없더라도 매치가 된다. bool 쿼리가 filter context 안에 있거나, `must` 또는 `filter` 중에 하나라도 있는 경우에만 매칭된다. `minimum_should_match` 이 값을 지정해서 컨트롤할 수 있다.
- `must_not` : 이 쿼리와 매칭되지 않아야 한다.

```
{
  "query": {
    "bool": {
      "must": {
        "term": {"title": "the"}
      },
      "must_not": {
        "term": {"contents": "world"}
      },
      "should": [
        {"term": {"title": "awesome"}},
        {"term": {"title": "elastic"}}
      ]
    }
  }
}
```

4.3 검색 API – 집계

Aggregation 은 번역하면 "집계" 라는 뜻 이지만 Elasticsearch 의 기능명 이기 때문에 보통 Elastic Stack 관련 세미나 또는 블로그 포스트에서는 원문대로 **aggregation** 혹은 **애그리게이션** 으로 많이 표현합니다. 이 책에서도 **aggregation** 또는 **aggs** 로 표현하도록 하겠습니다.

Aggregation 의 사용 방법은 다음과 같습니다. **_search API** 에서 query 문과 같은 수준에 지정자 `aggregations` 또는 `aggs` 를 명시하고 그 아래 임의의 aggregation 이름을 입력한 뒤 사용할 aggregation 종류와 옵션들을 명시합니다. 한번의 쿼리로 aggregation 여러 개를 입력할 수도 있습니다. 아래는 aggregation을 입력하는 예제입니다.

request | response

aggregations 입력

```
GET <인덱스명>/_search
{
  "query": {
    … <쿼리 구문> …
  },
  "aggs": {
    "<임의의 aggregation 1>": {
      "<aggregation 종류>": {
        … <aggreagation 구문> …
      }
    },
    "<임의의 aggregation 2>": {
      "<aggregation 종류>": {
        … <aggreagation 구문> …
      }
    }
  }
}
```

min, max, sum, avg

가장 흔하게 사용되는 metrics aggregations 은 **min**, **max**, **sum**, **avg** aggregation 입니다. 순서대로 명시한 필드의 **최소**, **최대**, **합**, **평균** 값을 가져오는 aggregation 입니다. 다음은 sum aggregation을 이용해서 my_stations 에 있는 전체 데이터의 **passangers** 필드값의 합계를 가져오는 예제입니다.

request | response

my_stations 인덱스의 passangers 필드 합 (sum) 을 가져오는 aggs 결과


```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 10,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [ ]
  },
  "aggregations" : {
    "all_passangers" : {
      "value" : 41995.0
    }
  }
}
```

4.3 검색 API – 집계

stats

min, **max**, **sum**, **avg** 값을 모두 가져와야 한다면 다음과 같이 **stats** aggregation을 사용하면 위 4개의 값 모두와 count 값을 한번에 가져옵니다.

request | response

stats 로 passangers 필드의 min, max, sum, avg 값을 가져오는 aggs

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "passangers_stats": {
      "stats": {
        "field": "passangers"
      }
    }
  }
}
```

cardinality

필드의 값이 모두 몇 종류인지 분포값을 알려면 **cardinality** aggregation을 사용해서 구할 수 있습니다. Cardinality 는 일반적으로 text 필드에서는 사용할 수 없으며 **숫자** 필드나 **keyword**, **ip** 필드 등에 사용이 가능합니다. 사용자 접속 로그에서 IP 주소 필드를 가지고 실제로 접속한 사용자가 몇명인지 파악하는 등의 용도로 주로 사용됩니다. 다음은 my_stations 인덱스에서 **line** 필드의 값이 몇 종류인지를 계산하는 예제입니다.

request | response

line 필드의 값이 몇 종류인지를 가져오는 aggs

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "uniq_lines ": {
      "cardinality": {
        "field": "line.keyword"
      }
    }
  }
}
```

위 쿼리 결과 `"uniq_lines " : { "value" : 3 }` 처럼 실제로 line 필드에는 "1호선", "2호선", "3호선" 총 3 종류의 값들이 있습니다.

4.3 검색 API – 집계

percentiles, percentile_ranks

값들을 백분위 별로 보기 위해서 **percentiles** aggregation 의 사용이 가능합니다. 먼저 passangers 필드에 percentiles aggregation 적용 한 것을 확인 해 보겠습니다.

request | response

passangers 필드의 백분위를 가져오는 aggs

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "pass_percentiles": {
      "percentiles": {
        "field": "passangers"
      }
    }
  }
}
```

percentiles aggregation은 디폴트로 **1%**, **5%**, **25%**, **50%**, **75%**, **95%**, **99%** 구간에 위치 해 있는 값들을 표시 해 줍니다. 백분위 구간을 직접 지정하고 싶으면 **percents** 옵션을 이용해서 지정이 가능합니다. 다음은 **20%**, **60%**, **80%** 백분위의 값을 가져오는 percentiles aggregation 입니다.

request | response

passangers 필드의 백분위를 지정해서 가져오는 aggs

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "pass_percentiles": {
      "percentiles": {
        "field": "passangers",
        "percents": [ 20, 60, 80 ]
      }
    }
  }
}
```

4.3 검색 API 집계

Bucket aggregation 은 주어진 조건으로 분류된 **버킷** 들을 만들고, 각 버킷에 소속되는 도큐먼트들을 모아 **그룹으로 구분**하는 것입니다. 각 버킷 별로 포함되는 도큐먼트의 개수는 **doc_count** 값에 기본적으로 표시가 되며 각 버킷 안에 metrics aggregation 을 이용해서 다른 계산들도 가능합니다. 주로 사용되는 bucket aggregation 들은 **Range**, **Histogram**, **Terms** 등이 있습니다.

range

range 는 숫자 필드 값으로 범위를 지정하고 각 범위에 해당하는 버킷을 만드는 aggregation 입니다. **field** 옵션에 해당 필드의 이름을 지정하고 **ranges** 옵션에 배열로 **from**, **to** 값을 가진 오브젝트 값을 나열해서 범위를 지정합니다. 다음은 passangers 값이 각각 1000 미만, 1000~4000 사이, 4000 이상 인 버킷들을 생성하는 예제입니다.

request | response

range aggs 를 이용해서 passangers 필드의 값을 버킷으로 구분

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "passangers_range": {
      "range": {
        "field": "passangers",
        "ranges": [
          {
            "to": 1000
          },
          {
            "from": 1000,
            "to": 4000
          },
          {
            "from": 4000
          }
        ]
      }
    }
  }
}
```

histogram

histogram 도 range 와 마찬가지로 숫자 필드의 범위를 나누는 aggs 입니다. 앞에서 본 range 는 from 과 to 를 이용해서 각 버킷의 범위를 지정했습니다. histogram 은 from, to 대신 **interval** 옵션을 이용해서 주어진 간격 크기대로 버킷을 구분합니다. 다음은 passangers 필드에 간격이 2000인 버킷들을 생성하는 예제입니다.

request | response

histogram aggs 를 이용해서 passangers 필드의 값을 버킷으로 구분

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "passangers_his": {
      "histogram": {
        "field": "passangers",
        "interval": 2000
      }
    }
  }
}
```

4.3 검색 API 집계

date_range, date_histogram

range 와 **histogram** aggs 처럼 숫자 외에도 날짜 필드를 이용해서 범위별로 버킷의 생성이 가능합니다. 이 때 사용되는 **date_range**, **date_histogram** aggs 들은 특히 시계열 데이터에서 날짜별로 값을 표시할 때 매우 유용합니다. **date_range** 는 **ranges** 옵션에 `{"from": "2019-06-01", "to": "2016-07-01"}` 와 같이 입력하며 **date_histogram** 은 **interval** 옵션에 `day`, `month`, `week` 와 같은 값들을 이용해서 날짜 간격을 지정할 수 있습니다. 다음은 **date_histogram** 으로 **date** 필드를 **1개월** 간격으로 구분하는 예제입니다.

request | response

date_histogram 을 이용해서 date 값을 1개월 간격의 버킷으로 구분

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "date_his": {
      "date_histogram": {
        "field": "date",
        "interval": "month"
      }
    }
  }
}
```

terms

앞에서 살펴 본 **(date_)range**, **histogram** 은 모두 숫자, 날짜를 가지고 구간을 나누는 aggregation 이었습니다. **terms** aggregation 은 **keyword** 필드의 문자열 별로 버킷을 나누어 집계가 가능합니다. keyword 필드 값으로만 사용이 가능하며 분석된 text 필드는 일반적으로는 사용이 불가능합니다. 다음은 my_stations 인덱스에서 **station.keyword** 필드를 기준으로 버킷들을 만드는 예제입니다.

request | response

terms 을 이용해서 station 값을 별로 버킷 생성

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "stations": {
      "terms": {
        "field": "station.keyword"
      }
    }
  }
}
```

terms aggregation 에는 **field** 외에도 가져올 버킷의 개수를 지정하는 **size** 옵션이 있으며 디폴트 값은 **10** 입니다. 인덱스의 특정 keyword 필드에 있는 모든 값들을 종류별로 버킷을 만들면 가져와야 할 결과가 매우 많기 때문에 먼저 도큐먼트 개수 또는 주어진 metrics 연산 결과가 가장 많은 버킷 들을 샤드별로 계산해서 상위 몇개의 버킷들만 coordinate 노드로 가져오고, 그것들을 취합해서 결과를 나타냅니다. 이 과정은 검색의 query 그리고 fetch 과정과 유사합니다.

4.3 검색 API 집계

Bucket Aggregation 으로 만든 버킷들 내부에 다시 `"aggs" : { }` 를 선언해서 또다른 버킷을 만들거나 Metrics Aggregation 을 만들어 사용이 가능합니다. 다음은 terms aggregation을 이용해서 생성한 **stations** 버킷 별로 **avg** aggregation을 이용해서 **passangers** 필드의 평균값을 계산하는 **avg_psg_per_st** 을 생성하는 예제입니다.

request | response

terms aggs 아래에 avg aggs 사용

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "stations": {
      "terms": {
        "field": "station.keyword"
      },
      "aggs": {
        "avg_psg_per_st": {
          "avg": {
            "field": "passangers"
          }
        }
      }
    }
  }
}
```

stations 버킷들 별로 **avg_psg_per_st** 라는 aggregation이 실행된 것을 확인할 수 있습니다. 버킷 안에 또 다른 하위 버킷을 만드는 것도 가능합니다. 다음은 terms aggregation 을 이용해서 line.keyword 별로 **lines** 버킷을 만들고 그 안에 또다시 terms aggregation을 이용한 **stations_per_lines** 버킷을 만든 예제입니다.

request | response

terms aggs 아래에 하위 terms aggs 사용

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "lines": {
      "terms": {
        "field": "line.keyword"
      },
      "aggs": {
        "stations_per_lines": {
          "terms": {
            "field": "station.keyword"
          }
        }
      }
    }
  }
}
```

4.3 검색 API 집계

Aggregation 중에는 다른 **metrics** aggregation의 결과를 새로운 입력으로 하는 **pipeline** aggregation이 있습니다. pipeline 에는 다른 버킷의 결과들을 다시 연산하는 **min_bucket**, **max_bucket**, **avg_bucket**, **sum_bucket**, **stats_bucket**, 이동 평균을 구하는 **moving_avg**, 미분값을 구하는 **derivative**, 값의 누적 합을 구하는 **cumulative_sum** 등이 있습니다. Pipeline aggregation 은 `"buckets_path": "<버킷 이름>"` 옵션을 이용해서 입력 값으로 사용할 버킷을 지정합니다. 다음은 my_stations 에서 **date_histogram**을 이용해서 월별로 나눈 passangers 의 합계 sum을 다시 **cumulative_sum**을 이용해서 누적값을 구하는 예제입니다.

request | response

passangers 의 값을 입력으로 받는 cumulative_sum aggs 실행

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "months": {
      "date_histogram": {
        "field": "date",
        "interval": "month"
      },
      "aggs": {
        "sum_psg": {
          "sum": {
            "field": "passangers"
          }
        },
        "accum_sum_psg": {
          "cumulative_sum": {
            "buckets_path": "sum_psg"
          }
        }
      }
    }
  }
}
```

서로 다른 버킷에 있는 값들도 bucket_path에 `>` 기호를 이용해서 "부모>자녀" 형태로 지정이 가능합니다. 다음은 sum_bucket 을 이용해서 **mon>sum_psg** 버킷에 있는 passangers 필드값의 합을 구하는 예제입니다.

request | response

다른 부모의 자녀 버킷에 있는 필드를 입력으로 받는 pipeline aggs

```
GET my_stations/_search
{
  "size": 0,
  "aggs": {
    "mon": {
      "date_histogram": {
        "field": "date",
        "interval": "month"
      },
      "aggs": {
        "sum_psg": {
          "sum": {
            "field": "passangers"
          }
        }
      }
    },
    "bucket_sum_psg": {
      "sum_bucket": {
        "buckets_path": "mon>sum_psg"
      }
    }
  }
}
```