ELASTICSEARCH AGGREGATIONS EXPLAINED

ElasticSearch lets you do the equivalent of a SQL GROUP BY COUNT and AVERAGE functions. They call these **aggregations**.

In other words, if you are looking at nginx web server logs you could:

- group each web hit record by the city from where the user came
- count them

So this would give you something like:

City	web hits
Paris	20
London	30
Berlin	40

In SQL this would be something like:

```
select city, count(city) from logs
 group by city
```

Here we illustrate this using the simplest use case, web logs. Follow the previous doc to populate your ElasticSearch instance with some nginx web server logs if you want to follow along.

Aggregation
Because ElasticSearch is concerned with performance, there are some rules on what kind of fields you can aggregate. You can group by any numeric field but for text fields that have to be of type

keyword or have **fielddata=true.**

You can think of keyword as being like an index. When we loaded the nginx data, we did not create the index mapping first. We let ElasticSearch build that on-the-fly. So it chose to index all the text fields since it map those of type keyword, like this:

```
"geoip" : {
          "dynamic" : "true",
          "properties" : {
            "city_name" : {
              "type" : "text",
              "norms" : false,
              "fields" : {
                "keyword" : {
                  "type" : "keyword",
                  "ignore_above" : 256
                }
              }
            }
```

Now, aggregations is a complicated topic. There are many variations. You can nest them to build up complex queries. Here we look at the simplest, most common use case: **bucket aggregation.** That is like the select, count SQL statement to produce a count by value.

In the example below, we want to count web hits by the city name.

The general pattern to build up the statement is:

- Use **aggs**, which is short for **aggregations.**
- Give the agg a name. Here we use **cityName.**
- Tell it what field to use. Here we use the dot notation **geoip.city_name** since city_name is a property of geo_ip. In other words in a deeply nested JSON structure put a dot as you go down the hierarchy. For a JSON array, you would use ElasticSearch scripting, a topic we have not covered yet.
- Add the word **keyword**, to tell it to use that index.
- Give it the aggregation operation type. here we use **terms**. You can also use **ave** (average) and some others.

You can also use filters, which we illustrate further below.

So this query products of a count of web hits by city.

```
curl -XGET --user $pwd --header 'Content-Type: application/json'
https://58571402f5464923883e7be42a037917.eu-central-1.aws.cloud.es.io:9243/lo
gstash/_search?pretty -d '{
        "aggs": {
                "cityName": {
                        "terms": {
                                "field": "geoip.city_name.keyword",
                                "size": 10
                        }
```

Here are the results. We gave it the default size of 10, meaning how far it should go. Since we have 18 cities in our data, "sum_other_doc_count" : 8 means it left off 8 records. Remember that ElasticSearch has many rules to keep performance high.

Notice that under each with these is a doc_count. So we had 6 web hits from the city of Montpellier.

```
"aggregations" : {
    "cityName" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 8,
      "buckets" :
    }
  }
}
```

Here we count the same data by **response**, e.g., 200, 404, etc. Since respond is of type long it's not necessary to add **keyword** to the end.

```
curl -XGET --user $pwd --header 'Content-Type: application/json'
https://58571402f5464923883e7be42a037917.eu-central-1.aws.cloud.es.io:9243/lo
gstash/_search?pretty -d '{
        "aggs": {
                "responses": {
                        "terms": {
                                "field": "response",
                                "size": 10
                        }
                }
        }
}'
```

Here are the results. Since there are only 4 types of responses in our data it showed all of them since that is < 10.

```
"aggregations" : {
    "responses" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" :
    }
  }
```

Adding a Query

Here is an example using a query to filter the results. This is from FDA drug interaction data that we will explore in our upcoming posts on analytics.

This query counts drug interactions by drug type, which is the whole purpose of the FDA database, to track drug interactions and side effects. So this is a bucket type aggregation query and subquery

```
curl -XGET --user $pwd --header 'Content-Type: application/json'
https://58571402f5464923883e7be42a037917.eu-central-1.aws.cloud.es.io:9243/fda/_search?pretty -d '{
"query": {
    "term": {
      "patient.drug.medicinalproduct.keyword": {
        "value": "METRONIDAZOLE"
      }
    }
  },
      "aggs" : {
         "drug": {
             "terms" : {
                "field": "patient.drug.medicinalproduct.keyword"

           }
         },
         "adverseaffect" : {
             "terms" : {
               "field": "patient.reaction.reactionmeddrapt.keyword"
             }
       }
   }
}'
```