Lab 3 : Start playing with search & aggregation queries

We would like be able to find some plays in the Shakespeare data set (111 396 docs) very easily.
This could be used (for example) to build a front end for users in Library to find plays they are looking for.

During this lab we will see the kind of queries we would use to do such a front end.
We could build a PHP, Java or whatever language page to generate these requests against localhost:9200.
In order to simplify the lab, we will write queries directly in Kibana > Dev Tools.

Please do not delete the queries you write in Dev Tools, you will be asked to provide it at the end of the lab.

Before to start, in order to make this lab more interesting, we gonna split the Shakespeare index currently having only 1 shard, into a new index with 2 shards, and work on this new index during this lab.

Make Shakespeare index readonly :

```
PUT /shakespeare/_settings
{
  "settings": {
    "index.blocks.write": true
  }
}
```

Create a new shakespeare-catalog index with 2 shards from this existing shakespear index :

```
POST shakespeare/_split/shakespeare-catalog
{
  "settings": {
    "index.number_of_shards": 2
  }
}
```

Delete the old 1 shard Shakespeare index

```
DELETE shakespeare
```

In the normal life, you would make use of aliases, so in case you need to change the underlying index name, for the application it is completely transparent.

Now that our index is ready, let’s search for all plays that match “The edge of war” in the “text_entry” field:

```
GET /shakespeare-catalog/_search
{
  "query": {
    "match": {
      "text_entry": "The edge of war"
    }
  }
}
```

3.1 Do a screenshot of your search results and add it to your report

You should see the following informations:

- **Took :** = Time in ms
- **Shards :** number of shards crossed as part of this search (should be 2 here)
- **Hits total :** the number of docs matching the request (by default if > at 10 000, it says 10 000 gte)
- **Max score :** the “score” of the doc matching the most the query term
- **Hits :** the list of the TOP 10 documents matching the requests (this is the default size, but you can add a size parameter to define how many documents you want to retrieve as part of the request)

You can see that the search “out of the box” is already very powerful. It return first the play containing exactly “the edge of war”, then some others plays with “the edge of …” but not “war”. This is why the next docs have a lower score.

But we have a LOT of docs (> 10 000) coming back from the search. This is because by default it will do a OR with all the words in the search query : THE or EDGE or OF or WAR.

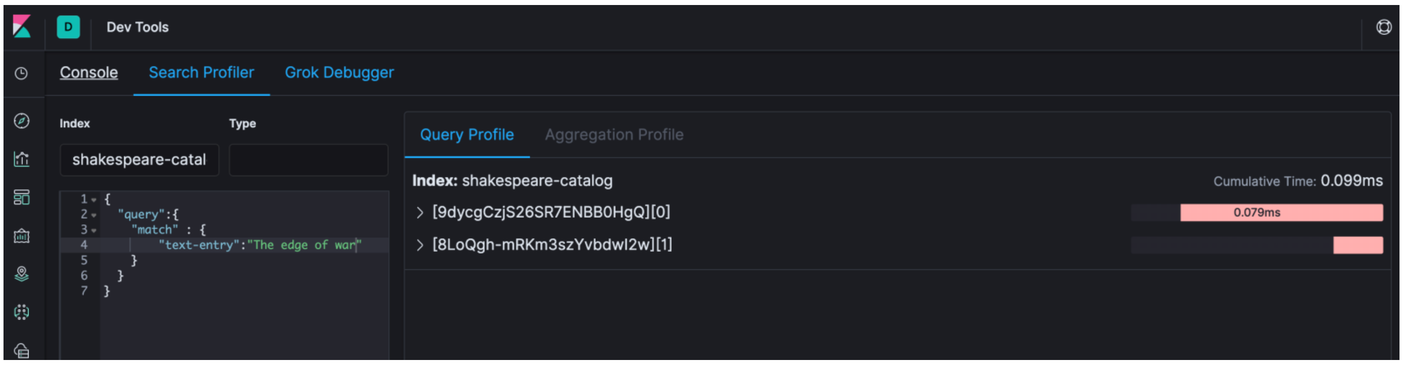
If you want to know the exact number of documents matching, you need to execute the request with this setting :

```
GET /shakespeare-catalog/_search
{
  "track_total_hits": true,
  "query": {
    "match": {
      "text_entry": "The edge of war"
    }
  }
}
```

But this is slower, so we consider if displaying the exact number of docs matching is an info you need for your end users or not.

Add the exact number of hits in your report that you will get from this request.

Let’s do a small pause here and go to the Search profiler in Dev Tools.
Select index = shakespeare-catalog - then write the query as you did in dev tool. Execute it :



shakespeare-catal

```
{
  "query":{
    "match" : {
        "text-entry":"The edge of war"
    }
  }
}
```

Index: shakespeare-catalog — Cumulative Time: 0.099ms

```
> [9dycgCzjS26SR7ENBB0HgQ][0]    0.079ms
> [8LoQgh-mRKm3szYvbdwl2w][1]
```

On the right you will see the shards that the search is crossing to grab the results.
You can see that you have the shard ID & the shard NUMBER.

Execute it multiple times. You can see that you always cross Shards 0,1 but for each of these shards number, the id is switching from one value to another. This is because the request will sometimes go to the primary shard or to the replica shard (both being on different nodes with different ID).

This illustrate what we discussed in the previous chapter.

3.2 Do a screenshot of your search profiler and add it to your report

---

Let’s come back to the Console in dev tools, and go further with search queries.

Let’s say we want to be much more restrictive on the results we get from the queries. We want only docs that contain ALL the words of the query. Let’s add an “AND” operator :

```
GET /shakespeare-catalog/_search
{
  "query": {
    "match": {
      "text_entry": {
          "query": "The edge of war",
          "operator": "AND"
      }
    }
  }
}
```

Now you get only 1 result, which is correct. But from a user experience perspective, presenting only 1 results vs 32 000+ with a OR query, is probably a little bit too restrictive.

Maybe it would be smarter to present only documents that contains AT LEAST 2 of the words in the search query, let’s do that with the “minimum_should_match” parameter :

```
GET /shakespeare-catalog/_search
{
  "query": {
    "match": {
      "text_entry": {
          "query": "The edge of war",
          "minimum_should_match": "2"
      }
    }
  }
}
```

With 2 minimum should match, we now get 6 912 results and still the most relevant at the top.
It’s better, but it is still a lot of results, let’s have a look at the results with a low score, by sorting the results by ascending order. We will get the 10 docs with the lowest score :

```
GET /shakespeare-catalog/_search
{
  "query": {
    "match": {
      "text_entry": {
          "query": "The edge of war",
          "minimum_should_match": "2"
      }
    }
  },
  "sort": [
    {
      "_score": {
        "order": "asc"
      }
    }
  ]
}
```

What do you see ? The docs return don’t contain EDGE or WAR. But they do contain THE and OF. Obviously we don’t want these results to be displayed, because it doesn’t make sense for the end user. Only main words should be taken into considerations. We would like to filter out “THE” & “OF”, this is the role of analyzers. Typically analyzers depend on the language so they can apply specific rules to filter out some keywords, …. Analyzers are defined in the mapping at the time of the index creation ….

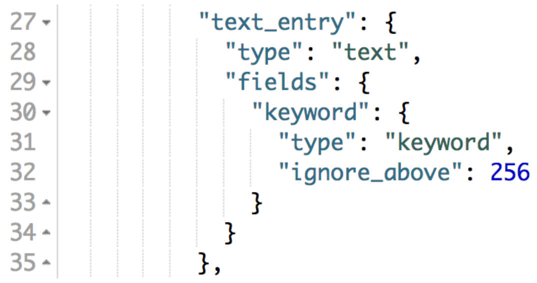
WAIT – we created a mapping for Shakespeare index back in time, don’t we ? YES

```
PUT /shakespeare
{
 "mappings": {
   "properties": {
    "speaker": {"type": "keyword"},
    "play_name": {"type": "keyword"},
    "line_id": {"type": "integer"},
    "speech_number": {"type": "integer"}
    }
  }
}
```

And this mapping was copied to shakespeare-catalog when we splitted the index.

But we didn’t specify anything for the “text_entry” field. So the default was applied (remember that ES try to guess the type based on the content of the first docs). Let’s have a look :

GET /shakespeare-catalog/_mapping



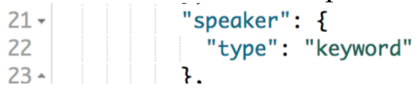
```
        "text_entry": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
```

So as text_entry contain some text, Elasticsearch decided to set is as “text”. It means it will be analyzed with the “default” analyzer. Also, it decided to create a sub field keyword, of type “keyword”. It means that it will be a copy of the value that will be NOT analyzed. This is used for aggregations. It’s probably useless for this field.

→ With these default settings for text fields, Elasticsearch make it possible to search and run aggregations on any text fields out of the box. But depending on the use case, you might not want / need to run search or aggregations on some specific fields. Make sure to set the right mapping.

Example :

We defined in the Shakespeare index mapping that spearker should be of type “keyword” only.



```
        "speaker": {
          "type": "keyword"
        },
```

What does it mean ? It means that we will not analyze it (try to remove some token or characters). We will just keep it as it is. But we will be able to run some aggregations on it such as COUNT the number of docs per speaker. This will be useful to show to the end user a list of all speakers and the corresponding number of plays for each. We will discuss this later in this lab.

Coming back to the text_entry field, we will define a proper mapping :

- No more keyword sub field
- Add a text sub field with an English analyzer

But you remember that we cannot change the mapping once the index is created ….

So first we will create a new index shakespeare-catalog-2 with the right mapping :

```
PUT /shakespeare-catalog-2
{
 "mappings": {
    "properties": {
      "speaker": {
        "type": "keyword"
      },
      "play_name": {
        "type": "keyword"
      },
      "line_id": {
        "type": "integer"
      },
      "speech_number": {
        "type": "integer"
      },
      "text_entry": {
        "type": "text",
        "fields": {
          "english": {
            "type": "text",
            "analyzer": "english"
          }
        }
      }
    }
 }
}
```

And then we will reindex the content of shakespeare-catalog into shakespear-catalog-2 :

```
POST _reindex
{
 "source": {
   "index": "shakespeare-catalog"
 },
 "dest": {
   "index": "shakespeare-catalog-2",
   "version_type": "internal"
 }
}
```

After a few seconds, check if you have the same number of docs in both indices via :

```
GET /_cat/indices/
```

3.3 Do a screenshot of the result of this command

Go back to the console. Let's reexcute the previous query but switching to the text_entry.english instead of text_entry :

```
GET /shakespeare-catalog-2/_search
{
  "query": {
    "match": {
      "text_entry.english": {
          "query": "The edge of war",
          "minimum_should_match": "2"
      }
    }
  },
  "sort": [
    {
      "_score": {
        "order": "asc"
      }
    }
  ]
}
```

Now we have only 1 result.

To see the difference between the two analyzers, you can execute these two queries :

```
GET /_analyze/
{
  "analyzer": "default",
  "text":"The edge of war"
}

GET /_analyze/
{
  "analyzer": "english",
  "text":"The edge of war"
}
```

You can see that when using English analyzer "the" and "of", are considered as stop words and removed. So with our new requests we are looking at docs that contains at least "edge" and "war", so only 1 doc is matching.

Let’s remove the minimum should match condition :

```
GET /shakespeare-catalog-2/_search
{
  "query": {
    "match": {
      "text_entry.english": {
          "query": "The edge of war"
      }
    }
  },
  "sort": [
    {
      "_score": {
        "order": "asc"
      }
    }
  ]
}
```

We now have 422 results, and even the last results contain at least “edge” or “war”, this sounds like a good user experience.

So our final query, fitting our requirements is as simple as :

```
GET /shakespeare-catalog-2/_search
{
  "query": {
    "match": {
      "text_entry.english": {
          "query": "The edge of war"
      }
    }
  },
  "size": 20
}
```

3.4 Do a screenshot of this search query results and add it to your report

The key here was to use the right analyzer. We can do very advance search queries with multiple fields match, boost, must or should match, … All of that is out of the scope of this lab but you can search online if you want to know more.

3.5 Do copy / paste all the queries you wrote in Dev Tools during this lab and add it to your report

Then cleanup a bit and delete the old shakespeare-catalog index :

DELETE shakespeare-catalog

Let’s play with aggregations

Now that we have a good “search”, in order to provide a better search experience, we would like to show “facets” / “filters” to the end user. We can easily achieve that by doing “terms aggregations” queries :

This query below will give us the list of plays and number of docs belonging to each play.

```
GET /shakespeare-catalog-2/_search
{
  "aggs" : {
        "play_names" : {
            "terms" : { "field" : "play_name" }
        }
    },
  "size": 0
}
```

3.6 Do a screenshot of your aggregation query results and add it to your report

This could be used to display a menu on the home page of our search application for example.
We specify size=0 here because we don’t want to actually retrieve some documents.

Now, let’s say we want to show facets after a user search.

Let’s reuse a previous search query, and now ask for a term aggregation on the “speaker field”.

```
GET /shakespeare-catalog-2/_search
{
  "query": {
    "match": {
      "text_entry": "The edge of war"
    }
  },
   "aggs" : {
        "speakers" : {
            "terms" : { "field" : "speaker" }
        }
    }
}
```

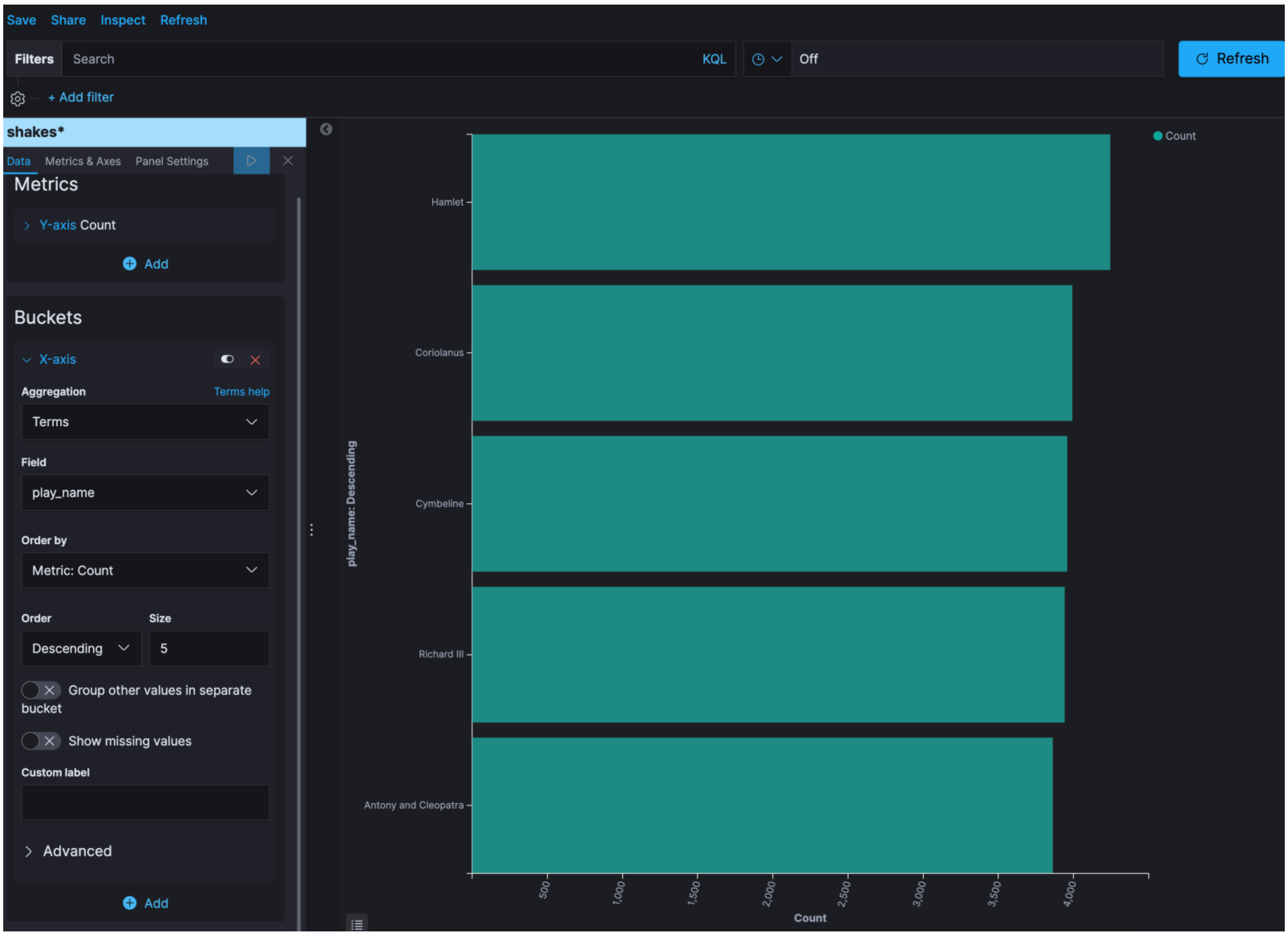
3.7 Do a screenshot of your search & aggregation query results and add it to your report

Thanks to that, the end user will now have the document matching “the edge of war” and will see a “speaker” facets belonging to these results. Then the end user could use these values to filter the search easily…

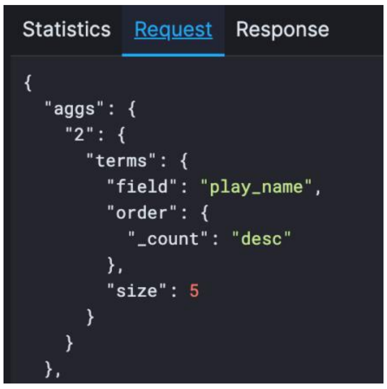
There is a ton of different aggregations we can do with Elasticsearch :
https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html

Let’s go to kibana in order to have a better overview of what’s possible …

First, let’s create a horizontal bar visualization for the shakes* index, for the X-Axis, select Terms aggregation on play_name. Click on “play” to see the result :

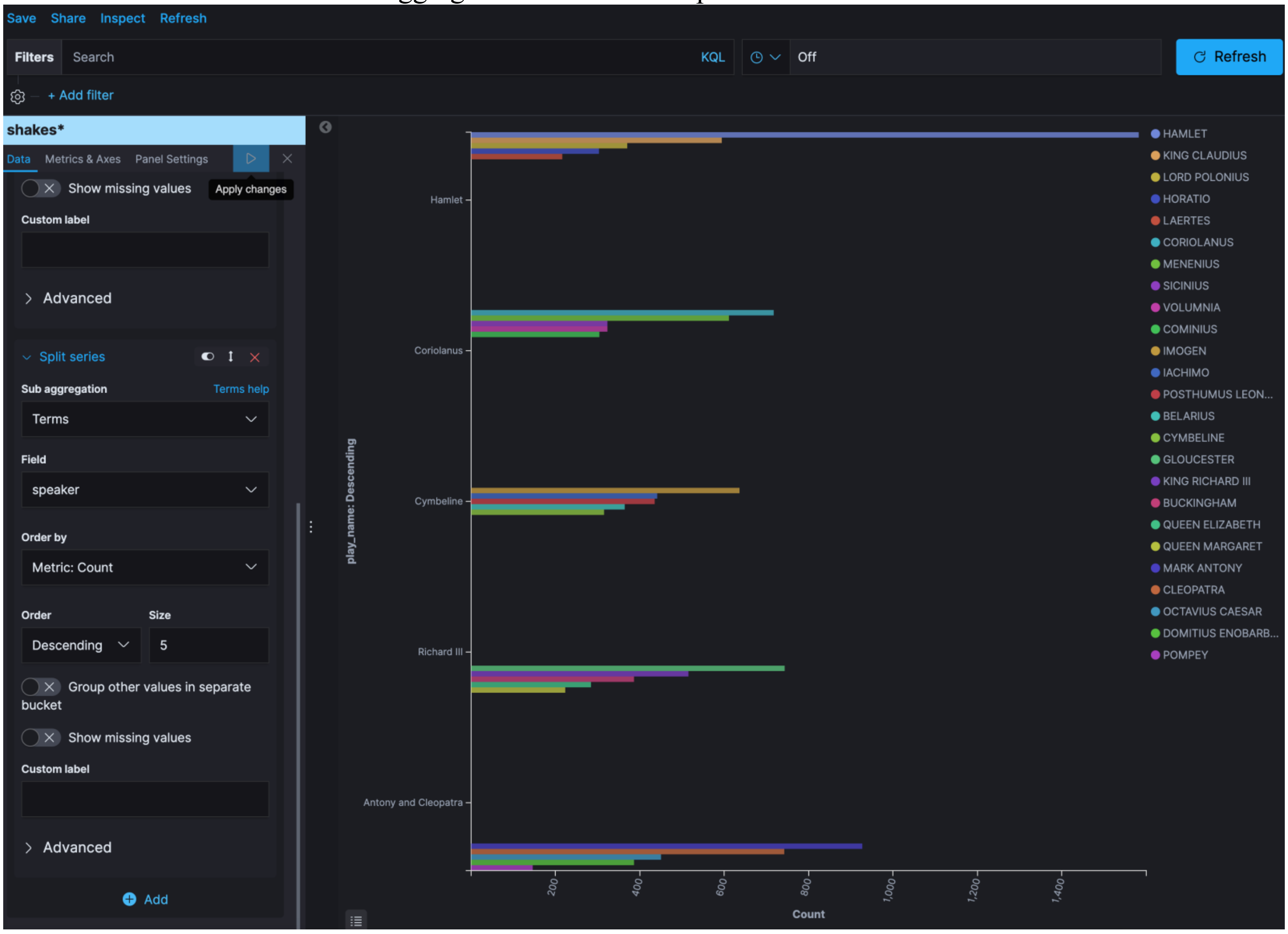


At the top, click on "Inspect" then select View:Requests in the right top corner and then finally the request tab - You can see that the request is almost the same than the one we wrote by ourselves. Except Kibana named the aggregated field "2" while we named it "play_names" on our side and also specify a size for the number of buckets & the order (desc).



```
{
  "aggs": {
    "2": {
      "terms": {
        "field": "play_name",
        "order": {
          "_count": "desc"
        },
        "size": 5
      }
    }
  },
```

Let’s do a sub bucket by asking Kibana to list the top 5 speakers of each play. Just click on “Add” > “Split series” on the left and add a sub aggregation on the term “speaker :



Show missing values

Custom label

Split series

Sub aggregation: Terms

Field: speaker

Order by: Metric: Count

Order: Descending

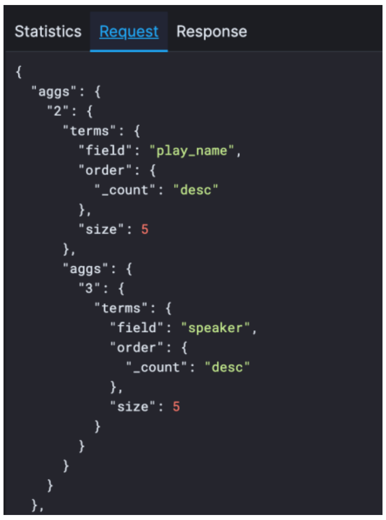
Size: 5

Group other values in separate bucket

Show missing values

Custom label

Inspect the request again, it’s very easy, we just add an aggregation within the first aggregation :



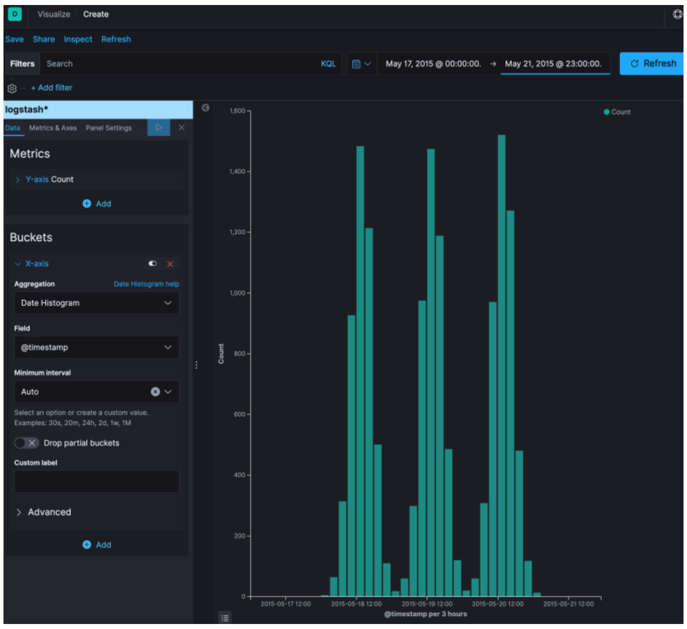
```
{
  "aggs": {
    "2": {
      "terms": {
        "field": "play_name",
        "order": {
          "_count": "desc"
        },
        "size": 5
      },
      "aggs": {
        "3": {
          "terms": {
            "field": "speaker",
            "order": {
              "_count": "desc"
            },
            "size": 5
          }
        }
      }
    }
  },
```

And look at the response, within each bucket from the first aggregation you will have sub-buckets for the second aggregation. Save your visualization.

Play around with the settings and see the impact on the visualisation, on the request and response.

Terms aggregation is very powerful, you can have many level of sub-buckets aggregations and Elasticsearch will still be very fast at returning results ….

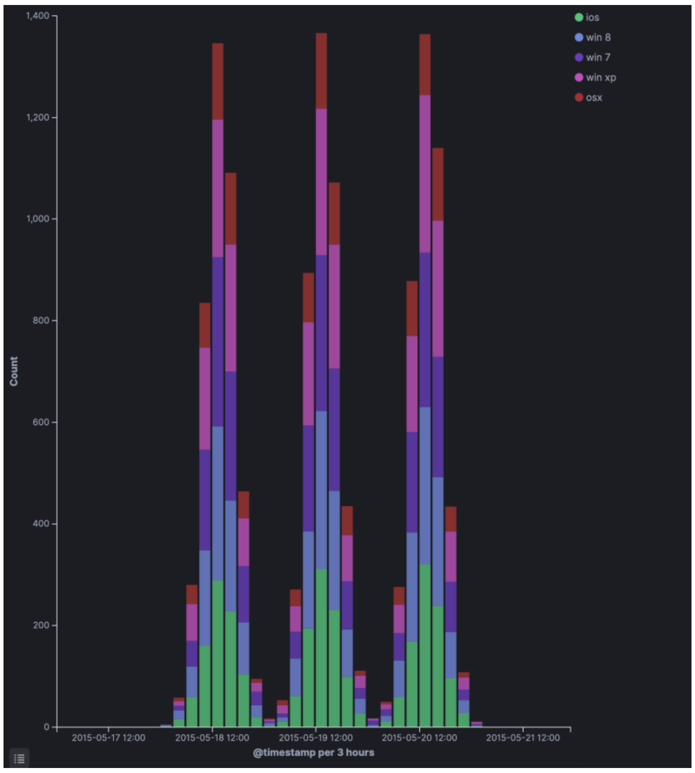
Now create a vertical bar visualization on logstash* - In the time picker at the top right corner select May 17 2015 to May 21 2015 (it's old data). Then in the settings on the left, keep Metrics as count, but in Buckets select Date Histogram and field timestamp :



Look at the request and response. Switch from Interval auto to daily. What change in the graph, request & response ?

3.8 Do a screenshot of the response in the inspect panel and add it to your report

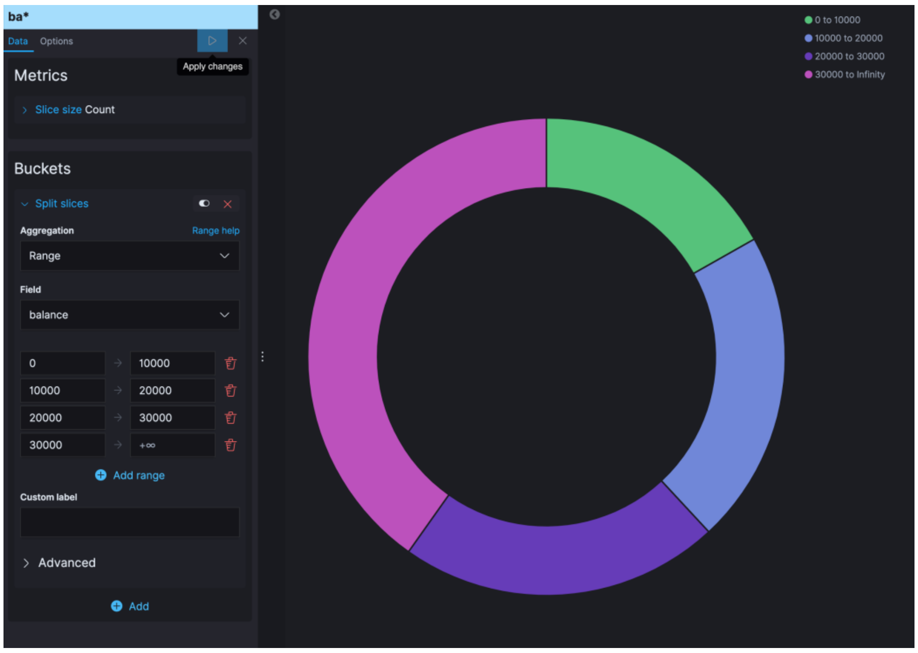
Go back to Auto interval and split series with a terms aggregation on field machine.os.keyword :



You now have the distribution of OS seen in logs files over time. Save your visualization.
Look at the request & response in inspect panel. Again very easy to request and to consume the response.

3.9 Do a screenshot of the response in the inspect panel and add it to your report

Last, add a pie visualization, where we count and split slices with a Range aggregation on the balance field and 4 buckets :



Slice size Count

Split slices

Aggregation: Range

Field: balance

0	10000
10000	20000
20000	30000
30000	+∞

Custom label

● 0 to 10000
● 10000 to 20000
● 20000 to 30000
● 30000 to Infinity

3.10 Do a screenshot of the response in the inspect panel and add it to your report

Add the sub bucket aggregation of your choice …

3.11 Do a screenshot of the visualization and settings and add it to your report

Save your visualization.

Create a markdown visualization with your name and save it.

Create a dashboard adding the 3 visualizations we created around aggregations + the markdown visualization and save it.

3.12 Do a screenshot of the dashboard and add it to your report

You now understand a little bit the power of aggregations. You saw that Kibana can be your best friend in order to easily build aggregations whether you plan to visualize it in Kibana or just to use the query for an external system.