

---

# TP4 – NOSQL ELASTICSEARCH

---

Manon GARDIN

Alexandre GARNIER

Tiphaine KACHKACHI

Matias OTTENSEN

Dataset : Restaurants\_Inspections

## Chapter 1 – Preprocessing

### Fixing Json format

To import our data in Kibana, a specific manipulation of the data was required to properly prepare it for indexing. The main purpose of this manipulation was to transform a JSON file containing a series of instance (in this case information about different restaurants) into a format compatible with the Elasticsearch \_bulk API.

The format required by the Elasticsearch \_bulk API differs slightly from the standard JSON structure, as it requires the inclusion of a specific header line before each JSON document to be indexed. This header line tells Elasticsearch what action to take (in this case, index) as well as metadata about the document, such as the index in which the document should be stored (\_index) and a unique identifier for the instance (\_id).

To accomplish this transformation, we developed a Python script that reads the original JSON file and generates a new file, where each document is preceded by the appropriate header line.

The script iterates on each line of the source file, treating each line as a separate JSON instance. And for each instance, it will generate a header line containing the index action and the target index \_index.

```
with open(input_file_path, 'r') as input_file, open(output_file_path, 'w') as output_file:
    # Lire chaque ligne du fichier
    for line in input_file:
        # Charger la ligne en tant qu'objet JSON
        document = json.loads(line)

        # Convertir les timestamps en dates pour chaque grade (en supprimant le sous-attribut $date)
        for grade in document["grades"]:
            grade["date"] = timestamp_to_date(grade["date"]["$date"])

        # Préparer le préfixe d'indexation avec l'_id correspondant au restaurant_id du document
        index_prefix = {
            "index": {
                "_index": "restaurants",
                "_id": int(document["restaurant_id"])
            }
        }

        # Écrire le préfixe d'indexation et le document au fichier de sortie
        output_file.write(json.dumps(index_prefix) + '\n')
        output_file.write(json.dumps(document) + '\n')
```

## Our dataset

See below the schema of our dataset. This schema will help us understand the attributes we have in this dataset, and their type :

```
{
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "address": {
        "type": "object",
        "properties": {
          "building": {
            "type": "string"
          },
          "coord": {
            "type": "object",
            "properties": {
              "type": {
                "type": "string"
              },
              "coordinates": {
                "type": "array",
                "items": {
                  "type": "number"
                }
              }
            }
          },
          "street": {
            "type": "string"
          },
          "zipcode": {
            "type": "string"
          }
        }
      },
      "borough": {
        "type": "string"
      },
      "cuisine": {
        "type": "string"
      },
      "grades": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "date": {
              "type": "object",
              "properties": {
                "$date": {
                  "type": "number"
                }
              }
            }
          }
        }
      }
    }
  },
}
```

```

        "grade": {
            "type": "string"
        },
        "score": {
            "type": "number"
        }
    }
},
"name": {
    "type": "string"
},
"restaurant_id": {
    "type": "string"
}
},
"required": ["address", "borough", "cuisine", "grades", "name",
"restaurant_id"]
}
}

```

## Cleaning data

We chose to change the date format to have it easier to read. We originally had a timestamp which is difficult to understand, and we changed it to 'YYYY-MM-DD' format.

For that, we added this feature in our python script with a function `timestamp_to_date` :

```

def timestamp_to_date(timestamp):
    dt_object = datetime.utcfromtimestamp(timestamp / 1000) # Convertir de millisecondes en secondes
    formatted_date = dt_object.strftime('%Y-%m-%d') # Format DD/MM/YYYY
    return formatted_date

```

Also, we could see in the Json schema before, that the date in grades was contained in "date.\$date". We chose to change it, so that the date is directly contained in the attribute "date". It will then be easier to query this attribute.

```

# Convertir les timestamps en dates pour chaque grade (en supprimant le sous-attribut $date)
for grade in document["grades"]:
    grade["date"] = timestamp_to_date(grade["date"]["$date"])

```

This time again we chose not to take care of the null values because they are not being problematic in our case.

## Chapter 2 – Queries

In order to not repeat ourselves in every query, let's explain the main body of a simple query :

- “GET /restaurants/\_search”: This line indicates that the query is a search (\_search) performed on the index named restaurants.
- “query”: This is the main part of the query that defines the search criteria.
- “bool”: This block specifies a boolean query. Boolean queries allow us to combine multiple queries like must, should, must\_not, and filter.

The response of Elasticsearch query are not in a pretty format, so we chose not to show the full results of our queries.

### Simple queries

#### #1 - Display the restaurants that have Italian cuisine

```
GET /restaurants/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "cuisine": "italian"
        }
      }
    }
  }
}
```

In our case, we only use “must” here as the Boolean query. Then, “match” is a clause of the query that is used to perform a text search on the specified fields. In it, we specify that specific field (here “cuisine”) and the value we want it to take (here “italian”).

The query returns 1537 values, every one of them having “Italian” cuisine.

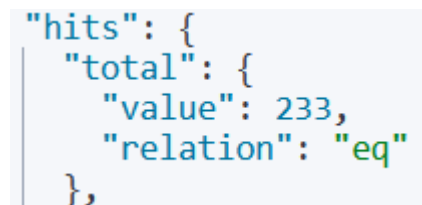
```
"hits": {
  "total": {
    "value": 1537,
    "relation": "eq"
  },
}
```

**#2 - Display the Italian restaurants located in Queens borough**

```
GET /restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "cuisine": "italian"
          }
        },
        {
          "match": {
            "borough": "queens"
          }
        }
      ]
    }
  }
}
```

Here, we execute two simple “match” conditions that we can contain in a “must” array. This time, we want a match on the cuisine but also on the borough.

The result will be every Italian restaurant in Queens, and we have 233 results.



```
"hits": {
  "total": {
    "value": 233,
    "relation": "eq"
  },
  ...
}
```

**#3 - Display all restaurants located in an "Avenue"**

```
GET /restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "wildcard": {
            "address.street": "*avenue*"
          }
        }
      ]
    }
  }
}
```

For this query, we use another filter than “match” this time. “wildcard” allows us to find every instance that have a street that contains the word “avenue” in it. Writing “\*avenue\*” will make the query find the word even if it is preceded or followed by other characters.

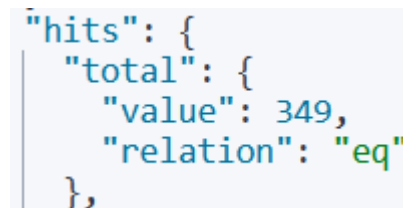
A lot of values have been returned with this query, and Kibana cannot display all of them.

**#4 - Display all restaurants that had at least one score greater than 50**

```
GET /restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "grades.score": {
              "gt": 50
            }
          }
        }
      ]
    }
  }
}
```

We use another block again, which is “range”. It will find all instances where at least one of the scores in the grades.score field is greater than 50. The property “gt”: 50 means “greater than 50”, filtering instances that have a grade score above.

We understand that the higher the score is, the lower the grade will be. So, this query is useful for identifying restaurants that received particularly low ratings. The query returns 349 restaurants in this case.



```
"hits": {
  "total": {
    "value": 349,
    "relation": "eq"
  },
}
```

**#5 - Display restaurants that have a least a score greater than 50 and is not American or Italian cuisine**

```
GET /restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "grades.score": {
              "gt": 50
            }
          }
        }
      ],
      "must_not": [
        {
          "match": {
            "cuisine": "american"
          }
        }
      ]
    }
  }
}
```

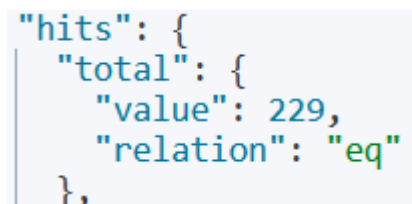
```

    },
    {
      "match": {
        "cuisine": "italian"
      }
    }
  ]
}
}
}

```

In this query, we combine several Boolean queries. We use the same “range” we did for the query #4, and we add a “must\_not” query. This block will filter the instance and won’t consider those with the specified value. Which means that the query won’t show restaurants whose “cuisine” is American or Italian.

As expected, we will have less value than in the query #4, because we won’t have every American or Italian restaurants.



```

"hits": {
  "total": {
    "value": 229,
    "relation": "eq"
  },

```

## #6 - Display all Japanese restaurants that have been graded in December of 2014

```

GET /restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "cuisine": "japanese"
          }
        }
      ],
      "filter": {
        "range": {
          "grades.date": {
            "gt": "2014-12-01",
            "lt": "2014-12-31"
          }
        }
      }
    }
  }
}

```

The first criterion of this query requires the “cuisine” field to match the word “japanese”. This filters out restaurants to return only those that offer Japanese cuisine. The second criterion applies a range filter to the “grades.date” field. This filter requires that the grade date be after December 1<sup>st</sup>,

2014 ("gt": "2014-12-01") and before December 31th, 2014 ("lt": "2014-12-31"), targeting grades given in December 2014.

Using a new block “filter” instead of “must” for the “range” query will not have an impact on our result, but it allows us to filter without affecting their relevance score, which improves performance by taking advantage of the filter cache for having fast and reusable queries.

```
"hits": {
  "total": {
    "value": 108,
    "relation": "eq"
  },
}
```

## Complex queries

### #1 - Display the number of restaurants per cuisine

```
GET /restaurants/_search
{
  "size": 0,
  "aggs": {
    "restaurant_per_cuisine": {
      "terms": {
        "field": "cuisine.keyword"
      }
    }
  }
}
```

This query performs an aggregation :

- The “size”: 0 permits to not return the instances themselves. We only need to count the number of restaurants by type of cuisine.
- It uses an aggregation of type terms on the “cuisine.keyword” field, which groups the instances by the unique value of that field.
- What does “.keyword” mean ? The .keyword is used to aggregate the exact values of the “cuisine” field.

```
"aggregations": {
  "restaurant_per_cuisine": {
    "doc_count_error_upper_bound": 0,
    "sum_other_doc_count": 9246,
    "buckets": [
      {
        "key": "American ",
        "doc_count": 6181
      },
      {
        "key": "Chinese",
        "doc_count": 2418
      },
      {
        "key": "Café/Coffee/Tea",
        "doc_count": 1214
      },
      {
        "key": "Pizza",
        "doc_count": 1163
      }
    ]
  }
}
```



**#2 - Display the number of distinct cuisine in Manhattan**

```
GET /restaurants/_search
{
  "size": 0,
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "borough": "Manhattan"
          }
        }
      ]
    }
  },
  "aggs": {
    "distinct_cuisine": {
      "cardinality": {
        "field": "cuisine.keyword"
      }
    }
  }
}
```

This query is specifically targeting restaurants located in the borough of Manhattan thanks to a “match” condition in the “must” clause. Next, it uses a “cardinality” aggregation named “distinct\_cuisine” to calculate the number of unique “cuisine” types among these restaurants.

This aggregation is performed on the “cuisine.keyword” field, which allows us to count the unique number of “cuisine” values, considering only restaurants in Manhattan. It’s an effective way to discover the variety of cuisines on offer in Manhattan. Our query returned 82 different cuisines.

```
"aggregations": {
  "distinct_cuisine": {
    "value": 82
  }
}
```

**Hard queries****#1 - Display the 20 better-scored cuisine in descending order**

```
GET /restaurants/_search
{
  "size": 0,
  "aggs": {
    "avg_score_per_cuisine": {
      "terms": {
        "field": "cuisine.keyword",
        "size": 20,
        "order": {
          "avg_score": "desc"
        }
      }
    }
  },
}
```

```

    "aggs": {
      "avg_score": {
        "avg": {
          "field": "grades.score"
        }
      }
    }
  }
}

```

This query performs an aggregation that calculates the average score assigned to each type of “cuisine”, with the results ordered by that average score in descending order. Here are the details of how it works:

- Main aggregation “avg\_score\_per\_cuisine”: It uses a “terms” aggregation on the “cuisine.keyword” field to group the instances by “cuisine” type.
- Nested aggregation “avg\_score”: For each “cuisine” group identified by the “terms” aggregation, this “avg” aggregation calculates the average score from the “grades.score” field. This gives an average of the scores for each “cuisine”.
- The “size”: 20 : It limits the number of “cuisines” to return. It’ll show the 20 most common “cuisine” types in the index.
- Order of results: The results of the “terms” aggregation are ordered according to the values of the nested aggregation “avg\_score” in descending order (“desc”). This means that the “cuisine” types with the highest average score will appear first.

The purpose of this query is to identify which kitchens receive the best ratings on average. The result won’t be showing us anything because the average is distorted by the number of restaurants (a few restaurants will have more impact on the mean). But it is coherent and showing what the query is asking for.

```

"aggregations": {
  "avg_score_per_cuisine": {
    "doc_count_error_upper_bound": -1,
    "sum_other_doc_count": 22495,
    "buckets": [
      {
        "key": "Iranian",
        "doc_count": 2,
        "avg_score": {
          "value": 17.125
        }
      },
      {
        "key": "Hawaiian",
        "doc_count": 3,
        "avg_score": {
          "value": 15.545454545454545
        }
      },
      {
        "key": "Chinese/Japanese",
        "doc_count": 59,
        "avg_score": {

```

## #2 - Display the most significant terms in the French restaurant's name

A “term” aggregation will act differently if the text field is parsed or not. Here, we want to access to every word of the attribute “name”. So, to make this query, we must proceed a mapping:

```
PUT restaurants/_mapping
{
  "properties": {
    "name": {
      "type": "text",
      "fielddata": true
    }
  }
}
```

By default, Elasticsearch disables fielddata for text fields because it can be expensive in terms of performance. However, enabling fielddata is required to parse the “name” field and to perform some aggregation operations on it. Then, we can do the query :

```
GET /restaurants/_search
{
  "size": 0,
  "query": {
    "match": {
      "cuisine": "french"
    }
  },
  "aggs": {
    "name_terms": {
      "terms": {
        "field": "name",
        "size": 10
      }
    }
  }
}
```

This query will filter restaurants to return only those that serve French cuisine (using “match” query).

Then it aggregates on the individual terms found in the “name” field of the filtered instances. This means that it identifies and counts the frequency of individual words that appear in the names of French restaurants.

The result will show the 10 most common words that appear in the names of these restaurants (“size”: 10).

```
"aggregations": {
  "name_terms": {
    "doc_count_error_upper_bound": 0,
    "sum_other_doc_count": 575,
    "buckets": [
      {
        "key": "le",
        "doc_count": 40
      },
      {
        "key": "cafe",
        "doc_count": 34
      },
      {
        "key": "la",
        "doc_count": 30
      },
      {
        "key": "bistro",
        "doc_count": 28
      },
      {
        "key": "restaurant",
        "doc_count": 12
      }
    ]
  }
}
```