

# BigData - Semaine 6

Pierre Nerzic

février-mars 2019

Nous allons étudier une autre sorte de base de données, Elasticsearch, tournée vers l'indexation et la recherche rapide d'informations parmi des méga-données.



Elasticsearch est un système qui regroupe un très grand nombre de fonctionnalités. Ce cours ne peut pas en faire l'inventaire et ne sera qu'un survol rapide de l'essentiel.

NB: c'est la version 1 du cours, elle contient sûrement quelques erreurs.

# Indexation et recherche

## Présentation rapide

Elasticsearch est l'aboutissement actuel des techniques d'indexation des documents, qui permettent de retrouver rapidement un document en fonction de mots-clés, d'expressions régulières, de fragment de texte, etc.

Un système numérique, proposé en 1876 par Dewey et régulièrement amélioré, par exemple par la *Online Computer Library Center* (OCLC), permet d'indexer les livres d'une bibliothèque. Un code numérique, appelé *cote*, ayant au moins 3 chiffres (classe, division, section) est attribué à chaque ouvrage. Ainsi tous les livres du même thème sont rangés par cote, au même endroit (et non pas dans l'ordre alphabétique des auteurs).

Ce système a été dépassé par internet. Chercher sur le contenu de toutes sortes de documents impose une indexation plus poussée.

## Elasticsearch et Lucene

Elasticsearch utilise **Apache Lucene** en interne. C'est une API Java libre permettant d'indexer des documents. L'index qu'elle produit fait entre 20 et 30 % de la taille des documents indexés, mais toutes sortes de recherches sont possibles, et les résultats sont classés par pertinence (*score*).

Lucene demande de programmer pour effectuer l'indexation : surcharger des classes, paramétrer et assembler des instances. Par exemple, redéfinir la manière dont les mots sont découpés dans un document, les mots à ignorer dans un texte (articles et mots de liaison). Voir **ce tutoriel**.

NB: Lucene évolue très rapidement<sup>1</sup>. Des classes et méthodes apparaissent, d'autres disparaissent. . .

---

<sup>1</sup>Version 7.7.0 mi-février 2019, 7.7.1 début mars et 8.0.0 mi-mars.

# Lucene, petit aperçu

## Exemple de programme Lucene, initialisation

Voici un petit programme Java qui illustre Lucene, le projet complet est dans [ExempleLucene.zip](#) :

```
// dossier pour stocker l'index
Directory index = MMapDirectory.open(Paths.get("index.dir"));

// analyseur de document, ici, c'est celui de base
StandardAnalyzer analyzer = new StandardAnalyzer();

// créer un écrivain d'index
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter writer = new IndexWriter(index, config);
```

Ça continue sur les transparents suivants...

## Exemple de programme Lucene, indexation

On continue avec l'ajout d'un document (à mettre dans une méthode pour insérer de nombreux documents similaires) :

```
// document
Document d1 = new Document();

// champs
d1.add(new TextField("titre", "Elasticsearch", Field.Store.YES))
d1.add(new StringField("isbn", "978-1449358549", Field.Store.YES)
d1.add(new IntPoint("pages", 687));

// enregistrement dans l'index
writer.addDocument(d1);

// fermeture de l'écrivain
writer.close();
```

## Exemple de programme Lucene, recherche

Ensuite, on crée une requête de recherche :

```
// construction d'une requête à l'aide d'un "assistant"
BooleanQuery.Builder builder = new BooleanQuery.Builder();

// le titre doit de préférence contenir "search"
Query query1 = new TermQuery(new Term("title", "search"));
builder.add(query1, BooleanClause.Occur.SHOULD);

// le nombre de pages doit être entre 150 et 400 inclus
Query query2 = IntPoint.newRangeQuery("pages", 150, 400);
builder.add(query2, BooleanClause.Occur.MUST);

// requête prête à être exécutée
BooleanQuery query = builder.build();
```

## Exemple de programme Lucene, résultats

Enfin, on lance la recherche :

```
// lancement de la recherche, 10 résultats au max
IndexReader reader = DirectoryReader.open(index);
IndexSearcher searcher = new IndexSearcher(reader);
TopScoreDocCollector collector=TopScoreDocCollector.create(10);
searcher.search(query, collector);

// affichage
for (ScoreDoc hit: collector.topDocs().scoreDocs) {
    Document doc = searcher.doc(hit.doc);
    System.out.println(doc.get("isbn")+" "+doc.get("title"));
}

// fermeture
reader.close();
```

# Elasticsearch

# Elasticsearch

Elasticsearch (ES) intègre Lucene pour effectuer l'indexation et la recherche, mais au lieu d'obliger à programmer en Java, il offre une interface REST plus souple.

ES est distribué de manière transparente et extensible (*scalable*) sur un groupe de machines (*cluster*).

Enfin, ES est accompagné de nombreux outils pour exploiter toutes sortes de données :

- analyse de données, graphiques
- apprentissage automatique (*machine learning*)

# Interface REST

Une interface REST (*Representational State Transfer*) consiste en un serveur HTTP couplé à une base de données :

- L'**URL** indique l'adresse de la donnée concernée, appelée *end-point*. Par exemple `http://serveur/produit/8763`, `http://serveur/client/342/achat/5/produit/8763`.
- L'**action** : lorsqu'on construit une requête HTTP, on indique une « méthode » parmi : GET, POST, PUT, DELETE. GET sert à lire une information, POST à en créer, PUT à modifier et DELETE à supprimer une information.
- Les **paramètres** sont encodés en JSON dans la « charge utile ».
- Le **code d'état** indique le succès de l'action : 200 ok, 201 créé, 202 modifié, 404 non trouvé, 409 en double...

Un service Web conforme à cette norme est appelé RESTful.

## Utilisation d'une interface REST

Le problème est de construire des requêtes HTTP avec un URL, une méthode et un contenu. Ce n'est pas commode :

- Avec une extension pour Firefox : **RESTclient**, *mais bug ?*
- En ligne de commande :

```
curl [-X méthode] url [-d contenu]
```

Exemples :

```
curl -X PUT http://serveur/produit/8763/nom -d Kiwi
```

```
curl -X DELETE http://serveur/produit/8763
```

- Par programme Python :



```
import requests
requests.post("http://serveur/produits",
             json={"id": "8763", "type": "fruit", "nom": "Pomme"})
requests.put("http://serveur/produit/8763/nom", "Kiwi")
requests.delete("http://serveur/produit/6541")
```

## Configuration bash pour travailler

Dans la suite du cours, on va faire de nombreuses références au serveur REST. Pour que ça soit plus pratique, on définit une variable shell :



```
SERVEUR='http://master:9200' # à l'IUT
```

Ainsi, on pourra faire :



```
curl -X GET "$SERVEUR/arbres/arbre/1"
```

Et on définit aussi un alias `curljson` comme ceci :



```
alias curljson='curl -H "Content-Type: application/json"'
```

Cet alias facilite l'envoi de documents JSON qui sont au cœur de ES, voir plus loin.

# Modèle de données dans Elasticsearch

## Index, types et documents

ES stocke des « documents ». Ils peuvent être du simple texte, ou être constitués de différents champs. Un document possède un identifiant unique, c'est ce qu'on appellerait *n-uplet* dans un SGBD.

Soit la liste des arbres de Paris. Dans un SGBD classique, on en aurait une base *Paris* contenant une table *Arbres*. Celle-ci contiendrait des *n-uplet*, chacun représentant un arbre avec différentes *colonnes* : taille, année...

Dans ES, on construira un **index** *Arbres* contenant des données du **type** *Arbre*. Ce type définit chaque **document** de l'index par différents **champs** : taille, année... Pour d'autres données sur Paris, on construirait un autre index, et/ou avec un autre type.

Attention, le mot *index* a donc deux sens : l'équivalent d'une BDD et une table pour accélérer les recherches.

## Index inversé

Chaque propriété de chaque document est indexée, ce qui permet de retrouver très rapidement tous les documents possédant telle valeur pour telle propriété.

Dans un SGBD classique, on construit un **index** sur une colonne, permettant de retrouver tous les n-uplets possédant une certaine valeur sur cette colonne.

Dans ES, on les appelle **index inversés**. ES construit la liste de tous les mots différents présents dans les documents et crée un index. On peut ensuite trouver immédiatement tous les documents qui contiennent tel mot.

## Mécanismes internes

En quelques mots :

- Elasticsearch est installé sur un *cluster* de machines appelées *nodes*.
- Les *index* regroupent des *documents* structurés en *types*.
- Quand un index est trop grand pour tenir sur un seul *node*, il est découpé en morceaux appelés *shards* (éclats).
- On peut également configurer ES pour répliquer les *shards* sur plusieurs machines.

De cette manière, les recherches sont distribuées, et donc très rapides même sur du Big Data.

NB: la notion de *type* est en cours de disparition. Il n'y a déjà qu'un seul *type* par index, et dans la version 7, les *types* auront disparu.

# JavaScript Object Notation

Les documents sont représentés en JSON (format concurrent de XML). Voici l'exemple de l'un des arbres de Paris :

```
{
  "geopoint": {
    "lat": 48.857140829,
    "lon": 2.2953345531
  },
  "arrondissement": 7,
  "genre": "Maclura",
  "espece": "pomifera",
  "famille": "Moraceae",
  "annee": 1935,
  "hauteur": 13,
  ...
}
```

## Stockage de données

Un tel document est placé sur ES par une requête PUT :

```
curljson -X PUT "$SERVEUR/arbres/arbre/1" -d '{
  "geopoint": {"lat": 48.857140829, "lon": 2.2953345531},
  "arrondissement": 7, ...
}'
```

L'URL \$SERVEUR/arbres/arbre/1 indique le serveur ES, l'index concerné, le type et enfin l'identifiant du document.

L'index arbres est créé s'il n'existe pas. Un second PUT sur le même identifiant écrase le précédent (mise à jour).

Le schéma de la table (du type arbre) est déterminé automatiquement en fonction des données ou préparé à l'avance, voir un peu plus loin.

## Stockage de données, suite

Dans le cas, préférable, où les données JSON sont dans un fichier, employer l'une des deux commandes :

```
curljson -X PUT "$SERVEUR/..." -d @fichier.json
```

```
cat fichier.json | curljson -X PUT "$SERVEUR/..." -d @-
```

L'alias `curljson` permet d'ajouter l'entête `Content-Type` indiquant que c'est du JSON.

L'option `-d @-` de `curl` fait lire les données sur `stdin`.

Cependant, il est rare d'avoir un seul document dans un fichier. Le transparent suivant montre comment insérer de nombreux documents d'un seul coup.

## Insertions en masse (*bulk loading*)

Le principe est d'insérer plusieurs documents ensemble par une seule requête POST. Ces documents sont mis dans un seul fichier (ou tube Unix), avec une alternance de deux types de lignes :

```
{"index":{"_index":"INDEX","_type":"TYPE","_id":"ID"}}\n{"champ1":"valeur1", "champ2":"valeur2", ... }\n
```

Les lignes impaires indiquent quel index, quel type et quel document affecter ; les lignes paires fournissent le contenu du document.

Ensuite, on transfère ce fichier par :



```
curl -H "Content-Type: application/x-ndjson" \  
-XPOST "$SERVEUR/_bulk" --data-binary @fichier.bulk
```

Le type MIME est x-ndjson, pas json, car c'est du *newline delimited JSON* : NDJSON. D'autre part, on poste sur `/_bulk`.

## Insertions en masse, préparation des données

Soit le fichier des arbres, `arbres.csv` à fournir à ES. Il faut séparer ses champs et en faire du NDJSON. Ce n'est pas très facile, alors je propose un script Python, `csv2ndjson.py` simple à utiliser : 

```
python csv2ndjson.py -i arbres -t arbre arbres.csv | \  
curl -H "Content-Type: application/x-ndjson" \  
-XPOST "$SERVEUR/_bulk" --data-binary @-
```

Le script produit les lignes nécessaires pour créer l'index indiqué par l'option `-i`, et le type de l'option `-t`. Le fichier CSV doit avoir une ligne de titres, et les champs séparés par des ;

En réponse, Elasticsearch affiche un bilan en JSON :

```
{ "took": 2258, "errors": false, "items": [ ... ] }
```

## Schéma d'un index

Lors de l'ajout du premier document, Elasticsearch déduit les types des champs de manière automatique, mais pas forcément correcte. Il est souvent nécessaire de définir le schéma manuellement avant toute insertion de données.

Dans le vocabulaire ES, on ne parle pas de schéma mais de *mappings* associés à un *type*. Ne pas confondre ce *type* qui représente une collection de documents, avec les types des champs.

ES gère de nombreux types de champs :

- texte : `text` ou `keyword` voir les explications page 29
- numériques : `long`, `integer`, `double`, `float`...
- divers : `boolean`, `date`, `geo_point`, `ip`
- tableau, objet JSON...

## Définition du schéma d'un index

Pour définir les types des champs (*mappings*) d'un index et d'un type, il faut émettre une requête PUT contenant la définition :

```
curljson -X PUT "$SERVEUR/INDEX?include_type_name=true" -d '{
  "mappings": {
    "TYPE": {
      "properties": {
        "CHAMP1": { "type": "TYPE1" },
        "CHAMP2": { "type": "TYPE2" },
        ...
      }
    }
  }
}'
```

L'option `include_type_name` prépare à la prochaine disparition des types (version 7 de ES).

## Types imbriqués

Certains champs peuvent être complexes. On met "properties" au lieu de "type" pour les définir :

```
"TYPE": {  
  "properties": {  
    "CHAMP1": { "type": "TYPE1" },  
    "CHAMP2": {  
      "properties": {  
        "CHAMP2_1": { "type": "TYPE2_1" },  
        "CHAMP2_2": { "type": "TYPE2_2" },  
        ...  
      }  
    },  
    ...  
  }  
}
```

On y accède avec une notation pointée : CHAMP2.CHAMP2\_1

## Définition du schéma d'un index, exemple

Voici les mappings du type arbre de l'index arbres :

```
"arbre": {
  "properties": {
    "espece":          { "type": "keyword" },
    "annee plantation": { "type": "integer" },
    "circonference":   { "type": "float" },
    "geopoint":        { "type": "geo_point" },
    "adresse":         { "properties": {
      "arrondissement": { "type": "integer" },
      "lieu":            { "type": "text" },
      ...
    }},
  }
}
```

## Types text et keyword

Ces deux types sont destinés à contenir des chaînes de caractères. Ils diffèrent dans la manière où le texte est indexé.

- Un champ `text` sert à contenir un texte quelconque qui est découpé en mots, dont chacun est indexé (index inversé), ce qui permet de faire des recherches sur des extraits de ce texte.
- Un champ `keyword` est pour une chaîne qui est mémorisée en tant qu'entité unique, un mot-clé. Il est impossible de faire une recherche sur des extraits, on ne peut chercher que sur la totalité du mot-clé.

On utilisera `text` pour un texte quelconque, et `keyword` pour une chaîne identifiante (adresse mail, n° de téléphone, etc.)

## Type text ou keyword ? Dilemme

Cela pose un dilemme pour certains champs. Soit par exemple, le nom commun d'un arbre. On rencontre « Séquoia géant » et « Séquoia sempervirent ».

Si on définit ce champ en tant que `keyword`, on ne pourra pas chercher les arbres dont le nom commun contient « géant ». On ne pourra le faire que si on le définit en tant que `text`.

Par contre, en tant que `text`, on ne pourra pas chercher la chaîne exacte « Séquoia géant », ES proposera aussi les autres séquoias et arbres géants. Cependant il leur donnera un score plus faible.

NB: il est possible que je rajoute un transparent sur le calcul du score en fonction de la ressemblance entre le texte et ce qu'on cherche.

## Types multiples

Pour résoudre le dilemme précédent, Elasticsearch propose d'associer plusieurs types au même champ (*multi-fields*), voir la [documentation](#).

```
"CHAMP": {  
  "type": "text",  
  "fields": {  
    "SOUS-NOM": { "type": "keyword" }  
  }  
},  
...
```

Le champ peut être accédé en tant que text par son nom et en tant que keyword par la notation pointée CHAMP.SOUS-NOM. En général, on met "raw", ou "keyword" à la place de "SOUS-NOM".

## Analyse des textes

Lorsqu'on ajoute un document, ses champs de type `text` sont découpés en mots et chaque mot est placé dans un index inversé, afin de retrouver instantanément les documents qui contiennent tel ou tel mot.

Déjà, les mots sont tous mis en minuscules. Mais chaque langue a ses spécificités concernant le découpage des mots et l'indexation de leurs variations (cheval = chevaux, etc.). Pour le français, on peut aider Elasticsearch à mieux indexer les mots en rajoutant un *analyzer* à la définition des champs `text` :

```
"CHAMP": {  
  "type":      "text",  
  "analyzer": "french"  
},  
...
```

# Recherches de documents

## Récupération d'un document précis

Pour récupérer un document par son identifiant, par exemple 97 :

```
curl -X GET "$SERVEUR/arbres/arbre/97"
```

ES retourne le document ainsi que des informations de contrôle :

```
{ "_index": "arbres",  
  "_type": "arbre",  
  "_id": "97",  
  "_version": 1,  
  "found": true,  
  "_source": {  
    "geopoint": {"lat": 48.83025320, "lon": 2.414005874},  
    "espece": "Acer",  
    ...  
  }  
}
```

# Recherches

Les recherches se font en interrogeant `/INDEX/_search`. Soit on fournit un URL qui spécifie la requête, soit c'est dans la charge utile en JSON, voir plus loin.

La recherche la plus simple consiste à demander tous les n-uplets enregistrés, à faire un GET sur `_search` sans filtre :

```
curl -X GET "$SERVEUR/arbres/_search"
```

L'option `-X GET` est inutile car présente par défaut.

On peut rajouter l'option `?pretty` pour un affichage lisible :

```
curl "$SERVEUR/arbres/_search?pretty"
```

Cette option `pretty` peut se rajouter à toutes les requêtes (bulk load, etc). Alors attention, s'il y a d'autres paramètres dans l'URL, il faut écrire `&pretty` pour respecter la syntaxe d'un URL.

## Recherches par URL avec critères

Pour filtrer les résultats sur la valeur d'un champ, on rajoute l'option `?q=champ:valeur`

```
curl "$SERVEUR/arbres/_search?q=genre:Acer"
```

Dans le cas général, espaces ou requête complexe, il faut encoder l'URL avec certaines options de `curl`. Voici un exemple :

```
Q='nom\ commun:Oranger\ des\ Osages'  
curl --get "$SERVEUR/arbres/_search" --data-urlencode "q=$Q"
```

La variable shell `$Q` rend le tout plus lisible, mais mettre un `\` devant les espaces et tous les caractères réservés comme `+=&|><!...`

Mettre `--data-urlencode "pretty"` à la fin pour un meilleur affichage.

## Critères complexes

Elasticsearch offre une syntaxe simple mais puissante (**doc**) :

- des opérateurs de comparaison qu'on place après le « : »

```
Q='hauteur:>30'
```

```
Q='annee\ plantation:<=1699'
```

```
Q='annee\ plantation:[1700 TO 1799]'
```

- les mots clés AND, OR et NOT et les ( ) pour combiner des filtres

```
Q='nom\ commun:Cèdre\ à\ encens AND adresse.arrondissement:8'
```

```
Q='(genre:Acer OR genre:Maclura) AND NOT hauteur:>30'
```

- les signes + et - pour exiger ou refuser des termes

```
Q='adresse.rue:(+Avenue -Foch -Quai)'
```

```
Q='adresse.rue:Avenue AND -adresse.rue:Foch'
```

## Résultats de recherche

Le résultat de la recherche est un document JSON généré par ES. Il contient différentes informations, ainsi que les documents trouvés :

```
{ "hits": {  
  "total": 3, "max_score": 2.5389738,  
  "hits": [ {  
    "_index": "arbres", "_type": "arbre",  
    "_id": "97",  
    "_score": 2.5389738,  
    "_source": {  
      "geopoint": "(48.8302532096, 2.41400587444)",  
      "arrondissement": "20",  
      ...  
    }  
  }  
]}
```

On peut noter la présence d'un score, c'est celui de Lucene. Il dépend de la coïncidence entre les mots cherchés et le document.

## Limitation des résultats

Pour éviter d'avoir trop de résultats, on peut ajouter une limite aux recherches, que ce soient des recherches par URL comme précédemment ou par DSL comme ce qui suit.

```
curl "$SERVEUR/arbres/_search?q=${Q}&size=N"
```

Le paramètre  $N$  donne le nombre de résultats voulus. C'est équivalent à un `LIMIT N` en SQL.

## Recherches avec DSL

Ce type de recherche, uniquement dans l'URL, est beaucoup trop simple. Pour faire mieux, ES propose un langage appelé DSL, *Domain Specific Language* (**doc**). On y écrit les requêtes en JSON.

Voici un aperçu, avec la recherche des arbres *Acer* :

```
curljson "$SERVEUR/arbres/_search?pretty" -d '{
  "query": {
    "match": {
      "genre": "Acer"
    }
  }
}'
```

Dans la suite, on ne mentionnera que la partie JSON.

# Requêtes JSON DSL

Les requêtes DSL ont cette structure générale :

```
{
  "query": {
    CLAUSE
  }
}
```

La clause définit les critères avec cette structure JSON :

```
"OPÉRATEUR": {
  "PARAMÈTRE": "VALEUR",
  "PARAMÈTRE": "VALEUR",
  ...
}
```

Voir l'exemple précédent.

## Opérateurs de recherche de texte

Deux opérateurs ont un peu le même rôle : chercher les documents par un texte. Cela dépend du type du champ : keyword ou text.

- `{"term": {"CHAMP": "VALEUR"}}` pour un champ de type keyword ou numérique, ES trouve les documents dont le champ est exactement égal à la valeur, voir [doc de term](#).

```
{ "query": { "term": { "nom commun": "Séquoia géant" }}}  
{ "query": { "term": { "hauteur": 42.0 }}}
```

- `{"match": {"CHAMP": "TEXTE"}}` et `{"match_phrase": {"CHAMP": "TEXTE"}}` pour un champ de type text, ES évalue la similitude entre le texte et ce que contient le champ, voir [doc de match](#).

```
{ "query": { "match": { "nom commun": "géant" }}}
```

## Opérateurs de recherches d'intervalles

- "range": {"CHAMP": {"COMPAR": "VALEUR", ...}}  
cherche les documents dont le champ correspond aux valeurs selon les comparateurs :
  - "gt" : greater than, "gte" : greater or equal,
  - "lt" : less than, "lte" : less or equal.

On cherche les arbres dont la hauteur est dans l'intervalle [40, 50[ :

```
{  
  "query": {  
    "range": {  
      "hauteur": { "gte": 40, "lt": 50 }  
    }  
  }  
}
```

## Opérateurs de recherche géographique

- Pour chercher les documents, dont un champ est de type `geo_point`, en fonction d'un éloignement à un point fixe :

```
{"geo_distance": {  
  "distance": "DISTANCE",  
  "CHAMP": { "lat": LATITUDE, "lon": LONGITUDE }  
}}
```

La distance est écrite en mètres : `m` ou kilomètres : `km`. Exemple :

```
{"query": {  
  "geo_distance": {  
    "distance": "1km",  
    "geopoint": { "lat": 48.864, "lon": 2.253 }  
  }  
}}
```

## Assemblage booléen

Pour des combinaisons de conditions :

```
{"bool": {  
  "must":      CLAUSE D'OBLIGATION,  
  "must_not":  CLAUSE D'INTERDICTION,  
  "should":    CLAUSE SOUHAITÉE,  
  "filter":    CLAUSE DE FILTRAGE  
}}
```

- "must" : ce qui doit être dans les documents,
- "must\_not" : ce qui doit être absent des documents,
- "should" : indique des alternatives,
- "filter" : pour des conditions non évaluées par un score.

NB: on peut mettre plusieurs clauses à l'aide d'un tableau JSON.

## Assemblage booléen, exemple

Afficher les arbres « Acer » qui ne sont ni dans le 12<sup>e</sup>, ni dans le 16<sup>e</sup> arrondissement :

```
{ "query": {  
  "bool": {  
    "must": { "match": { "genre": "Acer" } },  
    "must_not": [  
      { "term": { "arrondissement": 12 } },  
      { "term": { "arrondissement": 16 } }  
    ]  
  }  
}}
```

Remarquez le tableau pour grouper plusieurs clauses.

## Assemblage booléen, autre exemple

Afficher les arbres « Ginkgo » qui sont dans le 8<sup>e</sup> ou le 12<sup>e</sup> arrondissement, malheureusement, les Ginkgo des autres arrondissements sont aussi indiqués, mais avec un score plus faible :

```
{"query": {  
  "bool": {  
    "must": { "match": { "genre": "Ginkgo" }},  
    "should": [  
      {"term": { "arrondissement": 8 }},  
      {"term": { "arrondissement": 12 }}  
    ]  
  }  
}}
```

## Assemblage booléen, dernier exemple

Afficher uniquement les arbres « Ginkgo » qui sont dans le 8<sup>e</sup> ou 12<sup>e</sup> arrondissement, mais comme on utilise `filter`, le score est nul :

```
{ "query": {  
  "bool": {  
    "filter": {  
      "bool": {  
        "must": {  
          "match": { "genre": "Ginkgo" }  
        },  
        "should": [  
          { "term": { "arrondissement": 8 } },  
          { "term": { "arrondissement": 12 } }  
        ]  
      }  
    }  
  }  
}
```

## Opérateur de présence de valeurs

- {"exists": {"field": "CHAMP"}} cherche les documents qui possèdent ce champ et qui n'est pas null ou vide.

```
{ "query": { "exists": { "field": "hauteur" }}}}
```

- Pour l'inverse, on utilise un assemblage booléen :

```
{"query": {  
  "bool": {  
    "must_not": { "exists": { "field": "hauteur" }}  
  }  
}}
```

## Classement des résultats

ES permet de trier les résultats autrement que par leur score. Il suffit de rajouter un élément "sort" avant la requête :

```
{
  "sort": [
    { "CHAMP": "asc" ou "desc"},
    ...
  ],
  "query": ...
}
```

On peut simplifier quand il n'y a qu'un critère de tri :

```
{
  "sort": { "hauteur": "desc"},
  "query": { "match": { "genre": "Acer" }}
}
```

# Projection

Pour ne récupérer que certains champs des données recherchées, on ajoute une clause `_source` à la requête :

```
curljson "$SERVEUR/arbres/_search" -d '{
  "_source": false ou "CHAMP" ou [liste de champs],
  "query": {
    ...
  }
}'
```

Avec `false` seules les métadonnées sont affichées (nombre de résultats, score max, etc.). Si on indique des champs, seuls ces champs seront affichés en plus des métadonnées.

# Agrégations

## Réduction des données

Pour calculer une *réduction* (distinct, moyenne, somme, min, max...) des valeurs d'un champ :

```
curljson "$SERVEUR/INDEX/_search?size=0" -d '{
  "aggs": {
    "NOM": {
      "OPERATEUR": { "field": "CHAMP" }
    }
  }
}'
```

Le paramètre `size=0` de l'URL demande de ne pas afficher les documents ayant servi au calcul. D'autre part, ce n'est plus une "query" mais un "aggs". Le symbole "NOM" sert à nommer le résultat, par exemple "moyenne". Voir la [documentation](#).

## Opérateurs de réduction

- classiques : "sum", "avg", "min", "max"
- "percentiles" : retourne un aperçu de la distribution des données sous forme de **centiles** : quelle est la valeur telle qu'il y en a tant % plus petites, ex: 50% donne la valeur médiane.
- "percentile\_ranks" : permet de savoir quels pourcentages des données se trouvent en dessous de bornes fournies.

```
{
  "aggs": {
    "repartition années": {
      "percentile_ranks": {
        "field": "annee plantation",
        "values": [1700, 1800, 1900, 2000]
      }
    }
  }
}
```

## Opérateurs de réduction, suite

- Il manque un agrégateur classique à la liste précédente : le comptage des documents obtenus par une requête. Pour faire cela, on envoie une recherche non pas sur le service `_search`, mais sur `_count` :

```
curl "$SERVEUR/arbres/_count?q=genre:Acer"
```

- L'opérateur `"terms"` permet de grouper les valeurs identiques (distinctes), à condition que le champ soit un keyword :

```
{  
  "aggs": {  
    "genres": {  
      "terms": { "field": "genre" }  
    }  
  }  
}
```

# Divers

# Suppression de données

Il suffit d'envoyer une requête de type DELETE :

- Supprimer l'un des documents par son identifiant :

```
curl -X DELETE "$SERVEUR/arbres/arbre/IDENTIFIANT"
```

- Supprimer l'index entier :

```
curl -X DELETE "$SERVEUR/arbres"
```

- Supprimer certains documents, désignés par une recherche :

```
curljson -X POST "$SERVEUR/arbres/arbre/_delete_by_query" -d '{  
  "query": CLAUSE  
}'
```

# Kibana

# Présentation

Il s'agit d'une interface Web permettant de réaliser tous les traitements vus jusqu'ici beaucoup plus facilement. Cette interface intègre de nombreux outils, par exemple d'ajout de données, de recherche, de visualisation, d'analyse et d'intelligence artificielle.

PB: on ne pourra pas utiliser Kibana en TP car

- Le serveur ne supportera pas la charge de travail de plusieurs personnes simultanément.
- Il faut installer un module payant appelé x-pack pour ajouter la notion de compte dans Kibana et permettre à plusieurs personnes d'avoir des données personnelles.

## Illustration

5 hits New Save Open Share Reporting Auto-refresh

account\_number:<100 AND balance:>47500 Options 🔍

Add a filter +

ba\* 🔍 \_source

Selected Fields

- ? \_source

Available Fields

- Fields
- t \_id
- t \_index
- # \_score
- t \_type
- # account...
- t address
- # age
- # balance
- t city
- t email
- t employer
- t firstname
- t gender
- t lastname
- t state

1

- account\_number: 32 balance: 48,066 firstname: Dillard lastname: Mcpherson age: 34 gender: F address: 782 Quentin Street employer: Quailcom email: dillardcpherson@quailcom.com city: Veguito state: IN \_id: 32 \_type: account \_index: bank \_score: 2

2

- account\_number: 85 balance: 48,735 firstname: Wilcox lastname: Sellers age: 20 gender: M address: 212 Irving Avenue employer: Confrenzy email: wilcox sellers@confrenzy.com city: Kipp state: MT \_id: 85 \_type: account \_index: bank \_score: 2

3

- account\_number: 78 balance: 48,656 firstname: Elvira lastname: Patterson age: 23 gender: F address: 834 Amber Street employer: Assitix email: elvirapatterson@assitix.com city: Dunbar state: TN \_id: 78 \_type: account \_index: bank \_score: 2

4

- account\_number: 8 balance: 48,868 firstname: Jan lastname: Burns age: 35 gender: M address: 699 Visitation Place employer: Glasstep email: janburns@glasstep.com city: Wokulla state: AZ \_id: 8 \_type: account \_index: bank \_score: 2

5

- account\_number: 97 balance: 49,671 firstname: Karen lastname: Trujillo age: 40 gender: F address: 512 Cumberland Walk employer: Tsunami email: karentrujillo@tsunami.com city: Fredericktown state: MD \_id: 97 \_type: account \_index: bank \_score: 2

Collapse