



# Elastic Search

## Le moteur de recherche

---

David THIBAU – 2024

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- Introduction
  - L'offre
  - Concepts de base
  - Les API ELS
- Indexation et documents
  - Document API
  - Librairie clientes
  - Pipelines d'ingestion
  - API Search Lite
- Mapping et analyseurs
  - Types et champs full-text
  - Contrôler le mapping
  - Mapping dynamique
  - Analyseur custom
  - API de Réindexation
- Recherche avec DSL
  - Syntaxe DSL
  - Principaux opérateurs
  - Tri et pertinence
  - Contrôle du score
- Recherche avancée
  - Recherche avec préfixe
  - Recherche phrase
  - Recherche floue
  - Surbrillance
  - Agrégations
  - Géo-localisation
- Monitoring
  - Index et usage
- Annexes



# Introduction

---

## **L'offre**

Concepts de base

Les APIs



# Introduction

---

La société *elastic* propose une suite de produit nommé ElasticStack qui s'adresse à 2 aspects du BigData :

- **La Recherche en temps réel**

Des données indexées en permanence sont disponibles à la recherche, la latence due à l'indexation est négligeable (Near Real Time)

- **Analyse de données en temps réel.**

Les données sont agrégées en temps-réel afin de fournir des visualisations et tableau de bords permettant de détecter des informations pertinentes.

La clé de voûte de cette suite est le produit ***ElasticSearch***



# *ElasticSearch*

*ElasticSearch* est un serveur offrant une API REST permettant :

- De stocker et indexer tous types de données  
(Documents bureautique, tweet, fichiers journaux, métriques, ...)
- D'effectuer des requêtes de recherche  
(structurées, full-text, langage naturel, géographiques)
- D'effectuer différents types d'agrégations multi-critères



# Caractéristiques de l'offre

---

- ✓ Architecture massivement **distribuée et scalable** en volume et en charge => Big Data, performance remarquable
- ✓ **Haute-disponibilité** via la réplication
- ✓ **Muti-tenancy** ou multi-index. La base documentaire contient plusieurs index dont les cycles de vie sont complètement indépendants
- ✓ Basée sur la librairie de référence **Lucene** => Recherche full-texte, prise en compte des langues, du langage naturel, ...
- ✓ Stockage structuré des documents mais **sans schéma préalable**, possibilité d'ajouter des champs à un schéma existant
- ✓ **API RESTFu**l très complète
- ✓ **OpenSource**

A decorative graphic consisting of a vertical black line and several overlapping squares in blue, red, and yellow.

# Apache Lucene

---

- Projet Apache Démarré en 2000, utilisé dans de nombreux produits
- La « couche basse » d'ELS : une librairie Java pour écrire et rechercher dans les fichiers **d'index.**
- ELS ajoute à lucène :
  - Le clustering : distribution et la scalabilité,
  - Une API REST
  - La simplification de la configuration
  - Des fonctionnalités d'analyse de données (agrégation, machine learning)



# ELS vs SolR

La fondation Apache propose **SolR** qui est très proche d'ELS en termes de fonctionnalités

|   | SOLR                               | ELS   |
|---|------------------------------------|---|
| <b>Installation &amp; Configuration</b> | Documentation très détaillée       | Simple et intuitif                                    |
| <b>Indexation/ Recherche</b>            | Orienté Texte                      | Texte et autres types de données pour les agrégations |
| <b>Scalability</b>                      | Cluster via ZooKeeper et SolRCloud | Nativement en cluster                                 |
| <b>Communauté</b>                       | Importante mais stagnante          | A explosé ces dernières années                        |
| <b>Documentation</b>                    | Très complète et très technique    | Très complète, facile d'accès, bcp de tutoriaux       |





# Distributions

---

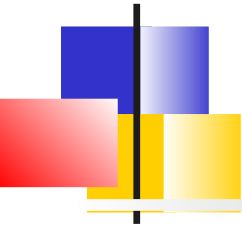
La distribution permettant d'installer *ElasticSearch* est disponible sous plusieurs formes :

- Archive ZIP, TAR
- Package : yum, apt, ..
- Containers

## Versions

- 8.0.0 : Février 2022
- 7.0.0 : 10 Avril 2019
- 6.0.0 : Novembre 2017
- 5.0.0 : Octobre 2016
- 2.4.1 : Septembre 2016
- 2.4.0 : Août 2016

# Configuration



Il existe plusieurs fichiers principaux de configuration :

- ***elasticsearch.yml*** : Propriétés propres au nœud ELS
- ***jvm.options*** : Options de la JVM
- ***log4j2.properties*** : Verbose et support de traces
- Annuaire utilisateur et certificats SSL

La configuration par défaut permet de démarrer rapidement après une indexation. En production, il faut quand même modifier certains paramètres.

Depuis la version 8.x, la sécurité est activée par défaut



# *Bootstrap check*

---

Au démarrage ELS effectue des vérifications sur l'environnement . Si ces vérifications échouent :

- En mode développement, des warning sont affichés dans les logs
- En mode production (écoute sur une adresse publique), ELS ne démarre pas

Ces vérifications concernent entre autre :

- Dimensionnement de la heap pour éviter les redimensionnements et le swap
- Limite sur le nombre de descripteurs de fichiers très élevée (65,536)
- Autoriser 2048 threads
- Taille et zones de la mémoire virtuelle (pour le code natif Lucene)
- Filtre sur les appels système
- Vérification sur le comportement lors d'erreur JVM ou de *OutOfMemory*



# Introduction

---

L'offre  
**Concepts de base**  
Les APIs



# Cluster

---

Un **cluster** est un ensemble de serveurs (nœuds) qui contient l'intégralité des données et offre des capacités de recherche sur les différents nœuds

- Il est identifié par son nom unique sur le réseau local (par défaut : "*elasticsearch*").

=> Un cluster peut être mono-nœud

=> Un nœud ne peut pas appartenir à 2 clusters distincts



# Nœud

---

Un **nœud** est un processus *e/s* identifié par un nom unique qui a rejoint un cluster

Le nombre de nœuds dans un cluster n'est pas limité, les nœud peuvent être ajoutés supprimés sans (trop) perturber le cluster

Un nœud peut avoir un ou plusieurs rôles :

- **master** : Peut être élu pour coordonner le cluster
- **data** : Stocke les données
- **ingest** : Ingère les données
- **ml** : Exécutes des jobs de Machine Learning
- ...



# Index

---

Un **index** est une collection de documents qui ont des caractéristiques similaires

- Par exemple un index pour les données client, un autre pour le catalogue produits et encore un autre pour les commandes

Un index est identifié par un nom (en minuscule)

- Le nom est utilisé pour les opérations de mise à jour ou de recherche

Dans un cluster, on peut définir autant d'index que l'on veut



# Shard

---

Un index peut stocker une très grande quantité de documents qui peuvent excéder les limites d'un simple nœud.

Pour pallier ce problème, ELS permet de sous-diviser un index en plusieurs parties nommées ***shards***

- Le nombre de shards est défini à la création de l'index

Chaque *shard* est un index indépendant contenant un sous-ensemble des données hébergé sur un nœud *data* du cluster





# Apports du sharding

---

Le sharding permet :

- De **scaler** le volume de contenu
- De **distribuer** et paralléliser les opérations => augmenter les performances

La mécanique interne de distribution lors de l'indexation et d'agrégation de résultat lors d'une recherche est complètement gérée par ELS et donc transparente pour l'utilisateur



# Réplica

---

Pour pallier à toute défaillance, un shard peut être **répliqué**

Le nombre de réplique est défini à la création mais peut être modifié dynamiquement

La réplication permet

- La **haute-disponibilité** dans le cas d'une défaillance d'un nœud (Une réplique ne réside jamais sur le nœud hébergeant le shard primaire)
- De **scaler** le volume des requêtes car les recherches peuvent être exécutées sur toutes les répliques en parallèle .



# Document

---

Un **document** est l'unité basique d'information stocké par les shards.

Il est constitué d'un ensemble de champs typés

Il peut être représenté avec JSON



# Introduction

---

L'offre  
Concepts de base  
**Les APIs**



# API d'ELS

---

ELS expose donc une API REST utilisant JSON

L'API est divisée en catégorie

- API **document** (CRUD sur document)
- API **d'index** (CRUD sur Index)
- API de **recherche** (*\_search*)
- API **cluster** : monitoring du cluster (*\_cluster*)
- API **cat** : Format de réponse tabulée pour gestion du cluster (*\_cat*)
- API **ingest** : Ingestion et transformation des données sources
- API **ML** : Gestion des jobs de ML
- ...



# Introduction

---

Les différentes API respectent un ensemble de conventions communes

- Possibilité de travailler sur plusieurs index
- Fonctions de calcul de Date pour spécifier *les* noms d'index.
- Options/paramètres communs



# Multiple index

---

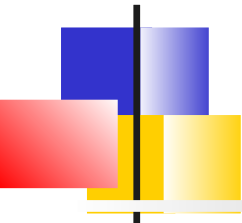
La plupart des APIs qui référencent un index peuvent s'effectuer sur plusieurs index :

test1, test2, test3

\*test

+test\*, -test3

\_all



# Noms d'index avec Date Math

---

La résolution d'index avec Date Math permet de restreindre les index utilisés en fonction d'une date.

Il faut que les index soient nommés avec des dates

La syntaxe est :

**<static\_name{date\_math\_expr{date\_format|time\_zone}}>**

- date\_math\_expr : Expression qui calcul une date
- date\_format : Format de rendu
- time\_zone : Fuseau horaire

Exemples :

GET /<logstash-{now/d}>/\_search => **logstash-2017.03.05**

GET /<logstash-{now/d-1d}>/\_search => **logstash-2017.03.04**

GET /<logstash-{now/M-1M}>/\_search => **logstash-2017.02**





# Options communes (1)

---

Les paramètres utilisent l'**underscore casing**

**?pretty=true** : JSON bien formaté

**?format=yaml** : Format yaml

**?human=false** : Si la réponse est traitée par un outil

**?filter\_path** : Filtre de réponse, permettant de spécifier les données de la réponse

Ex : GET `/_cluster/state?pretty&filter_path=nodes`

**?flat\_settings=true** : Les settings sont renvoyés en utilisant la notation « . » plutôt que la notation imbriquée

**?error\_trace=true** : Inclut la stack trace dans la réponse



# Alternatives Clients

---

- Clients REST classiques :  
*curl*, navigateurs avec add-ons, SOAPUI, ...
- Bibliothèques fournies par Elastic :  
*Java, Javascript, Python, Groovy, Php, .NET, Perl*  
Ces bibliothèques permettent l'équilibrage de charge sur les nœuds du cluster
- Enfin, le client le plus confortable est la  
*Dev Console* de Kibana

# DevConsole Kibana

- La **DevConsole** est composée de 2 onglets :
- L'éditeur permettant de composer les requêtes
  - L'onglet de réponse

La console comprend une syntaxe proche de Curl

```
PUT /megacorp/_doc/1
{
  "first_name" : "John",
  "last_name" : "Smith",
  "age" : 25,
  "about" : "I love to go rock climbing",
  "interests": [ "sports", "music" ]
}
```

```
1 - {
2   "_index" : "megacorp",
3   "_type" : "_doc",
4   "_id" : "1",
5   "_version" : 1,
6   "result" : "created",
7   "_shards" : {
8     "total" : 2,
9     "successful" : 1,
10    "failed" : 0
11  },
12  "_seq_no" : 0,
13  "_primary_term" : 1
14 }
```



# Fonctionnalités

---

- La console permet une traduction des commandes CURL dans sa syntaxe
- Elle permet l'auto-complétion, l'auto indentation
- Passage sur une seule ligne de requête (utile pour les requêtes BULK)
- Permet d'exécuter plusieurs requêtes
- Peut changer de serveur Elasticsearch
- Raccourcis clavier
- Historique des recherche



# Indexation et Documents

---

## **Document API**

Librairies clientes  
Pipelines d'ingestion  
API Search Lite



# Introduction

---

*ElasticSearch* est une base documentaire distribuée.

Il est capable de stocker et retrouver des structures de données (sérialisées en documents JSON) en temps réel

- Les documents sont constitués de champs.
- Chaque champ a un type
- Certains champs de type texte sont analysés pour permettre des recherche full-text



# Exemple document

---

```
{
  "name": "John Smith",      // String
  "age": 42,                 // Nombre
  "confirmed": true,         // Booléen
  "join_date": "2014-06-01", // Date
  "home": {                 // Imbrication
    "lat": 51.5,
    "lon": 0.1
  },
  "accounts": [             // tableau de données
    {
      "type": "facebook",
      "id": "johnsmith"
    }, {
      "type": "twitter",
      "id": "johnsmith"
    }
  ]
}
```



# Méta-données

---

Des méta-données sont également associées à chaque document.

Les principales méta-données sont :

- ***\_index*** : L'emplacement où est stocké le document
- ***\_id*** : L'identifiant unique
- ***\_version*** : N° de version du document
- ...





# Introduction

---

L'API document est l'API permettant les opérations CRUD sur les documents

- Création : *POST*
- Récupération à partir de l'ID : *GET*
- Mise à jour : *PUT*
- Suppression : *DELETE*



# Indexation et *id* de document

---

Un document est identifié par ses méta-données *\_index* et *\_id*.

Lors de l'indexation (insertion dans la base), il est possible

- de fournir l'ID
- ou de laisser ELS le générer



# Exemple avec Id

---

**POST** `/ {index} / _doc / {id}`

```
{  
  "field": "value",  
  ...  
}  
...  
{  
  "_index": {index},  
  "_id": {id},  
  "_version": 1,  
  "created": true  
}
```



# Exemple sans Id

---

**POST** `/_{index}/_doc/`

```
{
  "field": "value",
  ...
}
...
{
  "_index": {index},
  "_type": {type},
  "_id": "wM00SFhDQXGZAWDf0-drSA",
  "_version": 1,
  "created": true
}
```



# Récupération d'un document

---

La récupération d'un document peut s'effectuer en fournissant l'identifiant complet :

**GET** `/_{index}/_doc/{id}?pretty`

La réponse contient le document et ses méta-données. Exemple :

```
{  "_index" : "website",
    "_type" : "blog",
    "_id" : "123",
    "_version" : 1,
    "found" : true,
    "_source" : {
      "title": "My first blog entry",
      "text": "Just trying this out...",
      "date": "2014/01/01"    } }
```



# Partie d'un document

---

Il est possible de ne récupérer qu'une partie des données.

Exemples :

**# Les méta-données + les champs title et text**

GET /website/\_doc/123?\_source=title,text

**# Juste la partie source sans les méta-données**

GET /website/\_doc/123/\_source

**# Vérifier qu'un document existe (Retour 200)**

HEAD /website/\_doc/123



# Mise à jour

---

Les documents stockés par ELS sont immuables.

=> La mise à jour d'un document consiste à indexer une nouvelle version et à supprimer l'ancienne.

La suppression est asynchrone et s'effectue en mode batch  
(suppression de toutes les répliques)



# Mise à jour d'un document

---

**PUT /website/\_doc/123**

```
{  
  "title": "My first blog entry",  
  "text": "I am starting to get the hang of this...",  
  "date": "2014/01/02"  
}  
...  
{  
  "_index" : "website",  
  "_type" : "blog",  
  "_id" : "123",  
  "_version" : 2,  
  "created": false  
}
```





# Création ... if not exist

---

2 syntaxes sont possible pour créer un document seulement si celui-ci n'existe pas déjà dans la base :

PUT /website/\_doc/123?**op\_type=create**

Ou

PUT /website/\_doc/123/**\_create**

Si le document existe, ELS répond avec un code retour 409



# Suppression d'un document

---

```
DELETE /website/_doc/123
```

```
...
```

```
{
```

```
  "found" : true,
```

```
  "_index" : "website",
```

```
  "_type" : "blog",
```

```
  "_id" : "123",
```

```
  "_version" : 3
```

```
}
```



# Concurrence des mises à jour

---

Elasticsearch gère la concurrence des accès à sa base par une approche **optimiste**

En s'appuyant sur le champ **\_version**, il s'assure qu'une requête de mise à jour s'applique sur la dernière version du document.

Si ce n'est pas le cas (i.e., le document a été mis à jour par une autre thread entre-temps), il répond avec un code d'erreur 409

```
{  
  "error" : "VersionConflictEngineException[[website][2] [blog][1]:  
version conflict, current [2], provided [1]]",  
  "status" : 409  
}
```



# Mise à jour partielle

---

Les documents sont immuables : ils ne peuvent pas être changés, seulement remplacés

- La mise à jour de champs consiste donc à ré-indexer le document et supprimer l'ancien

L'API ***\_update*** utilise le paramètre ***doc*** ou ***script*** pour fusionner les champs fournis avec les champs existants



# Exemple

---

POST /website/\_update/1

```
{  
  "doc" : {  
    "tags" : [ "testing" ],  
    "views": 0  
  }  
}  
...  
{  
  "_index" : "website",  
  "_id" : "1",  
  "_type" : "blog",  
  "_version" : 3  
}
```



# Utilisation de script

---

Un script (*Groovy* ou *Painless*) peut être utilisé pour changer le contenu du champ `_source` en utilisant une variable de contexte : ***ctx.\_source***

Exemples :

```
POST /website/_doc/1/_update {
  "script" : "ctx._source.views+=1"
}
POST /website/_doc/1/_update {
  "script" : {
    "source" : "ctx._source.tags.add(params.new_tag)",
    "params" : {
      "new_tag" : "search"
    }
  }
}
```

N.B Les scripts peuvent également être chargés à partir de l'index particulier *.scripts* ou de fichier de configuration. Ils peuvent être utilisés dans d'autres contextes qu'une mise à jour.



# Bulk API

---

L'API **bulk** permet d'effectuer plusieurs ordres de mise à jour (création, indexation, mise à jour, suppression) en 1 seule requête

– => C'est le mode batch d'ELS.

Attention : Chaque requête est traitée séparément. L'échec d'une requête n'a pas d'incidence sur les autres.  
=> Pas de notion de transactions.



# Format de la requête

---

Le format de la requête est :

```
{ action: { metadata }}\n
```

```
{ request body }\n
```

```
{ action: { metadata }}\n
```

```
{ request body } \n
```

...

Le format consiste à des documents JSON sur une ligne concaténés avec le caractère **\n**.

- Chaque ligne (même la dernière) doit se terminer par **\n**
- Les lignes ne peuvent pas contenir d'autre **\n**  
=> Le document JSON ne peut pas être joliment formaté





# Syntaxe

---

La ligne action peut spécifier :

- ***create*** : Création d'un document non existant
- ***index*** : Création ou remplacement d'un document
- ***update*** : Mise à jour partielle d'un document
- ***delete*** : Suppression d'un document.

Elle précise les méta-données : *\_id*, *\_index*



# Exemple

---

```
POST /website/_bulk // website est l'index par défaut
{ "delete": { "_id": "123" }}
{ "create": { "_index": "website2", "_id": "123" }}
{ "title": "My first blog post" }
{ "index": {  }
{ "title": "My second blog post" }
{ "update": { "_id": "123"} }
{ "doc" : {"title" : "My updated blog post"} }
```



# Réponse

---

```
{
  "took": 4,
  "errors": false,
  "items": [
    { "delete": { "_index": "website",
      "_type": "_doc", "_id": "123",
      "_version": 2,
      "status": 200,
      "found": true
    } },
    { "create": { "_index": "website",
      "_type": "_doc", "_id": "123",
      "_version": 3,
      "status": 201
    } },
    { "create": { "_index": "website",
      "_type": "_doc", "_id": "EiwfApScQiiy7TIKFxRCTw",
      "_version": 1,
      "status": 201
    } },
    { "update": { "_index": "website",
      "_type": "_doc", "_id": "123",
      "_version": 4,
      "status": 200
    } }
  ]
}
```



# Indexation et Documents

---

Document API

**Librairies clientes**

Pipelines d'ingestion

API Search Lite



# Clients Elastic Search

---

Elastic fournit et supporte des clients écrits en différents langages :

- Java
- JavaScript
- .NET
- PHP
- Perl
- Python
- Ruby
- Go
- Eland
- Rust



# Principes

---

Pour chaque technologie, le principe consiste à construire un client à partir de la topologie du cluster ELS.

Le client permet de répartir la charge des requêtes

Ensuite, les appels de l'API sont effectués en utilisant

- Des méthodes dédiées au langage (Java, .NET)
- Directement le corps JSON de la requête (Javascript)

Des méthodes spécifiques sont proposées pour gérer la tolérance aux pannes



# Exemple Java

---

```
RestHighLevelClient client = new RestHighLevelClient(  
    RestClient.builder(  
        new HttpHost("localhost", 9200, "http"),  
        new HttpHost("localhost", 9201, "http")));  
  
GetRequest getRequest = new GetRequest(  
    "posts", "doc", "1");  
GetResponse getResponse = client.get(getRequest,  
    RequestOptions.DEFAULT);  
client.close();
```



# Exemple Javascript

---

```
var elasticsearch = require('elasticsearch');
var client = new elasticsearch.Client({
  host: 'localhost:9200',
  log: 'trace'
});
client.search({
  q: 'pants'
}).then(function (body) {
  var hits = body.hits.hits;
}, function (error) {
  console.trace(error.message);
});
```





# Indexation et Documents

---

Document API  
Librairies clientes  
**Pipelines d'ingestion**  
API Search Lite



# Concepts de l'ingestion

---

Certains nœuds du cluster peuvent être spécialisés en nœud de type ***ingest-nodes***. Leurs CPU sont alors utilisés pour pré-traiter des documents avant leur indexation

Les pré-traitements sont identifiés par une ***pipeline***

Le nom de la pipeline est précisée via le paramètre *pipeline* sur une requête bulk ou d'indexation

```
PUT my-index/_doc/my-id?pipeline=my_pipeline_id  
{ "foo": "bar"}
```



# Pipeline

---

Une **pipeline** est définie par 2 champs :

- Une description
- Une liste de **processeurs**

Les processeurs correspondent au ré-traitement effectués et s'exécutent dans leur ordre de déclaration



# Processeurs

---

ELS fournit de nombreux processeurs.

- Enrichissement  
*append, date\_index\_name, attachment, geo\_ip, ...*
- Transformation  
*convert, grok, rename, gsub, split, ...*
- Filtre  
*drop, remove, ...*
- Gestionnaire d'erreurs ou de routing  
*fail, pipeline, reroute, ...*
- Scripting  
*for\_each, script, ...*



# Processeur attachment

---

Le processeur ***attachment*** permet d'extraire le texte des fichiers dans les formats bureautiques les plus communs

Il utilise *Tika*



# Ingest API

---

L'API ***\_ingest*** permet de gérer les pipelines :

- ***PUT*** pour ajouter ou mettre à jour une pipeline
- ***GET*** pour retourner une pipeline
- ***DELETE*** pour supprimer
- ***SIMULATE*** pour simuler un appel à une pipeline



# Usage

---

## #Création de la pipeline nommée attachment

```
PUT _ingest/pipeline/attachment
{
  "description" : "Extract attachment information",
  "processors" : [
    { "attachment" : { "field" : "data" } }
  ]
}
```

## #Utilisation de la pipeline lors de l'indexation d'un doc.

```
PUT my_index/_doc/my_id?pipeline=attachment
{
  "data":
  "e1xydGYxXGFuc2kNCkxvcmVtIGlwc3VtIGRvbG9yIHNpdCBhbWV0DQpccGFyIH0="
}
```



# Résultat

---

```
GET my_index/my_type/my_id
{
  "found": true,
  "_index": "my_index",
  "_type": "my_type",
  "_id": "my_id",
  "_version": 1,
  "_source": {
    "data":
    "e1xydGYxXGFuc2kNCkxvcmVtIGlwc3VtIGRvbG9yIHNpdCBhbWV0DQpccGFyIH0=",
    "attachment": {
      "content_type": "application/rtf",
      "language": "ro",
      "content": "Lorem ipsum dolor sit amet",
      "content_length": 28
    }
  }
}
```





# Indexation et Documents

---

Document API  
Librairies clientes  
Pipelines d'ingestion  
**API Search Lite**



# APIs de recherche

---

Il y a donc 2 API de recherche :

- Une version **simple** qui attend que tous ses paramètres soient passés dans la chaîne de requête
- La version **complète** composée d'un corps de requête JSON qui utilise un langage riche de requête appelé DSL



# Search Lite

---

La version *lite* est cependant très puissante, elle permet à n'importe quel utilisateur d'exécuter des requêtes lourdes portant sur l'ensemble des champs des index.

Ce type de requêtes peut être un trou de sécurité permettant à des utilisateurs d'accéder à des données confidentielles ou de faire tomber le cluster.

=> En production, on interdit généralement ce type d'API au profit de DSL



# Recherche vide

---

GET /\_search

Retourne tous les documents de tous les index du cluster



# Réponse

---

```
{
  "hits" : {
    "total" : 14,
    "hits" : [ {
      "_index": "us",
      "_type": "tweet",
      "_id": "7",
      "_score": 1,
      "_source": {
        "date": "2014-09-17",
        "name": "John Smith",
        "tweet": "The Query DSL is really powerful and flexible",
        "user_id": 2
      }
    }, ... RESULTS REMOVED ... ],
    "max_score" : 1
  }, tooks : 4,
  "_shards" : {
    "failed" : 0,
    "successful" : 10,
    "total" : 10
  },
  "timed_out" : false
}
```



# Champs de la réponse

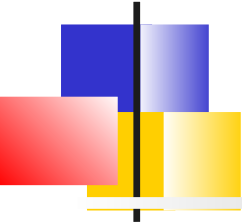
---

***hits*** : Le nombre de document qui répondent à la requête, suivi d'un tableau contenant l'intégralité des 10 premiers documents. Chaque document a un élément `_score` qui indique sa pertinence. Par défaut, les documents sont triés par pertinence

***took*** : Le nombre de millisecondes pris par la requête

***shards*** : Le nombre total de shards ayant pris part à la requête. Certains peuvent avoir échoués

***timeout*** : Indique si la requête est tombée en timeout. Il faut avoir lancé une requête de type :  
`GET /_search?timeout=10ms`



# Limitation à un index, un type

---

***\_search*** : Tous les index, tous les types

***/gb/\_search*** : Tous les types de l'index gb

***/gb,us/\_search*** : Tous les types de l'index gb et us

***/g\*,u\*/\_search*** : Tous les types des index commençant par g ou u

***/gb/user/\_search*** : Tous les documents de type user dans l'index

***/gb,us/user,tweet/\_search*** : Tous les documents de type user ou tweet présents dans les index gb et us

***/\_all/user,tweet/\_search*** : Tous les documents de type user ou tweet présents dans tous les index



# Pagination

---

Par défaut, seul les 10 premiers documents sont retournés.

ELS accepte les paramètres *from* et *size* pour contrôler la pagination :

- ***size*** : indique le nombre de documents devant être retournés
- ***from*** : indique l'indice du premier document retourné



# Paramètre *q*



L'API search lite prend le paramètre *q* qui indique la chaîne de requête dans la syntaxe lucene

La chaîne est parsée en une série de

- Termes : (Mot ou phrase)
- Et d'opérateurs (AND/OR)

La syntaxe support :

- Les caractères joker et les expressions régulières
- Le regroupement (parenthèses)
- Les opérateurs booléens (OR par défaut, + : AND, - : AND NOT)
- Les intervalles : Ex : [1 TO 5}
- Les opérateurs de comparaison



# Syntaxe Lucene

---

Recherche d'un terme sur le champ par défaut :

elastic

Sur un champ particulier

title:elastic

Recherche d'une phrase

title:"elastic search"

Combinaison opérateur par défaut OR

elastic search

elastic OR search

elastic AND search

+elastic +search

elastic NOT search

elastic -search

(+elastic +search) OR kibana



# Syntaxe Lucene (2)

---

## Recherche par préfixe:

ela\*

elas?tic

## Recherche de proximité

"elastic search"~5

elastik~2

## Recherche sur un intervalle

[10 TO 20]

{10 TO 20}

## Boosting

elastic^2 search



# Exemples search lite

---

GET /\_search?q=tweet:elasticsearch

Tous les documents de type tweet dont le champ full text *match* « elasticsearch »

+name:john +tweet:mary

Tous les documents dont le champ full-text *name* correspond à « john » et le champ *tweet* à « mary »

Le préfixe - indique des conditions qui ne doivent pas matcher



# Champ *\_all*

---

Lors de l'indexation d'un document, ELS concatène toutes les valeurs de type string dans un champ full-text nommé ***\_all***

C'est ce champ qui est utilisé si la requête ne précise pas de champ

GET /\_search?q=mary

Si le champ *\_all* n'est pas utile, il est possible de le désactiver



# Exemple plus complexe

---

La recherche suivante utilise les critères suivants :

- Le champ *name* contient « mary » ou « john »
- La date est plus grande que « 2014-09-10 »
- Le champ *\_all* contient soit les mots « aggregations » ou « geo »

***+name:(mary john) +date:>2014-09-10 +  
(aggregations geo)***

Voir doc complète :

<https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html#query-string-syntax>



# Autres paramètres de la query string

---

Les autres paramètres disponibles sont :

- **df** : Le champ par défaut, utilisé lorsque aucun champ n'est précisé dans la requête
- **default\_operator** (AND/OR) : L'opérateur par défaut. Par défaut OR
- **explain** : Une explication du score ou de l'erreur pour chaque hit
- **\_source** : *false* pour désactiver la récupération du champ *\_source*.  
Possibilité de ne récupérer que des parties du document avec *\_source\_include* & *\_source\_exclude*
- **sort** : Le tri. Par exemple *title:desc,\_score*
- **timeout** : Timeout pour la recherche. A l'expiration du timeout, les hits trouvés sont retournés



# Mapping et Analyseurs

---

## **Types simples et champs full text**

Contrôler le mapping

Mapping dynamique

Analyseur custom

Réindexation





# Types simples supportés

---

ELS supporte :

- Les chaînes de caractères : *string*
  - *text* ou *keyword* (pas analysé)
- Les numériques : *byte* , *short* , *integer* , *long* , *float* , *double* , *token\_count*
- Les booléens : *boolean*
- Les dates : *date*
- Les octets : *binary*
- Les intervalles : *integer\_range* , *float\_range* , *long\_range* , *double\_range* , *date\_range*
- Les adresses IP : *IPV4* ou *IPV6*
- Les données de géo-localisation : *geo\_point* , *geo\_shape*
- Une requête (structure JSON) : *percolator*



# Valeur exacte ou full-text

---

Les chaînes de caractère indexées par ELS peuvent être de deux types :

- **keyword** : La valeur est prise telle quelle, des opérateurs de type filtre peuvent être utilisés lors de la recherche  
Foo!= foo
- **text** : La valeur est analysée et découpée en termes ou token. Des opérateurs de recherche full-text peuvent être utilisés lors des recherche. Cela concerne des données en langage naturel



# Index inversé

---

Afin d'accélérer les recherches sur les champs *text*, ELS utilise une structure de donnée nommée **index inversé**

Cela consiste en une liste de mots unique où chaque mot est associé aux documents dans lequel il apparaît.

La liste de termes est constituée après une phase d'analyse du champ texte originel.



# Exemple

---

|    | A           | B                   |
|----|-------------|---------------------|
| 1  | term        | docs                |
| 2  | pizza       | 3, 5                |
| 3  | solr        | 2                   |
| 4  | lucene      | 2, 3                |
| 5  | sourcesense | 2, 4                |
| 6  | paris       | 1, 10               |
| 7  | tomorrow    | 1, 2, 4, 10         |
| 8  | caffè       | 3, 5                |
| 9  | big         | 6                   |
| 10 | brown       | 6                   |
| 11 | fox         | 6                   |
| 12 | jump        | 6                   |
| 13 | the         | 1, 2, 4, 5, 6, 8, 9 |



# Étapes de l'analyse

---

Les **analyseurs** transforment un texte en un flux de “termes”.

Ils sont constitués de 3 types de composants

- Les **filtres de caractères** préparent le texte en effectuant du remplacement de caractères ( & devient et) ou en supprimant (suppression des balises HTML)
- Les **tokenizers** splittent un texte en une suite d'unité lexicale : les termes
- Les **filtres** prend en entrée un flux de terme et le transforme en un autre flux de terme



# Analyseurs prédéfinis

---

- ELS propose des analyseurs directement utilisables :
  - **Analyseur Standard** : C'est l'analyseur par défaut. Le meilleur choix lorsque le texte est dans des langues diverses. Il consiste à :
    - Séparer le texte en mots
    - Supprime la ponctuation
    - Passe tous les mots en minuscule
  - **Analyseur simple** : Sépare le texte en token de 2 lettres minimum puis passe en minuscule
  - **Analyseur d'espace** : Sépare le texte en fonction des espaces
  - **Analyseurs de langues** : Ce sont des analyseurs spécifiques à la langue. Ils incluent les « stop words » (enlève les mots les plus courant) et extrait la racine d'un mot. C'est le meilleur choix si l'index est en une seule langue

# Test des analyseurs

GET /\_analyze?analyzer=standard

Text to analyze

Réponse :

```
{
  "tokens": [ {
    "token": "text",
    "start_offset": 0,
    "end_offset": 4,
    "type": "<ALPHANUM>",
    "position": 1
  }, {
    "token": "to",
    "start_offset": 5,
    "end_offset": 7,
    "type": "<ALPHANUM>",
    "position": 2
  }, {
    "token": "analyze",
    "start_offset": 8,
    "end_offset": 15,
    "type": "<ALPHANUM>",
    "position": 3
  } ] }
```



# Mapping et Analyseurs

---

Types simples et champs full text

**Contrôler le mapping**

Mapping dynamique

Analyseur custom

Réindexation





# Mapping

---

ELS est capable de générer dynamiquement le mapping

- Il *devine* alors le type des champs

On peut voir le mapping d'un index par :

GET /gb/\_mapping



# Réponse *\_mapping*

---

```
{ "gb": {  
  "mappings": {  
    "tweet": {  
      "properties": {  
        "date": {  
          "type": "date",  
          "format": "dateOptionalTime"  
        },  
        "name": {  
          "type": "text"  
        },  
        "tweet": {  
          "type": "text"  
        },  
        "user_id": {  
          "type": "long"  
        }  
      }  
    }  
  }  
}
```



# Type et Mapping

---

Un mapping définit pour chaque champ d'un document

- Le type de donnée
- Si champ de type *text*, comment ELS analyse le champ
- Les méta-données associées



# Comportement par défaut

---

Lorsque ELS détecte un nouveau champ *String* dans un type de document, il le configure automatiquement comme 2 champs :

- 1 champ *text* utilisant l'analyseur standard.
- 1 champ *keyword* nommé *<field\_name>.keyword*

Si ce n'est pas le comportement voulu, il faut explicitement spécifier le **mapping** lors de la création de l'index



# Mapping personnalisé

---

Un mapping personnalisé permet (entre autre) de :

- Faire une distinction entre les champs string de type full-text ou valeur exacte
- Utiliser des analyseurs spécifiques
- Optimiser un champ pour le matching partiel (voir + loin)
- Spécifier des formats de dates personnalisés
- Définir plusieurs types et/ou analyseurs pour le même champ

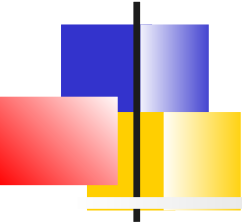


# Spécification du mapping

---

On peut spécifier le mapping :

- Lors de la création d'un index
  - Lors de l'ajout d'un nouveau type
  - Lors de l'ajout d'un nouveau champ dans un type existant
- => On ne peut pas modifier un champ déjà indexé



# Exemple (création)

---

PUT /gb

```
{
  "settings": { ... any settings : shards, replica,... },
  "mappings": {
    "properties" : {
      "tweet" : {
        "type" : "text",
        "analyzer": "english"
      },
      "date" : {
        "type" : "date"
      },
      "name" : {
        "type" : "keyword"
      },
      "user_id" : {
        "type" : "long"
      }
    }
  }
}
```



# Exemple (Ajout)

---

```
PUT /gb/_mapping/tweet
```

```
{
```

```
  "properties" : {
```

```
    "tag" : {
```

```
      "type" : "keyword"
```

```
    }
```

```
  }
```

```
}
```





# Double mapping

---

Un besoin relativement courant est d'indexer un même champ de plusieurs façons . L'attribut ***fields*** permet de définir des sous-champs ayant une autre stratégie d'indexation

```
"tweet": {  
  "type": "string",  
  "analyzer": "english",  
  "fields": {  
    "raw": {  
      "type": "keyword",  
    }  
  }  
}
```

Le nouveau sous-champ *tweet.raw* n'est pas analysé



# Mapping et Analyseurs

---

Types simples et champs full text

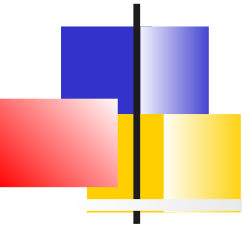
Contrôler le mapping

**Mapping dynamique**

Analyseur custom

Réindexation

# Dynamic Mapping



La détection de type lors de l'ajout d'un nouveau champ est appelée le ***dynamic mapping***

Les règles de détection peuvent être customisées via 2 moyens :

- L'activation ou la désactivation du *dynamic mapping*
- L'édition des gabarits dynamique (***dynamic templates***) spécifiant les règles de mapping pour les nouveaux champs



# Activation/Désactivation du dynamic mapping

---

La propriété **dynamic** permet de spécifier le comportement du *dynamic mapping* :

- **true** (défaut) : Chaque nouveau champ est automatiquement ajouté
- **false** : Tout nouveau champ est ignoré
- **strict** : Tout nouveau champ lève une exception

La spécification de *dynamic* peut s'effectuer sur l'objet racine ou sur une propriété particulière

```
PUT /my_index
{
  "mappings": {
    "my_type": {
      "dynamic": "strict",
      "properties": {
        "title": { "type": "string"},
        "stash": { "type": "object", "dynamic": true }
      }
    }
  }
}
```



# Règles de détection par défaut

---

Par défaut, en fonction du type JSON, ElasticSearch applique des correspondances :

***null*** : Pas d'ajout de champ

***true*** ou ***false*** : Champ *boolean*

***Point flottant*** : Champ *float*

***integer*** : Champ *long*

***object*** : champ *object*

***array*** : Dépend de la première valeur non nulle du tableau

***String*** :

- Soit un champ *date* (Possibilité de configurer les formats de détection)
- Soit un *double* ou *long* (Possibilité de configurer les formats de détection)
- Soit un champ *text* avec un sous-champ *keyword*.



# Exemple : Configuration des formats de détection

---

```
PUT my_index
{
  "mappings": {
    "dynamic_date_formats": ["MM/dd/yyyy"],
    "numeric_detection": true
  }
}
```



# Règles de détection

---

Les règles de détction permettent de définir un mapping en fonction :

- Du **datatype** détecté par ELS (propriété *match\_mapping\_type*).
- Du **nom** du champ ( propriétés *match*, *unmatch* ou *match\_pattern*).
- Du **chemin** complet du champ (propriétés *path\_match* et *path\_unmatch*).

Le nom d'origine du champ *{name}* et le type détecté *{dynamic\_type}* peuvent être utilisés comme variable lors de la spécification des règles



# Exemple : Ajouts de règles custom

---

PUT my\_index

```
{ "mappings": { "my_type": {  
  "dynamic_templates": [  
    { "integers": {  
      "match_mapping_type": "long",  
      "mapping": {  
        "type": "integer"  
      } } },  
    { "strings": {  
      "match_mapping_type": "string",  
      "mapping": {  
        "type": "text",  
        "fields": {  
          "raw": {  
            "type": "keyword",  
            "ignore_above": 256  
          } } } } } ] } } }
```





# Mapping et Analyseurs

---

Types simples et champs full text  
Contrôler le mapping  
Mapping dynamique  
**Analyseurs custom**  
Réindexation



# Analyseur explicite

---

- Lorsque ELS détecte un nouveau champ *String* dans un type de document,
  - il le configure automatiquement comme champ full-text et applique l'analyseur standard.
  - Ajoute un sous-champ non analysé (type keyword)
- Si ce n'est pas le comportement voulu, il faut explicitement spécifier le **mapping** pour le type de document



# Analyseurs

---

Les composants de l'analyseur standard sont :

- Le filtre de token standard (qui ne fait rien)
- Le tokenizer standard qui sépare sur les frontières de mots
- Le filtre lowercase qui passe tout en minuscule

Il est possible de surcharger cette configuration par défaut en ajoutant un filtre ou en remplaçant un existant



# Création par surcharge

---

Dans l'exemple suivant, un nouvel analyseur *fr\_std* pour l'index *french\_docs* est créé. Il utilise la liste prédéfinie des *stopwords* français :

```
PUT /french_docs
{
  "settings": {
    "analysis": {
      "analyzer": {
        "fr_std": {
          "type": "standard",
          "stopwords": "_french_"
        }
      }
    }
  }
}
```



# Création complète

---

Il est possible de créer ses propres analyseurs en combinant des filtres de caractères, un tokenizer et des filtres de tokens :

- 0 ou n : Filtres de caractères
- 1 : tokenizer
- 0 ou n : Filtre de tokens

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": { ... custom character filters ... },
      "tokenizer": { ... custom tokenizers ... },
      "filter": { ... custom token filters ... },
      "analyzer": { ... custom analyzers ... }
    }
  }
}
```



# Example

---

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": {
        "&_to_and": {
          "type": "mapping",
          "mappings": [ "&=> and " ]
        }, "filter": {
          "my_stopwords": {
            "type": "stop",
            "stopwords": [ "the", "a" ]
          }
        }
      },
      "analyzer": {
        "my_analyzer": {
          "type": "custom",
          "char_filter": [ "html_strip", "&_to_and" ],
          "tokenizer": "standard",
          "filter": [ "lowercase", "my_stopwords" ]
        }
      }
    }
  }
}
```



# Affectation à un index

---

Il faut ensuite associer l'analyseur à un champ *string*

```
PUT /my_index/_mapping
{
  "properties": {
    "title": {
      "type": "text",
      "analyzer": "my_analyzer"
    }
  }
}
```



# Analyseurs prédéfinis

Elasticsearch fournit des analyseurs pré-définis pouvant être utilisés directement :

- **Standard** (par défaut) : Sépare en mot, enlève la ponctuation, passe en minuscule, supporte les stop words
- **Simple** : Sépare en token dès qu'il trouve un caractère qui n'est pas une lettre. Passe en minuscule
- **Whitespace** : Se base sur les espaces pour la tokenization. Ne passe pas en minuscule
- **Stop** : Comme l'analyseur simple avec du support pour les stop words.
- **Keyword** : Ne fait rien. Prend le texte tel quel
- **Pattern** : Utilise une expression régulière pour la tokenization. Passe en minuscule et supporte les stop words.
- **Language** : english, french, ...
- **Fingerprint** : Crée une empreinte du document pouvant être utilisé pour tester la duplication.





# Filtres fournis

---

ELS fournit de nombreux filtres permettant de mettre au point des analyseurs personnalisés. Citons :

- **Length Token** : Suppression de mots trop courts ou trop longs
- **N-Gram** et **Edge N-Gram** : Analyzeur permettant d'accélérer les suggestion de recherche
- **Stemming filters** : Algorithme permettant d'extraire la racine d'un mot
- **Phonetic filters** : Représentation phonétique des mots
- **Synonym** : Correspondance de mots
- **Keep Word** : Le contraire de stop words
- **Limit Token Count** : Limite le nombre de tokens associés à un document
- **Elison Token** : Gestion des apostrophes (français)



# Utilisation de synonymes

---

Les synonymes peuvent être utilisés pour fusionner des mots qui ont quasiment le même sens.

Ex : joli, mignon, beau

Ils peuvent également être utilisés afin de rendre un mot plus générique.

- Par exemple, oiseau peut être utilisé comme synonyme à pigeon, moineau, ...



# Ajout d'un filtre synonyme

---

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonym_filter": {
          "type": "synonym",
          "synonyms": ["british,english", "queen,monarch"]
        }
      },
      "analyzer": {
        "my_synonyms": { "tokenizer": "standard",
                        "filter": ["lowercase","my_synonym_filter"]
        }
      }
    }
  }
}
```



# 3 utilisations

---

Le remplacement de synonyme peut se faire de 3 façons :

- Expansion simple : Si un des termes est rencontré, il est remplacé par tous les synonymes listés  
*"jump,leap,hop"*
- Contraction simple : un des termes rencontré est remplacé par un synonyme  
*"leap,hop => jump"*
- Expansion générique : un terme est remplacé par plusieurs synonymes  
*"puppy => puppy,dog,pet"*



# Mapping et Analyseurs

---

Types simples et champs full text  
Contrôler le mapping  
Mapping dynamique  
Analyseurs custom  
**Réindexation**



# Réindexation

---

Il n'est pas possible d'effectuer certains changements à posteriori sur un index. (ajout d'analyseur par exemple)

Pour réorganiser un index, la seule solution consiste à réindexer, i.e. créer un nouvel index avec la nouvelle configuration et copier tous les documents du vieil index vers le nouveau.

Cela s'effectue généralement avec une recherche de type « *scan and scroll* » et l'API « *bulk* » pour grouper les mises à jour



# Utilisation

---

L'utilisation s'effectue en positionnant les paramètres *search\_type* et *scroll* qui précise la durée pendant laquelle le scroll est ouvert

```
GET /old_index/_search?scroll=1m // 1 minute
{
  "query": { "match_all": {}}, "size": 1000
}
```

La réponse ne contient pas de hit mais un **scroll\_id** (Base 64 string) qui peut être utilisé pour récupérer le premier lot de documents

```
POST /_search/scroll?scroll=1m
c2Nhbjs10zEx0DpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlB0zEx0TpRNV9aY1VyUVM4U0
NMd2pjWlJ3YWlB0zExNjpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlB0zExNzpRNV9aY1Vy
UVM4U0NMd2pjWlJ3YWlB0zEyMDpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlB0zE7dG90YW
XfaGl0czox0w==
```

La réponse contient un autre **scroll\_id** permettant d'obtenir le lot suivant. Il faut alors répéter les requêtes jusqu'à ce qu'il n'y ait plus de documents



# API *\_reindex*

---

L'API ***\_reindex*** consiste à recopier les documents d'un index vers un autre index.

L'index cible peut avoir une configuration différente que l'index source (réplique, shard, mapping)

```
POST _reindex
{
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter"
  }
}
```





# API *\_reindex*

Il est possible de filtrer les documents recopiés et même d'interroger un cluster distant.

```
POST _reindex
{
  "source": {
    "remote": {
      "host": "http://otherhost:9200", // Cluster distant
      "socket_timeout": "1m",
      "connect_timeout": "10s"
    },
    "index": "source", // Sélection de l'index
    "size": 10000, // limitation sur la taille
    "query": {
      "match": { // limitation par une query
        "test": "data"
      }
    }
  },
  "dest": {
    "index": "dest"
  }
}
```



# Recherche avec DSL

---

## **Syntaxe DSL**

Principaux opérateurs

Tri et pertinence

Contrôle du score



# Introduction DSL

---

Les recherches avec un corps de requête offrent plus de fonctionnalités qu'une recherche simple.

En particulier, elles permettent :

- De combiner des clauses de requêtes plus facilement
- D'influencer le score
- De mettre en surbrillance des parties du résultat
- D'agréger des sous-ensemble de résultats
- De retourner des suggestions à l'utilisateur
- ...



# GET ou POST

---

La RFC 7231 qui traite de la sémantique HTTP ne définit pas des requêtes GET avec un corps

– => Seuls certains serveurs HTTP le supportent

ELS préfère cependant utiliser le verbe **GET** car cela décrit mieux l'action de récupération de documents

Cependant, afin que tout type de serveur HTTP puisse être mis en frontal de ELS, les requêtes GET avec un corps peuvent également être effectuées avec le verbe **POST**



# Recherches vides

---

```
GET /_search
```

```
{}
```

```
GET /index_2014*/type1,type2/_search
```

```
{}
```

```
GET /_search
```

```
{
```

```
  "from": 30,
```

```
  "size": 10
```

```
}
```



# Paramètre query

---

Pour utiliser DSL, il faut passer une requête dans le paramètre **query** :

```
GET /_search
```

```
{ "query": YOUR_QUERY_HERE }
```



# Structure de la clause Query

---

```
{  
  QUERY_NAME/OPERATOR: {  
    ARGUMENT: VALUE,  
    ARGUMENT: VALUE, ...  
  }  
}
```

Exemple :

GET /\_search

```
{  
  "query": {  
    "match": { "tweet": "elasticsearch" }  
  }  
}
```



# Principe DSL

---

DSL peut être vu comme un arbre syntaxique qui contient :

- Des clauses de requête **feuille**.  
Elles correspondent à un type de requête (*match*, *term*, *range*) s'appliquant à un champ. Elles peuvent s'exécuter seules
- Des clauses **composées**.  
Elles combinent d'autres clauses (feuille ou composée) avec des opérateurs logiques (*bool*, *dis\_max*) ou altèrent leurs comportements (*constant\_score*)

De plus, les clauses peuvent être utilisées dans 2 contextes différents qui modifient leur comportement

- Contexte **requête ou full-text**
- Contexte **filtre**





# Exemple Combinaison

---

```
{  
  "bool": {  
    "must": { "match": { "tweet": "elasticsearch" } },  
    "must_not": { "match": { "name": "mary" } },  
    "should": { "match": { "tweet": "full text" } }  
  }  
}
```



# Ex : Combinaison de combinaisons

---

```
{
  "bool": {
    "must": { "match": { "email": "business
opportunity" }},
    "should": [
      { "match": { "starred": true }},
      { "bool": {
        "must": { "folder": "inbox" }},
        "must_not": { "spam": true } }
    ]
  },
  "minimum_should_match": 1
}
```



# Distinction entre requête et filtre

---

DSL permet d'exprimer 2 types de requête

- Les **filtres** sont utilisés pour les champs contenant des valeurs exactes.  
Leur résultat est de type booléen, i.e. un document satisfait un filtre ou pas
- Les **recherches** calculent un score de pertinence pour chaque document trouvé.  
Le résultat est trié par le score de pertinence



# Activation des contextes

---

Le contexte *recherche* est activée dès lors qu'une clause est fournie en paramètre au mot-clé ***query***

Le contexte filtre est activée dès lors qu'une clause est fournie en paramètre au mot-clé ***filter*** ou ***must\_not***



# Exemple

---

GET /\_search

```
{
  "query": { // activation du contexte requête
    "bool": { // Les clauses bool, must et match sont exécutées dans un contexte requête
      "must": [
        { "match": { "title": "Search" } },
        { "match": { "content": "Elasticsearch" } }
      ],
      "filter": [ // activation du contexte filtre
        { "term": { "status": "published" } }, // exécuté dans un contexte filtre
        { "range": { "publish_date": { "gte": "2015-01-01" } } } // contexte filtre
      ]
    }
  }
}
```



# Performance filtre/recherche

---

La sortie de la plupart des filtres est une liste simple de documents qui satisfont le filtre. Le résultat peut être facilement caché par ELS

Les recherches doivent trouver les documents correspondant au mots-clés et en plus calculer le score de pertinence.

=> Les recherches sont donc plus lourdes que les filtres et sont plus difficilement cachable

=> L'objectif des filtres est de réduire le nombre de documents devant être examinés par la recherche



# Recherche avec DSL

---

Syntaxe DSL

**Principaux opérateurs**

Tri et pertinence  
Contrôle du score



# Opérateur de combinaison

---

***bool*** : Permet de combiner des clauses avec :

- ***must*** équivalent à ET avec contribution au score
- ***filter*** idem must mais ne contribue pas au score
- ***must\_not*** équivalent à NOT
- ***should*** équivalent à OU





# Types de recherche

---

Les requêtes textuelles peuvent être classifiées en 2 :

- Les requêtes n'effectuant pas d'analyse et opérant sur un terme unique (*term*, *fuzzy*).
- Les requêtes qui appliquent l'analyseur sur les termes recherchés correspondant au champ recherché (*match*, *query\_string*) .



# Opérateurs sans analyse

---

***term*** : Utilisé pour filtrer des valeurs exactes :

```
{ "term": { "age": 26 } }
```

***terms*** : Permet de spécifier plusieurs valeurs :

```
{ "terms": { "tag": [ "search", "full_text",  
"nosql" ] } }
```

***range*** : Permet de spécifier un intervalle de date ou nombre :

```
{ "range": { "age": { "gte": 20, "lt": 30  
} } }
```

***exists*** et ***missing*** : Permet de tester si un document contient ou pas un champ

```
{ "exists": { "field": "title" } }
```



## *match\_all, match\_none*

---

La recherche ***match\_all*** retourne tous les documents. C'est la recherche par défaut si aucune recherche n'est spécifiée. Tous les documents sont considérés également pertinents, ils reçoivent un `_score` de 1.

```
{ "match_all": {} }
```

Le contraire de *match\_all* est ***match\_none*** qui ne retourne aucun document.

```
{ "match_none": { } }
```



# *match*

---

La recherche ***match*** est la recherche standard pour effectuer une recherche exacte ou full-text sur presque tous les champs.

- Si la requête porte sur un champ full-text, il analyse la chaîne de recherche en utilisant le même analyseur que le champ,
- si la recherche porte sur un champ à valeur exacte, il recherche pour la valeur exacte

```
{ "match": { "tweet": "About Search" } }
```



# OR par défaut

---

*match* est une requête booléenne qui par défaut analyse les mots passés en paramètres et construit une requête de type OR. Elle peut être composée de :

- ***operator*** (and/or) :
- ***minimum\_should\_match*** : le nombre de clauses devant matcher
- ***analyzer*** : l'analyseur à utiliser pour la chaîne de recherche
- ***lenient*** : Pour ignorer les erreurs de types de données



# Exemple

---

GET /\_search

```
{
  "query": {
    "match" : {
      "message" : {
        "query" : "this is a test",
        "operator" : "and"
      }
    }
  }
}
```



# Contrôler la combinaison

---

```
GET /my_index/my_type/_search
{
  "query": { "match": {
    "title": {
      "query": "quick brown dog",
      "minimum_should_match": "75%" }
  } } }
```

Documents contenant 75 % des mots précisés



# *multi\_match*

---

La requête ***multi\_match*** permet d'exécuter la même requête sur plusieurs champs :

```
{"multi_match": { "query": "full text search", "fields":  
[ "title", "body" ] } }
```

Les caractères joker peuvent être utilisés pour les champs

Les champs peuvent être boostés avec la notation ^

```
GET /_search  
{  
  "query": {  
    "multi_match" : {  
      "query" : "this is a test",  
      "fields" : [ "subject^3", "text*" ]  
    }  
  }  
}
```





# *query\_string*

---

Les requêtes de types ***query\_string*** utilise un parseur pour comprendre la chaîne de requêtes. (le même que pour la recherche lite)

Il a de nombreux paramètres (default\_field, analyzer, ...)

```
GET /_search
{
  "query": {
    "query_string" : {
      "default_field" : "content",
      "query" : "title:(quick OR brown)"
    }
  }
}
```



# *simple\_query\_string*

---

L'opérateur ***simple\_query\_string*** permet de ne pas provoquer d'erreur de parsing de la requête de recherche.

Il peut être utilisé directement avec la saisie d'un utilisateur

Il prend la syntaxe simplifiée :

- + pour AND
- | pour OR
- - pour la négation
- ...



# Validation des requêtes

---

```
GET /gb/tweet/_validate/query?explain
{ "query": { "tweet" : { "match" : "really powerful" } }
...
{
  "valid" : false,
  "_shards" : { ... },
  "explanations" : [ {
    "index" : "gb",
    "valid" : false,
    "error" : "org.elasticsearch.index.query.QueryParseException:
[gb] No query registered for [tweet]"
  } ]
}
```



# Recherche avec DSL

---

Syntaxe DSL  
Principaux opérateurs  
**Tri et pertinence**  
Contrôle du score



# Tri selon un champ

---

Lors de requêtes de type filtre, il peut être intéressant de fixer le critère de tri.

Cela s'effectue via le paramètre **sort**

GET /\_search

```
{
  "query" : {
    "bool" : {
      "filter" : { "term" : { "user_id" : 1 } }
    }
  }, "sort": { "date": { "order": "desc" } }
}
```



# Réponse

---

Dans la réponse, la propriété `_score` n'est alors pas calculée et la valeur du champ de tri est précisée pour chaque document

```
"hits" : {  
  "total" : 6,  
  "max_score" : null,  
  "hits" : [ {  
    "_index" : "us",  
    "_type" : "tweet",  
    "_id" : "14",  
    "_score" : null,  
    "_source" : {  
      "date": "2014-09-24",  
      ...  
    },  
    "sort" : [ 1411516800000 ]  
  },  
  ...  
}
```



# Plusieurs critères de tri

---

Il est possible de combiner un critère de tri avec un autre critère ou le score. Attention l'ordre est important

```
GET /_search
{
  "query" : {
    "bool" : {
      "must": { "match": { "tweet": "manage text search" }},
      "filter" : { "term" : { "user_id" : 2 }}
    }
  }, "sort": [
    { "date":
      { "order": "desc" }},
    { "_score": { "order": "desc" }}]
}
```



# Tri sur les champs multivalués

---

Il est possible d'utiliser une fonction d'agrégation pour trier des champs multivalués (*min, max, sum, avg, ...*)

```
"sort": {  
  "dates": {  
    "order": "asc",  
    "mode": "min"  
  }  
}
```





# Pertinence

Le score de chaque document est représenté par un nombre à virgule (*\_score*). Plus ce chiffre est élevé, plus le document est pertinent.

L'algorithme utilisé par ELS est dénommé ***term frequency/inverse document frequency***, ou ***TF/IDF***. Il prend en compte les facteurs suivants :

- La ***fréquence du terme*** : combien de fois apparaît le terme dans le champ
- La ***fréquence inverse au niveau document*** : Combien de fois apparaît le terme dans l'index ? Plus le terme apparaît moins il a de poids.
- La ***longueur du champ normalisée*** : Plus le champ est long, moins les mots du document sont pertinents

En fonction du type de requête (requête floue, ...) d'autres facteurs peuvent influencer le score



# Explication de la pertinence

ELS permet d'obtenir une explication du score grâce au paramètre ***explain***

```
GET /_search?explain
```

```
{ "query"
: { "match" : { "tweet" : "honeymoon" }}
}
```

Le paramètre *explain* peut également être utilisé pour comprendre pourquoi un document match ou pas.

```
GET /us/tweet/12/_explain
```

```
{
"query" : {
  "bool" : {
    "filter" : { "term" : { "user_id" : 2 }},
    "must" : { "match" : { "tweet" : "honeymoon" }}
  } } }
```



# Réponse

---

```
"_explanation": {
  "description": "weight(tweet:honeyoon in 0)
[PerFieldSimilarity], result of:",
  "value": 0.076713204,
  "details": [ {
    "description": "fieldWeight in 0, product of:",
    "value": 0.076713204,
    "details": [ {
      "description": "tf(freq=1.0), with freq of:",
      "value": 1,
      "details": [ {
        "description": "termFreq=1.0",
        "value": 1
      } ]
    }, {
      "description": "idf(docFreq=1, maxDocs=1)",
      "value": 0.30685282
    }, {
      "description": "fieldNorm(doc=0)",
      "value": 0.25,
    } ]
  } ]
} ] }
```



# Regroupement

---

Le paramètre ***collapse*** permet de regrouper les résultats par la valeur d'un champ.

ELS ne retourne alors qu'un document par valeur distincte du groupe.

Ce regroupement est effectué après le tri.

```
GET /slides/_search?_source_includes=name
{
  "query": {
    "match_all": {}
  },
  "collapse": {
    "field": "content_type"
  }
}
```

```
"hits": [
  {
    "_source": {"name": "DPMicroServices.pdf" },
    "fields": { "content_type": [ "application/pdf" ] }
  },
  {
    "_source": { "name": "Birt.odp" },
    "fields": { "content_type": [ "application/html" ] }
  },
]
```



# Recherche avec DSL

---

Syntaxe DSL  
Principaux opérateurs  
Tri et pertinence  
**Contrôle du score**



# Cas de *multi\_match*

---

Le paramètre ***type*** précise le fonctionnement de l'opérateur et le calcul du score.

- ***best\_fields*** (défaut) : Trouve les documents qui match avec un des champs mais utilise le meilleur champ pour affecter le score
- ***most\_fields*** : Combine le score de chaque champ
- ***cross\_fields*** : Concatène tous les champs et utilise le même analyseur
- ***phrase*** : Utilise un *match\_phrase* sur chaque champ et combine le score de chaque champ
- ***phrase\_prefix*** : Utilise un *match\_phrase\_prefix* sur chaque champ et combine le score de chaque champ



# *multi\_match*

---

La requête ***multi\_match*** permet de facilement exécuter la même requête sur plusieurs champs. Le caractère joker peut être utilisé ainsi que le boost individuel

```
{  
  "multi_match": {  
    "query": "Quick brown fox",  
    "type": "best_fields",  
    "fields": [ "*_title^2", "body" ],  
    "tie_breaker": 0.3,  
    "minimum_should_match": "30%"  
  }  
}
```



# Cas de bool

---

```
GET /my_index/my_type/_search
```

```
{ "query": {  
  "bool": {  
    "must": { "match": { "title": "quick" }},  
    "must_not": { "match": { "title": "lazy" }},  
    "should": [  
      { "match": { "title": "brown" }},  
      { "match": { "title": "dog" }}  
    ] } } }
```

Le calcul de la pertinence s'effectue en additionnant le score de chaque clause *must* ou *should* et en divisant par 4





# Boosting clause

---

Il est possible de donner plus de poids à une clause particulière en utilisant le paramètre ***boost***

```
GET /_search
{
  "query": { "bool": {
    "must": { "match": {
      "content": { "query": "full text search", "operator": "and" }}}},
    "should": { "match": {
      "content": { "query": "Elasticsearch", "boost": 3 }}}
  } } }
```



# Opérateur boosting

L'opérateur **boosting** permet d'indiquer une clause qui réduit le score des documents qui matchent

GET /\_search

```
{
  "query": {
    "boosting" : {
      "positive" : {
        "term" : {"text" : "apple"}
      },
      "negative" : {
        "term" : { "text" : "pie tart fruit crumble tree" }
      },
      "negative_boost" : 0.5 // requis le malus au document qui matche
    }
  }
}
```



# *dis\_max*

Au lieu de combiner les requêtes via *bool*, il peut être plus pertinent d'utiliser ***dis\_max***

*dis\_max* est un OR mais le calcul de la pertinence diffère

*Disjunction Max Query* signifie : retourne les documents qui matchent une des requêtes et retourne le score de la requête qui matche le mieux

```
{ "query": { "dis_max": {  
  "queries": [  
    { "match": { "title": "Brown fox" }},  
    { "match": { "body": "Brown fox" }}  
  ]  
} } }
```



# *tie\_breaker*

---

Il est également possible de tenir compte des autres requêtes qui match en leur donnant une plus petite importance.

C'est le paramètre ***tie\_breaker***

```
{  
  "query": {  
    "dis_max": { "queries": [  
      { "match": { "title": "Quick pets" }},  
      { "match": { "body": "Quick pets" }}  
    ], "tie_breaker": 0.3 }  
  } }  
}
```

Le calcul du score est alors effectué comme suit :

- 1. Prendre le score de la meilleure clause .
- 2. Multiplier le score de chaque autres clauses qui matchent par le *tie\_breaker* .
- 3. Les ajouter et les normaliser



# Indexation multiple

---

Une technique très courante pour affiner la pertinence de documents et d'indexer plusieurs fois le même champ en utilisant des analyseurs différents.

Ensuite, un analyseur est utilisé pour trouver le plus de documents possibles, les autres pour différencier la pertinence des documents retournés.



# Indexation multiple

---

PUT /my\_index

```
{ "settings": { "number_of_shards": 1 },  
  "mappings": { "my_type": {  
    "properties": {  
      "title": { "type": "text", "analyzer": "english",  
        "fields": {  
          "std": { "type": "text", "analyzer":  
"standard" }  
        }  
      }  
    }  
  }  
}
```



# Utilisation

---

Utilisation du même champ indexé plusieurs fois

```
GET /my_index/_search
{
  "query": {
    "multi_match": {
      "query": "jumping rabbits",
      "type": "most_fields", // bool OR plutot que dis_max
      "fields": [ "title^10", "title.std" ]
    }
  }
}
```



# Indexation multiple

---

L'indexation multiple est également souvent utilisée pour concaténer dans un autre champ, plusieurs champs (variante du *\_all*).

```
PUT /my_index
{
  "mappings": {
    "person": {
      "properties": {
        "first_name": { "type": "string", "copy_to": "full_name" },
        "last_name": { "type": "string", "copy_to": "full_name" },
        "full_name": { "type": "string" }
      }
    }
  }
}
```





# *cross\_fields*

---

Si on a oublié d'effectuer une indexation multiple, on peut indiquer lors d'une requête multi-champs que l'on veut traiter ces champs comme un seul champ.

```
GET /books/_search
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "cross_fields",
      "fields": [ "last_name", "first_name" ]
    }
  }
}
```



# Recherches avancées

---

## **Recherche avec préfixe**

Recherche phrase

Recherche floue

Surbrillance

Agrégations

Géo-localisation



# Matching partiel

---

Le matching partiel permet aux utilisateurs de spécifier une portion du terme qu'il recherche

Les cas d'utilisation courant sont :

- Le matching de codes postaux, de numéro de série ou d'autre valeur *not\_analyzed* qui démarre avec un préfixe particulier ou même une expression régulière
- Recherche à la volée : des recherches effectuées à chaque caractère saisie permettant de faire des suggestions à l'utilisateur
- Le matching dans des langages comme l'allemand qui contiennent de long nom composés



# Filtre *prefix*

---

Le filtre ***prefix*** est une recherche s'effectuant sur le terme. Il n'analyse pas la chaîne de recherche et assume que l'on a fourni le préfixe exact

```
GET /my_index/address/_search
{
  "query": {
    "prefix": { "postcode": "W1" }
  }
}
```



# Wildcard et regexp

Les recherche **wildcard** ou **regexp** sont similaires à *prefix* mais permet d'utiliser des caractère joker ou des expressions régulières

```
GET /my_index/address/_search
{
  "query": {
    "wildcard": { "postcode": "W?F*HW" }
  }
}
```

```
GET /my_index/address/_search
{
  "query": {
    "regexp": { "postcode": "W[0-9].+" }
  }
}
```



# Indexation pour l'auto-complétion

---

L'idée est d'indexer les débuts de mots de chaque terme

Cela peut être effectué via un filtre particulier

***edge\_ngram***

```
{  
  "filter": {  
    "autocomplete_filter": { "type": "edge_ngram",  
                             "min_gram": 1,  
                             "max_gram": 20  
    }  
  }  
}
```

Pour chaque terme, il crée *n tokens* de taille minimum 1 et de maximum 20. Les tokens sont les différents préfixes du terme.



# Recherches avancées

---

Recherche avec préfixe

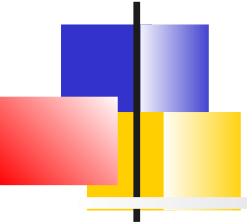
**Recherche phrase**

Recherche floue

Surbrillance

Agrégations

Géo-localisation



# *match\_phrase*

---

La recherche ***match\_phrase*** analyse la chaîne pour produire une liste de termes mais ne garde que les documents qui contient tous les termes dans le même position.

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": { "title": "quick brown fox" }
  }
}
```





# Proximité avec *slop*

Il est possible d'introduire de la flexibilité au phrase matching en utilisant le paramètre ***slop*** qui indique à quelle distance les termes peuvent se trouver.

Cependant, les documents ayant ces termes les plus rapprochés auront une meilleur pertinence.

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": {
        "query": "quick fox",
        "slop": 20 // ~ distance en mots
      }
    }
  }
}
```



# *match\_phrase\_prefix*

---

La recherche ***match\_phrase\_prefix*** se comporte comme *match\_phrase*, sauf qu'il traite le dernier mot comme un préfixe  
Il est possible de limiter le nombre d'expansions en positionnant *max\_expansions*

```
{  
  "match_phrase_prefix" : {  
    "brand" : {  
      "query": "johnnie walker bl",  
      "max_expansions": 50  
    }  
  }  
}
```



# Recherches avancées

---

Recherche avec préfixe

Recherche phrase

**Recherche floue**

Surbrillance

Agrégations

Géo-localisation



# Introduction

---

Pour augmenter les résultats d'une recherche en langage naturel, plusieurs techniques sont utilisées :

- Supprimer les diacritiques comme ´ , ^ , et ¨. Ainsi « rôle » match « role »
- Supprimer les distinctions entre singulier et pluriel, les conjugaisons en normalisant chaque mot à sa racine (stemming)
- Supprimer les mots trop communs (stopwords) comme « et » « le » « un »
- Inclure des synonymes afin que « UE » puisse matcher « Communauté européenne »
- Vérifier les fautes de typo ou travailler sur la phonétique



# Analyseur dédié à une langue

---

ELS fournit des analyseurs dédiés aux langues les plus courantes

Ces analyseurs effectuent en général 4 étapes :

- Sépare le texte en mots
- Passe chaque mot en minuscule
- Supprime les mots communs (stopwords)
- Remplace le terme par leur racine (Stem)

En fonction des langues, d'autres filtres peuvent être appliqués :

- Ex en FR, Elision Filter qui supprime les apostrophes et accents



# Gestion des typos

Pour gérer les erreurs de typo, les technique des ***matching flou (Fuzzy matching)*** peuvent être utilisées.

Ces techniques s'appuient sur la distance de Levenshtein, qui mesure le nombre d'édérations d'un simple caractère nécessaires pour passer d'un mot à un autre

- La plupart des erreurs de typo ou de mauvaise orthographe ont une distance de 1.

=> Ainsi, 80 % des erreurs pourraient être corrigés avec l'édition d'un seul caractère

Il est possible lors d'une requête de positionner le paramètre ***fuzziness*** qui donne donc la tolérance en distance de Levenshtein.

Ce paramètre peut également être positionné à la valeur AUTO :

- 0 pour les chaîne de 1 ou 2 caractères
- 1 pour les chaîne de 3, 4 ou 5 caractères
- 2 pour les chaîne de plus de 5 caractères

**Attention** : Pas de stemmer avec le mode fuzzy !



# Requête fuzzy

---

La recherche fuzzy est équivalent à une recherche par terme

```
GET /my_index/my_type/_search
{
  "query": {
    "fuzzy": {"text": "surprize" }
  }
}
```

2 paramètres peuvent être utilisées pour limiter l'impact de performance :

- ***prefix\_length*** : Le nombre de caractères initiaux qui ne seront pas modifiés.
- ***max\_expansions*** : Limiter les options que la recherche floue génère. La recherche fuzzy arrête de rassembler les termes proches quand elle atteint cette limite



# Fuzzy match

---

La paramètre **fuzziness** modifie la requête match en requête fuzzy

```
GET /my_index/my_type/_search
{
  "query": {
    "multi_match": {
      "fields": [ "text", "title" ],
      "query": "SURPRISE ME!",
      "fuzziness": "AUTO"
    }
  }
}
```





# Recherches avancées

---

Recherche avec préfixe  
Recherche phrase  
Analyse langage naturel  
**Surbrillance**  
Agrégations  
Géo-localisation



# Introduction

---

La surbrillance consiste à mettre en valeur le terme ayant matché dans le contenu original qui a été indexé.

Cela est possible grâce aux méta-données stockées lors de l'indexation

Le champ original doit être stocké par ELS



# Example

---

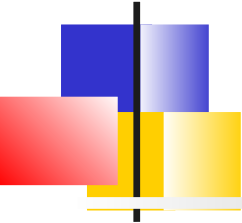
```
GET /_search
{
  "query" : {
    "bool" : {
      "must" : { "match" :
{ "attachment.content" : "Administration" } },
      "should" : { "match" : { "attachment.content" :
"Oracle" } }
    }
  },
  "highlight" : {
    "fields" : {
      "attachment.content" : {}
    }
  }
}
```



# Réponse

---

```
"highlight" : {  
    "attachment.content" : [  
        " formation Administration Oracle  
Forms/Report permet de voir tous les aspects nécessaires à",  
        " aux autres services Web de l'offre  
Oracle Fusion Middleware. Cependant, l'administration  
de ces",  
        " Surveillance. Elle peut-être un complément de la  
formation « Administration d'un serveur Weblogic",  
        "-forme Oracle Forms ou des personnes  
désirant migrer leur application client serveur Forms vers",  
        " Oracle Forms 11g/12c\nPré-requis : \  
nExpérience de l'administration système\nLa formation  
« Administation"  
    ]  
}
```



# Types de Highlighter

---

ELS utilise les « highlighters » de Lucene :

- **Plain** : Valeur par défaut qui donne les meilleurs résultats mais peut quelquefois causer des soucis de performance
- **Posting** : Utilisé si la propriété *index\_options* contient *offsets* dans le mapping. Plus rapide, travaille plutôt sur la phrase.
- **Fast-vector** : Utilisé si la propriété *term\_vector* est égale à *with\_positions\_offset* dans le mapping. Plus rapide, surtout pour les champs volumineux (>1MB), peut être affiné par la configuration, ...



# Forcer un highlighter

---

```
GET /_search
{
  "query" : {
    "bool" : {
      "must" : { "match" :
{ "attachment.content" : "Administration" } },
      "should" : { "match" : { "attachment.content" :
"Oracle" } }
    }
  },
  "highlight" : {
    "fields" : {
      "attachment.content" : { "type" : "plain" }
    }
  }
}
```



# Spécifier les balises

---

```
GET /_search
{
  "query" : {
    "bool" : {
      "must" : { "match" :
{ "attachment.content" : "Administration" } },
      "should" : { "match" : { "attachment.content" : "Oracle" } }
    }
  },
  "highlight" : {
    "pre_tags" : ["<tag1>"],
    "post_tags" : ["</tag1>"],
    "fields" : {
      "attachment.content" : { "type" : "plain" }
    }
  }
}
```



# Fragments mis en valeur

---

Il est possible de contrôler les fragments mis en valeur pour chaque champ :

- ***fragment\_size*** : donne la taille max du fragment mis en surbrillance
- ***number\_of\_fragments*** : Le nombre maximal de fragments dans ce champ





# Recherches avancées

---

Recherche avec préfixe

Recherche phrase

Recherche floue

Surbrillance

**Agrégations**

Géo-localisation



# Introduction

---

Les agrégations sont extrêmement puissantes pour le reporting et les tableaux de bord

Des énormes volumes de données peuvent être visualisées en temps-réel

- Le reporting change au fur et à mesure que les données changent

L'utilisation de la stack Elastic, Logstash et Kibana démontre bien tout ce que l'on peut faire avec les agrégations.

# Example (Kibana)





# Syntaxe DSL

---

Une agrégation peut être vue comme une unité de travail qui construit des informations analytiques sur un ensemble de documents.

En fonction de son positionnement dans l'arbre DSL, il s'applique sur l'ensemble des résultats de la recherche ou sur des sous-ensembles

Dans la syntaxe DSL, un bloc d'agrégation utilise le mot-clé **aggs**

**// Le max du champ *price* dans tous les documents**

```
POST /sales/_search?size=0
```

```
{
  "aggs" : {
    "max_price" : { "max" : { "field" : "price" } }
  }
}
```

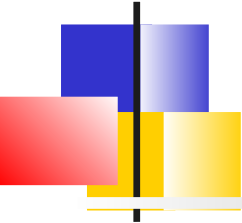


# Types d'agrégations

---

Plusieurs concepts sont relatifs aux agrégations :

- **Groupe ou Buckets** : Ensemble de document qui ont un champ à la même valeur ou partagent un même critère. Les groupes peuvent être imbriqués. ELS propose des syntaxes pour définir les groupes et compter le nombre de documents dans chaque catégorie
- **Métriques** : Calculs de métriques sur un groupe de documents (min, max, avg, ..)
- **Matrice** : (expérimentale) Opérations de classification selon différents champs, produisant une matrice des différentes possibilités. Le scripting n'est pas supporté pour ce type d'agrégation
- **Pipeline** : Une agrégation s'effectuant sur le résultat d'une agrégation



# Exemple Bucket

---

GET /cars/transactions/\_search

```
{  
  "aggs" : {  
    "colors" : {  
      "terms" : { "field" : "color.keyword" }  
    } } }  
}
```



# Réponse

---

```
{  
  ...  
  "hits": { "hits": [] },  
  "aggregations": {  
    "colors": {  
      "doc_count_error_upper_bound": 0, // incertitude  
      "sum_other_doc_count": 0,  
      "buckets": [  
        { "key": "red", "doc_count": 4 },  
        { "key": "blue", "doc_count": 2 },  
        { "key": "green", "doc_count": 2 }  
      ]  
    }  
  }  
}
```



# Exemple métrique

---

GET /cars/transactions/\_search

```
{  
  "size": 0,  
  "aggs" : {  
    "avg_price" : {  
      "avg" : {"field" : "price"}  
    }  
  }  
}
```





# Réponse

---

```
"hits": {  
  "total": 8,  
  "max_score": 0,  
  "hits": []  
},  
"aggregations": {  
  "avg_price": {  
    "value": 26500  
  }  
}
```



# Juxtaposition Bucket/Métriques

---

```
GET /cars/transactions/_search
{
  "size": 0,
  "aggs" : {
    "colors" : {
      "terms" : { "field" : "color.keyword" }
    },
    "avg_price" : {
      "avg" : {"field" : "price"}
    }
  }
}
```



# Réponse

---

```
{  
  ...  
  "hits": { "hits": [] },  
  "aggregations": {  
    "avg_price": {  
      "value": 26500  
    },  
    "colors": {  
      "doc_count_error_upper_bound": 0,  
      "sum_other_doc_count": 0,  
      "buckets": [  
        { "key": "red", "doc_count": 4 },  
        { "key": "blue", "doc_count": 2 },  
        { "key": "green", "doc_count": 2 }  
      ]  
    }  
  }  
}
```



# Imbrication

---

```
GET /cars/transactions/_search {  
  "aggs": {  
    "colors": {  
      "terms": { "field": "color" },  
      "aggs": {  
        "avg_price": {  
          "avg": { "field": "price" }  
        }, "make": {  
          "terms": { "field": "make" }  
        }  
      }  
    }  
  }  
}
```

# Réponse



---

```
{
  ...
  "aggregations": {
    "colors": {
      "buckets": [
        { "key": "red",
          "doc_count": 4,
          "avg_price": {
            "value": 32500
          },
          "make": { "buckets": [
            { "key": "honda", "doc_count": 3 },
            { "key": "bmw", "doc_count": 1 }
          ]
        }
      ],
    },
    ...
  }
}
```



# Agrégation et recherche

---

En général, une agrégation est combinée avec une recherche. Les buckets sont alors déduits des seuls documents qui matchent.

```
GET /cars/transactions/_search
```

```
{  
  "query" : {  
    "match" : { "make" : "ford" }  
  }, "aggs" : {  
    "colors" : {  
      "terms" : { "field" : "color" }  
    }  
  }  
}
```



# Spécification du tri

---

GET /cars/transactions/\_search

```
{
  "aggs" : {
    "colors" : {
      "terms" : { "field" : "color",
                  "order": { "avg_price" : "asc" }
                }, "aggs": {
                  "avg_price": {
                    "avg": {"field": "price"}
                  }
                }
      }
  }
}
```



# Types de bucket

---

Différents types de regroupement sont proposés par ELS

- Par terme, par filtre : Nécessite une tokenization du champ
- Par intervalle de valeurs
- Par intervalle de dates, par histogramme
- Par intervalle d'IP
- Par absence d'un champ
- Par le document parent
- Significant terms
- Par géo-localisation
- ...





# Intervalle de valeur

---

GET /cars/transactions/\_search

```
{
  "aggs": {
    "price": {
      "histogram": {
        "field": "price",
        "interval": 20000
      },
      "aggs": {
        "revenue": {
          "sum": { "field" : "price" }
        }
      }
    }
  }
}
```



# Histogramme de date

---

```
GET /cars/transactions/_search
```

```
{
```

```
  "aggs": {
```

```
    "sales": {
```

```
      "date_histogram": {
```

```
        "field": "sold",
```

```
        "interval": "month",
```

```
        "format": "yyyy-MM-dd"
```

```
      }
```

```
    } } }
```



## *significant\_terms*

---

L'agrégation ***significant\_terms*** est plus subtile mais peut donner des résultats intéressants (proche du machine-learning).

Cela consiste à analyser les données retournées et trouver les termes qui apparaissent à une fréquence *anormalement* supérieure

Anormalement signifie : par rapport à la fréquence pour l'ensemble des documents

=> Ces anomalies statistiques révèlent en général des choses intéressantes

# Fonctionnement



---

*significant\_terms* part d'un résultats d'une recherche et effectue une autre recherche agrégé

Il part ensuite de l'ensemble des documents et effectue la même recherche agrégé

Il compare ensuite les résultats de la première recherche qui sont anormalement pertinent par rapport à la recherche globale

Avec ce type de fonctionnement, on peut :

- Les personnes qui ont aimé ... ont également aimé ...
- Les clients qui ont eu des transactions CB douteuses sont tous allés chez tel commerçant
- Tous les jeudi soirs, la page untelle est beaucoup plus consultée
- ...



# Example

---

```
{  
  "query" : {  
    "terms" : { "force" : [ "British Transport Police" ] }  
  },  
  "aggregations" : {  
    "significantCrimeTypes" : {  
      "significant_terms" : { "field" : "crime_type" }  
    }  
  }  
}
```



# Réponse

---

```
"aggregations" : {  
  "significantCrimeTypes" : {  
    "doc_count": 47347, // Total résultat de requête  
    "buckets" : [  
      {  
        "key": "Bicycle theft",  
        "doc_count": 3640, // Nbr docs le résultat de requête  
        "score": 0.371235374214817,  
        "bg_count": 66799 // Nbr pour le total de l'index  
      }  
      ...  
    ]  
  }  
}
```

=> Le taux de vols de vélos est anormalement élevé pour  
« British Transport Police »



# Types de métriques

---

ELS propose de nombreux métriques :

- ***avg, min, max, sum***
- ***value\_count, cardinality*** :  
Comptage de valeur distinctes
- ***top\_hit*** : Les meilleurs documents
- ***extended\_stats*** : Fournit plein de métriques (count, sum, variance, ...)
- ***percentiles*** : percentiles



# Pipeline

---

Les agrégations pipelines travaillent sur le résultat d'une agrégation plutôt que sur les documents résultats.

Via leur paramètre ***bucket\_path***, ils indiquent une agrégation parente ou du même niveau





# Exemple

---

POST /\_search

```
{
  "aggs": {
    "my_date_histo": {
      "date_histogram": {
        "field": "timestamp",
        "calendar_interval": "day"
      },
      "aggs": {
        "the_sum": {
          "sum": { "field": "lemmings" }
        },
        "the_movavg": {
          "moving_avg": { "buckets_path": "the_sum" }
        }
      }
    }
  }
}
```



# Recherches avancées

---

Recherche avec préfixe

Recherche phrase

Recherche floue

Surbrillance

Agrégations

**Géo-localisation**



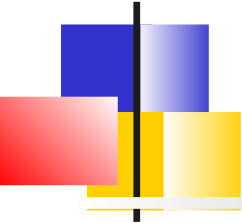
# Introduction

---

ELS permet de combiner la géo-localisation avec les recherches full-text, structurées et les agrégations

ELS a 2 modèles pour représenter des données de géolocalisation

- Le type ***geo\_point*** qui représente un couple latitude-longitude. Cela permet principalement le calcul de distance
- Le type ***geo\_shape*** qui définit une zone via le format GeoJSON. Cela permet de savoir si 2 zones ont une intersection



# Geo-point

Les Geo-points ne peuvent pas être détectés automatiquement par le *dynamic mapping*. Ils doivent être explicitement spécifiés dans le mapping:

```
PUT /attractions
```

```
{ "mappings": { "restaurant": {  
  "properties": {  
    "name": { "type": "string" },  
    "location": { "type": "geo_point" }  
  }  
} } }
```

```
----
```

```
PUT /attractions/restaurant/1
```

```
{"name": "Chipotle Mexican Grill", "location": "40.715, -74.011" }
```

```
PUT /attractions/restaurant/2
```

```
{ "name": "Pala Pizza", "location": { "lat":40.722,"lon": -73.989 } }
```

```
PUT /attractions/restaurant/3
```

```
{ "name": "Mini Munchies Pizza","location": [ -73.983, 40.719 ] }
```



# Filtres

---

4 filtres peuvent être utilisés pour inclure ou exclure des documents vis à vis de leur *geo-point* :

- ***geo\_bounding\_box*** : Les geo-points inclus dans le rectangle fourni
- ***geo\_distance*** : Distance d'un point central inférieur à une limite. Le tri et le score peuvent être relatif à la distance
- ***geo\_distance\_range*** : Distance dans un intervalle
- ***geo\_polygon*** : Les geo-points incluent dans un polygone



# Example

---

GET /attractions/restaurant/\_search

```
{  
  "query": {  
    "bool": {  
      "filter": {  
        "geo_bounding_box": {  
          "location": { "top_left": { "lat": 40.8, "lon": -74.0 },  
                        "bottom_right": { "lat": 40.7, "lon": -73.0 }  
        }  
      }  
    }  
  }  
}
```



# Agrégation

---

3 types d'agrégation sur les geo-points sont possibles

- ***geo\_distance*** (*bucket*): Groupe les documents dans des ronds concentriques autour d'un point central
- ***geohash\_grid*** (*bucket*): Groupe les documents par cellules (geohash\_cell, les carrés de google maps) pour affichage sur une map
- ***geo\_bounds*** (*metrics*): retourne les coordonnées d'une zone rectangle qui engloberait tous les geo-points. Utile pour choisir le bon niveau de zoom



# Example

---

```
GET /attractions/restaurant/_search
{
  "query": { "bool": { "must": {
    "match": { "name": "pizza" }
  },
  "filter": { "geo_bounding_box": {
    "location": { "top_left": { "lat": 40,8, "lon": -74.1 },
                  "bottom_right": { "lat": 40.4, "lon": -73.7 }
    }
  } } } },
  "aggs": {
    "per_ring": {
      "geo_distance": {
        "field": "location",
        "unit": "km",
        "origin": {
          "lat": 40.712,
          "lon": -73.988
        },
        "ranges": [
          { "from": 0, "to": 1 },
          { "from": 1, "to": 2 }
        ]
      }
    }
  }
}
```





# Geo-shape

---

Comme les champs de type *geo\_point* , les ***geo-shape*** doivent être mappés explicitement :

```
PUT /attractions
```

```
{
  "mappings": { "landmark": {
    "properties": {
      "name": { "type": "string" },
      "location": { "type": "geo_shape" }
    } } } }
```

```
---
```

```
PUT /attractions/landmark/dam_square
```

```
{
  "name" : "Dam Square, Amsterdam",
  "location" : {
    "type" : "polygon",
    "coordinates" : [[ [ 4.89218, 52.37356 ], [ 4.89205, 52.37276 ], [ 4.89301, 52.37274 ],
[ 4.89392, 52.37250 ], [ 4.89218, 52.37356 ] ]
  } }
```



# Exemple

---

```
GET /attractions/landmark/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "shape": {
          "type": "circle",
          "radius": "1km"
          "coordinates": [ 4.89994, 52.37815]
        }
      }
    }
  }
}
```



# Exploitation

---

Index et performance



# *node-stats* API

---

GET `_nodes/stats`

```
{
  "cluster_name": "elasticsearch_zach",
  "nodes": {
    "UNr6ZMf5Qk-YCPA_L18B0Q": {
      "timestamp": 1408474151742,
      "name": "Zach",
      "transport_address":
        "inet[zacharys-air/192.168.1.131:9300]",
      "host": "zacharys-air",
      "ip": [
        "inet[zacharys-air/192.168.1.131:9300]",
        "NONE"
      ],

```



# Sections indices

La section **indices** liste des statistiques agrégés pour tous les les index d'un nœud.

Il contient les sous-sections suivantes :

- **docs** : combien de documents résident sur le nœud, le nombre de documents supprimés qui n'ont pas encore été purgés
- **store** indique l'espace de stockage utilisé par le nœud
- **indexing** le nombre de documents indexés
- **get** : Statistiques des requêtes *get-by-ID*
- **search** : le nombre de recherches actives, nombre total de requêtes le temps d'exécution cumulé des requêtes
- **merges** fusion de segments de Lucene
- **filter\_cache** : la mémoire occupée par le cache des filtres
- **id\_cache** répartition de l'usage mémoire
- **field\_data** mémoire utilisée pour les données temporaires de calcul (utilisé lors d'agrégation, le tri, ...)
- **segments** le nombre de segments Lucene. (chiffre normal 50-150)



# Index stats API

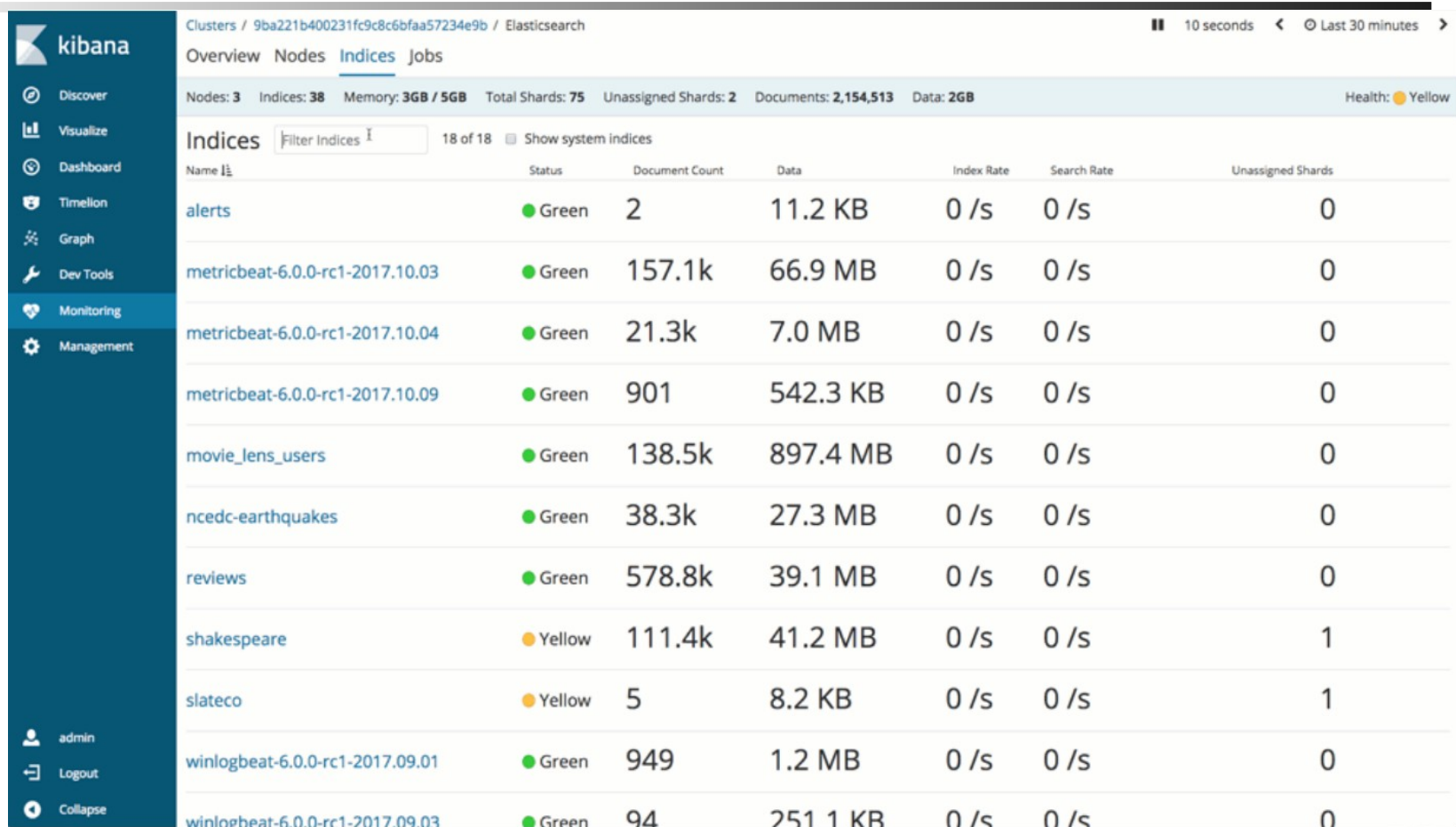
---

L'index stats API permet de visualiser des statistiques vis à vis d'un index

GET my\_index/\_stats

Le résultat est similaire à la sortie de *node-stats* : Compte de recherche, de get, segments, ...

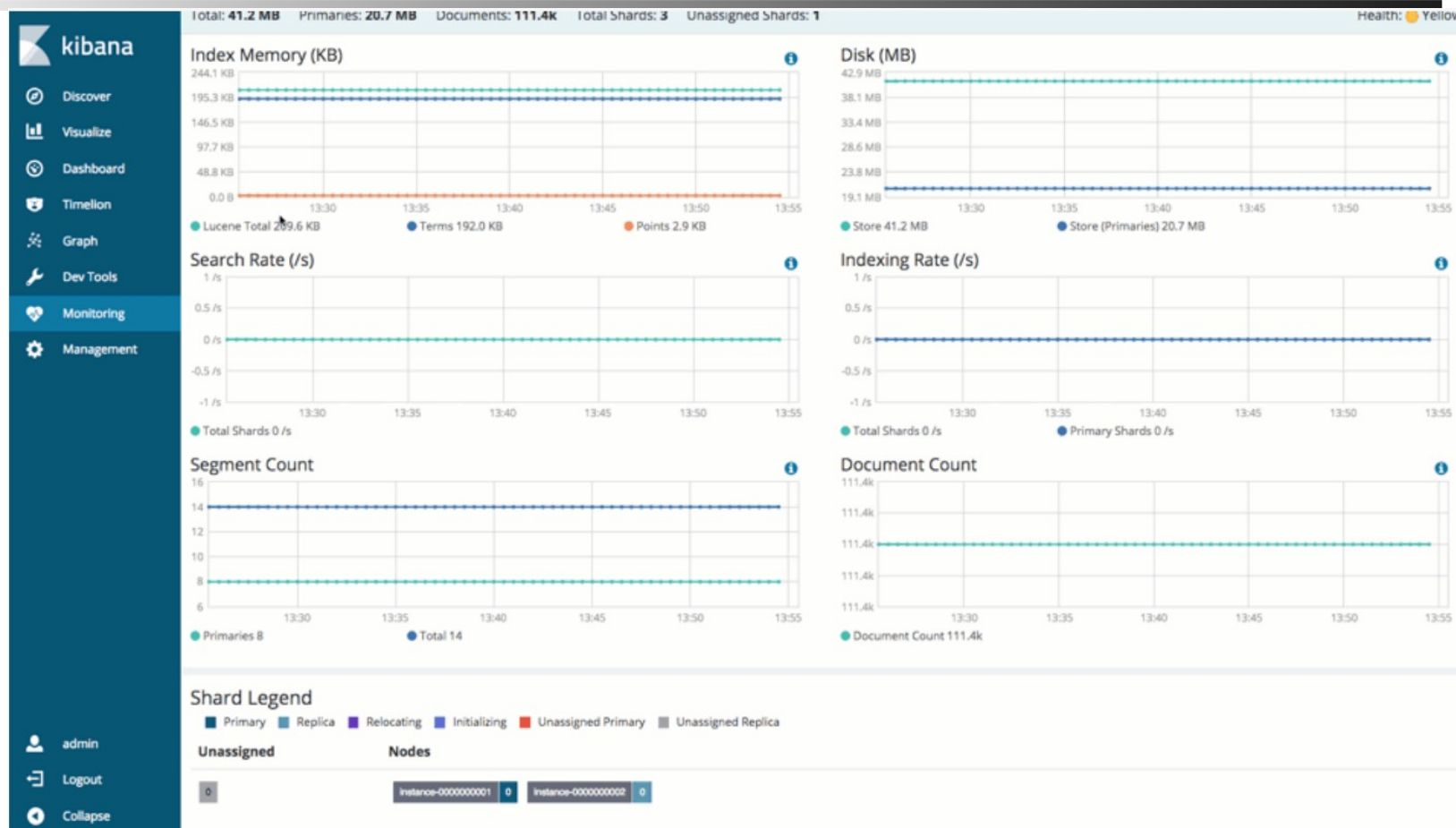
# Exemple XPack1



The screenshot shows the Kibana Monitoring dashboard for an Elasticsearch cluster. The left sidebar contains navigation links for Discover, Visualize, Dashboard, Timelion, Graph, Dev Tools, Monitoring (selected), and Management. The main content area displays the 'Indices' tab for a cluster with ID 9ba221b400231fc9c8c6faa57234e9b. A summary bar at the top of the indices section shows: Nodes: 3, Indices: 38, Memory: 3GB / 5GB, Total Shards: 75, Unassigned Shards: 2, Documents: 2,154,513, Data: 2GB, and Health: Yellow. Below this is a table of indices with columns for Name, Status, Document Count, Data, Index Rate, Search Rate, and Unassigned Shards. The table lists 18 indices, with the first 10 visible. The 'shakespeare' and 'slateco' indices are highlighted in yellow, indicating a 'Yellow' health status.

| Name                            | Status | Document Count | Data     | Index Rate | Search Rate | Unassigned Shards |
|---------------------------------|--------|----------------|----------|------------|-------------|-------------------|
| alerts                          | Green  | 2              | 11.2 KB  | 0 /s       | 0 /s        | 0                 |
| metricbeat-6.0.0-rc1-2017.10.03 | Green  | 157.1k         | 66.9 MB  | 0 /s       | 0 /s        | 0                 |
| metricbeat-6.0.0-rc1-2017.10.04 | Green  | 21.3k          | 7.0 MB   | 0 /s       | 0 /s        | 0                 |
| metricbeat-6.0.0-rc1-2017.10.09 | Green  | 901            | 542.3 KB | 0 /s       | 0 /s        | 0                 |
| movie_lens_users                | Green  | 138.5k         | 897.4 MB | 0 /s       | 0 /s        | 0                 |
| ncedc-earthquakes               | Green  | 38.3k          | 27.3 MB  | 0 /s       | 0 /s        | 0                 |
| reviews                         | Green  | 578.8k         | 39.1 MB  | 0 /s       | 0 /s        | 0                 |
| shakespeare                     | Yellow | 111.4k         | 41.2 MB  | 0 /s       | 0 /s        | 1                 |
| slateco                         | Yellow | 5              | 8.2 KB   | 0 /s       | 0 /s        | 1                 |
| winlogbeat-6.0.0-rc1-2017.09.01 | Green  | 949            | 1.2 MB   | 0 /s       | 0 /s        | 0                 |
| winlogbeat-6.0.0-rc1-2017.09.03 | Green  | 94             | 251.1 KB | 0 /s       | 0 /s        | 0                 |

# Monitoring Shards







# Segments Lucene

Chaque shard d'Elasticsearch est un index Lucene.

Un index Lucene est divisé en de petits fichiers : les **segments**

| Elasticsearch Index |         |                     |         |                     |         |                     |         |
|---------------------|---------|---------------------|---------|---------------------|---------|---------------------|---------|
| Elasticsearch shard |         | Elasticsearch shard |         | Elasticsearch shard |         | Elasticsearch shard |         |
| Lucene index        |         | Lucene index        |         | Lucene index        |         | Lucene index        |         |
| Segment             | Segment | Segment             | Segment | Segment             | Segment | Segment             | Segment |



# Fusion de segments

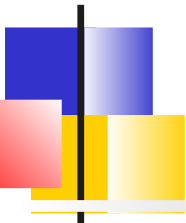
---

Lucene crée des segments lors de l'indexation. Les segments sont immuables

Lors d'une recherche, les segments sont traités de façon séquentielle  
=> Plus il y a de segments, plus les performances de recherche diminuent

Pour optimiser la recherche, Lucene propose l'opération **merge** qui fusionne de petits segments en de plus gros.

- C'est une opération assez lourde qui peut impacter les opérations d'indexation et de recherche. Elle est effectuée périodiquement par un pool de threads dédié
- On peut forcer une opération de merge :  
`curl -XPOST 'localhost:9200/logstash-2017.07*/_forcemerge?max_num_segments=1'`
- Pour que le merge réussisse, il faut que l'espace disque soit 2 fois la taille du shard



# Gestion du lifecycle des index

---

Il est possible d'automatiser des actions de gestion du cycle de vie des index comme :

- **Rollover** : Redirigez un alias pour commencer à écrire dans un nouvel index lorsque l'index existant atteint un certain âge, nombre de documents ou taille.
- **Shrink** : Réduire le nombre de shards primaires dans un index.
- **Force merge** : Réduire le nombre de segments dans chaque shards d'un index et libérer l'espace utilisé par les documents supprimés.
- **Freeze** : Rendre un index read-only et minimiser son empreinte mémoire
- **Delete** : Supprimer un index



# Mécanisme

---

En pratique, il faut associer une **stratégie de cycle de vie avec un gabarit d'index**, afin qu'il soit appliqué aux index nouvellement créés.

La stratégie de cycle de vie définit les conditions pour faire un index dans ses différentes de son cycle de vie :

- **Hot** : L'index est mis à jour et interrogé
- **Warm** : L'index n'est plus mis à jour mais encore interrogé.
- **Cold** : L'index n'est plus mis à jour et est rarement interrogé. Les informations sont toujours consultables, mais les requêtes sont plus lentes
- **Delete** : L'index n'est plus utilisé et peut être supprimé.



# Stratégies

---

Une stratégie de cycle de vie consiste à spécifier :

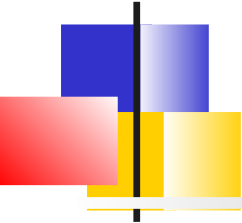
- Taille ou âge maximum pour effectuer le rollover.
- Le moment où l'index n'est plus mis à jour et où le nombre de shards primaires peut être réduit.
- Le moment pour forcer la fusion
- Le moment où on peut déplacer l'index vers un matériel moins performant.
- Le moment où la disponibilité n'est pas aussi critique et le nombre de répliques peut être réduit.
- Le moment où l'index peut être supprimé en toute sécurité.



# Création

---

```
PUT _ilm/policy/datastream_policy
{
  "policy": {
    "phases": {
      "hot": {
        "actions": {
          "rollover": {
            "max_size": "50GB",
            "max_age": "30d"
          }
        }
      },
      "delete": {
        "min_age": "90d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```



# Création de gabarit puis du premier index

---

```
PUT _template/datastream_template
{
  "index_patterns": ["datastream-*"],
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1,
    "index.lifecycle.name": "datastream_policy",
    "index.lifecycle.rollover_alias": "datastream"
  }
}
---
PUT datastream-000001
{
  "aliases": {
    "datastream": {
      "is_write_index": true
    }
  }
}
---
GET datastream-*/_ilm/explain
```



# Exploitation





# Changement de configuration

---

La plupart des configurations ELS sont dynamiques, elles peuvent être modifiées par l'API.

L'API *cluster-update* opère selon 2 modes :

- ***transient*** : Les changements sont annulés au redémarrage
- ***persistent*** : Les changements sont permanents. Au redémarrage, ils écrasent les valeurs des fichiers de configuration



# Example

---

```
PUT /_cluster/settings
{
  "persistent" : {
    "discovery.zen.minimum_master_nodes" : 2
  },
  "transient" : {
    "indices.store.throttle.max_bytes_per_sec" : "50mb"
  }
}
```



# Fichiers de trace

---

ELS écrit de nombreuses traces dans ES\_HOME/logs . Le niveau de trace par défaut est INFO

On peut le changer par l'API

```
PUT /_cluster/settings
```

```
{
```

```
"transient" : { "logger.discovery" : "DEBUG" }
```

```
}
```

OU

```
PUT /_cluster/settings {
```

```
"transient":{"logger._root":"DEBUG"}
```

```
}
```

Ne pas oublier la config. Initiale de *log4j42.properties*



# Quelques packages ELS

---

**org.elasticsearch.env** : Relatif à l'emplacement des données des nœuds (data)

**org.elasticsearch.indices.recovery** et  
**org.elasticsearch.index.gateway** : Recouvrement de shards

**org.elasticsearch.cluster.action.shard** : Statut des shards

**org.elasticsearch.snapshots** : Pour les snapshots&restore

**org.elasticsearch.http** : Connexion réseaux

**org.elasticsearch.marvel.agent.exporter** : Log du monitoring



# Slowlog

L'objectif du **slowlog** est de logger les requêtes et les demandes d'indexation qui dépassent un certain seuil de temps

Par défaut ce fichier journal n'est pas activé. Il peut être activé en précisant l'action (query, fetch, ou index), le niveau de trace ( WARN , DEBUG, ..) et le seuil de temps

C'est une configuration au niveau index

```
PUT /my_index/_settings
```

```
{ "index.search.slowlog.threshold.query.warn" : "10s",  
  "index.search.slowlog.threshold.fetch.debug": "500ms",  
  "index.indexing.slowlog.threshold.index.info": "5s" }
```

```
PUT /_cluster/settings
```

```
{ "transient" : {  
  "logger.index.search.slowlog" : "DEBUG",  
  "logger.index.indexing.slowlog" : "WARN"  
} }
```



# Annexes

---

Routing  
Distribution de la recherche  
Types complexes  
Mapping relation parent/enfant  
Requête avec jointure  
Client Java  
Pipeline d'agrégations



# Routing



# Résolution du shard primaire

---

Lors de l'indexation, le document est d'abord stocké sur le shard primaire.

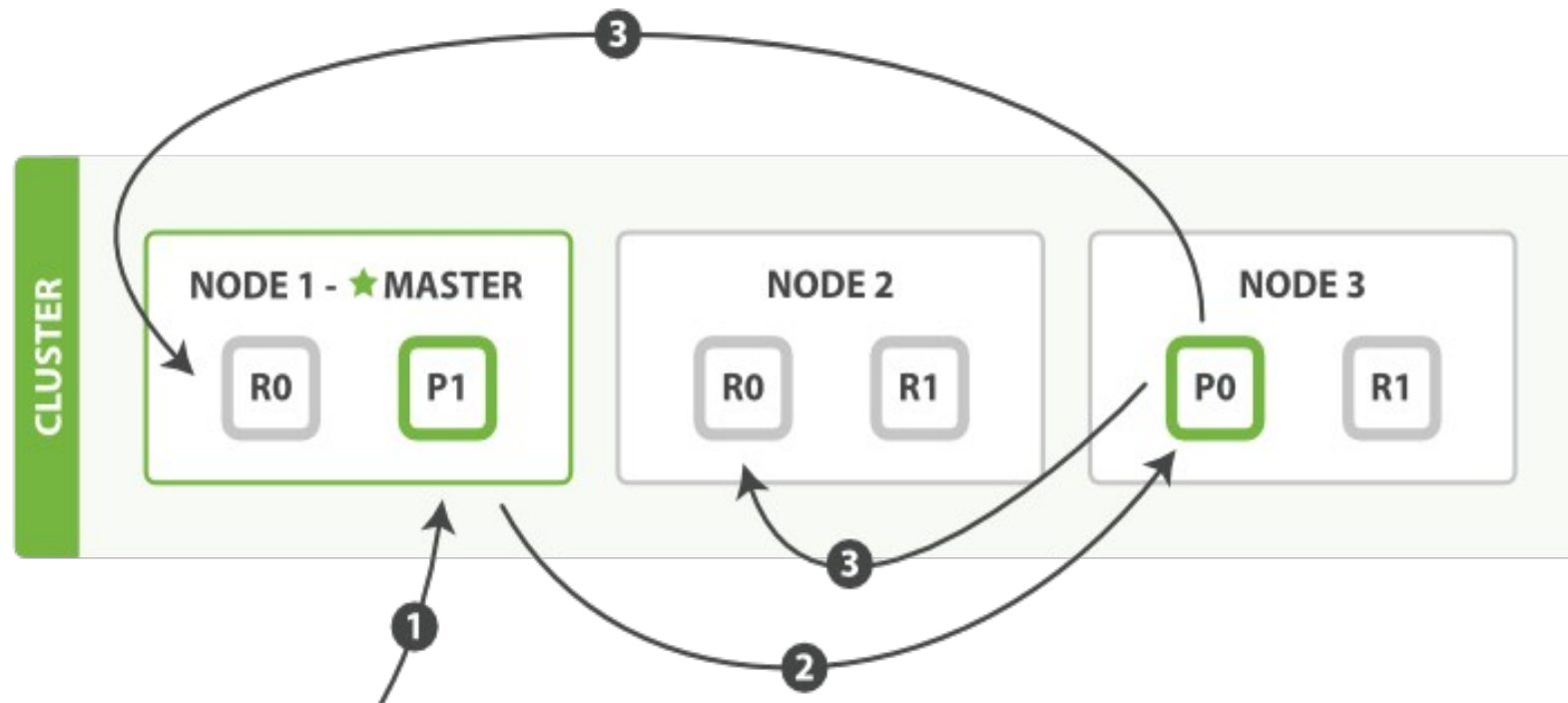
La résolution du n° de shard s'effectue grâce à la formule :

```
shard = hash(routing) % number_of_primary_shards
```

La valeur du paramètre ***routing*** est une chaîne arbitraire qui par défaut correspond à l'*id* du document mais peut être explicitement spécifiée



# Séquence d'une mise à jour sur une architecture cluster



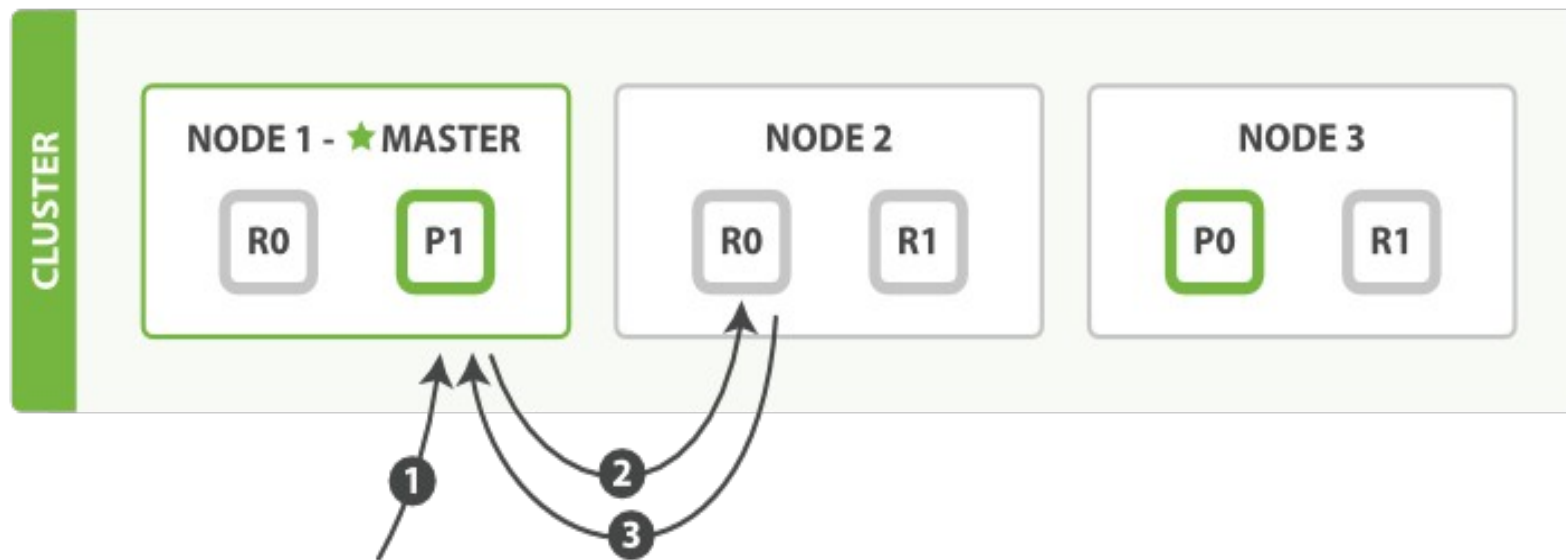


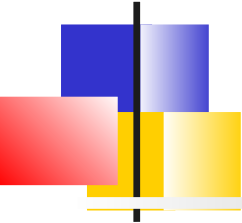
# Séquence d'une mise à jour sur une architecture cluster (2)

---

1. Le client envoie une requête d'indexation ou suppression vers le nœud 1
2. Le nœud utilise l'id du document pour déduire que le document appartient au shard 0 . Il transfère la requête vers le nœud 3 , ou la copie primaire du shard 0 réside.
3. Le nœud 3 exécute la requête sur le shard primaire. Si elle réussit, il transfère la requête en parallèle aux répliques résidant sur le nœud 1 et le nœud 2 . Une fois que les ordres de mises à jour des répliques aboutissent, le nœud 3 répond au nœud 1 qui répond au client

# Séquence pour la récupération d'un document





# Séquence pour la récupération d'un document

---

1. Le client envoie une requête au nœud 1.
2. Le nœud utilise l'*id* du document pour déterminer que le document appartient au shard 0. Des copies de shard 0 existent sur les 3 nœuds. Pour cette fois-ci, il transfère la requête au nœud 2.
3. Le nœud 2 retourne le document au nœud 1 qui le retourne au client.

Pour les prochaines demandes de lecture, le nœud choisira un shard différent pour répartir la charge (algorithme Round-robin)



## Distribution de la recherche



# Introduction

---

La recherche nécessite un modèle d'exécution complexe les documents correspondant à la recherche sont dispersés sur les shards

Une recherche doit consulter une copie de chaque shard de l'index ou des index sollicités

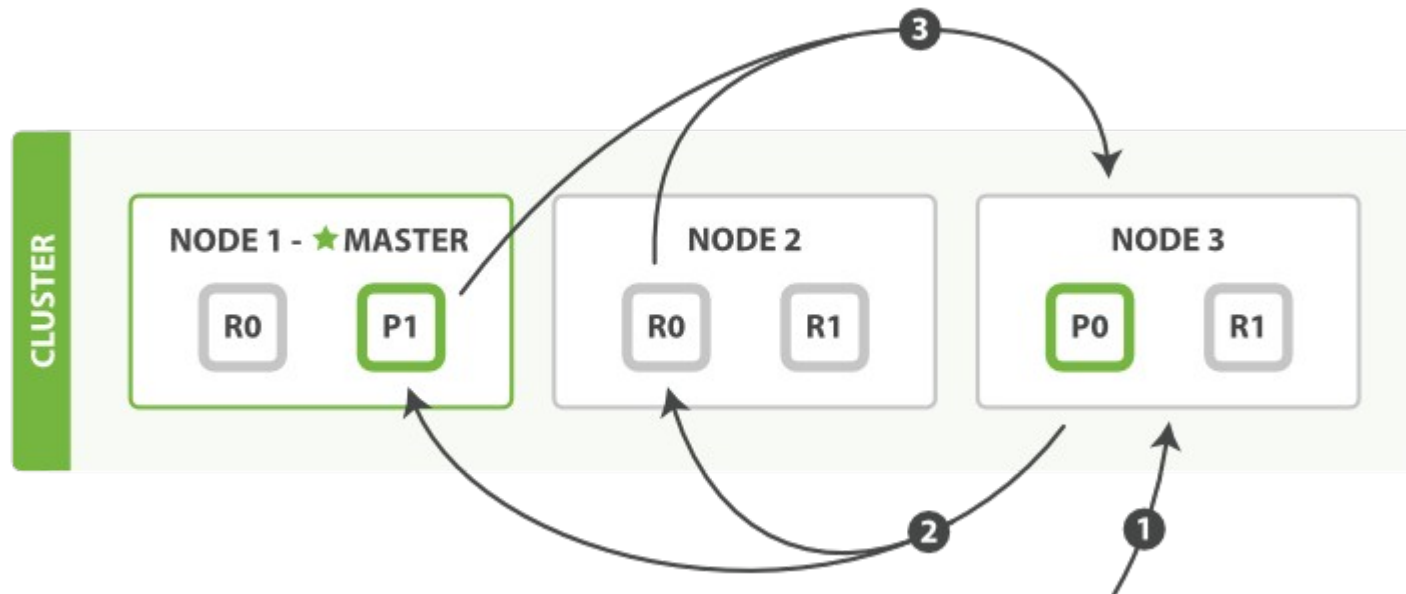
Une fois trouvés ; les résultats des différents shards doivent être combinés en une liste unique afin que l'API puisse retourner une page de résultats

La recherche est donc exécutée en 2 phases :

- ***query***
- ***fetch.***

# Phase de requête

Durant la 1ère phase, la requête est diffusée à une copie (primaire ou réplique) de tous les shards. Chaque shard exécute la recherche et construit une file à priorité des documents qui matchent





# Phase de requête

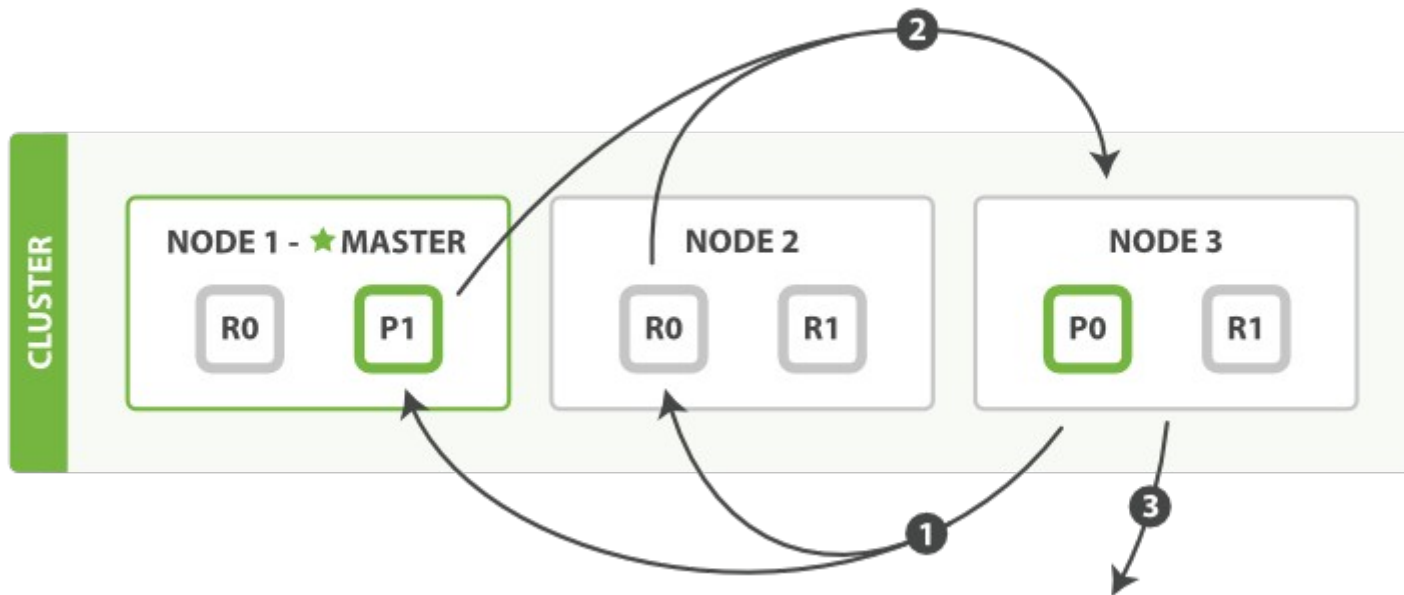
---

1. Le client envoie une requête de recherche au nœud 3 qui crée une file à priorité vide de taille *from + size* .
2. Le nœud 3 transfère la requête à une copie de chaque shard de l'index. Chaque shard exécute la requête localement et ajoute le résultat dans une file à priorité locale de taille *from + size* .
3. Chaque shard retourne les IDs et les valeurs de tri de tous les documents de la file au nœud 3 qui fusionne ces valeurs dans sa file de priorité



# Phase fetch

La phase de fetch consiste à récupérer les documents présents dans la file à priorité.





# Phase fetch

---

1. Le nœud coordinateur identifie quels documents doivent être récupérés et produit une requête multiple GET aux shards.
2. Chaque shard charge les documents, les enrichi si nécessaire (surbrillance par exemple) et les retourne au nœud coordinateur
3. Lorsque tous les documents ont été récupérés, le coordinateur retourne les résultats au client.

Types complexes



# Types complexes

---

En plus des types simples, ELS supporte

- Les **tableaux** : Il n'y a pas de mapping spécial pour les tableaux. Chaque champ peut contenir 0, 1 ou n valeurs  

```
{ "tag": [ "search", "nosql" ] }
```

  - Les valeurs d'un tableau doivent être de même type
  - Un champ à *null* est traité comme un tableau vide
- Les **objets** : Il s'agit d'une structure de données embarquées dans un champ
- Les **nested** : Il s'agit d'un tableau de structures de données embarquées dans un champ. Chaque élément du tableau est stocké séparément



# Mapping des objets embarqués

ELS détecte dynamiquement les champs objets et les mappe comme objet. Chaque champ embarqué est listé sous **properties**

```
{ "gb": {  
  "tweet": {  
    "properties": {  
      "tweet": { "type": "string" },  
      "user": {  
        "type": "object",  
        "properties": {  
          "id": { "type": "string" },  
          "age": { "type": "long"},  
          "name": {  
            "type": "object",  
            "properties": {  
              "first": { "type": "string" },  
              "last": { "type": "string" }  
            }  
          }  
        }  
      }  
    }  
  }  
}
```



# Indexation des objets embarqués

---

Un document Lucene consiste d'une liste à plat de paires clé/valeurs. Les objets embarqués sont alors convertis comme suit :

```
{  
  "tweet": [elasticsearch, flexible, very],  
  "user.id": [@johnsmith],  
  "user.age": [26],  
  "user.name.first": [john],  
  "user.name.last": [smith]  
}
```



# Attention !

---

```
"followers": [  
  { "age": 35, "name": "Mary White"},  
  { "age": 26, "name": "Alex Jones"},  
  { "age": 19, "name": "Lisa Smith"}]
```

Après indexation, cela donne :

```
{  
  "followers.age": [19, 26, 35],  
  "followers.name": [alex, jones, lisa, smith, mary, white]  
}
```

=> Lien entre age et nom perdu !

*q=age:19 AND name:Mary* retourne des résultats

=> Solution les ***Nested objects***



# Type nested

---

Pour garder l'indépendance de chaque objet, il faut déclarer un type ***nested***

Cela a pour effet de créer des « sous-documents » indépendants et chaque sous-document peut être recherché indépendamment (Voir nested query)

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "followers": {
          "type": "nested"
        }
      }
    }
  }
}
```





# Champs méta et relation parent/enfant



# Méta-données

---

Chaque document a des méta-données associés :

Identification :

- ***\_index, \_type, \_id***
- ***\_uid*** : Champ composite constitué de *\_type* et *\_id*

Document original :

- ***\_source*** : Le JSON original.
- ***\_size*** : La taille en octets

Indexation

- ***\_all*** : Contient toutes les valeurs des autres champs
- ***\_field\_names*** : Tous les champs qui contiennent une valeur non-null.

Routing

- ***\_parent*** : Utilisé pour créer une relation parent-enfant entre 2 types de document
- ***\_routing*** : Une valeur personnalisée pour le routing

Autres

- ***\_meta*** : Méta-données applicatives



# Relation parent/enfant

## 6.x

---

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "properties": {
        "my_join_field": { // Nom du champ
          "type": "join",
          "relations": {
            "question": "answer" // Relation One To Many
          }
        }
      }
    }
  }
}
```



# Relation parent/enfant

## 6.x

---

**# Indexation document parent**

```
PUT my_index/_doc/1?refresh
{
  "text": "This is a question",
  "my_join_field": {
    "name": "question"
  }
}
```

**# Indexation document enfant**

```
PUT my_index/_doc/3?routing=1&refresh
{
  "text": "This is an answer",
  "my_join_field": {
    "name": "answer",
    "parent": "1"
  }
}
```



# Jointures



# Introduction

---

ELS permet 2 types de requêtes s'apparentant à des jointures SQL

- Requête sur objets imbriqués : Ce type de requête permet de récupérer des documents en fonction de critères sur ses objets imbriqués
- Requête parent/enfant : Les requêtes *has\_child* et *has\_parent* permettent de retrouver un document en fonction de critère sur son parent ou ses enfants



# Requête imbriquée

---

Les requêtes imbriquées utilisent les paramètres suivants :

- ***nested*** : Encapsule les informations concernant la requête imbriquée
- ***path*** : référence le chemin vers l'objet imbriqué
- ***query*** : inclut la requête exécutée sur les objets imbriqués. Les champs référencés dans la requête doivent utiliser le chemin complet
- ***score\_mode*** : spécifie comment le score des enfants influence le score du document parent retourné. Par défaut *avg*, mais peut être *sum*, *min*, *max* et *none*.



# Example

---

GET /\_search

```
{
  "query": {
    "nested" : {
      "path" : "obj1",
      "score_mode" : "avg",
      "query" : {
        "bool" : {
          "must" : [
            { "match" : {"obj1.name" : "blue"} },
            { "range" : {"obj1.count" : {"gt" : 5}} }
          ]
        }
      }
    }
  }
}
```





# Requête *has\_child*

---

Le filtre ***has\_child*** spécifie 2 valeurs :

- ***type*** : Le type de documents sur lesquels portent la recherche
- ***query*** : La requête ELS

Il retourne des documents parents dont les enfants vérifient la requête

Il peut également préciser d'autres paramètres :

- ***score\_mode*** : Comment le score des enfants sont agrégés dans le parent
- ***min\_children, max\_children*** : Limitation sur le nombre d'enfants



# Exemple

---

GET /\_search

```
{
  "query": {
    "has_child" : {
      "type" : "blog_tag",
      "score_mode" : "min",
      "min_children": 2,
      "max_children": 10,
      "query" : {
        "term" : {
          "tag" : "something"
        }
      }
    }
  }
}
```



# Exemple

---

GET /\_search

```
{
  "query": {
    "has_child" : {
      "type" : "blog_tag",
      "score_mode" : "min",
      "min_children": 2,
      "max_children": 10,
      "query" : {
        "term" : {
          "tag" : "something"
        }
      }
    }
  }
}
```



# Requête *has\_parent*

---

Le filtre ***has\_parent*** spécifie 2 valeurs :

- ***parent\_type*** : Le type des documents parents
- ***query*** : La requête ELS

Il retourne des documents enfants dont les parents vérifient la requête

Il peut également préciser d'autres paramètres :

- ***score*** : Le score du parent est il agrégé dans l'enfant



# Example

---

GET /\_search

```
{
  "query": {
    "has_parent" : {
      "parent_type" : "blog",
      "score" : true,
      "query" : {
        "term" : {
          "tag" : "something"
        }
      }
    }
  }
}
```



# Requête *parent\_id*

---

La requête ***parent\_id*** renvoie tout simplement les enfants d'un parent particulier.

Elle est plus efficace que *has\_parent*

Exemple :

```
GET /my_index/_search
{
  "query": {
    "parent_id" : {
      "type" : "blog_tag",
      "id" : "1"
    }
  }
}
```



## *inner\_hits*

---

Lors de requêtes parent/child ou nested, les causes du match dans les objets embarqués (nested, parent ou child) ne sont pas affichées par défaut.

Le bloc ***inner\_hits*** permet de donner plus d'explications sur les résultats retournés



# Structure

---

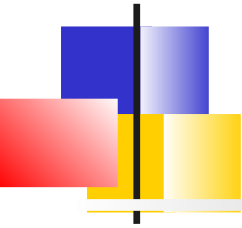
Requête:

```
"<query>" : {  
  "inner_hits" : {  
    <inner_hits_options>  
  }  
}
```

Réponse :

```
"hits": [  
  {  
    "_index": ...,  
    "_type": ...,  
    "_id": ...,  
    "inner_hits": {  
      "<inner_hits_name>": {  
        "hits": {  
          "total": ...,  
          "hits": [  
            {  
              "_type": ...,  
              "_id": ...,  
              ...  
            },  
            ...  
          ] } } },  
    ... }, ... ]
```





# Client Java



# 2 approches

---

2 API Java pour les clients REST sont fournies par elastic :

- API de bas niveau : Il faut sérialiser/désérialiser soi-même
- API de haut niveau : Méthodes spécifiques effectuant les sérialisation/désérialisation



# Apports API de bas-niveau

---

- Dépendances minimales
- Répartition de charge entre les nœuds du cluster
- Failover si défaillance de nœud
- Les nœuds ne répondant pas correctement sont pénalisés
- Connexions TCP persistantes
- Journalisation des requêtes et des réponses
- Découverte automatique des noeuds



# Dépendances

---

## Maven :

```
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>elasticsearch-rest-client</artifactId>
  <version>7.5.2</version>
</dependency>
```

## Gradle :

```
dependencies {
    compile 'org.elasticsearch.client:elasticsearch-rest-
client:7.5.2'
}
```



# Initialisation

---

Initialisation en fournissant une ou plusieurs adresses de nœud :

```
RestClient restClient = RestClient.builder(  
    new HttpHost("localhost", 9200, "http"),  
    new HttpHost("localhost", 9201, "http")).build();
```

Thread safe => généralement un singleton

*restClient.close()* quand il n'est plus requis



# Options avancées

---

Le builder propose d'autres options à l'instanciation :

- Positionner des entêtes
- Des listener pour écouter les défaillances de nœud
- Des sélecteurs de nœuds permettant de filtrer les nœuds sollicités
- La définition de callback permettant de modifier la configuration des requêtes et du client (timeout, ssl, ...)



# Requêtes

---

- ***performRequest*** : Appel synchrone

```
Request request = new Request(
    "GET",
    "/");
Response response = restClient.performRequest(request);
```

- ***performRequestAsync*** : Appel asynchrone avec un argument de type *ResponseListener* et retournant un objet *Cancellable*

```
Request request = new Request("GET", "/");
Cancellable cancellable = restClient.performRequestAsync(request,
    new ResponseListener() {
        public void onSuccess(Response response) { }
        public void onFailure(Exception exception) { }
    });
```



# Données de la requête

---

Ajout d'un paramètre :

```
request.addParameter("pretty", "true");
```

Positionnement du corps de la requête

```
request.setEntity(new NStringEntity(  
    "{ \"json\": \"text\" }",  
    ContentType.APPLICATION_JSON));  
request.setJsonEntity("{ \"json\": \"text\" }");
```

Possibilité de mutualiser les options dans une instance de type *RequestOption* (entêtes, sélecteur de nœud, buffer de réponses)





# Réponses

---

L'objet ***Response*** retourné par l'appel synchrone ou passé en argument à la méthode `onSuccess`

```
Response response = restClient.performRequest(new Request("GET", "/"));
RequestLine requestLine = response.getRequestLine();
HttpHost host = response.getHost();
int statusCode = response.getStatusLine().getStatusCode();
Header[] headers = response.getHeaders();
String responseBody = EntityUtils.toString(response.getEntity());
```

2 types d'exceptions :

- *IOException* : Communication
- *ResponseException* : Code!= 2xx



# Sniffer

Un sniffer permet de rafraîchir périodiquement la liste des nœuds

```
RestClient restClient = RestClient.builder(  
    new HttpHost("localhost", 9200, "http"))  
    .build();  
Sniffer sniffer = Sniffer.builder(restClient).build();
```

Le rafraîchissement peut également être provoqué lors d'une erreur

```
SniffOnFailureListener sniffOnFailureListener = new SniffOnFailureListener();  
RestClient restClient = RestClient.builder(new HttpHost("localhost", 9200))  
    .setFailureListener(sniffOnFailureListener)  
    .build();  
Sniffer sniffer = Sniffer.builder(restClient)  
    .setSniffAfterFailureDelayMillis(30000)  
    .build();  
sniffOnFailureListener.setSniffer(sniffer);
```



# High-Level

---

## Java 8

La version du client doit correspondre à celle des nœuds ELS (au moins la version majeure)

## Maven

```
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>elasticsearch-rest-high-level-client</artifactId>
  <version>7.5.2</version>
</dependency>
```

## Gradle

```
dependencies {
  compile 'org.elasticsearch.client:elasticsearch-rest-high-level-client:7.5.2'
}
```



# Initialisation

---

```
RestHighLevelClient client = new  
RestHighLevelClient(  
    RestClient.builder(  
        new HttpHost("localhost", 9200, "http"),  
        new HttpHost("localhost", 9201, "http")));
```

... .

```
// Quand plus nécessaire  
client.close();
```

Possibilité également de positionner les options via  
*RequestOption*



# Objets *Request*

L'API propose de nombreux objets Request en fonction de l'opération que l'on veut effectuer, des objets Response spécifique sont alors retournés :

***IndexRequest/IndexResponse*** pour indexer un document

***GetRequest/GetResponse*** pour récupérer un document

***CreateIndexRequest/CreateIndexResponse*** pour créer un index

***PutMappingRequest/AcknowledgedResponse*** pour spécifier un mapping

***SearchRequest/SearchResponse*** : recherche, indexation, suggestion

***MultiSearchRequest/MultiSearchResponse*** pour effectuer plusieurs requêtes en 1 seule requête HTTP

***ClusterHealthRequest/ClusterHealthResponse*** pour la santé du cluster

***PutPipelineRequest/AcknowledgedResponse*** pour positionner une pipeline

...



# Examples : Indexation

---

```
IndexRequest request = new IndexRequest("posts");
request.id("1");
String jsonString = "{" +
    "\"user\":\"kimchy\"," +
    "\"postDate\":\"2013-01-30\"," +
    "\"message\":\"trying out Elasticsearch\"" +
    "}";
request.source(jsonString, XContentType.JSON);

Map<String, Object> jsonMap = new HashMap<>();
jsonMap.put("user", "kimchy");
jsonMap.put("postDate", new Date());
jsonMap.put("message", "trying out Elasticsearch");
IndexRequest indexRequest = new IndexRequest("posts")
    .id("1").source(jsonMap);
```



# *IndexResponse*

---

```
String index = indexResponse.getIndex();
String id = indexResponse.getId();
if (indexResponse.getResult() == DocWriteResponse.Result.CREATED) {

} else if (indexResponse.getResult() == DocWriteResponse.Result.UPDATED) {

}
ReplicationResponse.ShardInfo shardInfo = indexResponse.getShardInfo();
if (shardInfo.getTotal() != shardInfo.getSuccessful()) {

}
if (shardInfo.getFailed() > 0) {
    for (ReplicationResponse.ShardInfo.Failure failure :
        shardInfo.getFailures()) {
        String reason = failure.reason();
    }
}
```



# *SearchSourceBuilder*

---

Lors d'une requête de type *SearchRequest*, un objet ***SearchSourceBuilder*** est utilisé pour spécifier tous les paramètres de requêtes

Cet objet s'appuie également sur des objets de type ***QueryBuilder***

```
SearchRequest searchRequest = new SearchRequest();
SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
sourceBuilder.query(QueryBuilders.termQuery("user", "kimchy"));
sourceBuilder.from(0);
sourceBuilder.size(5);
sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));
searchRequest.source(sourceBuilder);
```





# *SearchResponse et SearchHits*

---

L'objet retourné ***SearchResponse*** donne tous les informations sur la recherche et en particulier les ***SearchHits***

```
RestStatus status = searchResponse.status();
TimeValue took = searchResponse.getTook();
Boolean terminatedEarly =
searchResponse.isTerminatedEarly();
boolean timedOut = searchResponse.isTimedOut();
int totalShards = searchResponse.getTotalShards();
int successfulShards =
searchResponse.getSuccessfulShards();
int failedShards = searchResponse.getFailedShards();
SearchHits hits = searchResponse.getHits();
```



# *SearchHits*

---

```
TotalHits totalHits = hits.getTotalHits();
float maxScore = hits.getMaxScore();
SearchHit[] searchHits = hits.getHits();
for (SearchHit hit : searchHits) {
    String index = hit.getIndex();
    String id = hit.getId();
    float score = hit.getScore();
    String sourceAsString = hit.getSourceAsString();
    Map<String, Object> sourceAsMap = hit.getSourceAsMap();
    String documentTitle = (String) sourceAsMap.get("title");
    List<Object> users = (List<Object>) sourceAsMap.get("user");
    Map<String, Object> innerObject =
        (Map<String, Object>) sourceAsMap.get("innerObject");
}
```



# Exemple : Recherche multiple

---

```
MultiSearchRequest request = new MultiSearchRequest();
SearchRequest firstSearchRequest = new SearchRequest();
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
searchSourceBuilder.query(QueryBuilders.matchQuery("user", "kimchy"));
firstSearchRequest.source(searchSourceBuilder);
request.add(firstSearchRequest);
SearchRequest secondSearchRequest = new SearchRequest();
searchSourceBuilder = new SearchSourceBuilder();
searchSourceBuilder.query(QueryBuilders.matchQuery("user", "luca"));
secondSearchRequest.source(searchSourceBuilder);
request.add(secondSearchRequest);
```



# Agrégations pipeline



# Agrégations pipeline

---

Les agrégations de type pipeline travaillent à partir d'une sortie produite par une autre agrégation.

Il y en a de 2 types :

- **Parent** : Leurs entrées proviennent d'une agrégation parente et permettent de calculer de nouveaux groupements ou métriques sur les groupement parents
- **Sibling** : Leurs entrées proviennent d'une agrégation juxtaposée et permettent de calculer une nouvelle agrégation du même niveau

Dans les 2 cas, le chemin d'entrée est fourni par le paramètre ***bucket\_path***



# Syntaxe du *bucket\_path*

---

```
AGG_SEPARATOR      = '>' ;
METRIC_SEPARATOR    = '.' ;
AGG_NAME           = <le nom de l'aggregation> ;
METRIC              = <Le nom du metric> ;
PATH                = <AGG_NAME> [ <AGG_SEPARATOR>,
<AGG_NAME> ]* [ <METRIC_SEPARATOR>, <METRIC> ] ;
```

Par exemple :

```
my_bucket>my_stats.avg
```

i.e La valeur moyenne dans le métrique `my_stats` contenu dans le bucket `my_bucket`

Les chemins sont relatifs et ne peuvent pas remonter dans l'arborescence



# Quelques pipelines agrégations

---

avg\_bucket, moving\_avg (sibling): calcule la moyenne (mouvante) d'un autre métrique

derivative (sibling) : Fonction dérivée

min\_bucket, max\_bucket (sibling) : Min et max

sum\_bucket (sibling) : Somme

cumulative\_sum (parent) :

bucket\_script (parent) : Exécution d'un script

...



# Example

---

POST /\_search

```
{
  "aggs": {
    "my_date_histo": {
      "date_histogram": {
        "field": "timestamp",
        "interval": "day"
      },
      "aggs": {
        "the_sum": {
          "sum": { "field": "lemmings" }
        },
        "the_movavg": {
          "moving_avg": { "buckets_path": "the_sum" }
        }
      }
    }
  }
}
```





# Exemple 2

---

```
POST /_search
{
  "aggs" : {
    "sales_per_month" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "month"
      },
      "aggs": {
        "sales": {
          "sum": {
            "field": "price"
          }
        }
      }
    },
    "max_monthly_sales": {
      "max_bucket": {
        "buckets_path": "sales_per_month>sales"
      }
    }
  }
}
```



# Exemple 3

---

# Utilisation du path `_count`

POST /\_search

```
{
  "aggs": {
    "my_date_histo": {
      "date_histogram": {
        "field": "timestamp",
        "interval": "day"
      },
      "aggs": {
        "the_movavg": {
          "moving_avg": { "buckets_path": "_count" }
        }
      }
    }
  }
}
```