



Le moteur de recherche Elastic Search

David THIBAU – 2019

david.thibau@gmail.com



Agenda

- Introduction

- L'offre Elastic Search
- Distribution et installation
- Concepts de base
- Conventions API ELS
- La DevConsole de Kibana

- Indexation et documents

- Document API et Routing
- Clients Elasticsearch
- Ingestion de données et plugins
- API Search Lite
- Distribution de la recherche

- Mapping et analyseurs

- Présentation
- Analyseurs disponibles
- Spécifier le mapping
- Champs méta-données
- Gestion d'index

- Recherche avec DSL

- Syntaxe DSL
- Tri et pertinence

- Recherche avancées

- Recherche mots
- Recherche phrase
- Analyse du langage naturel
- Surbrillance
- Jointure
- Agrégations
- Géolocalisation

- Administration et monitoring

- API de monitoring
- Mise en production
- Exploitation

- Annexes



L'offre Elastic Search



Introduction

La société *elastic* commercialise et propose en OpenSource la suite Elasticsearch 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 RESTFuL** très complète
- ✓ **OpenSource**



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 :**
 - Un index contient des **documents**
 - Un document contient des **champs** (« fields »)
 - Un champ est constitué de **termes** (« terms »)

ELS lui apporte la scalabilité, une API REST, de la simplicité et des fonctionnalités d'agrégation





ELS vs SolR

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

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

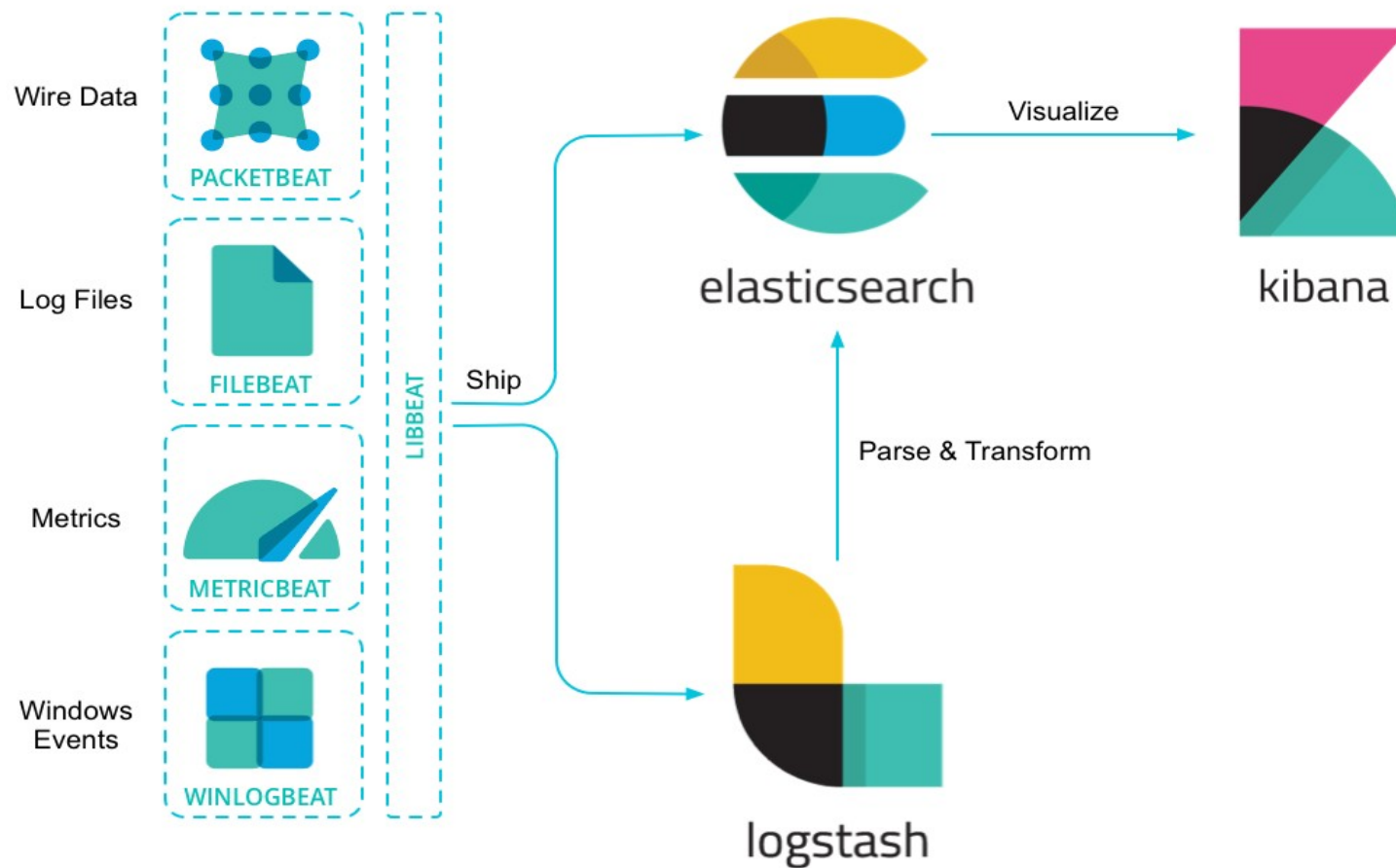


Elastic Stack

La suite Elastic Stack dédiée au BigData est composée de 3 principaux produits

- **Elastic Search** : Stockage, Indexation, Recherche
- **Kibana** : Génération de graphique pour le reporting
- **Logstash / Beat** : Récupération et centralisation des données d'entreprise

Architecture



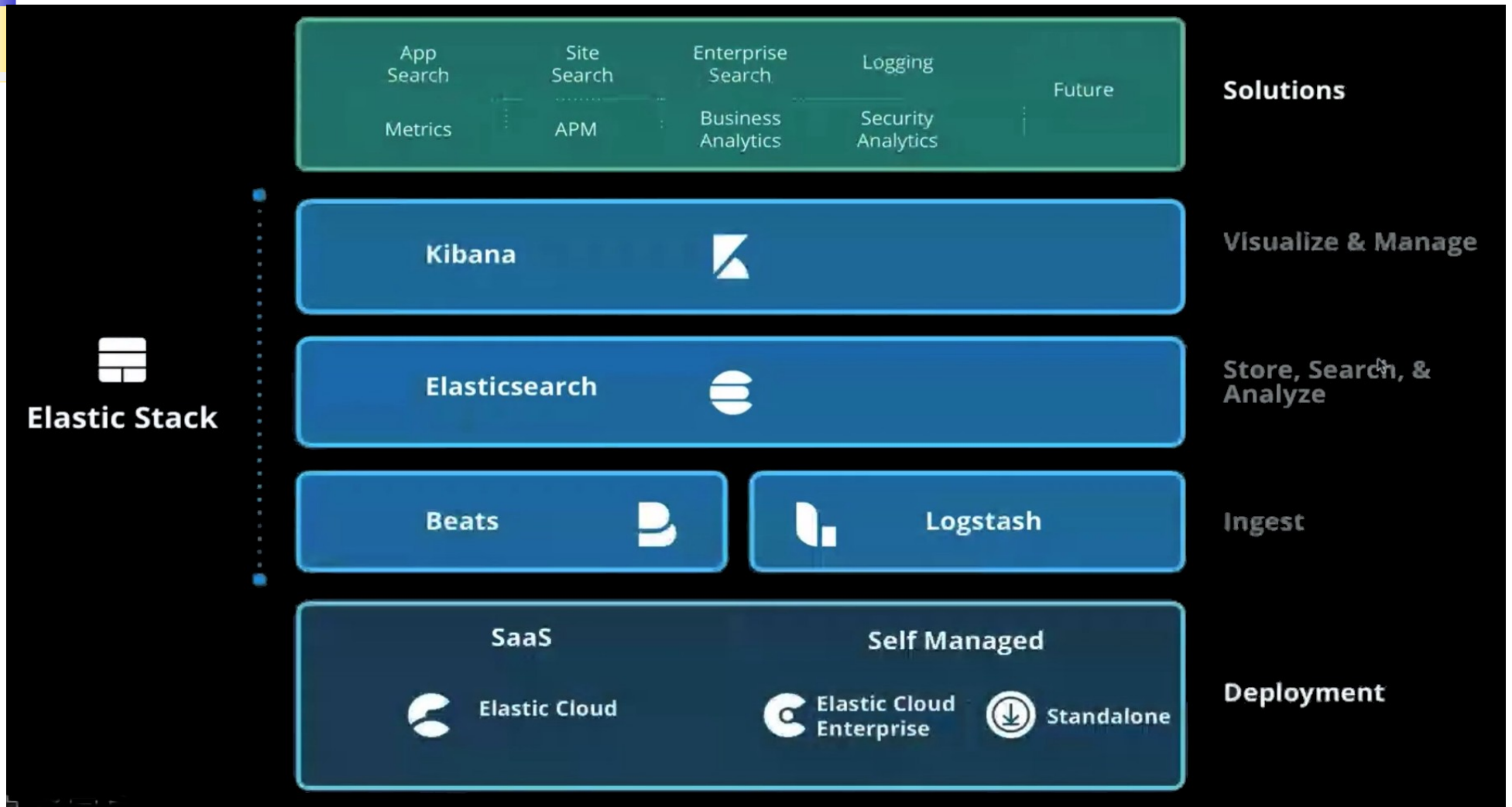


Offres Connexes

Autour des produits de base de la suite, la société Elastic propose :

- **X-Pack** offre partiellement payante intégré aux versions 6.3+
 - Sécurité, Monitoring
 - Alertes et Machine Learning
 - APM
 - Reporting PDF, planification d'envoi de rapports
- **ES-Hadoop** : Connecteur avec Apache Hadoop
- **Elastic Cloud** : Déléguer l'exploitation de son cluster ou créer un cloud d'entreprise

En résumé





Distributions et installation



Distributions

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

- ZIP, TAR, DEB ou RPM
- Également image Docker

Versions

- 7.5.1 : Décembre 2019
- 7.0.0 : 10 Avril 2019
- 6.8.6 : Décembre 2019
- 6.5.4 : Décembre 2018
- 6.0.0 : Novembre 2017
- 5.0.0 : Octobre 2016
- 2.4.1 : Septembre 2016
- 2.4.0 : Août 2016



Installation

1. Pré-requis Java > 8

2. Dézipper l'archive

3. Exécuter ELS

`bin/elasticsearch`

4. Accéder à ELS

`curl http://localhost:9200/`

Configuration



Il existe plusieurs fichiers principaux de configuration :

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

La configuration par défaut permet de démarrer rapidement après une indexation.

En production, il faut quand même modifier certains paramètres. Par exemple :

- Le nom du nœud : ***node.name***
- Les chemins : ***paths***
- Le nom du cluster ***cluster.name***
- L'interface réseau : ***network.host***



Le format YAML

Le format YAML permet de fixer les valeurs des propriétés de configuration.

Le format prend en compte les **:** et les **tabulations**

```
path:  
    data: /var/lib/elasticsearch  
    logs: /var/log/elasticsearch
```

Est équivalent à

```
path.data: /var/lib/elasticsearch  
path.logs: /var/log/elasticsearch
```



elasticsearch.yml

ELS apporte certaines fonctionnalités au format YAML :

- Les variables d'environnement sont accessibles via `${ENV_VAR}`
- Certaines propriétés peuvent être demandées au démarrage :
`${prompt.text}` , `${prompt.secret}`
- Les valeurs positionnées par la commande en ligne écrasent la valeur définie dans *elasticsearch.yml*
`./elasticsearch -Enode.name=David`



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*



Concepts de base



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 constitué que d'un seul nœud

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



Nœud

Un **nœud** est un simple serveur faisant partie d'un cluster.

Un nœud stocke des données, et participe aux fonctionnalités d'indexation et recherche du cluster.

- Un nœud est également identifié par un nom unique (généré automatiquement si pas renseigné)

Le nombre de nœuds dans un cluster n'est pas limité



Nœud maître

Dans un cluster un nœud est élu comme **nœud maître**, c'est lui qui est en charge de gérer la configuration du cluster et les créations d'index.

Pour toutes les opérations sur les documents (indexation, recherche), chaque nœud de données est **interchangeable** et un client peut s'adresser à n'importe lequel des nœuds

Dans de grosses architectures, des nœuds peuvent être spécialisés pour l'ingestion de données, les processus ML, et dans ce cas, ils ne participent pas aux fonctionnalités de recherche.



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



Type

A l'intérieur d'un index, on peut définir un ou plusieurs **types** de documents

Un type est une partition logique de l'index

Il définit des documents qui ont les mêmes champs

Déprécié dans la version 6.x, encore présent dans l'API mais ne plus s'appuyer dessus !

**=> Les index sont mono-type !!
Par convention, le type est `_doc`**



Document

Un **document** est l'unité basique d'information qui peut être indexée.

- Le document est un ensemble de champs (clé/valeur) exprimés avec le format JSON

A l'intérieur d'un index/type, on peut stocker autant de documents 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***

- A la création de l'index, il est possible de définir le nombre de shards

Chaque *shard* est un index indépendant qui peut être hébergé sur un des nœuds 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, il est recommandé d'utiliser des mécanismes de failover dans le cas où un nœud défaille

ELS permet de mettre en place des copies des shards : les **répliques**

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 .



Résumé

En résumé chaque index peut être divisé sur plusieurs shards.

Il peut être répliqué plusieurs fois

Une fois répliqué, chaque index a des shards primaires et des shards de réplication

Le nombre de shards et de répliques peuvent être spécifiés au moment de la création de l'index

Après sa création, le nombre de répliques peut être changé dynamiquement **mais pas** le nombre de shards

Par défaut, ELS 6.x alloue 5 shards primaires et une réplique, ELS 7.x 1 shard et 1 réplique



Conventions de l'API REST



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 **X-Pack** : Configuration des fonctionnalités X-Pack
- ...



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

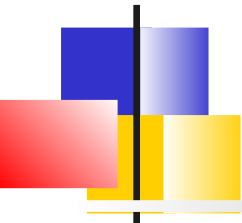
La plupart des APIs qui référencent un index peuvent s'effectuer sur plusieurs index :

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

Date Math : *+1h* : Ajout d'une heure, *-1d* : Soustraction d'une journée, */d* : Arrondi au jour le plus proche

?filter_path : Filtre de réponse, permettant de spécifier les données de la réponse

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



Options communes (2)

?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

Unités de temps : **d, h, m, s, ms, micros, nanos**

Unité de taille : **b, kb, mb, gb, tb, pb**

Nombre sans unités : **k, m, g, t, p**

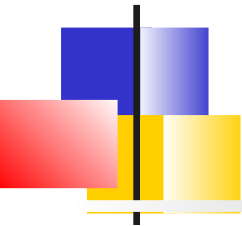
Unités de distance : **km, m, cm, mi, yd, ft**



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 le load-balancing 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

```
GET /_search
```

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



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
- Elle peut être désactivée (*console.enabled: false*)



Installation Kibana

La version de *Kibana* doit correspondre rigoureusement à celle d'*ElasticSearch*

La distribution inclut également la bonne version de *Node.js*

Elle est disponible sous différents formats :

- Archive compressée
- Package debian ou rpm
- Image Docker

Installation à partir d'une archive



```
wget  
https://artifacts.elastic.co/downloads/kibana/kibana-  
5.0.1-linux-x86_64.tar.gz  
sha1sum kibana-5.0.1-linux-x86_64.tar.gz  
tar -xzf kibana-5.0.1-linux-x86_64.tar.gz  
cd kibana/  
./bin/kibana
```



Structure des répertoires

bin : Script binaires dont le script de démarrage et le script d'installation de plugin

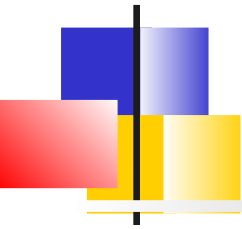
config : Fichiers de configuration dont *kibana.yml*

data : Emplacement des données, utilisé par Kibana et les plugins

optimize : Code traduit.

plugins : Emplacement des plugins. Chaque plugin correspond à un répertoire

Configuration



Les propriétés de Kibana sont lues dans le fichier ***conf/kibana.yml*** au démarrage

Les propriétés principales sont :

- ***server.host, server.port*** : défaut localhost:5601
- ***elasticsearch.url*** : <http://localhost:9200>
- ***kibana.index*** : Index d'ElasticSearch utilisé pour stocker les recherches et les tableaux de bord
- ***kibana.defaultAppId*** : L'application utilisée au démarrage Par défaut *discover*
- ***logging.dest*** : Fichier pour les traces. Défaut *stdout*
- ***logging.silent, logging.quiet, logging.verbose*** : Niveau de log



Indexation et Documents

Qu'est-ce qu'un document
Document API
Routing
API Search Lite
Distribution de la recherche



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, les autres sont indexés tel quel



Structure de données

Un document est donc une **structure de données**.

Le format utilisé par ELS est le format JSON

- Chaque champ a une ou plusieurs valeurs (tableau)
- Un champ peut également être une structure de données. (Imbrication)



Exemple

```
{
  "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
- ***_type*** : La classe de l'objet que le document représente. **(Déprécié avec 6.x, un index est mono-type ! Par convention _doc)**
- ***_id*** : L'identifiant unique
- ***_version*** : N° de version du document
- ...



Document API



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

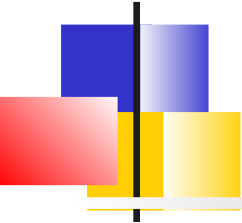
- de fournir l'ID
- ou de laisser ELS le générer



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

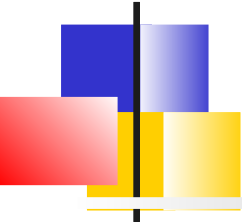
GET /website/_doc/123?_source=title,text

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*** pour fusionner les champs fournis avec les champs existants



Exemple

POST /website/_doc/1/_update

```
{  
  "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. L'API Bulk ne peut donc pas être utilisée pour mettre en place des 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 formattés



Syntaxe

La ligne action/metadata spécifie :

- l'action :
 - ***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.
- Les méta-données : *_id*, *_index*, *_type*



Exemple

POST /website/_bulk // **website est l'index par défaut**

```
{ "delete": { "_type": "_doc", "_id": "123" }}  
{ "create": { "_index": "website2", "_type": "_doc", "_id":  
"123" }}  
{ "title": "My first blog post" }  
{ "index": { "_type": "_doc" }}  
{ "title": "My second blog post" }  
{ "update": { "_type": "_doc", "_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
    } }
  ]
}
```



Routing



Résolution du shard primaire

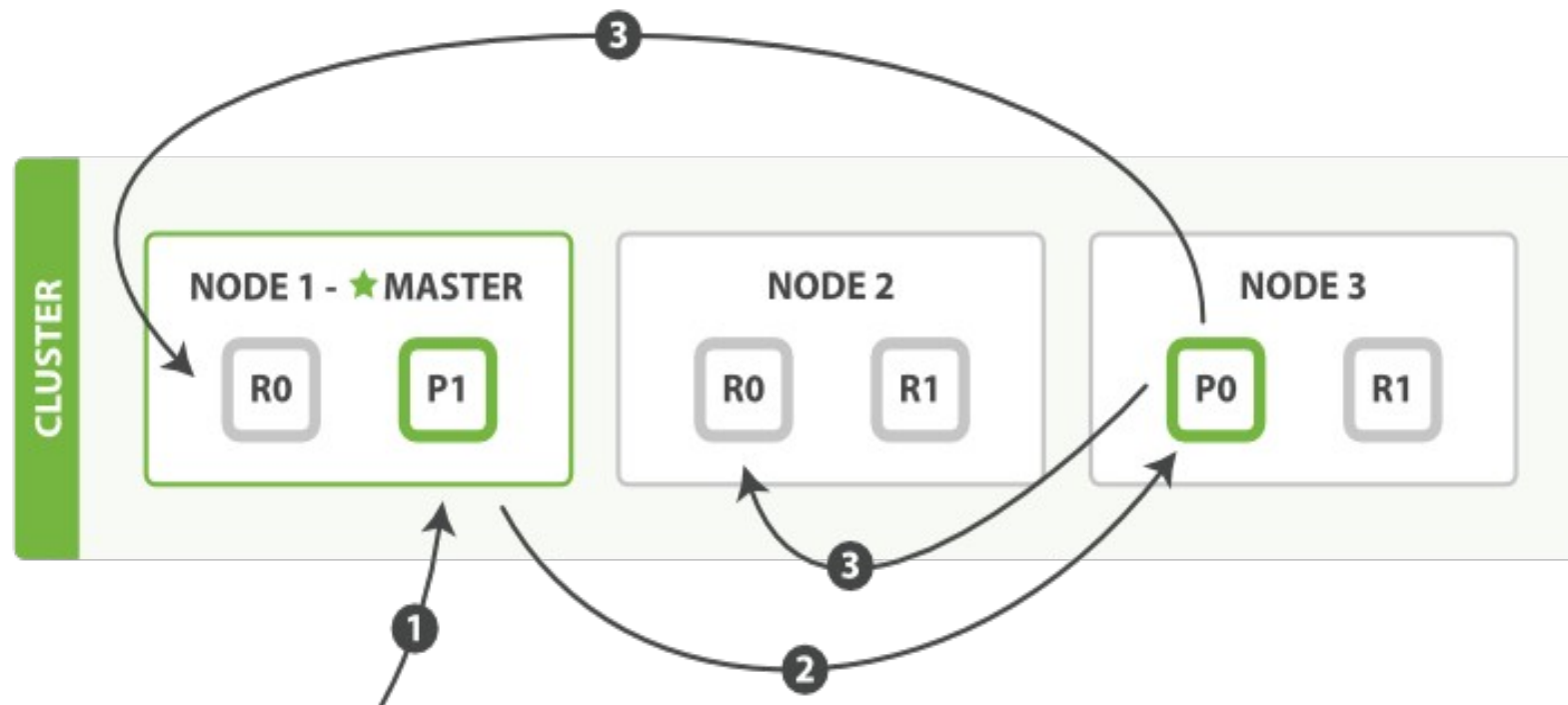
Lors de l'indexation, le document est d'abord stocké sur le shard primaire.

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

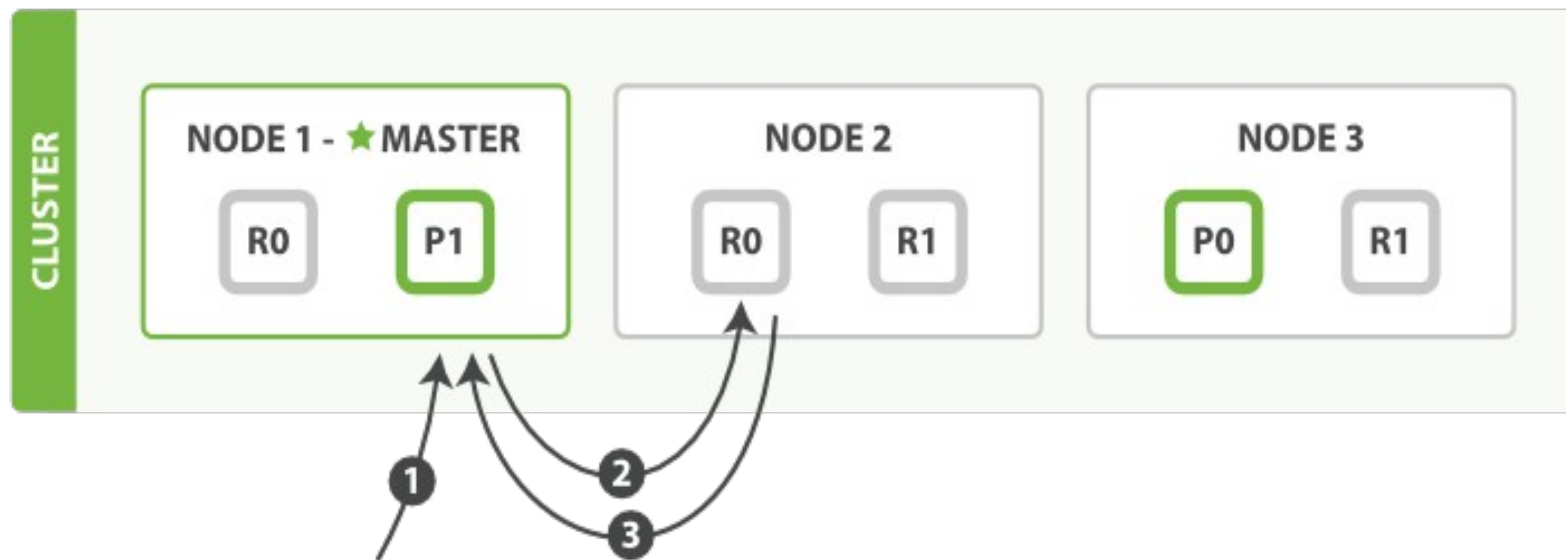


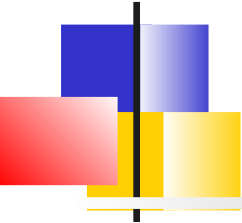


Séquence d'une mise à jour sur une architecture cluster (2)

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)



Clients Elastic Search



Clients Elastic Search

Elastic fournit et supporte des clients écrits en différents langages :

- Java REST Client : Appel REST en Java
- Java API : Appel direct en Java
- JavaScript API
- Groovy API
- .NET API
- PHP API
- Perl API
- Python API
- Ruby API



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);
});
```




Ingestion de données



Introduction aux plugins

Les fonctionnalités de ELS peuvent être augmentées via la notion de **plugin**

Les plugins contiennent des fichiers JAR, mais également des scripts et des fichiers de configuration

Ils doivent être installés sur chaque nœud du cluster

Après une installation, chaque nœud doit être redémarré



Installation

L'installation des plugins cœur (gérés par ELS) se fait en général de la même façon :

```
sudo bin/elasticsearch-plugin install [plugin_name]
```



Ingest plugins

Les plugins de type ***ingest*** permettent d'effectuer des traitements sur les données avant leur indexation .

ELS propose :

- ***Attachment Plugin*** : Extrait les données textes de pièces jointes en différents formats (PPT, XLS, PDF, ...). Il utilise la librairie Apache Tika.
- ***Geoip Plugin*** : Ajoute des informations concernant l'emplacement d'une adresse IP. Il utilise par défaut le champ *geoip*
- ***User agent plugin*** : Extrait des détails à partir de l'entête HTTP User-Agent .



Ingest attachment

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

Il utilise *Tika*

Installation :

```
sudo bin/elasticsearch-plugin install ingest-attachment
```



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



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/my_type/my_id?pipeline=attachment
{
  "data":
  "e1xydGYxXGFuc2kNCkxvcmVtIGlwc3VtIGRvbG9yIHNpdCBhbWV0DQpccGFy
  IH0="
}
```



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
    }
  }
}
```



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
}
```



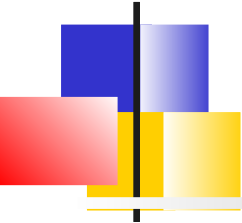
Champs de la réponse

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

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



Exemples search lite

GET /_all/tweet/_search?q=tweet:elasticsearch
Tous les documents de type tweet dont le champ full text *match* « elasticsearch »

+name:john +tweet:mary

GET /_search?q=%2Bname%3Ajohn+%2Btweet%3Amary

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



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

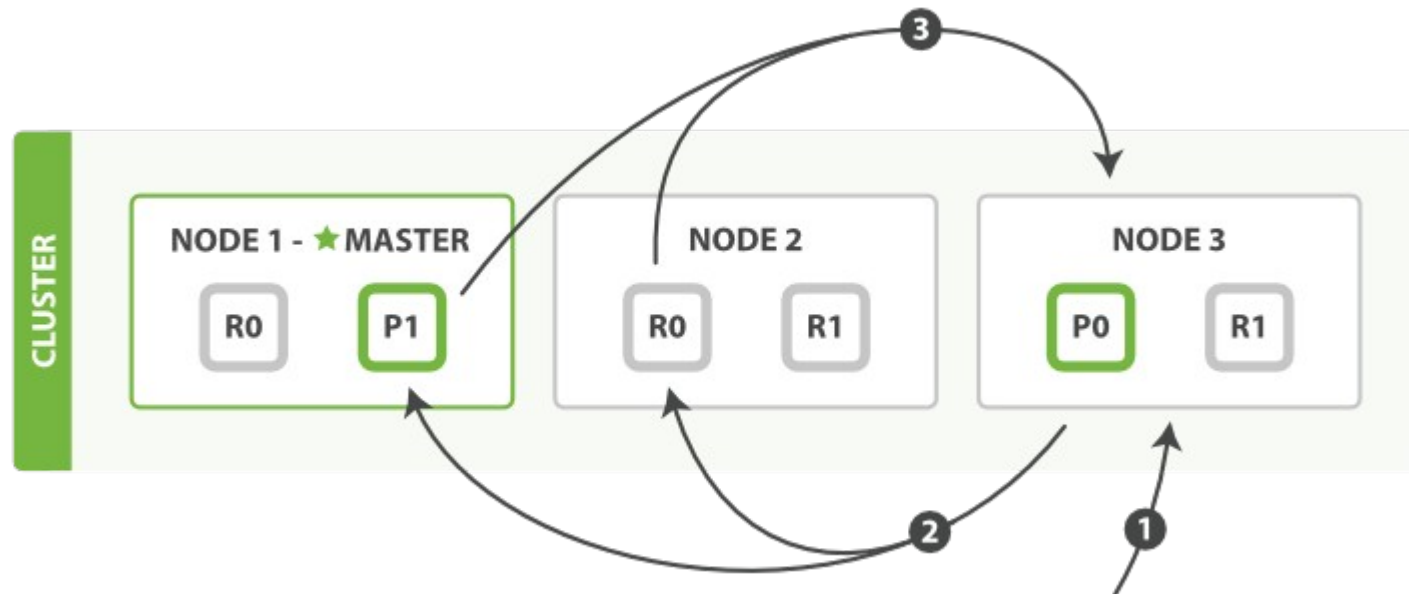
Une fois trouvés ; les résultats des différents shards doivent être combinés en une liste unique afin que l'API puisse retourner une page de résultats

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



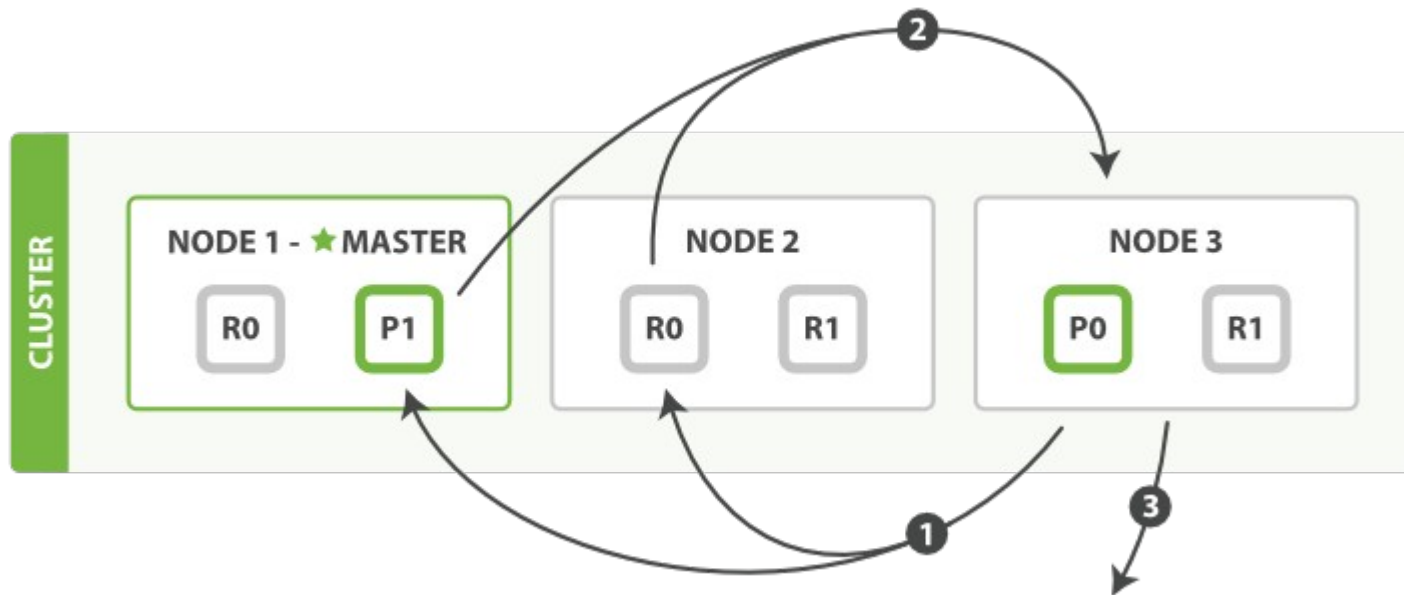


Phase de requête

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.



Mapping et Analyseurs

Types simples et champs full text
Analyseurs
Contrôler le mapping
Retour sur la configuration d'index
Types complexes
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 naturelS



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 méthode est consitué après une phase d'analyse du champ texte originel.

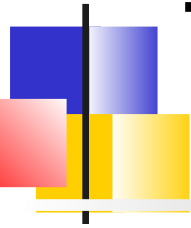


Exemple

	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



Analyseurs



Tokenisation et Normalisation

Pour construire l'index inversé, ELS doit séparer un texte en mots (**tokenisation**) et ensuite **normaliser** les mots afin que la recherche du terme « envoi » par exemple retourne des documents contenant : « envoi », « Envoi », « envois », « envoyer », ...

La tokenisation et la normalisation sont appelées **analyse**.

L'analyse s'applique sur les documents lors de l'indexation **ET** lors de la recherche sur les termes recherchés.

Étapes de l'analyse

- Les **analyseurs** transforment un texte en un flux de “token”. Ils peuvent être constitués d'une seule classe Java ou d'une combinaison de filtres, de tokenizer et de filtres de caractères
 - 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 tokens
 - Les **filtres** prend en entrée un flux de token et le transforme en un autre flux de token

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
  } ] }
```

Spécification des analyseurs

- 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.
- Si ce n'est pas le comportement voulu, il faut explicitement spécifier le **mapping** pour le type de document



Contrôle du mapping



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 :

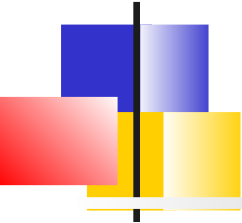
- Faire une distinction entre les champs string de type full-text ou valeur exacte
- Utiliser des analyseurs spécifiques
- Optimiser un champ pour le matching partiel (voir + loin)
- Spécifier des formats de dates personnalisés
- Définir plusieurs types et/ou analyseurs pour le même champ



Spécification du mapping

On peut spécifier le mapping :

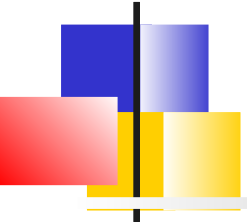
- Lors de la création d'un index
- Lors de l'ajout d'un nouveau type
- Lors de l'ajout d'un nouveau champ dans un type existant
=> On ne peut pas modifier un champ déjà indexé



Exemple (création)

PUT /gb

```
{  
  "mappings": {  
    "tweet" : {  
      "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 de stocker une seule fois un champ simple mais de l'indexer de plusieurs façons . Tous les types cœur d'ELS acceptent un paramètre **fields** permettant 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



Retour sur la création et la configuration d'index

Champs *_source* et *_all*
Contrôler le dynamic mapping
Analyseurs dédiés



Introduction

ELS permet de démarrer sans mise en place particulière

Par contre, pour une mise en production, il est nécessaire de tuner finement les processus d'indexation et de recherche afin de s'adapter à son cas d'utilisation

La plupart de ces personnalisations concernent les index



Création manuelle de l'index

Lors de la création, 2 blocs peuvent être configurées :

- **settings** : Principalement répliques et shards
- **mappings** : Les types de champs et les analyseurs

```
PUT /my_index
{
  "settings": { ... any settings ... },
  "mappings": {
    "type_one": { ... any mappings ... },
    ...
  }
}
```

Il est également possible de désactiver la création automatique

```
# config/elasticsearch.yml
```

```
action.auto_create_index: false
```



Répliques et shards

Dans la partie settings, les 2 principales configurations sont :

- Le **nombre de shards** : nombre de shards primaires (par défaut 5) . Cette configuration ne peut pas être changée après la création
- Le **nombre de répliques** : Par défaut 1, cette configuration peut être changée à tout moment

```
PUT /my_temp_index
{
  "settings": {
    "number_of_shards" : 1,
    "number_of_replicas" : 0
  } }
```



Champ *_source*

ELS stocke la string JSON (compressée) représentant le document dans le champ ***_source***.

Ce champ apporte plusieurs fonctionnalités :

- Le document complet est accessible à partir des résultats de recherche
- Les requêtes de mise à jour partielle sont possibles
- Si une réindexation est nécessaire, le champ *_source* peut être utilisé
- Les valeurs individuelles des champs peuvent être extraites du champ source lors des requêtes GET ou SEARCH
- Cela facilite le debugging



Configuration de *_source*

Si le comportement par défaut n'est pas désirable :

- Il est possible de désactiver le stockage du document source:

```
PUT /my_index
{ "mappings": { "my_type":
{ "_source": { "enabled": false }
} } }
```

- Ou d'affiner les champs à stocker via les attributs *excludes*,
includes

```
PUT /my_index
{ "mappings": { "my_type":
{ "_source": { "excludes": ["data","content"] }
} } }
```

Mais cependant, attention plus de surbrillance dans les recherches full-text

Champ *_all*

Le champ *_all* concaténant tous les champs n'est souvent plus utile lorsque l'on affine son index.

Pour le désactiver :

```
PUT /my_index/_mapping/  
{ "my_type": { "_all": { "enabled": false } } }
```

Pour spécifier les champs le constituant :

```
PUT /my_index/my_type/_mapping  
{ "my_type": { "include_in_all": false,  
  "properties": {  
    "title": { "type": "string", "include_in_all": true },  
    ...  
  } } }
```



copy_to

Le champ ***copy_to*** permet de créer son propre champ *_all*

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "properties": {
        "first_name": {
          "type": "text",
          "copy_to": "full_name"
        },
        "last_name": {
          "type": "text",
          "copy_to": "full_name"
        },
        "full_name": {
          "type": "text"
        }
      }
    }
  }
}
```

Dynamic Mapping



La détection et l'ajout automatique de nouveau champs est le ***dynamic mapping***

Les règles d'affectation des types de données 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

La propriété *mappings/_default_* est dépréciée dans la version 6.x



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": {
    "_doc": {
      "dynamic_date_formats": ["MM/dd/yyyy"]
    }
  }
}
```

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "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 de règle

```
{ "string_default": {  
  "match_mapping_type": "string",  
  "mapping": {  
    "type": "text",  
    "analyzer": "french",  
    "fields": {  
      "raw": {  
        "type": "keyword",  
        "ignore_above": 256  
      }  
    }  
  }  
}
```


Gabarits dynamiques



Les gabarits dynamiques sont des collections de règles nommées.

Ils peuvent être associés à un index ou un gabarit d'index (voir plus loin)

La syntaxe est la suivante :

```
"dynamic_templates": [  
  {  
    "my_template_name": { // Nom de la règle  
      ... match conditions ... // match_mapping_type, match, match_pattern, ...  
      "mapping": { ... } // Le mapping à utiliser  
    },  
    ...  
  ]
```

Les règles sont traitées dans l'ordre et la première qui match gagne



Example

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  
          } } } } } ] } } }
```



Analyseurs

L'analyseur par défaut est l'analyseur standard qui consiste en :

- Le tokenizer standard qui sépare sur les frontières de mots
- Le filtre de token standard (qui ne fait rien)
- Le filtre lowercase qui passe tout en minuscule
- Le filtre stopwords (qui contient un minimum de stop words)

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"

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"
  }
}
```



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



Syntaxe DSL



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



Principaux opérateurs



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" } }
```



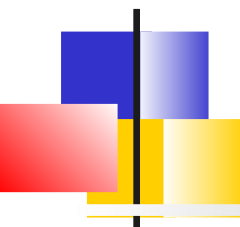
match_all, match_none

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]"
```

```
  } ]
```

```
}
```



Tri et pertinence



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:honey moon 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,
    } ]
  } ]
} ] }
```



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
Analyse langage naturel
Surbrillance
Jointures
Agrégations
Géo-localisation



Requêtes avec préfixe



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

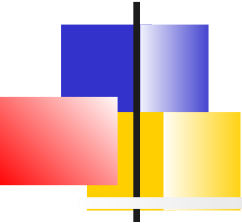
edge_ngram

```
{  
  "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.



Requêtes phrase



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  
    }  
  }  
}
```



Langage naturel et requêtes fuzzy



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"
    }
  }
}
```



Surbrilliance



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



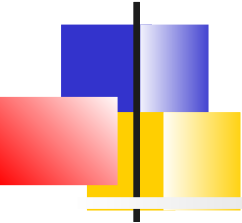
Example

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



Réponse

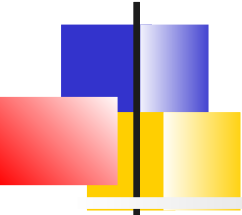
```
"highlight" : {  
    "attachment.content" : [  
        " formation Administration Oracle  
Forms/Report permet de voir tous les aspects nécessaires à",  
        " aux autres services Web de l'offre  
Oracle Fusion Middleware. Cependant, l'administration  
de ces",  
        " Surveillance. Elle 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



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"
        }
      }
    }
  }
}
```



Example

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": ...,  
              ...  
            },  
            ...  
          ] } } },  
    ... }], ... ]
```



Agrégations



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 une 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" }
        }
      }
    }
  }
}
```




Géo-localisation

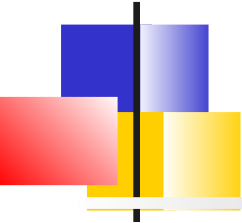


Introduction

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]
        }
      }
    }
  }
}
```



Administration

Surveillance du cluster
Déploiement en production
Exploitation



Monitoring X-Pack

ELS offre une API permettant d'obtenir certains métriques d'un cluster : *_cluster*

L'outil de surveillance recommandé par ELS est désormais inclus dans les fonctionnalités gratuites de X-Pack

Il sollicite périodiquement l'API, stocke les données dans ELS et fournit des dashboards Kibana très détaillé



Cluster Health API

```
GET _cluster/health
{
  "cluster_name": "elasticsearch_zach",
  "status": "green", // green, yellow or red
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 10,
  "active_shards": 10,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 0
}
```



Information au niveau des index

```
GET _cluster/health?level=indices
{
  "cluster_name": "elasticsearch_zach",
  "status": "red",
  ...
  "unassigned_shards": 20
  "indices": {
    "v1": {
      "status": "green",
      "number_of_shards": 10,
      "number_of_replicas": 1,
      "active_primary_shards": 10,
      "active_shards": 20,
      "relocating_shards": 0,
      "initializing_shards": 0,
      "unassigned_shards": 0
    },
```



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)



Section OS et processus

Ce sont les chiffres basiques sur la charge CPU et l'usage mémoire au niveau système

- CPU
- Usage mémoire
- Usage du swap
- Descripteurs de fichiers ouverts



Section JVM

La section **jvm** contient des informations critiques sur le processus JAVA

En particulier, il contient des détails sur la collecte mémoire (garbage collection) qui a un gros impact sur la stabilité du cluster

La chose à surveiller est le nombre de collectes majeures qui doit rester petit ainsi que le temps cumulé dans les collectes

collection_time_in_millis .

Si les chiffres ne sont pas bon, il faut rajouter de la mémoire ou des nœuds.



Section pool de threads

ELS maintient des pools de threads pour ses tâches internes.

En général, il n'est pas nécessaire de configurer ces pools.



File system et réseau

ELS fournit des informations sur votre **système de fichiers** : Espace libre, les répertoires de données, les statistiques sur les IO disques

Il y a également 2 sections sur le **réseau**

- **transport** : statistiques basiques sur les communications inter-nœud (port TCP 9300) ou clientes
- **http** : Statistiques sur le port HTTP. Si l'on observe un très grand nombre de connexions ouvertes en constante augmentation, cela signifie qu'un des clients HTTP n'utilise pas les connexions *keep-alive*. Ce qui est très important pour les performances d'ELS



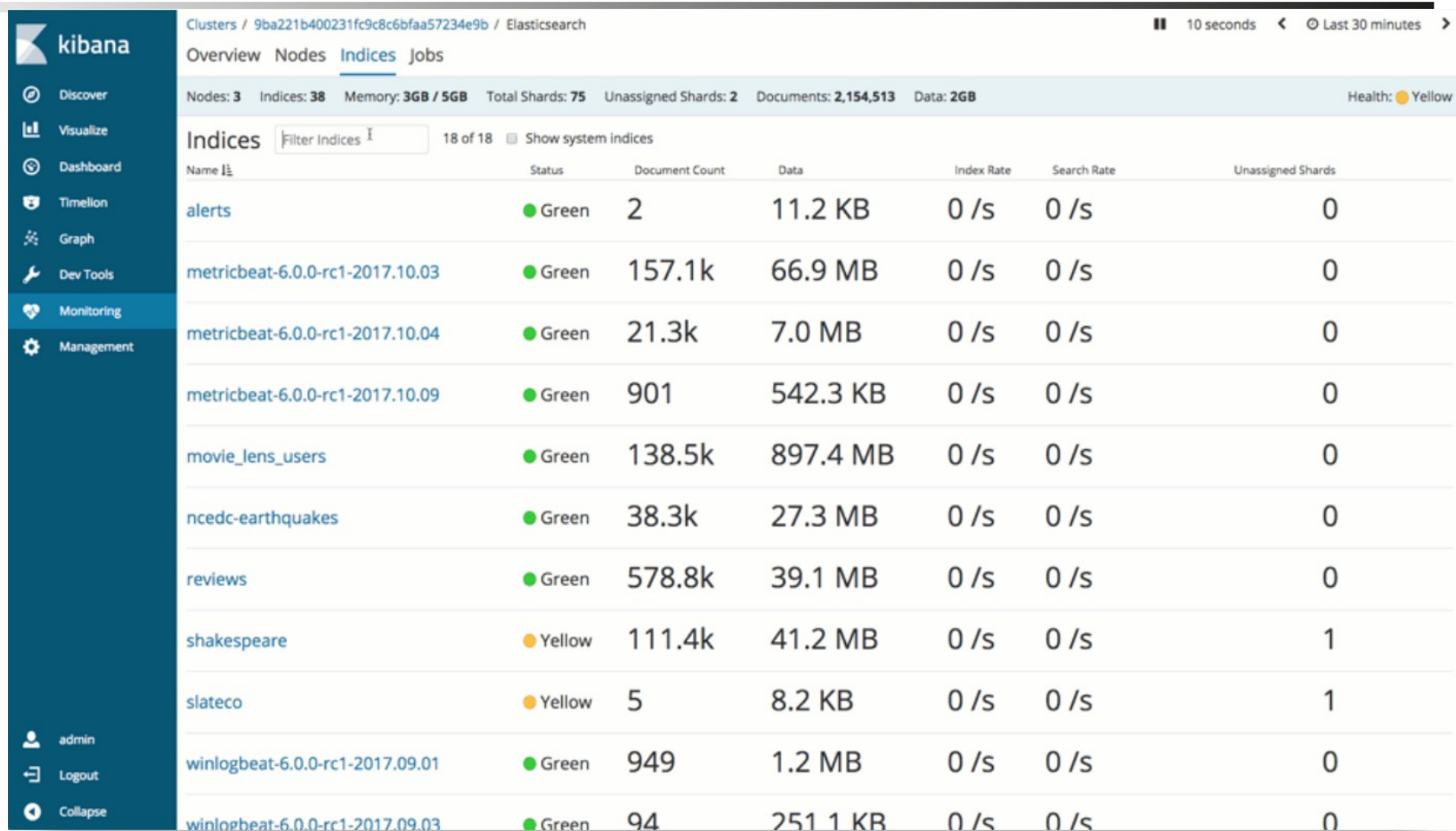
Index stats API

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, ...

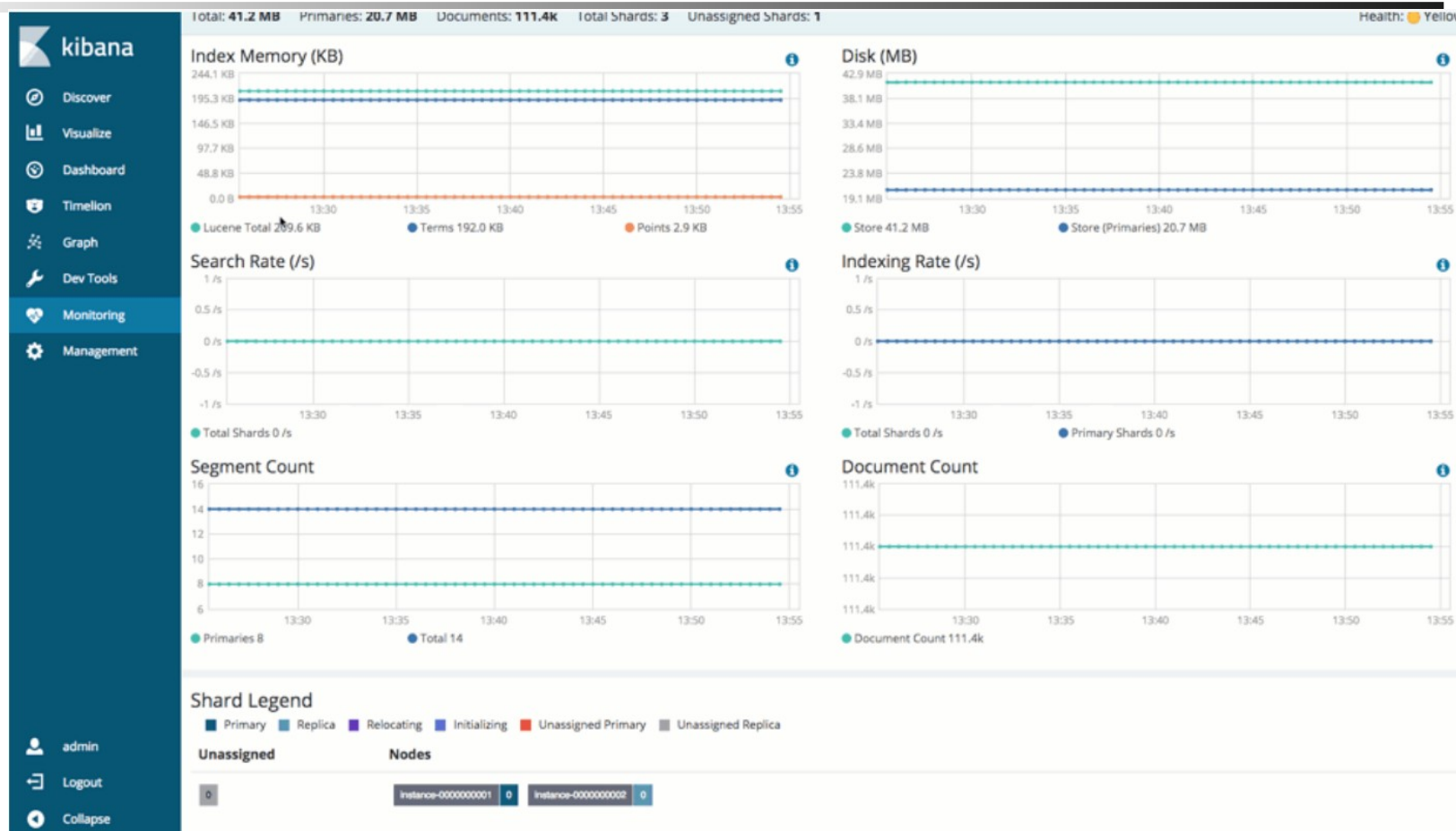
Example XPack1



The screenshot shows the Kibana Monitoring Dashboard. The left sidebar contains navigation links: Discover, Visualize, Dashboard, Timelion, Graph, Dev Tools, Monitoring (selected), and Management. The main content area displays the 'Indices' tab for an Elasticsearch cluster. At the top, a summary bar 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, a table lists 18 indices. The table columns are: Name, Status, Document Count, Data, Index Rate, Search Rate, and Unassigned Shards. The indices listed are: alerts, metricbeat-6.0.0-rc1-2017.10.03, metricbeat-6.0.0-rc1-2017.10.04, metricbeat-6.0.0-rc1-2017.10.09, movie_lens_users, ncedc-earthquakes, reviews, shakespeare, slateco, winlogbeat-6.0.0-rc1-2017.09.01, and winlogbeat-6.0.0-rc1-2017.09.03. The 'shakespeare' and 'slateco' indices are marked as Yellow, while the others are Green.

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





Production



Matériel

RAM : Le talon d'Achille de ELS. Une machine avec 64 GB est idéale ; 32 GB et 16 GB sont corrects

CPU : Favoriser le nombre de coeur plutôt que la rapidité du CPU

Disques : Si possible rapides, SSDs ?
Éviter NAS (network-attached storage)



JVM

Dernière version d'Oracle ou OpenJDK

Surtout ne pas modifier la
configuration de la JVM fournie par ELS.
Elle est issue de l'expérience



Gestion de configuration

Utiliser de préférence des outils de gestion de configuration comme Puppet, Chef, Ansible ...

sinon la configuration d'un cluster avec beaucoup de nœuds devient rapidement un enfer



Personnalisation d'une configuration

La configuration par défaut défini déjà beaucoup de choses correctement, Il y a donc peu à personnaliser. Cela se passe dans le fichier *elasticsearch.yml*

- Le **nom du cluster** (le changer de *elasticsearch*)
`cluster.name: elasticsearch_production`
- Le **nom des nœuds**
`node.name: elasticsearch_005_data`
- Les **chemins** (hors du répertoire d'installation de préférence)
`path.data: /path/to/data1,/path/to/data2`
`# Path to log files:`
`path.logs: /path/to/logs`
`# Path to where plugins are installed:`
`path.plugins: /path/to/plugins`



Spécialisation des nœuds

Tous les nœuds d'un cluster se connaissent mutuellement et peuvent rediriger des requêtes HTTP vers le nœud approprié. Il est possible de spécialiser les nœuds afin de répartir la puissance entre la charge de gestion des données et la charge d'ingestion.

Les différents types de nœuds sont :

- Nœuds pouvant être **maître** : ***node.master*** à *true*
- Nœuds de **données** : ***node.data*** à *true*. Détient les données et effectue les tâches d'indexation et de recherche
- Nœuds **d'ingestion** : ***node.ingest*** à *true*. Exécute les pipelines d'ingestion
- Nœuds de **coordination** : Nœuds acceptant les requêtes et redirigeant vers les nœuds appropriés
- Nœuds **tribe** ou **cross-cluster** (Version 6.x). Propriétés ***tribe.****
Nœuds pouvant effectuer des recherches vers plusieurs cluster.



Nœuds maître

Le nœud maître est responsable d'opérations légères :

- Création ou suppression d'index
- Surveillance des nœuds du cluster
- Allocations des shards

Dans un environnement de production, il est important de s'assurer de la stabilité du nœud maître.

Il est conseillé pour de gros cluster de ne pas charger les nœuds maîtres avec des travaux d'ingestion, d'indexation ou de recherche.

```
node.master: true  
node.data: false  
node.ingest: false  
search.remote.connect: false
```



Configurations des nœuds

Configuration d'un nœud de **données**

```
node.master: false  
node.data: true  
node.ingest: false  
search.remote.connect: false
```

Configuration d'un nœud **d'ingestion**

```
node.master: false  
node.data: false  
node.ingest: true  
search.remote.connect: false
```

Configuration d'un nœud de **coordination**

```
node.master: false  
node.data: false  
node.ingest: false  
search.remote.connect: false
```



Découverte et formation du cluster

Le **module de découverte** est responsable de découvrir les nœuds, d'élire un maître, de former un cluster et de publier l'état du cluster chaque fois qu'il change

Différents process sont exécutés par ce module :

- La découverte : les nœuds se découvrent lorsque le maître est inconnu
Au démarrage du cluster ou quand le maître courant est down
- Prise de décision basée sur un quorum
- Configurations de vote : les masters pouvant voter
Mise à jour des configuration lors d'ajout ou de retrait de nœuds master
- Premier démarrage (bootstrapping)
Au premier démarrage d'un cluster, le processus est différent
- Ajout/retrait de nœud maître
- Publication de l'état du cluster
- Détection de faute



Découverte

Ce processus utilise une liste d'adresses de départ : ***seed address*** ainsi que les adresses de tous les nœuds éligibles maîtres qui se trouvaient dans le dernier cluster connu

- Chaque nœud contacte les autres nœuds et s'échange les informations sur les nœuds éligible maître actifs
- Les seed address sont fournies dans la configuration, dans un fichier externe ou par un plugin (AWS EC2, Azure, GCE, ...)



Prise de décision

La prise de décision concerne l'élection d'un maître ou le retrait/ajout d'un nœud.

- Elle s'effectue par les nœuds éligibles maître.
- Il faut qu'un quorum de nœuds soit d'accord.
- Le nœud participant au vote sont les nœuds présent dans la configuration de vote

```
GET /_cluster/state?  
filter_path=metadata.cluster_coordination.last_committed_config
```



Bootstrapping

Lorsqu'un cluster démarre pour la première fois, il doit élire son premier nœud maître sans configuration de vote.

- La propriété ***cluster.initial_master_nodes*** fixe les nœuds participants au 1^{er} vote
- Cette configuration n'est ensuite plus requise pour les redémarrages

Personnalisation d'une configuration (2)

- ***minimum_master_nodes*** : Le nombre minimal de nœuds éligible comme master pour qu'une élection ait lieu. Le fixer à un quorum des nœuds de type master
`discovery.zen.minimum_master_nodes: 2`
- **Attributs pour le redémarrage**
Exemple attendre le démarrage de huit nœuds puis 5 minutes ou les 10 nœuds avant d'entamer le processus de recovery
`gateway.recover_after_nodes: 8`
`gateway.expected_nodes: 10`
`gateway.recover_after_time: 5m`



Mémoire heap

L'installation par défaut d'ELS est configuré avec 1 GB de heap (mémoire JVM). Ce nombre est bien trop petit

3 façons pour changer la taille de la heap .

- Le fichier ***jvm.options***
- Via la variable d'environnement ***ES_HEAP_SIZE***
export ES_HEAP_SIZE=10g
- Via la commande en ligne
./bin/elasticsearch ***-Xmx=10g -Xms=10g***

2 recommandations standard :

- donner 50% de la mémoire disponible à ELS et laisser l'autre moitié vide. En fait Lucene occupera allègrement l'autre moitié
- Ne pas dépasser 32Go



Swapping

Éviter le swapping à tout prix.

Éventuellement, le désactiver au niveau système

```
sudo swapoff -a
```

Ou au niveau ELS

```
bootstrap.memory_lock: true
```

(Anciennement `bootstrap.mlockall`)



Descripteurs de fichiers

Lucene utilise énormément de fichiers.
ELS énormément de sockets

La plupart des distributions Linux limite le nombre de descripteurs e fichiers à 1024.

Cette valeur doit être augmenté à 64,000



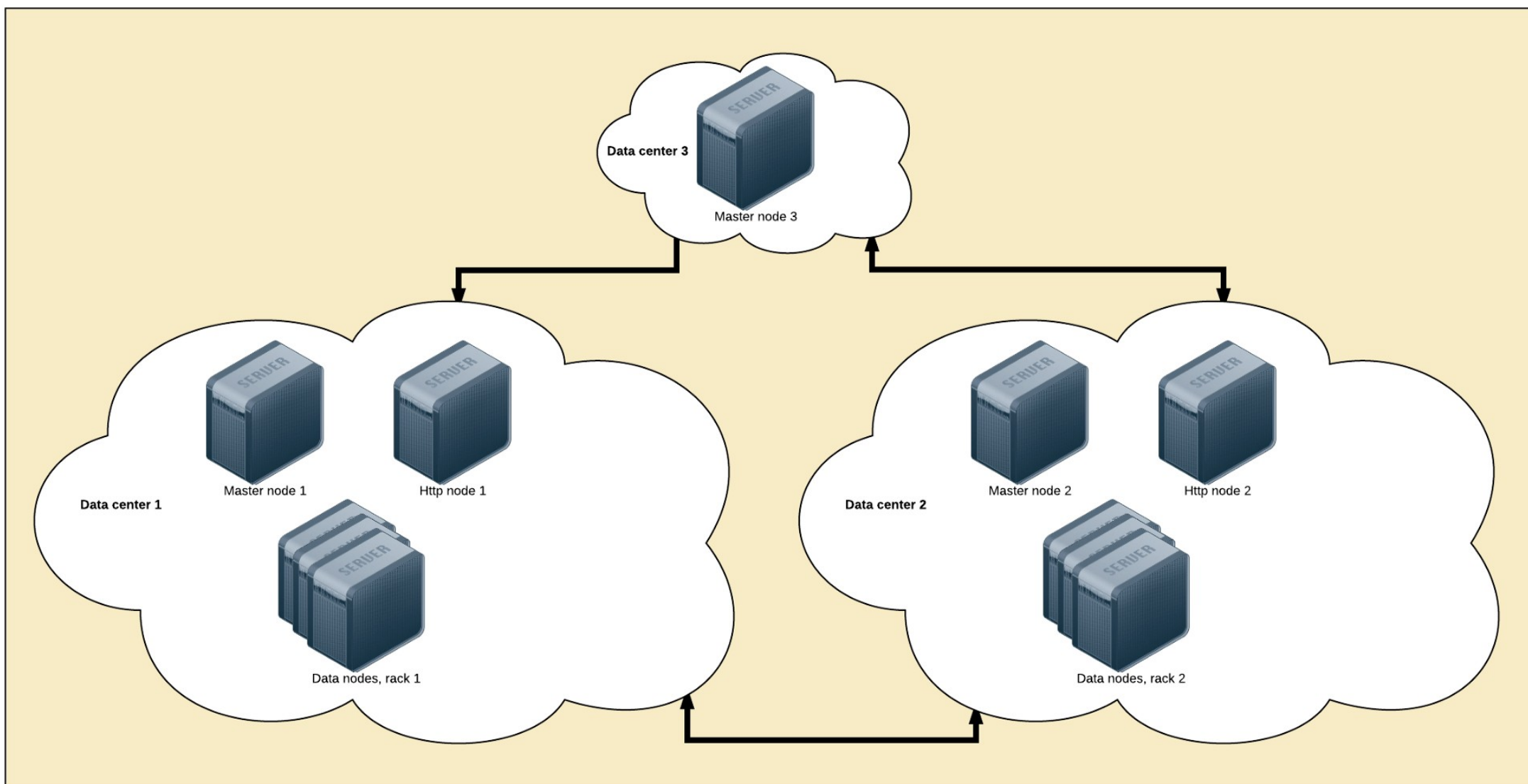
Architecture fault-tolerant

Une architecture tolérante aux pannes typique est de distribuer les nœuds sur différents data center

- au minimum 2 data center principaux contenant les nœuds de données et 1 de backup contenant un éventuel master node
- 3 master node
- 2 nœuds pour exécuter les requêtes http (1 par data center principal)
- Des nœuds de données distribués sur les 2 data center principaux



Architecture fault-tolerant





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



Dimensionnement du nombre de shards

Le nombre de shards est défini lors de la création de l'index. (Par défaut : 5)

Seulement la charge réelle permet de trouver la bonne valeur pour le nombre de shards.

Redimensionner un index en production nécessite une réindexation et éventuellement l'utilisation d'alias d'index.



Avantages pour de nombreux shards

Disposer de beaucoup de shards sur de gros indices et de gros cluster (Plus de 20 nœuds de données) apporte certains avantages :

- Meilleures allocations entre les nœuds
- De petits shards sur beaucoup de nœuds rend le processus de recovery plus rapide (Perte d'un nœud de données ou arrêt du serveur).
- Cela peut régler des problèmes mémoire, lorsque l'on exécute de grosses requêtes.
- Les gros shards rendent les processus d'optimisation de Lucene plus difficile. Lors d'une fusion de segments Lucene, il faut avoir 2 fois la taille du shard comme espace libre.

Par contre, avoir de nombreux shards peut surcharger le master et le cluster devient alors très instable



Recommandations

Repère :

- Des shards de 10GB semblent offrir un bon compromis entre la vitesse d'allocation, et la gestion du cluster .

=> Pour une moyenne de 2GB pour 1 million de documents :

- De 0 à 4 millions de documents par index: 1 shard.
- De 4 à 5 million documents par index: 2 shards
- > 5 millions documents : 1 shards par 5 millions.



Exemple de script de resizing

```
#!/bin/bash
for index in $(list of indexes); do
  documents=$(curl -XGET http://cluster:9200/${index}/_count 2>/dev/null | cut -f 2 -d : | cut -f 1 -d ',,')
  # Dimensionnement du nombre de shard en fonction du nbre de documents
  if [ $counter -lt 4000000 ]; then
    shards=1
  elif [ $counter -lt 5000000 ]; then
    shards=2
  else
    shards=$(( $counter / 5000000 + 1 ))
  fi

  new_version=$(( $(echo ${index} | cut -f 1 -d _ ) + 1 ))
  index_name=$(echo ${index} | cut -f 2 -d _)

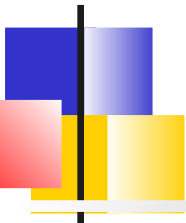
  curl -XPUT http://cluster:9200/${new_version}${index_name} -d '{
    "number_of_shards" : "${shards}"
  }'
  curl -XPOST http://cluster:9200/_reindex -d '{
    "source": {
      "index": "'${index}'"
    },
    "dest": {
      "index": "'${new_version}${index_name}'"
    }
  }'
done
```



Débit de stockage

ELS a 2 propriétés qui protègent contre des situation d'étranglement lors d'écriture

- ***indices.store.throttle.max_bytes_per_sec*** : limite le débit d'écriture. Par défaut il est de 10mb/s ; ce qui est peu. Elle peut éventuellement être augmentée :
`indices.store.throttle.max_bytes_per_sec: 2g`
- ***indices.store.throttle.type*** protège de trop nombreuses opérations de fusion. Elle peut être désactivée si on utilise l'API `_bulk`
`indices.store.throttle.type: "none"`



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

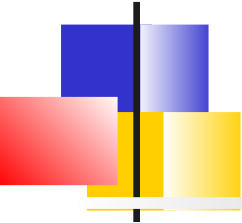
Une stratégie de cycle de vie consiste à spécifier :

- 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



Exemple

```
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"  
} }
```



Backup

Pour sauvegarder un cluster, l'API snapshot API peut être utilisé

Cela prend l'état courant du cluster et ses données et le stocke dans un dépôt partagé

Le premier snapshot est intégral, les autres sauvegardent les deltas

Les dépôts peuvent être de différents types

- Répertoire partagé (NAS par exemple)
- Amazon S3
- HDFS (Hadoop Distributed File System)
- Azure Cloud



Usage simple

```
PUT _snapshot/my_backup
```

```
{
```

```
"type": "fs",
```

```
"settings": {
```

```
  "location": "/mount/backups/my_backup"
```

```
} }
```

Ensuite

```
PUT _snapshot/my_backup/snapshot_1
```



Restauration

POST _snapshot/my_backup/snapshot_1/_restore

Le comportement par défaut consiste à restaurer tous les index existant dans le snapshot

Il est également possible de spécifier les index que l'on veut restaurer

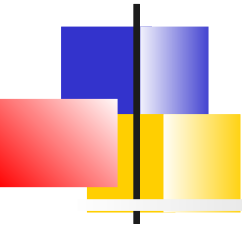


Security

<https://www.elastic.co/guide/en/elasticsearch/reference/current/security-getting-started.html>



Annexes



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)



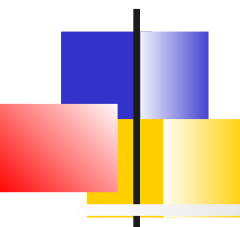
Réponses

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



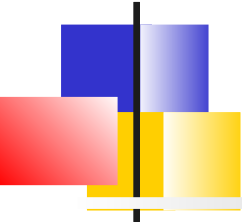
Sniffer

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);
```



Scripts



Scripting

Possibilité d'utiliser des :

- Inline Scripts
- Stored Scripts
- File Scripts

Langage Groovy (deprecated), Painless (expérimental), Natif Java via des plugins

Les valeurs du documents, le score peuvent être accédées via les scripts



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" }
        }
      }
    }
  }
}
```