
PLB
FORMATION INFORMATIQUE ET MANAGEMENT

Elastic Stack

David THIBAU – 2016

david.thibau@gmail.com

Agenda

- **Introduction**
  - L'offre ELK et ses cas d'usage
  - Les composants de la pile
  - Installation
- **ElasticSearch**
  - Concepts, architecture en cluster
  - Indexation API
  - Mapping et Analyseurs
  - Search API
  - Recherche avancée, agrégation
- **Logstash**
  - Concepts cœurs
  - Beats
  - Configuration
- **Kibana**
  - Présentation
  - Discover
  - Visualisations et tableaux de bord
  - Consoles, Management et plugins
- **Vers la production**
  - Architctures Logstash
  - Monitoring API Logstash
  - Replica et Shards
  - Monitoring API ElasticSearch
  - Points de surveillance
  - Exploitation

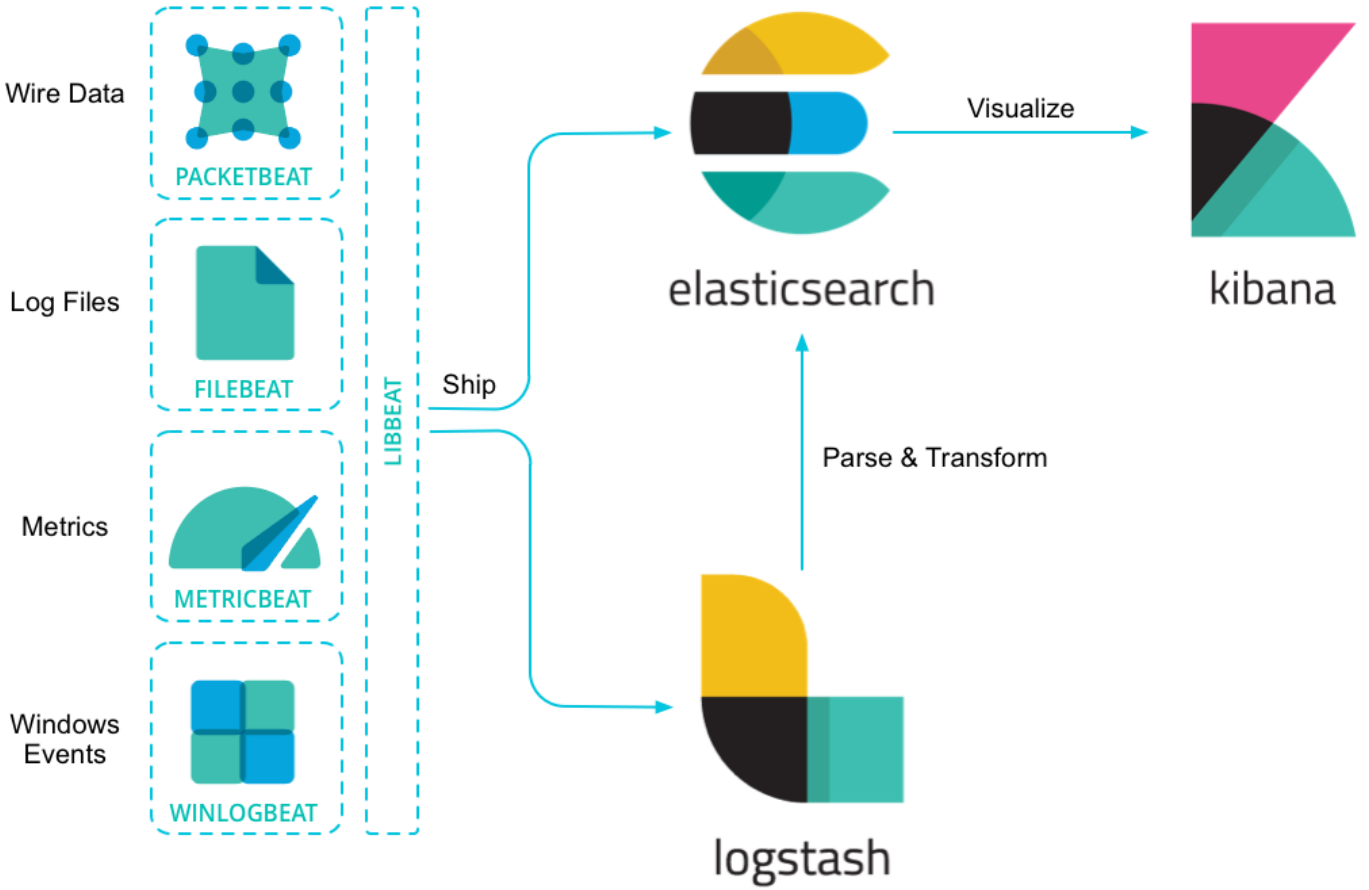
Introduction

- **Elastic Stack** (avant ELK) est un groupe d'outils facilitant l'analyse et la visualisation de données volumineuses
- Ces outils libres sont développés par la même structure, la société *Elastic*, qui encadre le développement communautaire et propose des services complémentaires (support, formation, intégration et hébergement cloud)

La pile

- Elastic Stack est composé des outils suivants :
  - ***Elasticsearch*** : Base documentaire NoSQL basée sur le moteur de recherche Lucene
  - ***Logstash*** : Outil de traitement des traces qui exécutent différentes transformations, et exporte les données vers des destinations diverses dont les index ElasticSearch
  - ***Beats*** : Agents spécialisés permettant de fournir les données à logstash
  - ***Kibana*** est une application Angular permettant de visualiser les données d'Elasticsearch

Architecture



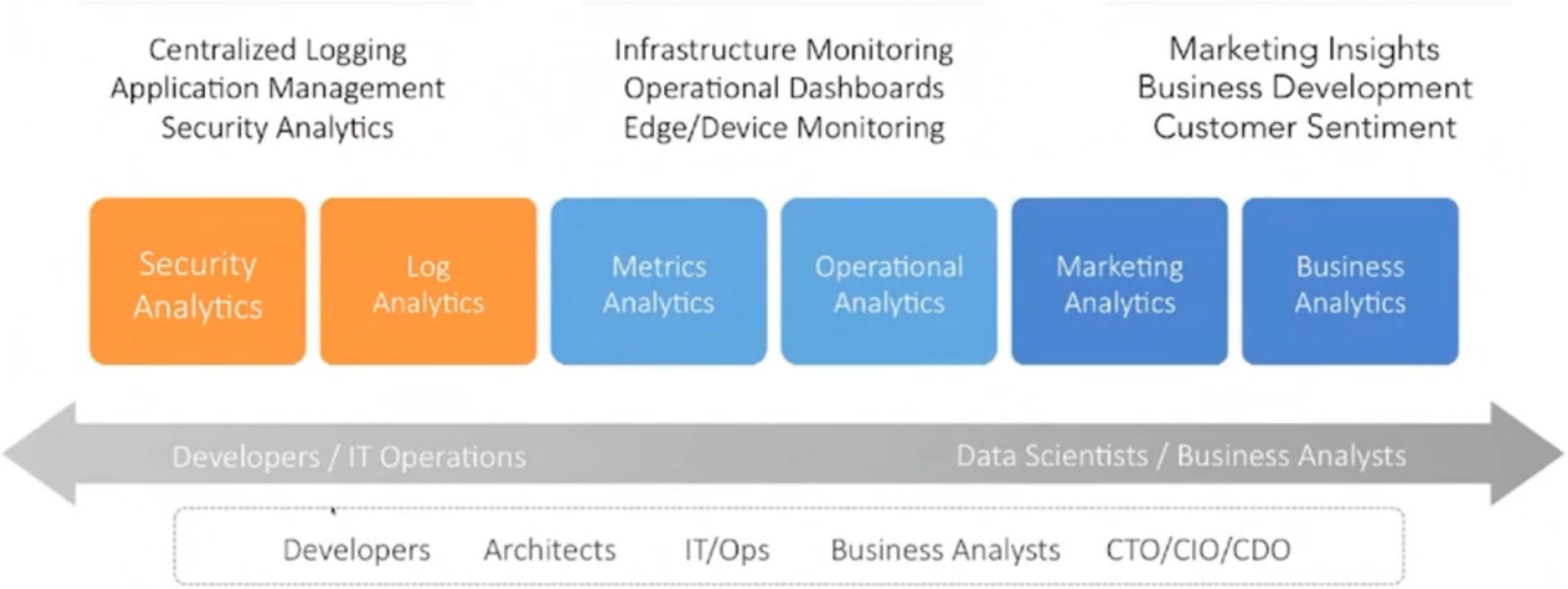
Autres outils

- D'autres outils connexes peuvent être ajoutés :
  - ***X-Pack*** : Fonctionnalités d'entreprise à la pile (Sécurité, monitoring, Alerte)
  - ***Apache Hadoop*** : Stockage BigData
  - ***Elastic Cloud*** : ELK As A Service

Cas d'usage

- La pile ELK est surtout utilisée pour l'analyse des traces ou fichiers journaux.
  L'analyse permet la surveillance d'infrastructure et d'application du point de vue de la performance, de la sécurité, de la conformité
- La pile a également d'autres cas d'usage
  - BI : Elle permet d'avoir une vue métier à 360°
  - Solution de recherche pour les applications webs ou mobiles
  - Comportement utilisateur, fréquentation, segmentation marketing
  - Gestion de risque métier et détection de fraude
  - L'Internet des objets connectés, (capteurs de santé, …)
  - Big Data

Elastic Use cases



Raisons du succès

- ELK est une plate-forme simple robuste et opensource. Ses principales atouts sont
  - Accès au données en temps-réel
  - Scalable en ressources (CPU, RAM) et en volume (Disque)
  - API REST
  - Intégration aisée avec langages et framework. Nombreux clients, Spring Boot, ...
  - Documentation disponible, importante communauté
  - Netflix, Facebook, Microsoft, LinkedIn, SalesForce et Cisco l'ont choisi !

Importance de l'analyse des traces

- Les architectures Microservices, le Cloud augmentent les probabilités qu'un élément de l'architecture dysfonctionne

  => L'ensemble de l'infrastructure doit continuellement être surveillé afin que les problèmes soient détectés au plus tôt
- Le volume des traces étant énorme (BigData). Des outils de recherche, d'agrégation et d'analyse sont nécessaires.
- Ces outils sont aussi bien à destination de l'exploitation technique que métier

Déploiement

- ELK peut être installé dans un entreprise
  - Cependant, l'exploitation de plate-forme peut être consommatrice de ressources surtout si celle-ci a tendance à scaler continuellement, a prendre en compte de nouveaux cas d'usage
- L'autre option est de choisir un fournisseur ELK as a Service. Elastic ou autres

Installation

Versions

- Depuis la 5.0, les différents produits de ELK ont la même numérotation en terme de versions
- Tous les produits font partie du même processus de release, ainsi lorsqu'un produit est releasé tous les produits de la pile sont releasés.
- Il faut donc maintenant juste s'assurer que tous les produits installés ont la même version.

Ordre d'installation

Il est recommandé d'installer la pile en respectant l'ordre suivant :

1. Elastic Search
   Éventuellement X-Pack pour ELS
2. Kibana
   Éventuellement X-Pack pour Kibana
3. Logstah
4. Beats
5. Éventuellement ElasticSearch Hadoop

Elastic Search

Concepts, cluster
Indexation API
Mapping et Analyseurs
Search API
Recherches avancées
Agrégations et géo-localisation

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

Noeud

- 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 (personnage de Marvel par défaut)
- 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 comme la création d'index, l'ajout de nœud dans le cluster
- Pour toutes les opérations sur les documents, chaque nœud du cluster est **interchangeable** et un client peut s'adresser à n'importe quel nœud du cluster pour effectuer une recherche ou indexer des données

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

Document

- Un **document** est l'unité basique d'information qui peut être indexée.
  - Le document est un ensemble de champs (clé/valeur) exprimé avec le format JSON
- A l'intérieur d'un index/type, on peut stocker autant de documents que l'on veut
- Un document d'un index doit être assigné à un type

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)
  - Il permet 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 alloue 5 shards primaires et une réplique

Installation

- Dans le monde Linux, la distribution d'ELS est disponible sous forme
  - D'archive tar.gz, gzip
  - Packages *deb* ou *rpm* téléchargeable à partir des dépôts d'ELS
  - Image docker
- Elastic fournit également des configurations pour *Puppet* et *Chef*
- Sur Windows, il faut télécharger un zip et utiliser *elasticsearch-service.bat* pour mettre ELS en service

Configuration

- Il existe 2 fichiers principaux de configuration :
  - ***elasticsearch.yml*** : Propriétés propre à ELS
  - ***log4j2.properties*** : Verbosité et support de traces

Par défaut, ElasticSearch est configuré pour voir 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***

Elasticsearch.yml

- 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
```

- Les variables d'environnement sont accessibles *${ENV_VAR}*
- Certaines propriétés peuvent être demandées au démarrage *${prompt.text}* , *${prompt.secret}*
- Les valeurs par défaut peuvent être positionnés par la commande en ligne. Elle s'applique si la valeur n'est pas définie dans *elasticsearch.yml*

Bootstrap check

- Au démarrage ELS effectue des vérifications sur l'environnement . Si ces vérifications échouent :
  - En mode développement, des warning sont affichées dans les logs
  - En mode production (écoute sur une adresse publique), ELS ne démarre pas
- Ces vérifications concernent :
  - Dimensionnement de la heap pour éviter les redimensionnements et le swap
  - Limite sur les nombre de descripteurs de fichiers très élevée (65,536)
  - Autoriser 2048 threads
  - Taille et zones de la mémoire virtuelle (pour le code natif Lucene)
  - Le type de JVM (interdit les JVM client)
  - Le garbage collector (interdit la collecte série), la collecte Garbage First si la JVM est trop vieille
  - Filtre sur les appels système
  - Vérification sur le comportement lors d'erreur JVM ou de *OutOfMemory*

Atelier

- Démarrage cluster ELS, monitoring simple

Indexation API

Introduction

- ElasticSearch est une base documentaire distribuée.
- Il est capable de stocker et retrouver des structures de données (sérialisés en documents JSON) en temps réel
- Toutes les données des champs sont indexés par défaut afin de permettre des recherches rapides

Structure de données

- Un document est donc une **structure de données**.
  - Il contient des champs et chaque champ a une ou plusieurs valeurs (tableau)
- Un champ peut également être une structure de données. (Imbrication)
- Le format utilisé par ELS est le format JSON

Exemple

```
{
  "name":  "John Smith",
  "age": 42,
  "confirmed": true,
  "join_date": "2014-06-01",
  "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
  - ***_id*** : L'identifiant unique

Indexation et id document

- Un document est identifié par ses méta-données _*index* , _*type* et _*id*.
- Lors de l'indexation (insertion dans la base Elastic), il est possible de fournir l'ID ou de laisser ELS le générer

Exemple avec Id

```
POST /{index}/{type}/{id}
{
  "field": "value",
  ...
}
….
{
  "_index": {index},
  "_type": {type},
  "_id": {id},
  "_version": 1,
  "created": true
}
```

Exemple sans Id

```
POST /{index}/{type}
{
  "field": "value",
  ...
}
….
{
  "_index": {index},
  "_type": {type},
  "_id": "wM0OSFhDQXGZAWDf0-drSA",
  "_version": 1,
  "created": true
}
```

Récupération d'un document

- La récupérant d'un document peut s'effectuer en fournissant l'identifiant complet :
- **`GET /{index}/{type}/{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/blog/123?_source=title,text

# Juste la partie source sans les méta-données

GET /website/blog/123/_source

# Vérifier qu'un document existe (Retour 200)

HEAD /website/blog/123
```

Mise à jour d'un document

```
PUT /website/blog/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/blog/123?op_type=create`
- Ou
- `PUT /website/blog/123/_create`
- Si le document existe, ELS répond avec un code retour 409

Suppression d'un document

```
DELETE /website/blog/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. (ELS peut également s'appuyer sur un champ version externe)
- Si ce n'est pas le cas (cela veut dire que 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 être changés seulement remplacées
  - La mise à jour de champs suit les mêmes règles : cela consiste donc à réindexer 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/blog/1/_update
{
  "doc" : {
    "tags" : [ "testing" ],
    "views": 0
  }
}
….
{
"_index" : "website",
"_id" : "1",
"_type" : "blog",
"_version" : 3
}
```

Utilisation de script

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

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

- Les scripts peuvent également être chargés à partir de l'index particulier *.scripts* ou du disque du serveur

Bulk API

- L'API bulk permet d'effectuer plusieurs ordres de mise à jour (création, indexation, mise à jour, suppression) en 1 seul 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 concaténer avec le caractère \n.
  - Chaque ligne (même la dernière) doit se terminer par \n simple ligne
  - 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 is then the default index of request
{ "delete": { "_type": "blog", "_id": "123" }}
{ "create": { "_index": "website2", "_type": "blog", "_id": "123" }}
{ "title": "My first blog post" }
{ "index": {  "_type": "blog" }}
{ "title": "My second blog post" }
{ "update": { "_type": "blog", "_id": "123", "_retry_on_conflict" :
3} }
{ "doc" : {"title" : "My updated blog post"} }
```

Réponse


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

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 OpenSourcce 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 nodes

- Des nœuds de type *ingest* sont utilisés pour pé-traiter des documents avant leur indexation
- Le nœud intercepte les requêtes d'indexation, applique des transformations et passe le document à l'API d'indexation
- Pour effectuer des pré-traitements, il faut définir une pipeline qui est une séquence de processeurs
- Chaque processeur effectue une transformation spécifique.
- Pour utiliser une pipeline ; il suffit de préciser le paramètre pipeline sur une requête bulk ou d'indexation
  `PUT my-index/my-type/my-id?pipeline=my_pipeline_id`
- `{ "foo": "bar"}`

Pipeline

- Une pipeline est donc une définition d'une série de processeurs qui s'exécutent dans le même ordre dans lequel ils sont déclarés
- Un pipeline consiste de 2 champs :
  - Une description
  - Une liste de processeurs

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

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`

Usage

```
PUT _ingest/pipeline/attachment
{
  "description" : "Extract attachment information",
  "processors" : [
    {  "attachment" : { "field" : "data" } }
  ]
}
….
PUT my_index/my_type/my_id?pipeline=attachment
{
  "data":
"e1xydGYxXGFuc2kNCkxvcmVtIGlwc3VtIGRvbG9yIHNpdCBhbWV0
DQpccGFyIH0="
}
```

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
    }
  }
```

Atelier Indexation

- Indexation de document, mise à jour, commandes BULK

Mapping et Analyseurs

Introduction

- Pour comprendre les résultats retournés par ELS, il faut connaître :
  - Le **mapping** : comment les données de chaque champ sont interprétés par ELS
  - L' **analyse** : comment les champs plein-texte sont analysés afin qu'il soit recherchables
  - Le **DSL** : le langage de requête utilisé par ELS

Valeur exacte ou full-text

- Les données stockées par ELS peuvent être de deux types :
  - Les **valeurs exactes**, des critères d'égalité stricte sont appliqués
    `Foo != foo`
    Ils concernent en général les données de type date, numériques mais également String
  - Les **données full-text** référencent des données texte, habituellement en langage naturel, les critères de recherche sont alors complètement différents

Mapping

- ELS est capable de générer dynamiquement le mapping
  - Il devine alors le type des champs. (full-text ou autre)
- Le mapping d'un type de données dans un index est accessible par :
- `GET /gb/_mapping/tweet`

Réponse _*mapping*

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

Index inversé

- ELS utilise une structure de donnée nommée **index inversé** afin d'accélérer les recherches full-text
- Cela consiste en une liste de mots unique où chaque mot est associé aux documents dans lequel il apparaît.

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

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.

Constitution des analyseurs

- ELS utilise 3 concepts pour traiter le texte des documents :
  - Les ***analyseurs*** de champs (utilisés à l'indexation et lors de la recherche) 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 définis

- 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 notre index est en une seule langue

Phases d'analyse

L'analyseur est utilisé lors de l'indexation et la recherche

- Indexation : Le flux de tokens résultants est ajouté à l'index et définit l'ensemble des termes (incluant la position, la taille, etc.) pour le champ
- Recherche : Les valeurs de la recherche sont analysées et transformées en un flux de token qui sont comparés à ceux de l'index

En général, on utilise le même analyseur pour l'indexation et la recherche pour un champ donné

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

Définition mapping

- Chaque document d'un index a un type. Chaque type a son propre mapping, ou définition de schéma.
- Un mapping définit les champs d'un type de document
  - Le type de donnée
  - Comment ELS traite le champ
  - Les méta-données associées

Types simples supportés

- ELS supporte :
  - Les chaînes de caractères : *string*
  - Les entiers : *byte* , *short* , *integer* , *long*
  - Les nombres à virgule : *float* , *double*
  - Les booléens : *boolean*
  - Les dates : *date*

Pour voir le mapping d'un type donné :

```
GET /gb/_mapping/tweet
```

Mapping personnalisé

- Un mapping personnalisé permet (entre autres) 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
  - Spécifier des formats de dates personnalisés

Champs full-text

- Les champs *string* sont considérés par défaut comme full-text et associés à l'analyseur standard.
- Cependant, 2 attributs de mapping peuvent être précisés
  - ***index*** pouvant prendre 3 valeurs :
    - ***analyzed*** : Par défaut
    - ***not_analyzed*** : Valeur exacte
    - ***no*** : Ne pas indexer ce champ
  - ***analyzer*** spécifie l'analyseur à utiliser
    - Utiliser un mot pré-défini (*standard*, *whitespace*, *french*)
    - Définir ses propres analyseurs

Mise à jour du mapping

- On peut spécifier le mapping :
  - à la création d'un index
  - À l'ajout d'un nouveau type
  - Ou mettre à jour un type existant en ajoutant un nouveau champ.
    => On ne peut pas modifier un champ déjà indexé
- Le point d'entrée de l'API est ***_mapping***

Exemple (création)

```
PUT /gb
{
  "mappings": {
     "tweet" : {
      "properties" : {
        "tweet" : {
          "type" :  "string",
          "analyzer": "english"
        },
        "date" : {
          "type" : "date"
        },
        "name" : {
          "type" : "string"
        },
         "user_id" : {
          "type" : "long"
        }
} } } }
```

Exemple (Ajout)

```
PUT /gb/_mapping/tweet
{
  "properties" : {
    "tag" : {
      "type" : "string",
      "index": "not_analyzed"
  }
}
}
```

Types complexes

- En plus des types simples, JSON et ELS supporte les valeurs *null*, les tableaux et les objets
  - **Tableau** : 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** sont des structures de données embarquées dans un champ

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" },
           "gender": { "type": "string" },
           "age": { "type": "long"},
           "name": { "type": "object",
             "properties":  {
               "full": { "type": "string" },
               "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.gender": [male],
"user.age": [26],
"user.name.full": [john, smith],
"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"}
```

- Cela donne :

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

- => Lien entre age et nom perdu !
  =>> Nested objects

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 (*strings*, *numbers*, *Booleans*, *Dates*) accepte un paramètre ***fields*** permettant de définir des sous-champs ayant une autre stratégie d'indexation

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

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

Atelier

- Visualisation de Mapping, définition de son propre mapping

Search API

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

Types d'API

- Il y a 2 types d'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
- 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 faire tomber le cluster.
- En production, on interdit généralement ce type d'API au profit de DSL

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

Introduction DSL

- Les recherches avec un corps de requête offrent plus de fonctionnalités qu'une recherche simple.
- En particulier, elles permettent :
  - De mettre en surbrillance des parties du résultat
  - 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

Requête DSL

- Le langage DSL permet d'exposer toute la puissance du moteur Lucene avec une interface JSON
- 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: {
    ARGUMENT: VALUE,
    ARGUMENT: VALUE,...
  }
}
```

Exemple :

```
GET /_search
{
  "query": {
    "match": { "tweet": "elasticsearch" }
  }
}
```

Combiner des clauses

- Une clause (bloc pouvant être combinés dans une requête) peut être de 2 types :
  - Clause **feuille** : (ex : *match*) utilisée pour comparer un (ou plusieurs) champs à la chaîne de requête
  - Clause **composée** : permettant combiner d'autres clauses (ex : *bool*)

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

Ex : Combinaison de combinaison

```
{
"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 deux types de requête
  - Les **filtres** sont utilisés pour les champs contenant des valeurs exactes. Leur résultat est de type booléen. 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

Performance

- La sortie de la plupart des filtre 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 e en plus calculer le score de pertinence.
- => Les recherches sont donc plus lourdes que les filtres et ne sont pas cachable
- =>L'objectif des filtres est de réduire le nombre de documents devant être examinés par la recherche

Filtres disponibles

- ***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" }}
```
- ***bool*** : Permet de combiner des clauses avec :
  - ***must*** équivalent à ET
  - ***must_not*** équivalent à NOT
  - ***should*** équivalent à OU

Recherches

- 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 pertinent, ils reçoivent un _score de 1
- `{ "match_all": {}}`
- La recherche ***match*** est la recherche standard pour effectuer une recheche exacte ou full texte 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" }}`

Recherches (2)

- La requête ***multi_match*** permet d'exécuter la même requête sur plusieurs champs :
- `{"multi_match": { "query": "full text search", "fields": [ "title", "body" ] } }`
- La recherche ***bool***, (comme le filtre) est utilisé pour combiner plusieurs clauses de recherche. Il combine également le score de chaque clause *must* ou *should* qui match.Elle accepte les paramètres suivants :
  - ***must*** : Clauses qui doivent matcher afin que le document soit inclus
  - ***must_not*** : Clauses qui ne doivent pas matcher afin que le document soit inclus
  - ***should*** : Si la clause match, elle augmente le score, sinon elle n'a pas d'effet
-

Combiner requêtes et filtres

- Les recherches composées peuvent combiner des recherches et des filtres
- Il est très utile d'appliquer un filtre à une recherche (limite les documents à évaluer) ou d’utiliser une recherche comme filtre (moins souvent)

- Il y a des clauses de recherche dédiés qui permettent d'encapsuler un filtre dans une recherche (et vice-versa)
- L'utilisation de ces clauses permettent souvent des gains de performance

Filtrer une recherche

```
GET _search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "text": "quick brown fox"
        }
      },
      "filter": {
        "term": {
          "status": "published"
        }
      }
    }
  }
}
```

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.QueryParsingException:
[gb] No query registered for [tweet]"
} ]
}
```

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

Réponse


```
"_explanation": {
  "description": "weight(tweet:honeymoon 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,
    } ]
} ] }
```

Recherche avancée

Recherche simple mot

```
GET /my_index/my_type/_search
{ "query": { "match": { "title": "QUICK!" } } }
```

- ELS exécute les étapes suivantes pour traiter cette requête :
  - 1. Vérification du type de champ . Le champ *title* est un champ full-text (analyzed) ce qui signifie que la requête doit être analysée
  - 2. Analyse : La chaîne ne recherche « QUICK! » Est transformé en « quick » par l'analyseur standard. Comme il n'y a qu'un seul terme, ELS peut exécuter la requête comme une recherche de type *term*
  - 3. Récupération des documents.
  - 4. Affectation du score pour chaque document

Recherche multi-mots

```
GET /my_index/my_type/_search
{ "query": {"match": {"title": "BROWN DOG!"} } }
```

- Lors de la recherche multi-mots, ELS doit combiner des requêtes de termes. Le comportement par défaut est de les combiner avec un OR.
- => La requête précédente renvoie les documents qui contiennent BROWN ou DOG (*should*)
- Il est naturellement possible de changer le comportement par défaut.

Contrôler la combinaison

```
GET /my_index/my_type/_search
{ "query": { "match": {
       "title": { "query": "BROWN DOG!",
"operator": "and" }
} } }
```

- Documents contenant BROWN et DOG

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

- Documents contenant 75 % des mots précisés

Combinaison manuelle

```
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 }}}
} } }
```

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 match une de ses 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

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

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

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

```
PUT /my_index
{ "settings": { "number_of_shards": 1 },
"mappings": { "my_type": {
  "properties": {
  "title": {"type": "string", "analyzer": "english",
    "fields": {
      "std": { "type": "string", "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" ]
} } }
```

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
} } } }
```

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].+" }
}
}
```

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_expansion*s

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

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 le différents préfixes du terme.

Atelier

- Search API

Agrégations et géo-localisation

Agrégations

- 2 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
  - **Metriques** : Les statistiques calculés sur les documents d'un groupe (min, max, avg, ..)
- ELS fournit de nombreux métriques et buckets prédéfinis
- Voir
  https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html

Exemple

```
GET /cars/transactions/_search
{
"aggs" : {
"colors" : {
   "terms" : { "field" : "color" }
} } }
```

- Les agrégations sont exécutées dans le contexte d'un résultat de recherche
- C'est donc juste un autre paramètre de haut-niveau

Réponse

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

Ajout d'un métrique

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

Groupes imbriqués

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

Réponse

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

Intervalle de regroupement et histogramme

```
GET /cars/transactions/_search?search_type=count
{
"aggs":{
  "price":{
    "histogram":{
      "field": "price",
      "interval": 20000
    },
   "aggs":{
     "revenue": {
      "sum": { "field" : "price" }
   }
} } } }
```

Histogramme de date

```
GET /cars/transactions/_search?search_type=count
{
"aggs": {
  "sales": {
    "date_histogram": {
      "field": "sold",
      "interval": "month",
      "format": "yyyy-MM-dd"
    }
} } }
```

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" }

} } }
```

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 trouve
- Il part ensuite de l'ensemble des documents et effectue la même recherche
- 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 allé chez tel commerçant
  - Tous les jeudi soirs, la page untelle est beaucoup plus consultée
  - ...

Géo-localisation

- 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 géolcalisation
  - 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

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

Exemple

```
GET /attractions/restaurant/_search
{
"query": {
"filtered": {
  "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*** : Groupe les documents dans des ronds concentriques autour d'un poin central
  - ***geohash_grid*** : Groupe les documents par cellules (*geohash_cell*, les carrés de google maps) pour affichage sur une map
  - ***geo_bounds*** : retourne les coordonnées d'une zone rectangle qui engloberait tous les geo-points. Utile pour choisir le bon niveau de zoom

Exemple

```
GET /attractions/restaurant/_search
{
"query": { "filtered": { "query": {
  "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 }
          ]
     }
} } }
```

TP

- Recherches avancées et agrégations

Logstash

Concepts cœur
Beats
Configuration

Logstash

- *Logstash* est un outil de **collecte de données temps-réel** capable d'unifier et de normaliser les données provenant de sources différentes
- A l'origine centré sur les fichiers de traces, il peut s'adapter à d'autres types d’événements
- *Logstash* est basé sur la notion de **pipeline** de traitement.
  Il permet de filtrer, mixer et orchestrer différentes entrées
- Extensible via des plugins, il se connecte à de nombreuses sources de données

Types de source

- Traces et métriques : Apache, log4j, syslog, Windows Event, JMX, …
- Web : Requêtes HTTP, Twitter, Hook pour GitHub, JIRA, Polling d'endpoint HTTP
- Support de persistance : JDBC, NoSQL, *MQ
- Capteurs diverses : Mobiles, Domotique, Véhicules connectés, Capteur de santé

Enrichissement de données

- Logstash permet également d'enrichir les données d'entrées.
  - Géo-localisation à partir d'une adresse IP. (lookup vers un service)
  - Encodage/décodage de données
  - Normalisation de dates
  - Requêtes Elastic Search pour l'enrichissement
  - Anonymiser des informations sensibles

Structure répertoires

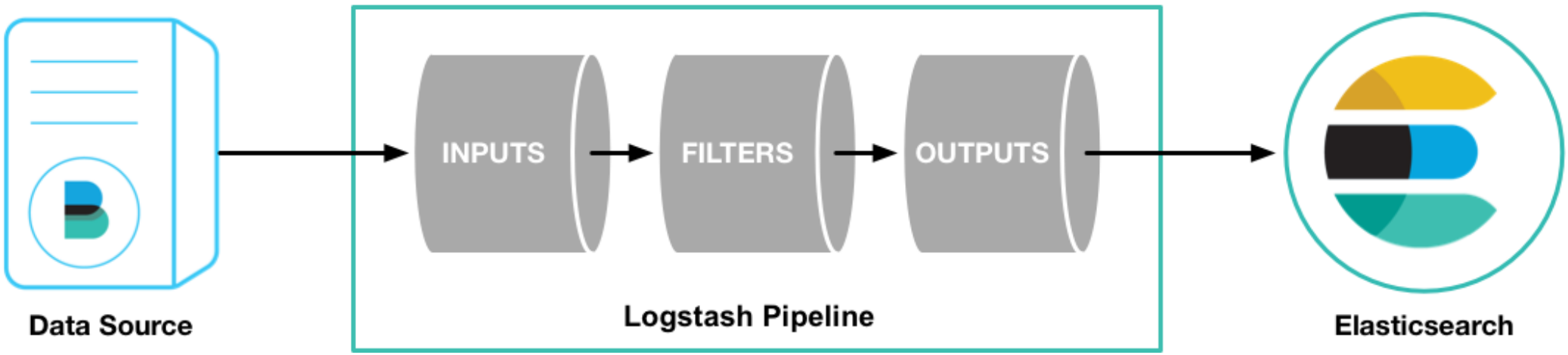
- ***bin*** : Scripts ; en particulier
  - *logstash* le script de démarrage
  - et *logstash-plugin* pour installer des plugins)
- ***settings*** : Fichiers de configuration ; en particulier
  - *logstash.yml*
  - et *jvm.options*
- ***logs*** : Fichiers de traces, configurable via : *path.logs*
- ***plugins*** : plugins locaux (non Ruby-Gem). Chaque plugin est contenu dans un répertoire configurable via *path.plugins*

Plugins

- Il est possible d'ajouter des plugins à l'installation :
  - en utilisant le dépôt *RubyGems.org* Exemple : `bin/logstash-plugin install logstash-output-kafka`
  - En utilisant un chemin local : `bin/logstash-plugin install /path/to/logstash-output-kafka-1.0.0.gem`
- Les plugins peuvent ensuite être mis à jour `bin/logstash-plugin update`

Pipeline Logstash

- Une **pipeline** logstash a au minimum :
  - Un plugin d'entrée pour lire les données
  - Une plugin de sortie pour écrire les données
  - Optionnellement des filtres entre les 2



Configuration

- Les pipelines sont généralement configurées dans des fichiers :

```
# The # character at the beginning of a line indicates a comment. Use
# comments to describe your configuration.
input {
}
# The filter part of this file is commented out to indicate that it is
# optional.
# filter {
#
# }
output {
}
```

- Ou la configuration peut être précisée sur la ligne de commande

```
bin/logstash -e 'input { stdin { } } output { stdout {} }'
```

Exemple

```
input {
    beats { port => "5043" }
}
filter {
    grok {
        match => { "message" => "%{COMBINEDAPACHELOG}"}
    }
    geoip {
        source => "clientip"
    }
}
output {
    elasticsearch {
        hosts => [ "localhost:9200" ]
    }
}
```

Multiple entrées/sorties

- *Logstash* est capable de traiter plusieurs entrées et les acheminer vers plusieurs sorties

```
input {
    twitter {
        consumer_key => "enter_your_consumer_key_here"
        consumer_secret => "enter_your_secret_here"
        keywords => ["cloud"]
        oauth_token => "enter_your_access_token_here"
        oauth_token_secret => "enter_your_access_token_secret_here"
    }
    beats {
        port => "5043"
    }
}
output {
    elasticsearch {
        hosts => ["IP Address 1:port1", "IP Address 2:port2", "IP Address 3"]
    }
    file {
        path => "/path/to/target/file"
    }
}
```

Les entrées

- Les entrées permettent d'insérer des données dans les pipelines *Logstash*.
- Les plus courant sont :
  - ***file*** : Lecture d'une fichier (<=> *tail -f*)
  - ***syslog*** : Écoute les messages *syslog* sur le port 514 et les parse au format RFC3164
  - ***redis*** : Lecture d'un serveur redis. Redis est souvent utilisé pour centraliser les événements provenant de différents installation Logstash
  - ***beats*** : Traite les événements de *FileBeats*

Champs

- Chaque événements traités par logstash est constitué de **champs**
  - Les champs deviendront les champs indexés par ElasticSearch
  - Il est possible de conditionner l'exécution d'un filtre ou d'un plugins de sortie à la présence d'un champ

Les filtres

- Les filtres sont les traitements intermédiaires d'une pipeline Logstash
- Il peuvent s'appliquer selon des conditions, i.e. l’événement remplit certains critères
- Les filtres les plus courants :
  - ***grok***: Parse et structure un texte arbitraire. 120 patterns sont prédéfinis dans *Logstash* correspondant aux formats de logs les plus courant
  - ***mutate***: Effectue des transformations sur les champs de l’événement (Renommage, suppression, remplacement, modification)
  - ***drop***: Supprime un événement Ex : DEBUG
  - ***clone***: Effectue une copie de l’événement en ajoutant ou enlevant des champs
  - ***geoip***: Ajoute des informations géographiques à partir de l’adresse IP

Les sorties

- Les sorties sont la phase finales d'une pipeline
- Les sorties les plus courantes sont :
  - ***elasticsearch***: Envoie les données à ElasticSearch pour indexation
  - ***file***: écrit l'événement sur un fichier
  - ***graphite***: Envoie les données à graphite, un outil open source pour stocker des données de graphiques
    http://graphite.readthedocs.io/en/latest/
  - ***statsd***: Envoie vers le démon *statsd* qui écoute sur UDP, agrège des données et envoie à des services backend pluggable

Codecs

- Les entrées et les sorties peuvent appliquer des ***codecs*** qui permettent d'encoder ou décoder les données sans utiliser de filtres particuliers
- Les codecs permettent de facilement séparer le transport de message du processus de sérialisation
- Les codecs les plus utilisés sont :
  - ***json*** : Encode ou décode les données au format JSON
  - ***multiline*** : fusionne des événements textes multi-ligne en une seule ligne. Ex : Exception Java et leur stacktrace
  - ***rubydebug*** : Utilisée pour le debugging. Permet de voir les champs trouvés par logstash

Modèle d'exécution

- Le modèle d'exécution de Logstash 5 distingue :
  - Les threads d'input
  - Les worker threads exécutant les filtres et les sorties
- Chaque entrée définie dans la configuration s'exécute dans sa propre thread
- Il écrit les événements dans une file synchronisée, i.e les écritures se terminent lorsque la lecture par une worker thread s'effectue
- Les worker thread effectuent des lecture par batch en utilisant un buffer
- Il passe ensuite le batch d’événements dans la pipeline
- La **taille du batch** et le **nombre de workers** sont configurables

Options configurables de la pipeline

- 3 options sont configurable pour une pipeline
  - ***--pipeline.workers*** ou ***-w*** : Le nombre de workers, généralement supérieur au nombre de CPU disponibles. Il faut augmenter ce chiffre afin que les CPU travaillent au maximum
  - ***--pipeline.batch.size*** ou ***-b*** : Nombre maximum d'événements qu'un worker peut collecter. Plus le nombre est grand, plus le débit et plus la mémoire augmente. Si on augmente trop ce chiffre, les performances se dégradent à cause des collectes mémoire
  - ***--pipeline.batch.delay*** : La latence de la pipeline. Si aucun nouveau événement n'arrive sous ce délai. Les événements dans le batch sont traités. Ce paramètre nécessite rarement un tuning

Beats

Introduction

- Les ***beats*** sont des convoyeurs de données
- Ils sont installés comme agent sur les différents serveurs et envoient leur données
  - Directement à Elastic Searrh
  - Ou à des pipeline *logstash* qui peuvent effectuer certaines transformations sur les données

Types de beats

- La distribution de beats fournie par ELK contient :
  - ***Packetbeat*** : Un analyseur de paquets réseau qui convoie des informations sur les transaction entre les différents serveurs applicatifs
  - ***Filebeat*** : Convoie les fichiers de trace des serveurs.
  - ***Metricbeat*** : Un agent de monitoring qui collecte des métriques sur l'OS et les services qui s'y exécutent
  - ***Winlogbeat*** : Événements Windows
- Il est possible de créer ses propre beats. ELK offre la librairie *libbeat* (écrit en Golang) comme support

Beats communautaires

- De nombreux beats communautaires sont également disponibles :
  - ***Apachebeat*** : Statut des serveurs Apache HTTPD
  - ***Elasticbeat*** : Statut d'un cluster ElasticSeearch. Il envoie ses données directement à Elasticsearch.
  - ***Execbeat*** : Exécute des commandes shells périodiquement et envoie la sortie standard vers Logstash ou Elasticsearch.
  - ***hsbeat*** : Métriques de performance de la VM Java HotSpot
  - ***httpbeat*** : Interroge périodiquement des endpoints HTTP(S) et envoie les réponses vers Logstash ou Elasticsearch.
  - ***jmxproxybeat*** : Lit les métriques JMX de Tomcat.
  - ***journalbeat*** : Convoie les logs de systemd/journald sur les systèmes Linux.

Beats communautaires (2)

- ***logstashbeat*** : Collecte les données de l'API de monitoring de Logstash et les index dans *Elasticsearch*.
- ***mysqlbeat*** : Exécute des requêtes SQL sur MySQL et envoie les résultats à *Elasticsearch*.
- ***nagioscheckbeat*** : Vérification Nagios et données de performance
- ***pingbeat*** : Envoie des ping ICMP pings à des cibles et stocke le *round trip time* (RTT) dans *Elasticsearch*.
- ***springbeat*** : Collecte les métriques de performance et de santé d'une application Spring Boot avec le module actuator activé.
- ***twitterbeat*** : Lit des tweets
- ***udpbeat*** : Envoie des traces via UDP
- ***wmibeat*** : Utilise WMI pour récupérer des métriques Windows configurable.

TP

- Installation

Inputs, filtres et outputs Logstash

Introduction

- Le fichier de configuration spécifie les plugins à utiliser et leur configuration
- Il référence les champs des données d'entrée et les traitements qui peuvent y être appliqués
- Des tests peuvent conditionner l'exécution d'un traitement
- Lors de l'exécution de *logstash*, l'option ***-f*** spécifie le fichier de configuration

Types des propriétés

- Les blocs de configuration sont différents selon les plugins, mais les valeurs de configuration appartiennent aux types suivants :
  - Types simples : Booléen, nombre ou chaînes de caractères
    ```
    ssl_enable => true  name => "Hello world"
    ```
  - Table de Hash
    ```
    match => {   "field1" => "value1"
                 "field2" => "value2" }
    ```
  - Taille en octets
    ```
    my_bytes => "10MiB"  # 10485760 bytes
    ```
  - Chaîne aveec gabarits : Uri, password, path
    ```
    my_path => "/tmp/logstash" my_uri => "http://foo:bar@example.net"
    my_password => "password"
    ```
  - Codec (valeur prédéfinie)
    ```
    codec => "json"
    ```
  - Commentaires
    ```
    # this is a comment
    ```

Référence des champs

```
{
  "agent": "Mozilla/5.0 (compatible; MSIE 9.0)",
  "ip": "192.168.24.44",
  "request": "/index.html"
  "response": {
    "status": 200,
    "bytes": 52353
  },
  "ua": {
    "os": "Windows 7"
  }
}
```

- Pour référencer un champ de haut niveau, il suffit de spécifier le champ. Ex : agent. Ou utiliser la notation [agent]
- Pour les champs imbriqués, il faut utiliser la notation [ ] . Ex [ua][os]

Exemples *sprintf*

```
output {
  statsd {
    increment => "apache.%{[response][status]}"
  }
}

output {
  file {
    path => "/var/log/%{type}.%{+yyyy.MM.dd.HH}"
  }
}
```

Configuration conditionnelle

- La configuration conditionnelle s'obtient en utilisant des instructions *if then else*

```
if EXPRESSION {
   ...
} else if EXPRESSION {
   ...
} else {
   ...
}
```

Exemples

```
filter {
  if [action] == "login" {
    mutate { remove_field => "secret" }
  } }

output {
  # Send production errors to pagerduty
  if [loglevel] == "ERROR" and [deployment] == "production" {
    pagerduty {
    ...
    } } }

if [foo] in ["hello", "world", "foo"] {
    mutate { add_tag => "field in list" }
  }
```

L'expression **if [foo]** retourne false si :

- [foo] n'existe pas dans l'évènement,
- [foo] existe mais est false ou null

Le champ *@metadata*

- Il existe un champ spécial : ***@metadata***
- Ce champ ne sera pas écrit sur les sorties par contre il peut être utilisé pendant tout le traitement Logstash

```
input { stdin { } }

filter {
  mutate { add_field => { "show" => "This data will be in the output" } }
  mutate { add_field => { "[@metadata][test]" => "Hello" } }
  mutate { add_field => { "[@metadata][no_show]" => "This data will not be in the output" } }
}

output {
  if [@metadata][test] == "Hello" {
    stdout { codec => rubydebug }
  }
}
```

Variables d'environnement

- une variable environnement peut être référencée par ***${var}*** dans le fichier de configuration
  - Des références à des variables indéfinies provoquent des erreurs
  - Les variables sont sensibles à la casse
  - Une valeur par défaut peut être fournie par ***${var:default value}***.

Exemple complet log Apache

```
input {
  file {
    path => "/tmp/*_log"
  }
}
# Traitements différents en fonction du type de log
filter {
  if [path] =~ "access" {
    mutate { replace => { type => "apache_access" } }
    grok {
      #gabarit connu par grok
      match => { "message" => "%{COMBINEDAPACHELOG}" }
    }
    Date {
      # Normalisation de la date
      match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]
    }
  } else if [path] =~ "error" {
    mutate { replace => { type => "apache_error" } }
  } else {
    mutate { replace => { type => "random_logs" } }
  }
}

output {
  elasticsearch { hosts => ["localhost:9200"] }
  stdout { codec => rubydebug }
}
```

Résultat pour une ligne de log d'accès


```
{
        "message" => "127.0.0.1 - - [11/Dec/2013:00:01:45 -0800] \"GET
/xampp/status.php HTTP/1.1\" 200 3891 \"http://cadenza/xampp/navi.php\" \"Mozilla/5.0
(Macintosh; Intel Mac OS X 10.9; rv:25.0) Gecko/20100101 Firefox/25.0\"",
     "@timestamp" => "2013-12-11T08:01:45.000Z",
       "@version" => "1",
           "host" => "cadenza",
       "clientip" => "127.0.0.1",
          "ident" => "-",
           "auth" => "-",
      "timestamp" => "11/Dec/2013:00:01:45 -0800",
           "verb" => "GET",
        "request" => "/xampp/status.php",
    "httpversion" => "1.1",
       "response" => "200",
          "bytes" => "3891",
       "referrer" => "\"http://cadenza/xampp/navi.php\"",
          "agent" => "\"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:25.0)
Gecko/20100101 Firefox/25.0\""
}
```

Exemple complet syslog

```
input {
  tcp { port => 5000 type => syslog }
  udp { port => 5000 type => syslog } }

filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp} %{SYSLOGHOST:syslog_hostname} %
{DATA:syslog_program}(?:\[%{POSINT:syslog_pid}\])?: %{GREEDYDATA:syslog_message}" }
      add_field => [ "received_at", "%{@timestamp}" ]
      add_field => [ "received_from", "%{host}" ]
    }
    date { match => [ "syslog_timestamp", "MMM  d HH:mm:ss", "MMM dd HH:mm:ss" ] }
  } }

output {
  elasticsearch { hosts => ["localhost:9200"] }
  stdout { codec => rubydebug }
}
```

Rechargement automatique de la configuration

- Si logstash est démarré avec l'option ***--config.reload.automatic***, il est alors capable de recharger dynamiquement sa configuration
  - La configuration est vérifiée toutes les *--config.reload.interval <seconds>* (3s par défaut)
- On peut également activer à posteriori le rechargement automatique par :
  kill -1 <pid>
- La reconfiguration consiste à vérifier la nouvelle config, et si elle est correcte arrêter la pipeline courante et en créer une autre.

Événements multi-lignes

- Lors d'événements multi-lignes, *Logstash* doit savoir quels lignes doivent faire partie de l'événement unique
- Le traitement des multi-lignes est effectué en général en premier dans le pipeline et utilise le filtre ou le codec ***multiline***
- 3 options importantes de configuration :
  - ***pattern :*** Expression régulière. Les lignes correspondantes sont considérées soit comme la suite des lignes précédentes ou le début d'un nouvel événement Il est possible d'utiliser les gabarits de *grok*
  - ***what*** : Qui peut prendre 2 valeurs : *previous* ou *next*
  - ***negate*** : Qui permet d'exprimer la négation

Exemple stack trace Java

```
Exception in thread "main" java.lang.NullPointerException
        at com.example.myproject.Book.getTitle(Book.java:16)
        at com.example.myproject.Author.getBookTitles(Author.java:25)
        at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

- Le filtre suivant indique qui si la ligne démarre avec des espaces, elle est considéré comme la suite de l'événement

```
input {
  stdin {
    codec => multiline {
      pattern => "^\s"
      what => "previous"
    }
  }
}
```

Le filtre *grok*

- ***grok*** est sûrement le filtre le plus utilisé. Il permet de générer différents champs ELS à partir d'une ligne de log.
- Il s'appuie sur des patterns qui sont déposés par la communauté sur GitHub (+120)
- Les patterns s'appuient sur les ***regexp*** et des patterns spécifiques peuvent être déclarés
- Un pattern Grok s'écrit : **`%{SYNTAX:SEMANTIC}`**
  - *SYNTAX* est un pattern associé à une expression régulière
  - *SEMANTIC* est le nom de champ que l'on veut créer

TP

- Logs serveur applicatif Java
- Syslog

Kibana

Introduction
Discover
Visualisations et tableau de bord
Console, Management et plugins

Introduction

- Kibana est une plateforme d'analyse et de visualisation fonctionnant avec ElasticSearch
- Il est capable de rechercher les données stockées dans les index d'ElasticSearch
- Il propose une interface web permettant de créer des tableaux de bord dynamique affichant le résultat des requêtes en temps réel
- Il s'exécute sur Node.js

Dashboard Kibana

Installation

- 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.por***t : défaut localhost:5601
  - ***elasticsearch.url*** : http://localhost:9200
  - ***kibana.index*** : Index d'ElasticSearch utilisé pour stocker les recherches et les tableaux de bord
  - ***kibana.defaulAppId*** : 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

Premier accès

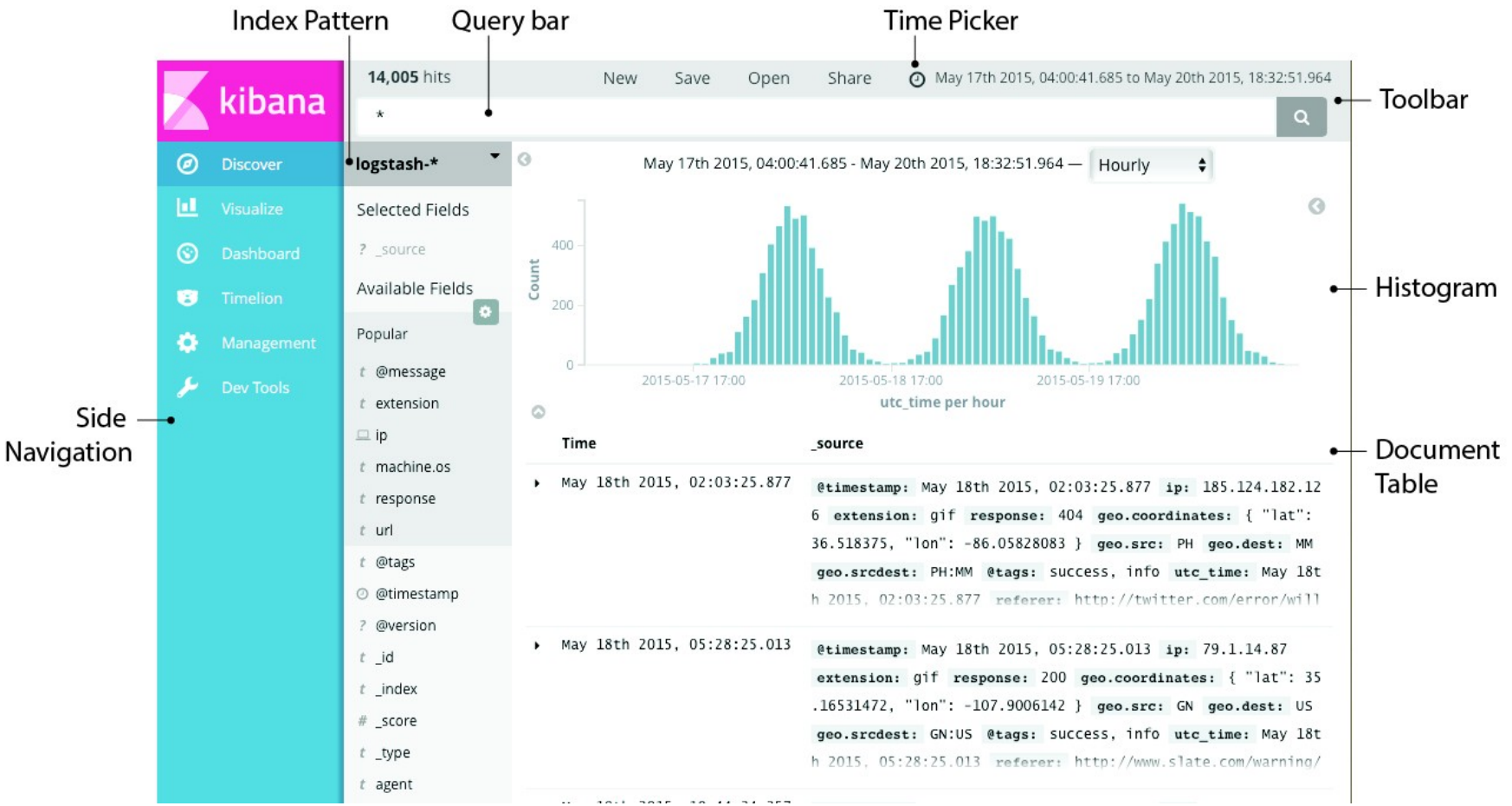
- Lors du premier accès, Kibana demande avec quels index d'ElasticSearch il doit se connecter
  - Les index sont résolus à partir d'un motif (pattern) ; ex : /* ou logstash-*
  - Eventuellement, indiquer le champ timestamp de l'index
- Kibana utilise le dynamic mapping d'ElasticSearch.
  => Si cette fonctionnalité est désactivée :
  - Il faut fournir explicitement le mapping pour la visualisation
  - Il faut autoriser le dynamic mapping pour l'index .kinbana

```
PUT .kibana
{  "index.mapper.dynamic": true }
```

Fonctionnalités

- Kibana propose 3 fonctionnalités principales :
  - ***Discover*** : Permet d'effectuer des recherche ES, de sélectionner les champs retournés et d'accéder à un document
  - ***Visualisation*** : Permet d'agréger les résultats de requêtes et de les afficher sous forme de graphique
  - ***Dashboard*** : Arranger en une seule page plusieurs visualisations
- Il propose également
  - ***Management*** : Gérer ses recherches
  - ***Console*** : Tester des requêtes REST vers ElasticSeearch

Discover


Index Pattern
Query bar
Time Picker
kibana
14,005 hits
New
Save
Open
Share
May 17th 2015, 04:00:41.685 to May 20th 2015, 18:32:51.964
Toolbar
*
Discover
Visualize
Dashboard
Timelion
Management
Dev Tools
logstash-*
Selected Fields
? _source
Available Fields
Popular
t @message
t extension
ip
t machine.os
t response
t url
t @tags
@timestamp
? @version
t _id
t _index
_score
t _type
t agent
May 17th 2015, 04:00:41.685 - May 20th 2015, 18:32:51.964 — Hourly
Count
400
200
0
2015-05-17 17:00
2015-05-18 17:00
2015-05-19 17:00
utc_time per hour
Histogram
Side Navigation
Time
_source
May 18th 2015, 02:03:25.877
@timestamp: May 18th 2015, 02:03:25.877 ip: 185.124.182.126 extension: gif response: 404 geo.coordinates: { "lat": 36.518375, "lon": -86.05828083 } geo.src: PH geo.dest: MM geo.srcdest: PH:MM @tags: success, info utc_time: May 18th 2015, 02:03:25.877 referer: http://twitter.com/error/will
May 18th 2015, 05:28:25.013
@timestamp: May 18th 2015, 05:28:25.013 ip: 79.1.14.87 extension: gif response: 200 geo.coordinates: { "lat": 35.16531472, "lon": -107.9006142 } geo.src: GN geo.dest: US geo.srcdest: GN:US @tags: success, info utc_time: May 18th 2015, 05:28:25.013 referer: http://www.slate.com/warning/
Document Table

Fenêtre temporelle

- Si un champ timestamp existe dans l'index :
  - Le haut de page affiche la distribution des documents sur une période (par défaut 15 mn)
  - La fenêtre de temps peut être changée

Recherche

- Les critères de recherche peuvent être saisis dans la barre de requête. Il peut s'agir de :
  - Un simple texte
  - Une requête Lucene
  - Une requête DSL avec JSON.
- Le résultat met à jour l'ensemble de la page et seuls les 500 premiers documents sont retournés dans l'ordre chronologique inverse
- Des filtres sur les champs peuvent être spécifiés
- La recherche peut être sauvegardée
- L'index pattern peut être modifié
- La recherche peut se rafraîchir automatiquement toutes les X temps

Gestion des filtres

- Plusieurs boutons sont disponibles sur les filtres :
  - Activation désactivation
  - Pin : Le filtre est conservé même lors d'un changement de contexte
  - Toggle : Filtre positif ou négatif
  - Suppression
  - Edition : On peut alors travailler avec la syntaxe DSL et créer des combinaisons de filtre

Visualisation des documents

- La liste affiche par défaut 500 documents (propriété *discover:sampleSize*)
- Il est possible de chosir le critère de tri
- Ajouter/Supprimer des champs
- Des statistiques sont également disponibles

Visualisation et tableau de bord

Introduction

- Les visualisations sont basées sur des recherches ES
- En utilisant des agrégations, on peut créer des graphiques qui montrent des tendances, des pics, ...
- Les visualisations peuvent ensuite être composées en tableau de bord

Types de visualisations

- ***Area*** : Visualise les contributions totales de plusieurs séries
- ***Table de données*** : Affichage en tableau des données agrégées
- ***Line*** : Compare différente séries
- ***Markdown*** : Format libre pour des information ou des instructions
- ***MetricDisplay*** : Une unique valeur.
- ***Pie*** : Camembert .
- ***Tile map*** : Associe le résultat de l'agrégation à un emplacement géographique
- ***Timeseries*** : Adopté aux échelles de temps
- ***Vertical bar*** : Histogramme permettant la comparaison de séries

Étapes de création

- Les étapes de création consiste donc à :
  1. Choisir un type de graphique
  2. Spécifier les critères de recherche ou utiliser une recherche sauvegardée
  3. Choisir le calcul d'agrégation pour l'axe des Y (count, average, sum, min, …)
  4. Choisir le critère de regroupement des données (bucket expression) (Histogramme de date, Intervalle, Termes, Filtres, Significant terms, …)
  5. Définir éventuellement des sous-agrégations

Bucket expressions

- **Histogramme Date** : Un histogramme de date est construit sur un champ entier pour lequel on a précisé une fenêtre temporel
- **Intervalle** : Entier, date ou IPV4
- **Terme** : Permet de présente les n meilleurs (ou plus basse) pertinence ordonné par la valeur agrégée
- **Filtres** : Il est possible d'ajouter des filtres de données à ce niveau
- **Significant Terms** : Agrégation *significant terms*
- **Geohash** : Agrégation sur les coordonnées géographiques (Data Table et carte)

Options

- Lors de plusieurs agrégations en Y, il est possible de spécifier leur mode d'affichage
  - ***Stacked*** : Empile les agrégations les unes sur les autres.
  - ***Overlap*** : Superposition avec transparence
  - ***Wiggle*** : Ombrage
  - ***Percentage*** : Chaque agrégation comme une portion du total
  - ***Silhouette*** : Chaque agrégation comme variance d'une ligne centrale
  - ***Grouped*** : Groupe les résultats horizontalement par les sous-agrégations (Grapique barre) .

Types de cartes

- Plusieurs types de carte sont disponibles :
  - Marqueurs circulaires à l'échelle : Adapte la taille des marqueurs en fonction de la valeur du métrique agrégé
  - Marqueurs circulaires ombragés : Affiche le marqueurs avec différrentes ombre en fonction de la valeur du métrique
  - Cellule Geohash ombragé : Affiche des cellules rectangulaires avec différentes ombres en fonction du métrique.
  - Heatmap : Applique du flou au marqueur circulaire et de l'ombrage en fonction du chevauchement

Tableau de bord

- Un tableau de bord affiche un ensemble de visualisations sauvegardées
- Il est possible de réarranger et retailler les visualisations
- Il est possible de partager les tableaux de bord par un simple lien

TP

- Prise en main Kibana

Console, Management et plugins

Console

- Le plugin Console offre une interface utilisateur permettant d'interagir avec l'API REST d'Elasticsearch.
- Elle 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*)

Management

- L'application management permet la configuration de Kibana et des objets que l'on peut sauvegarder
- Cette partie est pluggable et les plugins peuvent ajouter leur propres écrans de configuration

Fonctionnalités

- Définir les motifs d'index
- Eventuellement le champ comportant le timestamp et le format de la date
- Gérer les champs, le formattage associé
- Créer des champs scriptés (créer et renseigner à la volée). Ces champs peuvent être utilisés dans les visualisations mais pas dans les recherches
- Plein d'options avancées
- Gérer les objets sauvegardés (Recherches, Visualisation et tableaux de bord)

Plugins

- Des fonctionnalités additionnelles peuvent être implémentés par des plugins
- Kibana propose un script permettant leur installation
- Il est également possible de faire une installation manuelle, consistant à dézipper une archive dans le répertoire *plugins*

Quelques plugins

- X-Pack : Sécurité, Monitoring, Alerte. Inclus dans l'offre commerciale
- LogTrail : Interface technique spécialement pour les évènements de travce
- Timelion : Composeur pour les time séries
- Visualisations : Swimlanes, Gauge, Nuage de tags, Graphiques 3D

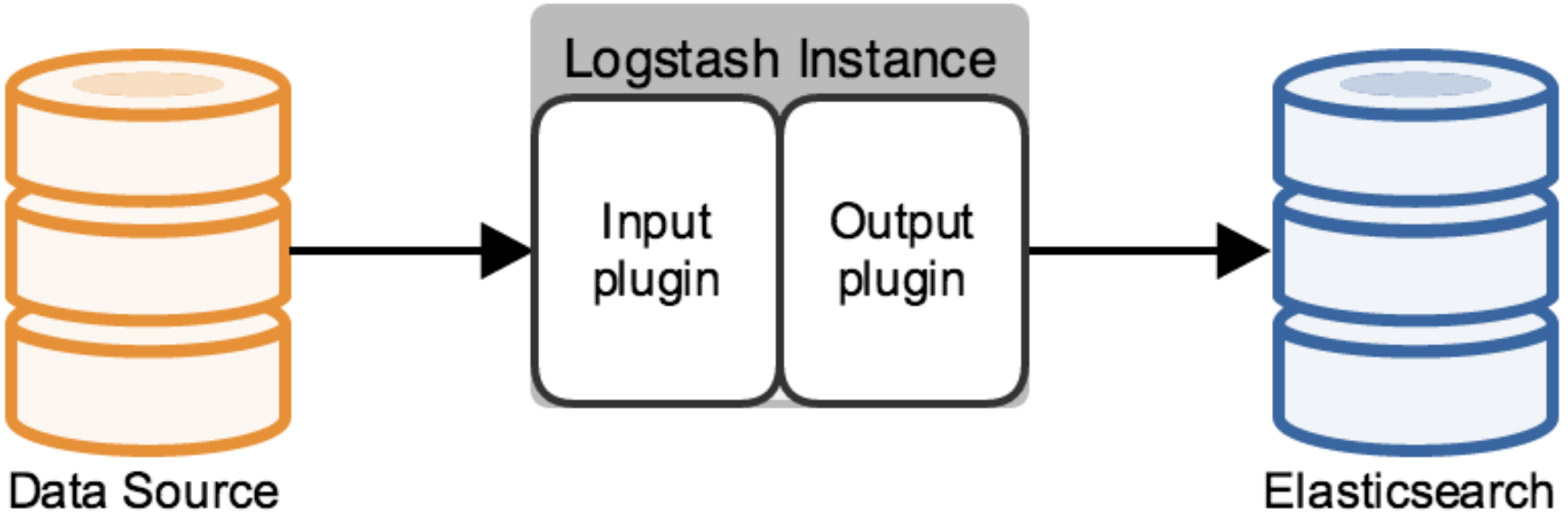
https://github.com/elastic/kibana/wiki/Known-Plugins

Déploiement en production

Architectures Logstash
Monitoring Logstash
Indexation, recherche vs Shards et Replica
Monitoring ES
Point de vérifications
Exploitation

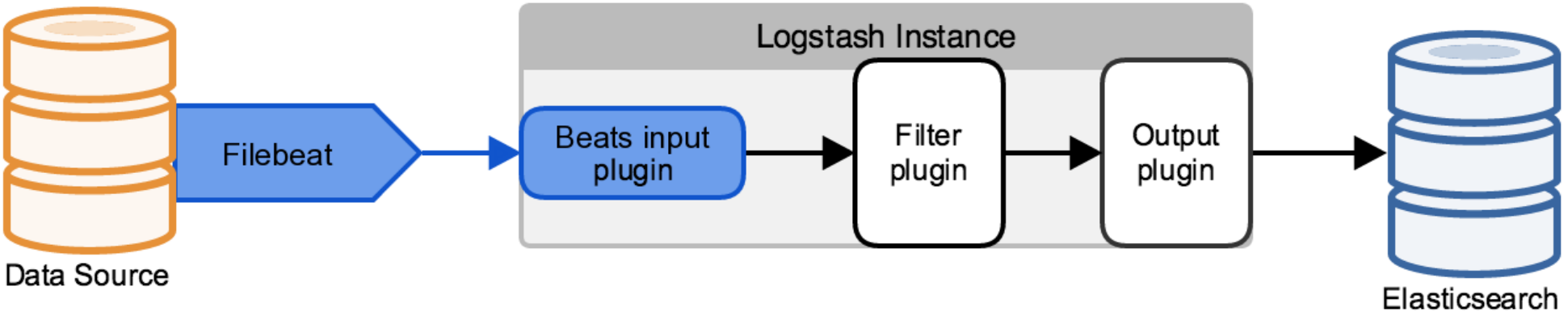
Architectures

Architecture minimale

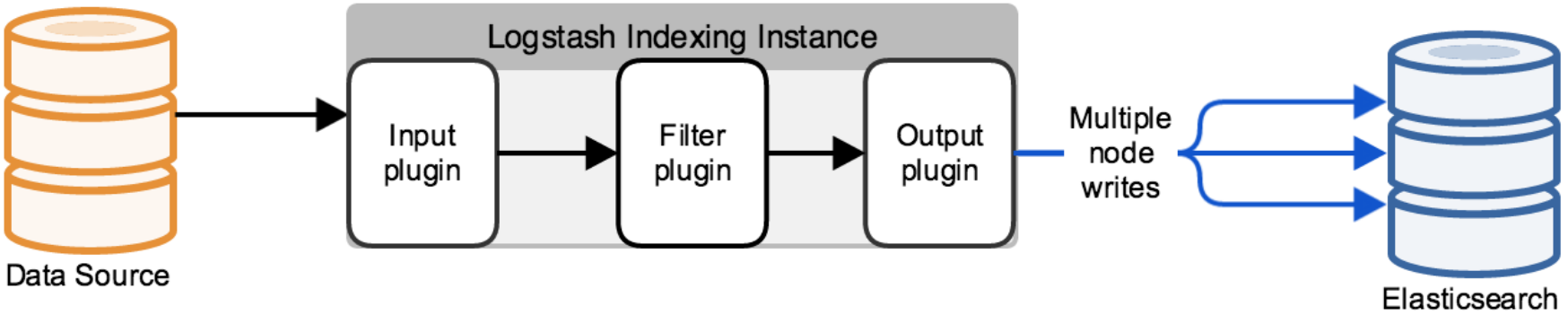


Architecture avec un Beat

- L'utilisation d'un *Beat* permet de déporter du traitement sur la machine source



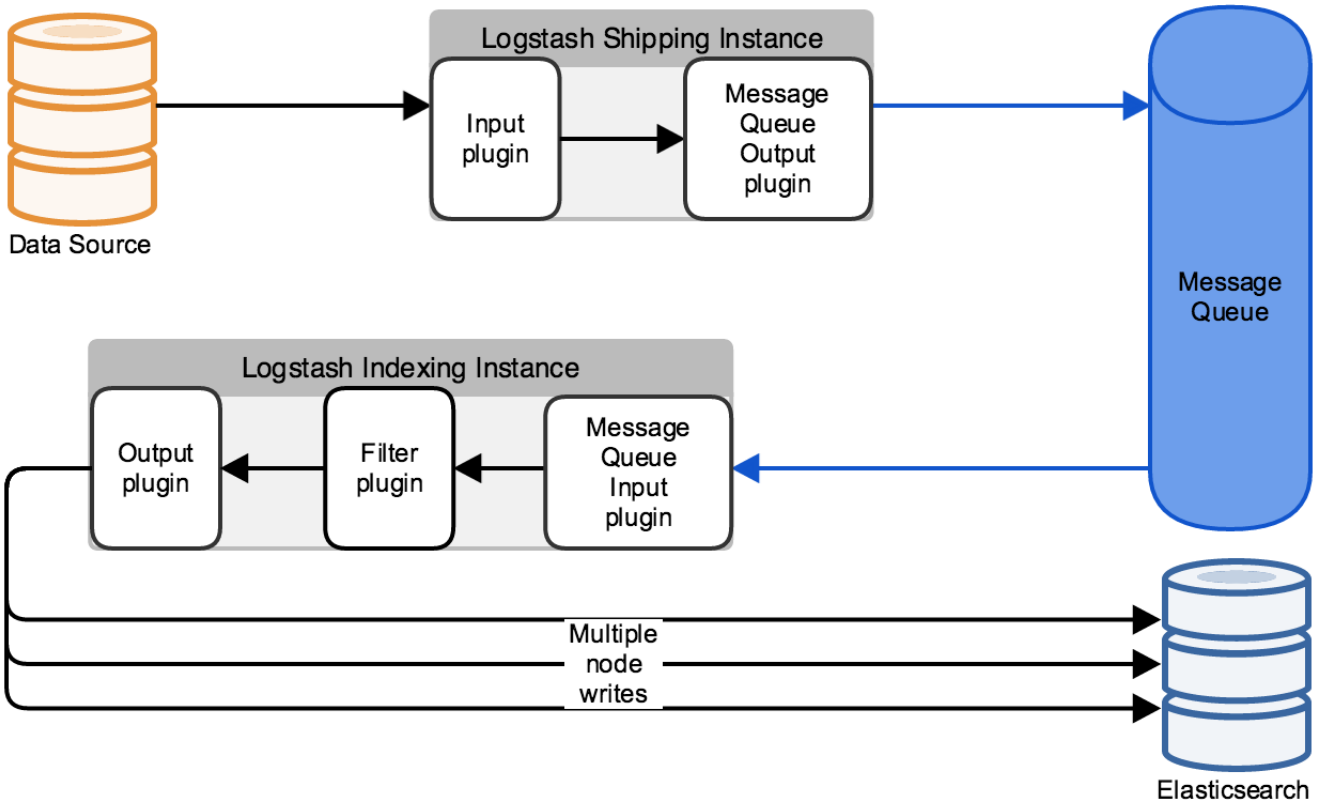
Load balancing sur les data nodes de ElasticSearch



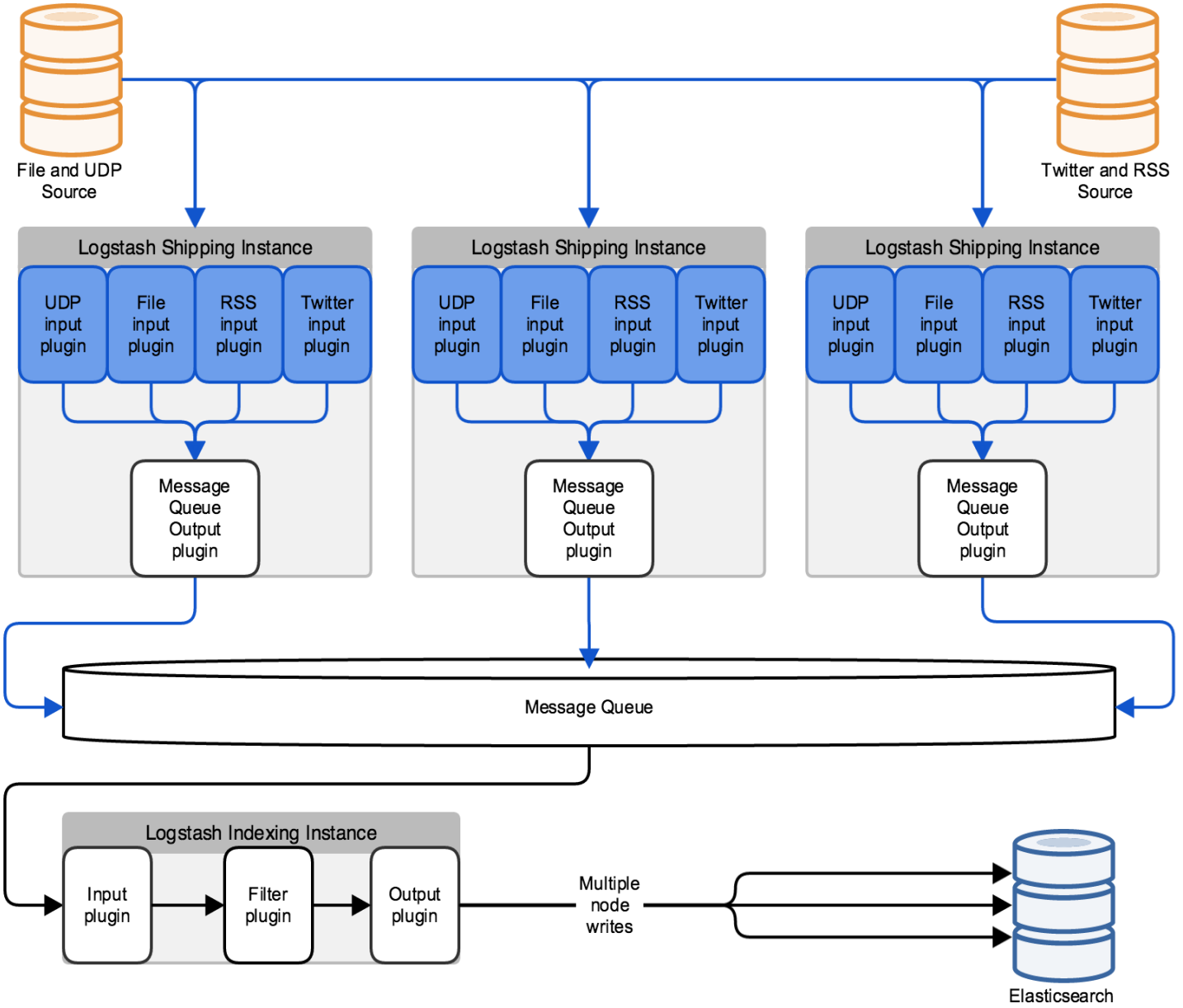
Message broker

- Afin de faire face à des pics de débit, l'architecture peut inclure des message brokers (Redis, Kafka, RabbitMQ). Cela peut soulager le travail d'indexation d'ElasticSearch
- Des instances de logstash écrivent vers une file de message
- D'autres lisent de la file, effectue les traitements et envoient vers ElasticSearch

Message broker



Haute disponibilité via des connections multiples



Architecture en tiers

- Un déploiement Logstash a typiquement un pipeline composé de différents tiers :
  - Le tiers d' entrée consomme les données des sources et est composé d'instance logstash avec les bons plugins d'entrée
  - Le broker de messages sert de buffer et de protection contre des pannes
  - Le tiers de filtrage trait et normalise les données
  - Le tiers d'indexation déplace les données traitées vers ElasticSearch
- Chaque tiers peut être scalés selon les besoins

API de monitoring

Introduction

- Logstash fournit une API REST pour la surveillance
- L'API est divisé en 4 domaines
  - **Node Info** : Informations sur un nœud
  - **Plugins** : Les plugins installés
  - **Node stats** : Métriques sur les nœuds
  - **Hot threads** : Threads avec un gros usage CPU
- Les réponses JSON peuvent être formattés par les paramètres :
  - *pretty=true*
  - *human=true*

Node info

- Sur une pipeline, le nombre de workers, la taille et le délai de batch
  `GET /_node/pipeline`
- Sur l'OS, les versions et les processeurs disponibles
  `GET /_node/os`
- Sur la JVM, processus, Heap et garbage collector
  `GET /_node/jvm`

Plugins info

```
GET /_node/plugins
{
  "total": 91,
  "plugins": [
    {
      "name": "logstash-codec-collectd",
      "version": "3.0.2"
    },
    {
      "name": "logstash-codec-dots",
      "version": "3.0.2"
    },
    .
    .
    .
  ]
```

Node stats

`GET /_node/stats/<types>`

- Soit tous les métriques, soit si limité à un type, types peut être :
  - ***jvm*** : JVM stats, threads, usage mémoire et garbage collectors.
  - ***process*** : processus, descripteurs de fichiers, consommation mémoire et usage CPU
  - ***mem*** : Usage mémoire .
  - ***pipeline*** : Métrique sur la pipeline Logstash

Exemple pipeline


```
{
  "pipeline": {
    "events": { "duration_in_millis": 7863504,"in": 100, "filtered": 100, "out": 100 },
    "plugins": {
      "inputs": [],
      "filters": [
        {
          "id": "grok_20e5cb7f7c9e712ef9750edf94aefb465e3e361b-2",
          "events": { "duration_in_millis": 48, "in": 100, "out": 100 },
          "matches": 100,
          "patterns_per_field": { "message": 1 },
          "name": "grok"
        },
        {
          "id": "geoip_20e5cb7f7c9e712ef9750edf94aefb465e3e361b-3",
          "events": { "duration_in_millis": 141, "in": 100, "out": 100 },
          "name": "geoip"
        }
      ],
      "outputs": [
        {
          "id": "20e5cb7f7c9e712ef9750edf94aefb465e3e361b-4",
          "events": { "in": 100, "out": 100 },
          "name": "elasticsearch"
        }
      ]
    },
    "reloads": {"last_error": null, "successes": 0, "last_success_timestamp": null, "last_failure_timestamp": null, "failures": 0 }
  }
```

Hot Threads

```
GET /_node/hot_threads
```

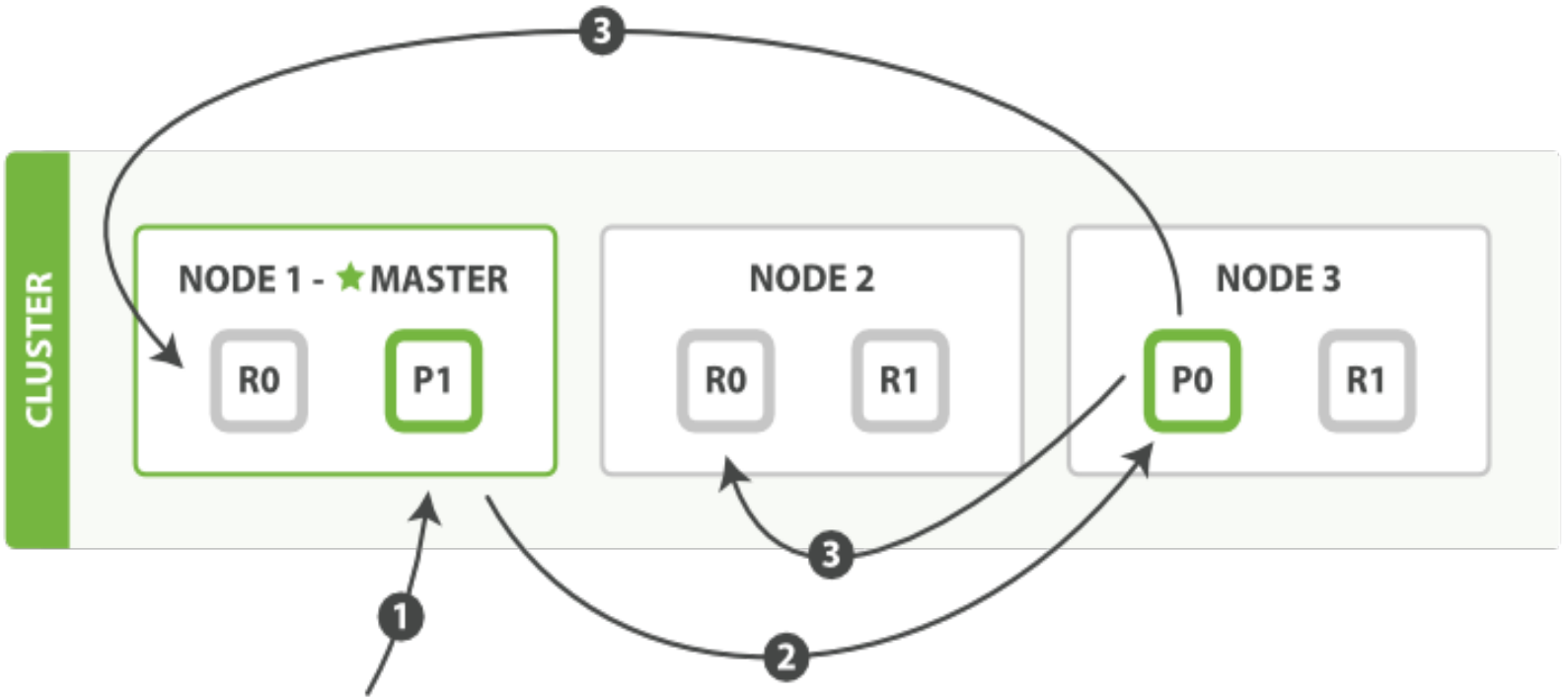
- Retourne des informations sur les threads qui prennent le plus de CPU
- Pour chaque thread :
  - Le pourcentage de CPU
  - Son état
  - Sa stack trace

Indexation et Recherche
Versus
Shards et Replica

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 uen chaîne arbitraire qui par défaut correspond à l'id du document mais peut être explicitement spécifié => Le nombre de shards primaire est donc fixé à la création de l'index et ne peut plus être changé

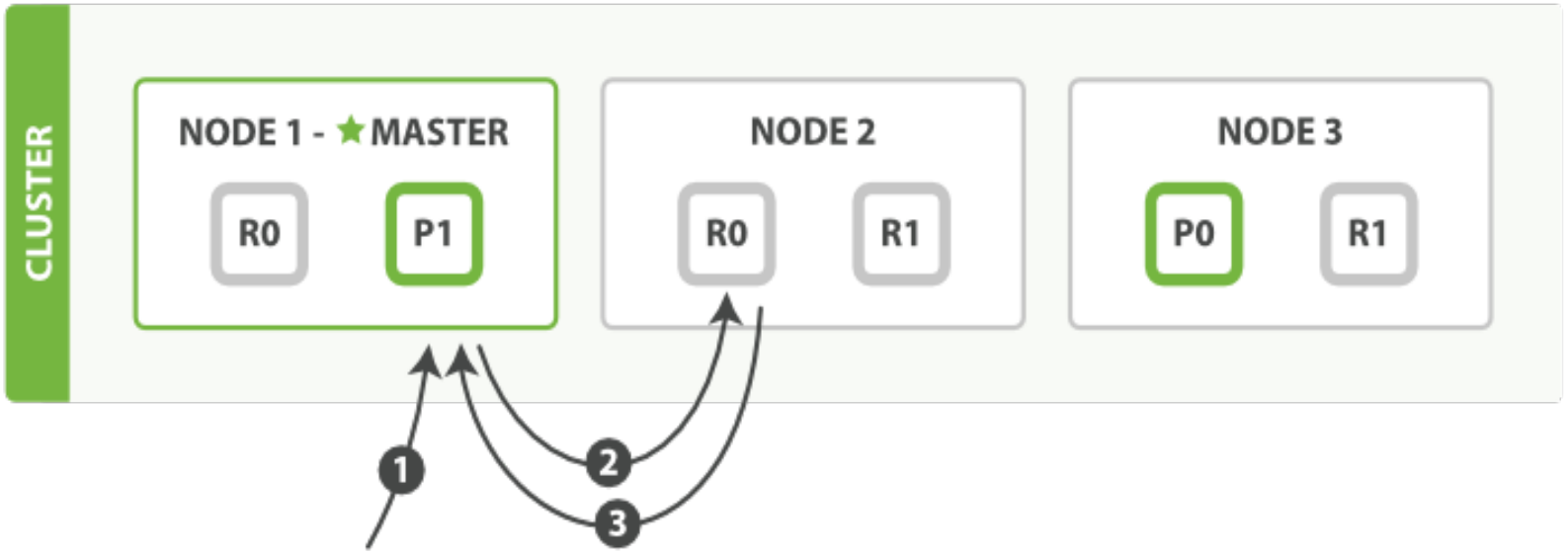
Séquence d'une mise à jour sur une architecture cluster



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 transfert 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 réplicas 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

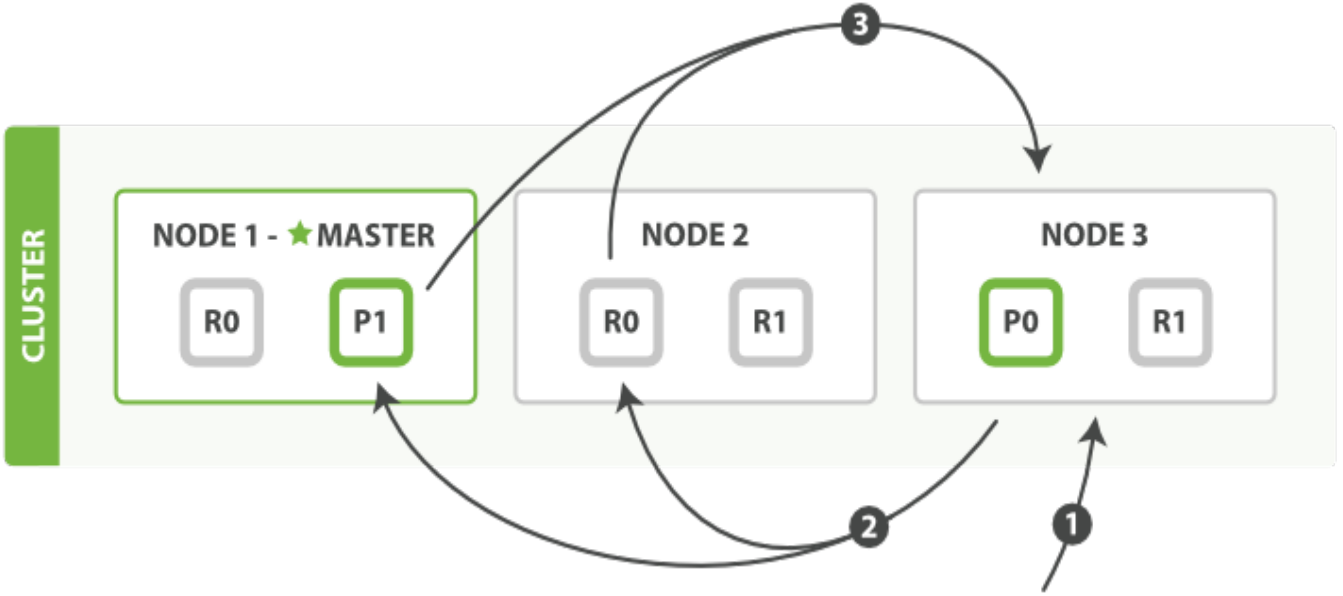
- 1. Le client envoie une requête au nœud 1.
- 2. Le nœud utilise l'id du document pour déterminer que le document appartient au shard 0. Des copies de shard 0 existent sur les 3 nœuds Pour cette fois-ci, il transfert la requête au nœud 2 .
- 3. Le nœud 2 retourne le document au nœud 1 qui le retourne au client.
- Pour les prochaines demandes de lecture, le nœud choisira un shard différent pour répartir la charge (algorithme Round-robin)

Recherche distribuée

- 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 don 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

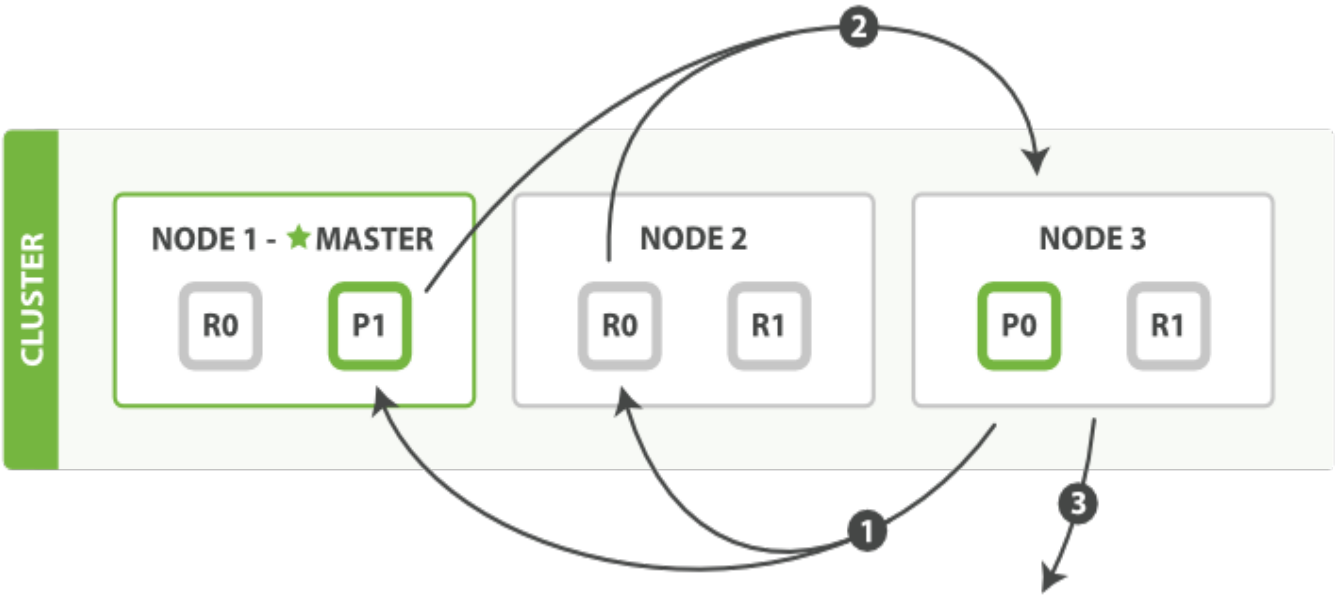


Phase de requête

- 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 dasn 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 coordinbateur 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.

Monitoring ElasticSearch

X-Pack

- ELS offre une API permettant d'obtenir certains métriques d'un cluster
- L'outil de surveillance recommandé par ELS (Marvel) a été intégré dans X-Pack
  - Il sollicite périodiquement l'API et stocke les données dans ELS.
  - Il est alors possible de construire des graphiques sur les données historisées

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_L18BOQ": {
- "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, ...

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
    ```

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
  ```
- En production, il est également recommandé d'utiliser ***unicast*** plutôt que *multicast*.
  ```
  discovery.zen.ping.multicast.enabled: false
  discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]
  ```

Mémoire heap

- L'installation par défaut d'ELS est configuré avec 1 GB de heap (mémoire JVM). Ce nombre est bien trop petit
- 2 façons pour changer la taille de la heap .
  - 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.
- Eventuellement, le désactiver au niveau système
- `sudo swapoff -a`
-
- Ou au niveau ELS
- `bootstrap.mlockall: true`

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

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"
647}
}
```

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" }`
- `}`

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 une dépôt partagé
- Le premier snapshot est intégral, les autres sauvegardent les deltas
- Les dépôts peuvent être de diffférents 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

MERCI !!

Pour votre attention