



# Elasticsearch

# Plan



- Introduction to search engines
- First steps with elasticsearch
- Document indexation
- Mapping
- Text analysis and extraction
- Search
- Aggregations
- Advanced search features
- Java clients and APIs
- Cloud and clusterisation
- Advanced features



# Introduction to search engines

# Definitions



*« A search engine is an information retrieval system designed to help find information stored on a computer system » Wikipedia*

# Search engine | Specialization



- Health, F&B,
- News, Informations,
- Travels, Transports, Mapping,
- Blogs, Social networks, Forums,
- Books, Videos, Pictures,
- Public services,
- Shopping, ...



# Search | Editors



- Sinequa
- Dassault Systems Exalead
- Antidot
- Celebros
- Oracle Endeca
- FredHopper
- ...



# Open source solutions



- Apache Lucene
- Apache Solr
- Elasticsearch
- Hibernate Search
- Compass
- Sphinx
- ...





# Needs / Requirements of a search engine

- Why do we need search engine?



# Needs / Requirements of a search engine





# Needs / Requirements of a search engine



# Key features



- Relevance (or Scoring)
  - The relevance of results is the most important feature of a search engine
  - Score computing can be modified:
    - Boost
    - Top hits / Always on top

jam singapore

All Images News Maps Videos More Settings Tools

About 22,000,000 results (0.33 seconds)

**Jam & Marmalade Singapore - Wide Variety Of Spreads Available**  
Ad [www.redmart.com/Jam\\_&\\_Marmalade](https://www.redmart.com/Jam_&_Marmalade) ▾  
Only At Your RedMart Online Store!  
Hassle-Free Shopping · Free Delivery Available · Christmas Shop · Everyday Low Prices  
Types: Chocolate Spreads, Honey, Jam, Marmalade, Kaya, Pate, Peanut Butter, Savoury Spread

Jams & Marmalade      Christmas Shop  
Peanut Butter      Spreads

**ONE.MOTORING - Traffic News**  
[https://www.onemotoring.com.sg/content/onemotoring/en/on\\_the.../traffic\\_news.html](https://www.onemotoring.com.sg/content/onemotoring/en/on_the.../traffic_news.html) ▾  
On the eve of Christmas (24th December 2016, Saturday), Weekend/Off-Peak Cars require a valid e-Day licence if used before or at 3pm while Revised Off-Peak ...

# Key features



- Auto-Suggest
  - Saves time
  - Direct user to better searches
  - Improved cache handling

The screenshot shows the Lazada SG mobile application interface. At the top, there is a navigation bar with the Lazada logo and categories: Electronics and Women's Fashion. Below the navigation bar, there is a promotional banner for a 'YEAR FURNI SA' sale. On the right side of the screen, a search bar contains the query 'chair'. Below the search bar, a list of suggestions is displayed: 'chair', 'chairs', 'chair folding', and 'chair leg foot cap'. Further down, a section titled 'Popular Products' lists four items:

- 4D GAMING CHAIR \* Racer Seat Chair \* high back boss chair \* Computer chair(Black&Red)  
SGD 119.00
- Basic Office Chair Rein S02 Black  
SGD 43.00
- High-End Economic Chair with steel base (Black) (16)  
SGD 79.90
- Mesh Pastel Floor Chair (Grey) (Free Delivery)  
SGD 39.90

# Key features



- Spell checking
  - Saves time
  - Avoid search with bad spelling
  - Complex implementation: needs dictionary for each supported language
  - Alternative possible using fuzzy search or synonyms

# Key features



- Faceting
  - Classify and analyze search results
  - Allow the user to drill down on a subcategory

The screenshot shows the Amazon.fr website interface for a search query. The search bar at the top contains "Spring Batch". The main content area displays search results for "Spring Batch" with 0 results found. To the left, there is a sidebar with various facets:

- Boutique:** «Toutes les boutiques»
- Livres:** Informatique et Internet (6), Sciences, Techniques et Médicale (1), Ésotérisme et Paranormal (1), Scolaire et Parascolaire (1), Entreprise et Bourse (2)
- Format:** Poche et Broché (8)
- Option d'expédition:** Livraison gratuite
- Moyenne des commentaires client:** ★★★★☆ & plus (1), ★★★☆☆ & plus (2), ★★☆☆☆ & plus (2), ★☆☆☆☆ & plus (2)
- Prix:** 20 à 50 EUR (0), Plus de 50 EUR (2)

The main search results area shows three items:

1. **Spring Batch in Action [With eBook]** By Cogoluegnes, Arnaud (Author) Paperback on (10 , 2011) de Arnaud Cogoluegnes (Broché - 7 octobre 2011)  
1 neuf à partir de EUR 57,99
2. **(Pro Spring Batch (New))** By Minella, Michael (Author) Paperback on (07 , 2011) de Michael Minella (Broché - 14 juillet 2011)  
1 neuf à partir de EUR 59,99
3. **Spring par la Pratique Spring 2.5 et 3.0 de**  
Acheter neuf: EUR 45,60 EUR 43,32  
4 neufs à partir de EUR 40,00  
Plus que 11 ex. Commandez vite !

To the right of the search results, there is a data visualization dashboard:

- Categories:** A donut chart showing proportions of different categories.
- Topics:** A donut chart showing proportions of topics, with one segment labeled "60.0% Apple".
- Sentiment:** A donut chart showing proportions of sentiment.
- Top 5 Authors:**

noznamo : 331 (2.03%)
Zoravv : 122 (0.75%)
tomrossa : 100 (0.61%)
Daniel Dolekai : 93 (0.57%)
MacBog.sk : 84 (0.51%)

A large line chart is also visible in the background of the dashboard area.

# Key features

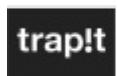


- More Like This
  - Trigger a new search from a document
  - Allows the system to find similar elements to the current one
  - Direct user to other products - can be used for recommendations

# Key features



- Orientation
  - Google « +1 »
- Personalized results
  - Machine learning : search engine « understands » the search requests of the user
  - Bubble effect
- Cloud usage
- Analyze and build abstracts of the results





# Set up your own search engine

- Whatever the technical solution used, the steps will always be:
  - To collect and index data
  - To search and return the results



# Different steps towards data indexing





# Content Collection



# Content Collection



- **What:** Identify and fetch data to index
- **Where:** Data source can differ a lot. Should you find it in:
  - Filesystems,
  - Databases,
  - Mails,
  - Websites,
  - Content Management Systems,
  - ...
- **How difficult:** Higher difficulty if data access is not the same for everyone (« superadmin », ACLs)
- **How often:** Data indexing can either be live or incremental
- **How:** Specific indexing tools like crawlers, connectors or batch processing are available for you
  - Solr, Nutch, Grub, Apache Droids, Heritrix, Aperture, ManifoldCF, beats...



# Document Creation



# Document Creation



- Adapt informations to fit the search engine model (document mapping)
- A document consists of several fields each with:
  - type or format
  - storage options like:
    - docvalues
    - \_source keeping
    - field storage
  - analysis options like:
    - Text extraction/tokenization
    - Scoring / highlighting options
    - Synonyms additions
    - Grammatical settings
    - Value modification / parsing (dates, date formats, scaled float, ...)
  - semantic options like:
    - Boosting some field
    - Business synonyms
- The structure of documents must fit future search and display requirements



# Datas extracts from a bookstore (1/2)

- Samples of « Book » documents

Identifier	Fields	Value(s)
1	title	Spring Batch in Action
	authors	Arnaud Cogoluegnes, Thierry Templier, Gary Gregory
	price	59.99
	currency	USD
	publication	10/2011
	language	en_US
2	title	Mahout in Action
	authors	Sean Owen, Robin Anil
	price	44.99
	currency	USD
	publication	10/2011
	language	en_US



# Datas extracts from a bookstore (2/2)

Identifier	Fields	Value(s)
3	title	L'art en théorie et en action
	authors	Nelson Goodman
	price	6.60
	currency	EUR
	publication	01/2009
	language	fr_FR
4	title	Spring par la pratique
	authors	Arnaud Cogoluegnes, Thierry Templier, Julien Dubois
	price	43.30
	currency	EUR
	publication	10/2011
	language	fr_FR



# Sample JSON document for a book

- Elasticsearch documents are indexed in JSON format

```
{  
  "title" : "Spring Batch in Action",  
  "authors" : [{"  
    "firstname" : "Arnaud", "lastname" : "Cogoluegnes"  
  }, {  
    "firstname" : "Thierry", "lastname" : "Templier"  
  }, {  
    "firstname" : "Gary", "lastname" : "Gregory"  
  }],  
  "price" : 59.99,  
  "currency" : "USD",  
  "publication" : "2010",  
  "language" : "en_US"  
}
```





# Document Analysis



# Document Analysis



- JSON documents are analyzed by the search engine:
  - as the default field type behavior when dynamic mapping is used. i.e field is unknown for the given document type
  - as chosen at mapping time for known field types
- As a result, the search engine analyzes each field and applies needed transformations
- Those tranformations allow us to get the expected behaviour:
  - Case insensitive searches
  - Synonyms
  - Phonetic search
  - Plural and conjugation
  - ...



# Analysis sample

- Original text

*Spring Batch in Action*

*Mahout in Action*

*The art in Theory and in Action*

*Working with Spring*

- after text analysis

*spring batch action*

*mahout action*

*art theory action*

*working spring*



# Document Indexing



# Document Indexing



- During this phase, documents are just added to the search engine index and storage as required by explicit or default mapping
- The internal organization of the search index depends on each implementation (Lucene for SOLR and Elasticsearch)
- Most search engine use an Inverted Index
  - each word/token references documents containing it
  - tradeoff: quick search but effort on indexing
- Index options have an effect on the disk size and the used memory for each document



# Document indexing sample 1/2

- after text analysis

*spring batch action*

*mahout action*

*art theory action*

*working spring*

- Unique terms

*spring, batch, action, mahout, art, theory, working*



# Document indexing samplee 2/2

- Unique terms

*spring, batch, action, mahout, art, theory, working*

- Inverted index construction sample

	<b>spring</b>	<b>batch</b>	<b>action</b>	<b>mahout</b>	<b>art</b>	<b>theory</b>	<b>working</b>
Spring Batch in Action	X	X	X				
Mahout in Action			X	X			
The art in Theory and in Action			X		X	X	
Working with Spring		X					X



# Different steps towards querying data





# Prepare business search options



# Prepare business search options



- A search form allows the user to query the search engine with keywords and categories as required by the business
- Each option should be able to filter out data and to drill down
- User experience is **very** important
- Business requirements as authentication and authorization checks may be part of your request
- For security and data management reasons, do not expose entire available options to the client desktops



# Forge your DSL requests



# Forge your DSL requests



- From the previously selected **keywords/filters**, your applications have to create a **search query**
- The application can also add **specific filters** on top of what the user selected from UI
  - e.g. filter out data depending on user role
- The query is executed on the search engine and a **relevance score** is calculated
- Do not forget to forge requests with user experience and performance attention in mind. **Caches** are useful.



# Search sample

- Searching « spring action » in the ***title*** inverted vector index

	spring	batch	action	mahout	art	theorie	pratique
Spring Batch in Action	O	X	O				
Mahout in Action			O	X			
L'art en théorie et en action			O		X	X	
Spring par la pratique	O						X



# Search relevance is a Search Engine keypoint

- A relevance factor is calculated : the **scoring**
- To calculate the relevance
  - How many **terms** does the document contain ?
  - How many of the **search options** are matching the document ?
  - How many times are the **matching terms** found in the document ?
  - What is the frequency of **matching terms** in the selected documents ?
- Relevance can be customized :
  - Per document and per field
  - During document indexing or during querying





# Parse results



# Parse results



- Delivers a paginated list of results by relevance if taken into account or specific chosen ordering options
- optionnaly delivers results for aggregations or highlighting requests
- Indicates `_index` state and request informations
- The UI can allow the user to drill down on the results with other requests resolution if needed or chaches if compatible with the user experience
- Results should not hide the processes applied by the search engine (e.g. spelling correction)
- Use your own implementation

# Open Source Solutions



# Apache Lucene



- Java open source project created in 2000 by
  - Doug Cutting (later also Nutch and Hadoop)
  - Erik Hatcher (Ant, Tapestry)
  - Otis Gospodnetic (Lucene, Hadoop)
- Current version is 6.6.x <http://lucene.apache.org/>
- Provides a search engine and utility classes
  - Analysis and Tokenization Classes
  - Search request builders
  - Document Builders
  - Index handlers
- Lucene is core to other open source search engines
  - low level API (filesystem, Document structure)
  - Difficult handling in multi threaded/distributed environments

# Apache Solr



- Java open source project
  - First release (1.3) in 2006
  - Merged with Apache Lucene in 2011
- Based on Apache Lucene library
- Current version is 6.6.x <http://lucene.apache.org/solr/>
- Same development cycle as Lucene
- Search server
  - HTTP/XML, JSON API
  - Distributed search
  - Administration interface
  - Master slave replication
  - Aggregation and geolocation support
  - Data importing tool (CSV, XML, Database)
  - Configuration mostly via XML files

# Elasticsearch



- Open source Java project
  - First release in 2010
  - Created by Shay Banon (also creator of Compass)
- Based on Apache Lucene and experience on Compass
- Current version 5.4 available on  
<https://www.elastic.co/downloads/elasticsearch>
- Clustered search server
  - REST/JSON API
  - Distributed search in clustered mode
  - Sized for big volumes
  - JSON, YAML, properties configuration
  - Schemaless datastructure
  - Multi-index Search
  - Aggregation and geolocation support



# Getting Started with Elasticsearch

# Elasticsearch



- Elasticsearch is a **search engine** developed in Java
  - Open source (Apache License 2), Distributed, RESTful
- Based on the Apache Lucene library, Elasticsearch masks its complexity
- Elasticsearch provides **Web Services** via an JSON HTTP/REST API
- Elasticsearch provides a **Java API** equivalent to Web services services
- Elasticsearch works with the concept of **cluster**
  - A cluster is a set of Elasticsearch instances
  - An instance of Elasticsearch is called a '**node**'
  - Improves the **performance** and the **availability** of the search engine



# Clustering (1/2)

- An Elasticsearch ***cluster***
  - Is identified by a ***unique name***
  - Is composed of **nodes** located on the ***same network***
  - Elects a ***single master*** node to:
    - Manage cluster state
    - Make decisions on data routing
    - Make rebalancing decision on cluster topography changes
    - Coordinate snapshot/restore operations

# Clustering (2/2)



- An Elasticsearch ***node***
  - Is a standard ***Java process*** which ***always join a cluster***
  - Can be added or removed from a cluster at runtime
  - Communicates with other nodes to index, handle queries, share cluster state and datas,...
  - Stores parts or none of the indexed data depending of its ***data*** role
  - Can be elected as ***master*** depending of its ***master eligible*** role
  - Can execute ***processing pipelines*** on incoming documents depending of its ***ingest*** role and ***processing parameters***
  - Provides HTTP REST/JSON Web Services or Java API to index, request, manage clusters, ... as each node can ***coordinate requests*** in cluster



# Index hold Document Types

- An **index** is a storage space for functionally related documents
  - It wears some structural settings like:
    - iOne or more **Document types definitions** also called **mappings**
    - **Text analyzers** available for use in all **Document types fields**
    - **Splitting storage strategy** (see the following slide with **sharding** definition)
    - **Lucene** management parameters
- A **document type** is the **index relative definition** of what should **contain** each document and **how to process** each of their **fields**
  - Document types can be dynamically discovered (Elasticsearch is **schemaless**)
  - Document types can be precisely defined with **mappings**
  - Document types can be partly defined and discovered



# Documents

- A **document** is a **partial**, **complete** or **more than** an **instance** of a document type
  - It represents the business data value to be indexed and on which searches will be performed
  - It is formatted in JSON and contains fields values

```
{  
    "title" : "Spring Batch in Action",  
    "authors" : ["Arnaud Cogoluegnes", "Olivier Bazoud"],  
    "price" : 20.80  
}
```



# Shards and Replicas (1/2)

- To enable clustering, **index storage** is divided into **one or more** parts called **shards**
  - A shard is a **partition** of an index and contains **non overlapping subsets** of all the index documents
  - As a result, each incoming document will be **coherently first rooted** to a chosen **shard** by the master
  - The number of **shards** is fixed at the index creation

*Sharding is a way to think up to **horizontal indexation scaling***

*Oversharding is a way to split more than it is today necessary in case of unexpected business needs increase*



# Shards and Replicas (2/2)

- **Shards** may have replicas
  - A replica is a copy of a shard
    - Strictly speaking it is really the same thing as its corresponding shard.
    - Its master attributed role is the only distinguishing factor
    - A replica is not first involved in indexation
    - A replica can respond to search requests just as efficiently as its corresponding shard
    - A replica can instantly be promoted as a shard by the master node of the cluster
  - Its update process takes place ***after corresponding shard document indexation*** or at ***creation/recovery time***
  - The number of replicas is configurable at runtime

*Replication is a way to horizontally scale searches as the documents are available on more than one node*

*It is also a resilience and high availability helper for the same reasons*

# Installation



- There are three deployment solutions:
  - Local deployment only for development purposes
  - Service deployment as RPM/DEB or Windows service for development and production purposes
  - Elastic Cloud Enterprise (**ECE**) deployment for a development/production platform as a service



# Local deployment (1/3)

- Elasticsearch 5.x requires a JRE 1.8.0\_73 or later (**JAVA\_HOME** needed)
- Distributed as an archive tar.gz or zip
- Start an Elasticsearch instance (No configuration required on first launch):
  - **./bin/elasticsearch** (Unix)
  - **./bin/elasticsearch.bat** (Windows)

```
> wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-5.4.0.tar.gz  
> tar -xzvf elasticsearch-5.4.0.tar.gz  
> cd elasticsearch-5.4.0  
> ./bin/elasticsearch
```



# Local deployment (2/3)

- You can run Elasticsearch with configuration overriding options

```
> ./bin/elasticsearch -Ecluster.name=production -Enode.name=node1
```



# Local deployment (3/3) - Directory layout

Directory	Description	Default path	Configuration
home	Base directory of Elasticsearch or \$ES_HOME		
bin	executables	\$ES_HOME/bin	
config	Configuration files	\$ES_HOME/config	path.conf
data	Index data. Can contain data from several nodes	\$ES_HOME/data	path.data
logs	Server logs	\$ES_HOME/logs	path.logs
plugins	Plugin files location	\$ES_HOME/plugins	

- Some paths are configurable :

- With a configuration file

```
path.data: /mnt/data
```

- As an program parameter

```
elasticsearch -Epath.data=/path/to/my/data/
```





# Service deployment on Linux (1/2)

- Use **DEB** when compatible with your ***Operating System***:
  - Debian
  - Ubuntu
  - Other Debian based
- Use **RPM** when compatible with your ***Operating System***:
  - Red hat
  - Centos
  - SLES
  - Open Suse
  - Other RPM-based system



# Service deployment on Linux (2/2)

- RPM installation sample with service enabling

```
> yum localinstall /mnt/admin/rpm/ELK/elasticsearch-5.1.2.rpm
Modules complémentaires chargés : langpacks, product-id, search-disabled- repos
Examen de /mnt/admin/rpm/ELK/elasticsearch-5.1.2.rpm ; elasticsearch-5.1.2-1.noarch
 Sélection de /mnt/admin/rpm/ELK/elasticsearch-5.1.2.rpm pour installation
Résolution des dépendances
--> Lancement de la transaction de test
--> Le paquet elasticsearch.noarch 0:5.1.2-1 sera installé
--> Résolution des dépendances terminée

Dépendances résolues
=====
Package Architecture Version Dépôt Taille
=====
Installation :
elasticsearch noarch 5.1.2-1 /elasticsearch-5.1.2 35 M

Résumé de la transaction
=====
Installation 1 Paquet

Taille totale : 35 M
Taille d'installation : 35 M
Is this ok [y/d/N]: y
Downloading packages:
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
Creating elasticsearch group... OK
Creating elasticsearch user... OK
Installation : elasticsearch-5.1.2-1.noarch 1/1
### NOT starting on installation, please execute the following statements to configure elasticsearch service to start automatically using systemd
sudo systemctl daemon-reload
sudo systemctl enable elasticsearch.service
### You can start elasticsearch service by executing
sudo systemctl start elasticsearch.service
Vérification : elasticsearch-5.1.2-1.noarch 1/1
Installé :
elasticsearch.noarch 0:5.1.2-1
Terminé !

> systemctl enable elasticsearch
> systemctl start elasticsearch
```



# Service deployment on Windows

- Start a local deployment
- Use the command `elasticsearch-service.bat` like this:

```
> bin\elasticsearch-service.bat install  
> bin\elasticsearch-service.bat stop
```



# Official documentation

- See

[https://www.elastic.co/guide/en/elasticsearch/reference/current/\\_installation.html](https://www.elastic.co/guide/en/elasticsearch/reference/current/_installation.html)

# Elastic Cloud Enterprise (1/2)



Also called **ECE**, this is a new way to install **Elasticsearch** to create an **Elastic platform as a service**.

## *The big picture:*

- Install some big machines (Licence is per 128 Go Ram)
- Install an allocator on the first of them
- Install other allocators pointing to the first one to make a cluster of allocators
- Connect to the admin interface to create one or more clusters:
  - business clusters
  - monitoring clusters
  - production clusters
  - development clusters
  - ...
- Connect to the admin interface to parameterize access, autorizations, ...

# Elastic Cloud Enterprise (2/2)



- Technically, **ECE** is a cluster of clusters:
  - It is a cluster of **Allocators** holding **cluster of Elasticsearch nodes**
  - It shares **underlying physical machine resources** via **docker containers instantiation**
  - It is resilient with the help of its **availability zones** which make Elastic docker nodes dispatched efficiently
- See the [architecture overview](#) if interested



# Configuration of a node (1/2)

- Elasticsearch has three main configuration files
- `elasticsearch.yml`, `log4j2.properties` and `jvm.options`
- Located in the `config` directory
- The main file is formatted in YAML (`.yml`)



# Configuration of a node (2/2)

- Here is a list of important Elasticsearch configuration options:
  - path.data and path.logs
  - cluster.name
  - node.name
  - bootstrap.memory\_lock
  - network.host
  - discovery.zen.ping.unicast.hosts
  - discovery.zen.minimum\_master\_nodes
- Look at **this** to see the details

*[https://www.elastic.co/guide/en/elasticsearch/reference/current/importan...  
settings.html](https://www.elastic.co/guide/en/elasticsearch/reference/current/importan...)*

- Do not forget other important **system settings**:

*[https://www.elastic.co/guide/en/elasticsearch/reference/current/system-...  
config.html](https://www.elastic.co/guide/en/elasticsearch/reference/current/system-...)*

# Plugins (1/2)



- Integrated plugin system (dedicated executable `elasticsearch-plugin`)
- Install an Elasticsearch plugin like this, only changing the plugin name:

```
> ./bin/elasticsearch-plugin install analysis-icu
=> Downloading analysis-icu from elastic
[=====] 100%
=> Installed analysis-icu
```



# Plugins (2/2)

- Many types of plugins :
  - Alerting,
  - Analysis,
  - Ingestion,
  - Discovery,
  - Mapper,
  - Scripting,
  - Security,
  - Backup,
  - Transport
- See the full list here:

<https://www.elastic.co/guide/en/elasticsearch/plugins/master/index.html>

# Communicate with Elasticsearch 1/2



- Via the ***Rest interface***
  - Language and platform agnostic
  - A lot of http clients or libraries exist
  - Decouple a little bit more from the Elasticsearch version than the java library
  - Templating is easy and minor changes(sorting) can be done without changing a line of code
  - <https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html>
  - Widely used, even in production
- Via the ***Java client*** library
  - <https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/index.html>
  - The internal API of Elasticsearch

# Communicate with Elasticsearch 2/2



- Other clients and front-ends
  - elastic and third-party libraries offering Elasticsearch clients for many languages:
    - php,
    - perl,
    - scala,
    - python
    - ...
  - Http tools like CuRL, wget, postman, kibana dev tools...
  - Connectors/Inputs/Outputs like MongoDB, Logstash, Beats, ...
  - Monitoring or administration interfaces like Kibana, Head, Kopf, ...
  - <https://www.elastic.co/guide/en/elasticsearch/client/index.html>



# Overview of the Rest API

- The Rest API template which make all API operations possible  
`http://host:port/[index]/[type]/[_action|id]`
- Requests and responses are formatted in JSON
  - **index** : Name of the index that is the target of the operation
  - **type**: Document type name
  - **\_action**: Name of the action to be performed
  - **id** : Document ID
  - **Method**: Depends on the type of operation to be performed on the resource: **GET, POST, PUT, DELETE**

# Rest tools



- From command line
  - cUrl

```
curl -XGET "http://localhost:9200/library/_search?pretty=true"

curl -XPOST "http://localhost:9200/library/library/_search" -d '
{ "query": { ... }}'

curl -XPUT "http://localhost:9200/library/library/1" --data-binary
@book1.json
```

- HTTPIe
- Browser plugins
  - Chrome: Postman, Advanced Rest Client, Insomnia
  - Firefox: Rest Client, Rest Easy
- Kibana Dev Console (formerly: Sense)



# Indexing first document 1/2

- A REST/JSON API enables to communicate with Elasticsearch, for example to index a document with Kibana dev tools:

```
PUT /library/book/1
{
  "title" : "Spring Batch in Action",
  "authors" : [
    {"firstname" : "Arnaud", "lastname" : "Cogoluegnes"},  

    {"firstname" : "Thierry", "lastname" : "Templier"},  

    {"firstname" : "Gary", "lastname" : "Gregory"}
  ],
  "price" : 59.99,
  "currency" : "USD",
  "publication" : "2011-10",
  "language" : "en_US"
}
```

- **library** is the name of the index where the document is stored
- **book** is the type of the document
- **1** is the document identifier



# Indexing first document 2/2

- As the document was provided in JSON format, the answer is returned in JSON too

```
{  
  "_index": "library",  
  "_type": "book",  
  "_id": "1",  
  "_version": 1,  
  "result": "created",  
  "_shards": {  
    "total": 2,  
    "successful": 1,  
    "failed": 0  
  },  
  "created": true  
}
```

# Document retrieval



- The REST/JSON API enables to retrieve a precise document as soon as we know its index, its type and its identifier :
- ***GET /library/book/1*** will return

```
{  
  "_index": "library",  
  "_type": "book",  
  "_id": "1",  
  "_version": 1,  
  "found": true,  
  "_source": {  
    "title": "Spring Batch in Action",  
    "authors": [ {  
      "firstname": "Arnaud", "lastname": "Cogoluegnes"  
    }, {  
      [...]  
    } ],  
    "price": 59.99,  
    "currency": "USD",  
    "publication": "2011-10",  
    "language": "en_US"  
  }  
}
```

# First search 1/2



- ***GET /library/book/\_search?q=action*** will return all books containing the «action» term

```
{  
    "took": 11,  
    "timed_out": false,  
    "_shards": { "total": 5, "successful": 5, "failed": 0},  
    "hits": {  
        "total": 1,  
        "max_score": 0.28004453,  
        "hits": [  
            {  
                "_index": "library", "_type": "book", "_id": "1",  
                "_score": 0.28004453,  
                "_source": {  
                    "title": "Spring Batch in Action",  
                    "authors": [  
                        [...]  
                    ],  
                    "price": 59.99,  
                    "currency": "USD",  
                    "publication": "2011-10",  
                    "language": "en_US"  
                }  
            }  
        ]  
    }  
}
```

# First search 2/2



- `hits.total` indicates the total number of results found
- For each result or `hit` :
  - Its coordinates : the `_index` it belongs to, the `_type` of the document, and its unique identifier `_id`
  - The document `_source`
  - The document relevance or `_score`



# Search on multiple types

- Indexing a **dvd** document in the index **library**

```
PUT /library/dvd/1
{
  "title" : "Looney Tunes: Back in Action",
  "authors" : [
    {"firstname" : "Timothy", "lastname" : "Dalton"},
    {"firstname" : "Jenna", "lastname" : "Elfman"},
    {"firstname" : "Brendan", "lastname" : "Fraser"}
  ],
  "release" : "2003-12"}
```





# Search on multiple types samples

- Search of the books and dvd containing « action » in the title

```
GET /library/book,dvd/_search?q=title:action
```

- Search on all types of documents containing « action » in the title

```
GET /library/_search?q=title:action
```

- Search on multiple fields

```
GET /library/_search?q=authors.lastname:templier OR authors.lastname:dalton
```

```
GET /library/_search?q=authors.\*:templier
```

```
GET /library/_search?q=\*.lastname:d*
```



# Search on multiple indexes samples

- Search books and dvd on multiple indexes

```
GET /library1,library2/book,dvd/_search?q=title:action
```

- Search on all indexes (`_all`)

```
GET /_all/_search?q=title:action
```

```
GET /_search?q=title:action
```

- Search on all indexes starting with "lib" except « library »

```
GET /lib*,!library/_search?q=Action
```





# Document Indexing



# Index | Shard & replica 1/3

- JSON documents are typed and stored in an index
- An **Index** is made of shards and replicas
- **Shard**
  - A shard is a fragment of the index
  - All the shards together constitute the index
- **Replica**
  - A replica is a copy of the index
  - Replicas ensure that no data of the index is lost when a node crashes
- By default, shards and replicas are automatically distributed among the different nodes of the Elasticsearch cluster



# Index | Shard & replica 2/3

- Default configuration in `elasticsearch.yml` :
  - `index.number_of_shards: 5`
  - `index.number_of_replicas: 1`
- Configuration for each index during the index creation
  - Number of shards defined **when the index is created**
  - The number of replicas can be **changed at run time**



# Index | Shard & replica 3/3



- **Adding shards** improves performances at indexing time and allows the distribution of big indices on several nodes
- **Adding replicas** improves performances at search time and ensures higher availability of the index
- In most cases the default configuration is fine, however only performance tests will help to find out the best shard/replica configuration

# Creating an index



- Create an index using the REST Create Index API:

```
PUT /library
```

- Create an index with settings, mappings and aliases:

```
PUT /library
{
  "settings" : {
    "number_of_shards" : 8,
    "number_of_replicas": 2,
    "analysis" : {
      "analyzer": {
        "content": {
          "type": "custom",
          "tokenizer": "whitespace"
        }
      }
    }
  },
  "mappings": { ... },           // Seen later on
  "aliases": { ... }            // Seen later on
}
```

- Indices are ***automatically created*** if they don't exist ***when a document is being indexed***



# Updating an index

- Update **settings** of an index using the REST Update Settings API:

```
PUT /library/_settings
{
  "index" : {
    "number_of_replicas": 6,
    "refresh_interval": "30s"
  }
}
```

- Main settings :
  - number\_of\_replicas** : number of replicas
  - refresh\_interval** : time interval (in seconds) for refreshing the index

**number\_of\_shards** is not allowed on update time



# Deleting an index

- Deletion using the REST [Delete index API](#):

```
DELETE /library           //deletes the library index  
DELETE /index1,index2,index3 //deletes several index at a time  
DELETE /_all              //deletes all indexes  
DELETE /*                 //deletes all indexes
```

- Deletion of all the indices can be disabled by configuring in the file [elasticsearch.yml](#):

```
action.destructive_requires_name : true
```



# Indexing documents (1/2)

- Indexing document using the REST **Index API**:
  - If your documents are subject to **CRUD** operations, you should use **PUT method** and give your documents chosen **business identifiers**

```
PUT /library/book/1
{
  "title": "7 Years Later...",
  "authors": "Guillaume Musso",
  "price": 20.80
}
----- Résultat -----
{
  "_index": "library",
  "_type": "book",
  "_id": "1",
  "_version": 1,
  "result": "created",
  "_shards": { ... },
  "created": true
}
```

*Impossible to get back a unique document to make  
CRUD operations without an identifier*



# Indexing documents (2/2)

- Without **CRUD** operations necessity, you should use **POST method** to get a document identifier **generated** for you

```
POST /library/book
{
  "title": "7 Years Later...",
  "authors": "Guillaume Musso",
  "price": 20.80
}
//--- Résultat -----
{
  "_index": "library",
  "_type": "book",
  "_id": "AVGRV_usnjDD0CQltR_N",
  "_version": 1,
  "result": "created",
  "_shards": { ... },
  "created": true
}
```

# Updating documents 1/3



- Updating a document means ***totally reindexing this document***:
  - The new document replaces its previous version in the index
  - The new document has a ***\_version*** field set to ***old document \_version + 1***
- Updating a document requires to know its identifier for a direct access

```
PUT /library/book/1
{
  "title": "7 Years Later...",
  "authors": "Guillaume Musso",
  "price": 20.80
}
//--- Résultat -----
{
  "_index": "library",
  "_type": "book",
  "_id": "1",
  "_version": 2,
  "result": "updated",
  "_shards": { ... },
  "created": false
}
```

# Updating documents 2/3



- The REST [Update API](#) allows to update 0-\* fields of documents via scripts
- Allows:
  - adding new fields,
  - adding new values to an existing field,
  - delete an existing field,
  - removing the document if a condition is verified...
- Seamlessly, script updates reindex the whole documents as a complete update:
  - It requires an enabled `_source` field
  - It internally uses the documents `_version` fields to detect concurrent accesses (more details on the `_version` field later on)



# Updating documents 3/3

- First option, using **script**:

```
POST /library/book/1/_update
{ "script": {
    "inline": "ctx._source.price += params.addition",
    "params": { "addition": 4 }
}}
```

- **ctx** map gives access to several information (**\_index**, **\_type**, **\_id** ...)
- possible to delete fields:  
`"inline" : "ctx._source.remove(\"price\")"`

- Second option, providing a partial **doc**:

```
POST /library/book/1/_update
{ "doc": {
    "price" : 25
}}
```

- merged with the current document
- no field deletion possible



# Deleting documents

- The REST Delete API allows to delete a document:

```
DELETE /library/book/1
```

# Bulk API 1/3



- Indexing document by document is ***not always performance efficient*** and the Rest **Bulk API** is here to help you with ***fast indexing***...
- This is the only non JSON Rest API and its format is like this:

```
POST /_bulk
{JSON object action + metadata}\n
{Optional JSON source}\n
{JSON object action + metadata}\n
{Optional JSON source}\n
...
...
```

- Output : JSON containing the result of each action in input action ordering
  - It contains HTTP response code
  - And an error message if applicable



# Bulk API 2/3

- Eligible actions : ***index / create / update / delete***
- No need to provide source for ***delete***
- Example of bulk request :

```
{ "create" : { "_index" : "library", "_type" : "book", "_id" : "43" } }
{ "author" : "Victor Hugo", "title": "Les Misérables" }
{ "delete" : { "_index" : "library", "_type" : "book", "_id" : "2" } }
{ "update" : { "_index" : "library", "_type" : "dvd", "_id" : "1" } }
{ "doc" : {"lang" : "fr"} }
...
```

# Bulk API 3/3



- Response :

```
[{"create": {"_index": "library", "_type": "book", "_id": "43", "_version": 1, "status": 201}, {"delete": {"_index": "library", "_type": "book", "_id": "2", "_version": 1, "status": 404, "found": false}, {"update": {"_index": "library", "_type": "dvd", "_id": "43", "status": 404, "error": {"type": "document_missing_exception", "reason": "[dvd][1]: document missing", "index": "library", "shard": "-1"}}, ...]
```

# Document identifier



- All the documents have an identifier
  - During the document indexing, the identifier can be specified (**PUT**) or automatically generated (**POST**)
  - The ***identifier and the document type*** identify a document within an index
  - The id is returned in the ***\_id*** field when performing a search
  - The id is used to retrieve the document in a **GET** request
  - The ***\_uid*** field is the unique identifier in the index ***\_uid = \_type + \_id***
- Whenever possible, use a business identifier
  - All operations on a document use its ***\_id***
  - Without the identifier, a search has to be performed first in order to retrieve the document



# Version

- Documents are automatically ***versioned*** at indexing time
  - Version number starts at 1
  - It is incremented with each indexing
  - The version number is returned during the indexing, but not when performing a search (by default)
- It is possible to manually manage version numbers
  - By using the version parameter and ***version\_type=external***

# Version | Optimistic Concurrency Control 1/4



- **Optimistic Concurrency Control**
  - Elasticsearch has ***no transaction support***
  - When indexing, it is possible to provide a version number using the ***version*** parameter
  - Elasticsearch will verify if the document is in the provided version before performing the requested operation
  - Useful to emulate a read-then-update transaction



# Version | Optimistic Concurrency Control 2/4

Example of requests that show Optimistic Concurrency Control

- Indexing without version (v2 → v3;)

```
PUT /library/book/1 {...}  
{"_index": "library", "_type": "book", "_id": "1", "_version": 3, "created": false}
```

- Indexing v2 (error of type **version\_conflict\_engine\_exception**)

```
PUT /library/book/1?version=2 {...}  
{  
  "error": {  
    "type": "version_conflict_engine_exception",  
    "reason": "[book][1]: version conflict, current [3], provided [2]",  
    "index": "library", "shard": "3"  
  },  
  "status": 409  
}
```

- Indexing v3 (v3 → v4)

```
PUT /library/book/1?version=3 {...}  
{"_index": "library", "_type": "book", "_id": "1", "_version": 4, "created": false}
```





# Version | Optimistic Concurrency Control 3/4

The current version is 3...

- Deleting v3 (error of type ***version\_conflict\_engine\_exception***)

```
DELETE /library/book/1?version=3

{
  "error": {
    "reason": "[book][1]: version conflict, current [4], provided [3]"
  },
  "status": 409
}
```



# Version | Optimistic Concurrency Control 4/4

- Example of request using `version_type=external`
  - Requires `version > existing version in the index`

```
PUT /library/book/1?version_type=external&version=4
{
  "title": "Spring Batch in Action"
}
//--- Résultat -----
{
  "_index": "library",
  "_type": "book",
  "_id": "1",
  "_version": 4,
  "created": false,
  [ ... ]
}
```

# Refresh



- **Near Real Time**
  - There is a small **delay** between the moment the document is **indexed** and the moment it is **available** through search
  - Default is **1 second**
  - Configurable for each index with the **refresh\_interval** setting
- Before a bulk indexing (**\_bulk**), it is recommended to increase the **refresh\_interval** in order to prevent too many index refreshes

# Refresh and Lucene segments



- Before being sent to Lucene, the indexed documents are **analyzed** and **stored** in a buffer in memory:
  - at this point, these documents **cannot** be **searched**.
- This buffer comes with a **Translog** file which shares the same lifecycle and is written to the **system disk cache**. It is used to persist commands in the buffer.
- The buffer is **flushed** to a Lucene segment when:
  - the buffer is **full**
  - during a Lucene **reopen**
  - on an Elasticsearch **refresh**:
    - **without** Lucene commit,
    - triggered according to the **index.refresh\_interval** setting value or though the API **\_refresh**
  - on an Elasticsearch **flush**:
    - triggers a Lucene **commit**,
    - **triggered** automatically by Elasticsearch or though the API **\_flush**

# Lucene segments and merges 1/2



- A Lucene segment is an ***immutable inverted vector index***
- The number of segments increases, ***consumes*** more and more ***resources*** (file pointers, caches, memory...)
- To overcome this problem, Elasticsearch decides to regularly ***merge*** the segments
- These merge operations comply with a ***merge policy*** optionally chosen at the creation of the index. (this is an ***expert*** setting)
- A ***merge*** consists of:
  - Deleting several segments
  - Deleting the associated caches
  - Building a bigger ***immutable*** segment to contain the previously deleted segments without keeping the documents ***marked*** as destroyed

# Lucene segments and merges 2/2



- A merge operation can be costly, use the following parameters at the **node** level to control the cost of the operation:
  - set ***indices.store.throttle.type*** to merge
  - set ***indices.store.throttle.max\_bytes\_per\_sec*** to xmb

See this simulation of Lucene merges to have a better understanding:

<https://www.youtube.com/watch?v=YW0bOvLp72E>

# Forced refresh



In order to force a refresh:

- Indexing operations accept a **refresh** parameter:

- Values:

- **refresh=false** (by default)
    - **refresh=true**: force a refresh
    - **refresh=wait\_for**: wait for a refresh (see **refresh\_interval**)

```
PUT /library/book/1?refresh=true
{
    "title": "Spring Batch in Action"
}
```

- A **\_refresh** operation exists at the index level:

```
POST /library/_refresh
```

- Usage:

- Useful for integration tests
  - Costly in terms of performance: use with care



# Operation Type

- During indexing, it is possible to specify the ***op\_type=create*** parameter or add the ***\_create*** URL path
- This allows to index a document only if does not already exist

```
PUT /library/book/1?op_type=create OR PUT /library/book/1/_create
{
  "title": "Spring Batch in Action"
}
//--- Résultat -----
{
  "error": [
    ...
  ],
  "type": "version_conflict_engine_exception",
  "reason": "[book][1]:version conflict,document already exists(current version[5]),
  "index_uuid": "KfVePdBoTzGkrQUYr43mjjg",
  "shard": "3",
  "index": "library"
},
"status": 409
}
```

# Other features



- When indexing a document it is also possible to specify:
  - The ***routing*** to be used, so that the document is added to a specific shard of the index
  - The ***parent*** document which will allow parent/child searches to be performed
  - A ***timeout***, after which an error is thrown if the operation takes longer than the provided timeout



# Index aliases (1/2)

- It is possible to create **aliases** for the indices
  - An alias is a pointer to one or more other **indices**
  - They can be created and deleted using the REST **Index alias API**

```
POST /_aliases
{
  "actions" : [{"  
    "add" : {  
      "index" : "library",  
      "alias" : "znk" }  
    }]  
}  
  
PUT library/_aliases/znk  
GET library/_aliases
```

- There is a default alias : **\_all**



# Index aliases (2/2)

- Aliases allows several usecases
  - **current** alias for time based indices
  - Joining multiple indices that are often used together
- Advances features
  - Filter : alias on a filtered index (based on a query)
  - Route : specify the routing on the index shards





# Mapping



# Role of the mapping

- A ***mapping*** defines the way a ***document*** is analyzed and indexed
- Defines the properties of a document type:
  - Is the original content of the document preserved?
  - Is it necessary to index a catch-all field?
- Defines the ***properties*** for each field of a ***document type***:
  - Is the field stored?
  - Is the field analyzed and with which analyzer?
  - Is it text, date, number?
  - Is it boosted?
- Mapping is stored on index level
- An index can have multiple mappings

# Dynamic mapping



- **Dynamic mapping** is the feature of detecting field types while indexing even if there is no explicit mapping defined for them
- This feature allows you to index and search text even without any configuration
- Dynamic mapping can be disabled
  - Globally or in index settings

```
"index.mapper.dynamic": false
```
  - for a document type or subobject:

```
"dynamic": "false"
```

--- Or ---

```
"dynamic": "strict" //throws an exception on unknown encountered field
```
- When Elasticsearch detects a field type the defaults for this type are applied

*Dynamic mapping does not enable fine tuned searches...*



# Default mapping

- A default (empty) mapping is used when indexing documents which do not have an explicit mapping

```
{"_default_": {}}
```

- By default this mapping is entirely dynamic
  - Elasticsearch will automatically detect all field types of the indexed document
  - The **default** mapping, to put in ***mappings section***, allows you to define the index ***properties*** for certain fields (`_all` in the following sample):

```
{ "_default_": {  
    "dynamic_date_formats": ["yyyy-MM-dd", "dd-MM-yyyy", "date_optional_time"],  
    "_all": {  
        "enabled": false  
    }  
},  
"doctype1": {},  
"doctype2": {  
    "_all": {  
        "enabled": true  
    }  
}}
```



# Create a mapping

- Create a mapping ***at index creation time*** using the ***mappings*** section

```
PUT /library
{
  "settings" : {...},
  "mappings" : {
    "book" : {
      "properties" : { ... }
    }
  }
}
```

- Add a mapping to ***an existing index*** via the REST Put mapping API

```
PUT /library/_mapping/book
{ "properties" : { ... }}
```



# Delete a mapping

- It is not possible to delete a mapping for a document type.

*You have to suppress the whole index*



# Retrieving mappings definitions

- The REST **Get Mapping API** allows to retrieve all existing mappings

```
GET /_all/_mapping          //Gets all mappings from all existing indexes  
GET /_mapping               //Gets all mappings from all existing indexes  
  
GET /framasoft,zenika/_mapping //Gets all mappings from all specific indexes.  
                                //Here framasoft and zenika indexes  
  
GET /_all/_mapping/book,dvd   //Gets mappings for books and dvd from all  
                                //existing indexes
```



# Update a mapping

- The REST [Put mapping API](#) allows to update a mapping

```
PUT /library/_mapping/book  
{ "properties" : { ... }}
```

- Elasticsearch tries to merge the old mapping, or the one that was created by default, with the new one:
  - You can add a field (Including adding a **multi-field** field)
  - You can change the **ignore\_above** limit for a **text** field
  - You can disable the **norms** but you will not be able to reenable it
  - You can change **index\_options** values as you want
  - You **never can change** the **name and type** of a field

NB1: Fields in the same index with the same name in two different types must have the same mapping

NB2: The new mapping will only be used for future indexed documents: Documents already indexed are neither impacted nor reindexed!



# Mapping sample

```
{ "properties" : {  
    "title" : {  
        "type" : "text",  
        "index" : true  
    },  
    "price" : {  
        "type" : "float",  
        "null_value" : 1.0,  
        "store" : true  
    },  
    "language" : {  
        "type" : "keyword",  
        "store" : true,  
        "null_value" : "en_US"  
    }  
}
```

Mapping of a document of type **book** with 3 fields:

- The field **title** is of type **text** is **indexed and analyzed** but **not stored**
- The field **price** is of type **float, stored** and has a **default value of 1.0**
- The field **language** is a **keyword, stored** but **not analyzed**, with a **default value en\_US**



# Properties (1/2)

- The ***properties section*** defines all fields that compose the ***document type***
- Each property must have a ***type***:
  - ***core types*** :
    - ***Text*** or ***Keyword*** datatypes
    - ***Numeric datatypes***(double, float, half\_float, scaled\_float, long, integer, short, byte)
    - ***Date datatypes***
    - ***Range datatypes***(integer\_range,float\_range,long\_range,double\_range, date\_range)
    - ***Boolean datatypes***
    - ***Binary datatypes***
  - ***Complex types***:
    - ***Array, object*** and ***nested datatypes***
  - ***Geo datatypes***:
    - ***geo\_point*** and ***geoshape datatypes***



# Propertieses (2/2)

- Other types:
  - ***Specialized datatypes***:
    - ***ip datatypes***
    - ***Completion datatypes***, a way of implementing autocomplete
    - ***Token count datatypes***, count the number of tokens in a string
    - ***Mapper-Murmur3 datatypes***, plugin needed - compute and store hashes
    - ***Attachment datatypes***, ingest plugin needed - extract text from a lot of file formats
    - ***Percolator datatypes***, accepts JSON formatted query for later percolator queries



# Properties (3/3)

- Each property may also have ***configuration attributes***:
  - Is the field indexed?
  - Is the field stored?
  - What is the format?
  - What is the default value?
  - What analyzers are used for indexing?

# Core Types



- Common parameters for all core types

Attribute name	Possible value	Description
index (Binary excepted)	<b>true</b> , false	Allow to index the field to make it searchable or not
store	true, <b>false</b>	Is the original content stored?
include_in_all (Binary and Boolean excepted)	<b>true</b> , false	Include the content of the field in the <b>_all</b> field ? Defaults to false if index is set to no, or if a parent object field sets include_in_all to false. Otherwise defaults to true.
boost (Binary excepted)	1.0	Boost factor

# Core Type | Text (1/3)



- Field of type **text** are used for indexing and searching text content

Attribute name	Possible values	Description
analyzer	<b>default index analyzer</b> , or the <b>standard analyzer</b>	Analyzer used for indexing analyzes and searches if no <b>search_analyzer</b> specified
search_analyzer	<b>analyzer attribute setting</b>	Analyzer used for search analyzes
search_quote_analyzer	<b>search_analyzer attribute setting</b>	Analyzer used for phrase search analyzes
norms	<b>true</b> , false	Indicates if field-length should be taken into account when scoring queries. Consumes some disk. It can be disabled for fields that are only used for sorting and aggregations
index_options	<b>docs</b> (not for text), <b>freqs</b> , <b>positions</b> (text) or offsets	Additional index informations. Used for highlighting and phrase queries.
term_vector	<b>no</b> , yes, <b>with_offsets</b> , <b>with_positions</b> , <b>with_positions_offsets</b>	A term vector stores the list of all terms and their matching informations from the source in a document. Used for highlighting.
Similarity	<b>BM25</b> , classic, boolean	Scoring algorithm

# Core Type | Text (2/3)



- Field of type **text** are used for indexing and searching text content

Attribute name	Possible values	Description
fielddata	true, <b>false</b>	<p>Another storage structure to <b>answer</b> the question: "<b>What is the value of this field for this document?</b>", i.e it enable or facilitate sorting and aggregations.</p> <p><b>Memory and cpu intensive</b> as built at <b>query time</b>, use it with caution.</p> <p>Text is not compatible with on disk index time built <b>docvalues</b>...</p>
fielddata_frequency_filter	min, max, min_segment_size	Reduces the number of terms loaded into memory, thus reducing memory usage. Depends on Terms frequency in all indexed documents
eager_global_ordinals	true, <b>false</b>	A new datastructure build on top of fielddata or docvalues to track terms ordinals across an entire shard and not per segment to speed up sorting or terms aggregation. Can be low to build on demand so it can be eagerly computed on refresh time.



# Core Type | Text (3/3)

- Field of type **text** are used for indexing and searching text content

Attribute name	Possible values	Description
position_increment_gap	integer value <b>(100)</b>	A fictive position increment to prevent match_phrase requests to match different values in a multi-valued field. If the match_phrase request tolerance is greater than the <b>position_increment_gap</b> a match could be found overlapping two different initial values. <a href="#">Sample</a>

fields

An object defining fields which are derived from the same input but with different configurations. Elasticsearch documentation refers to it as **multi-fields**.





# Core Type | Keyword (1/2)

- Field of type **keyword** are used to index strings as a **single token**.

NB: A normalizer can change some little things like case or asciifolding ...

Attribute	Possible values	Description
name		
normalizer	Name of an analysis normalizer( <b>null</b> )	similar to analyzer except that it guarantees that the analysis chain produces a single token.
norms	true, <b>false</b>	Indicates if field-length should be taken into account when scoring queries. Consumes some disk. It can be disabled for fields that are only used for sorting and aggregations
index_options	<b>docs</b> , freqs	Additional index information. Used for highlighting and phrase queries.
Similarity	<b>BM25</b> , classic, boolean	Scoring algorithm

# Core Type | Keyword (2/2)



Attribute name	Possible values	Description
docvalues	<b>true</b> , false	Another storage structure to <b>answer</b> the question: " <b>What is the value of this field for this document?</b> ", i.e it enable or facilitate sorting and aggregations. Unlike fielddata, docvalues are disk stored and are computed at index time. No OutOfMemory and more less cpu usage than fielddata made it activated by default
eager_global_ordinals	true, <b>false</b>	A new datastructure build on top of fielddata or docvalues to track terms ordinals across an entire shard and not per segment to speed up sorting or terms aggregation. Can be low to build on demand so it can be eagerly computed on refresh time.
ignore_above	Integer value( <b>2147483647</b> )	Length limitation
null_value		Default value of the field
fields		An object defining fields which are derived from the same input but with different configurations. Elasticsearch documentation refers to it as <b>multi-fields</b> .





# Core Type / Numerics

- Some field types for indexing numbers:

***double, float, half\_float, scaled\_float, long, integer, short, byte***

Attribute name	Possible value	Description
coerce	<b>true</b> , false	Convert strings to numbers, remove decimal fraction for integers
ignore_malformed	true, <b>false</b>	
null_value		Default value of the field
docvalues	<b>true</b> , false	Another storage structure to <b>answer</b> the question: " <b>What is the value of this field for this document?</b> ", i.e it enable or facilitate sorting and aggregations. Unlike fielddata, docvalues are disk stored and are computed at index time. No OutOfMemory and more less cpu usage than fielddata made it activated by default

# Core Type | Date 1/3



- **Date** fields are used to index dates and times in UTC (Coordinated Universal Time)
- Format is configurable for each field

Attribute name	Possible value	Description
format	<code>strict_date_optional_time</code> or <code>epoch_millis</code>	Built-in or custom Joda time pattern
locale	<b>Root locale</b>	
ignore malformed	true, <b>false</b>	
null_value		Default value of the field
docvalues	<b>true</b> , false	Another storage structure to <b>answer</b> the question: " <b>What is the value of this field for this document?</b> ", i.e it enable or facilitate sorting and aggregations. Unlike fielddata, docvalues are disk stored and are computed at index time. No OutOfMemory and more less cpu usage than fielddata made it activated by default

# Core Type | Date 2/3



- The pattern for the date string is configured via the attribute **format**
- The format is defined using the following elements:

Key	Description
yyyy	Year (in 4 digits)
yy	Year (in 2 digits)
MM	Month
dd	Day of month
HH	Hours in 24 hour syntax (00 to 23)
hh	Hours in 12 hour syntax (00 to 12)
mm	Minutes
ss	Seconds

# Core Type | Date 3/3



- Example date formats
  - yyyy/MM/dd → 2012/12/09
  - dd/MM/yy → 09/12/2012
  - yyyy/MM/dd HH:mm:ss → 2012/12/09 10:46:59
- The double pipe (||) allows to define multiple formats that are tested one after the other
  - yyyy/MM/dd HH:mm:ss || yyyy/MM/dd → 2012/12/09 10:46:59, 2012/12/09
  - The first format defined is used when returning a date in the search results
- Predefined formats
  - ISO formats: `date_optional_time`, `date_time`, `date`
  - Basic formats (20050403T020100): `basic_date_time`, `basic_date`
  - Other: `year`, `epoch_millis`

# Core Type | Range



- A new way to indicate a field is a range of values deriving from some core types:
  - *integer\_range*,
  - *float\_range*,
  - *long\_range*,
  - *double\_range*,
  - *date\_range*
- With this mapping, *range queries* accept a new *relation* parameter:
  - within,
  - contains,
  - *intersects*

Attribute name	Possible value	Description
coerce	<i>true</i> , false	Convert strings to numbers, remove decimal fraction for integers



# Core Type | Boolean

- Fields of type **boolean** are used for indexing true or false values
- Accepts **false**, "false", **true** and "true" values

Attribute name	Possible value	Description
null_value		Default value of the field
docvalues	<b>true</b> , false	Another storage structure to <b>answer</b> the question: " <b>What is the value of this field for this document?</b> ", i.e it enable or facilitate sorting and aggregations. Unlike fielddata, docvalues are disk stored and are computed at index time. No OutOfMemory and more less cpu usage than fielddata made it activated by default



# Core Type | Binary

- Fields of type **binary** can be used to store binary data in **base64**
- This field type is exclusively for **storing data**, it **can't be indexed**

Attribute name	Possible value	Description
docvalues	<b>true</b> , false	Another storage structure to <b>answer</b> the question: " <b>What is the value of this field for this document?</b> ", i.e it enable or facilitate sorting and aggregations. Unlike fielddata, docvalues are disk stored and are computed at index time. No OutOfMemory and more less cpu usage than fielddata made it activated by default



# Complex fields

- Elasticsearch provides also some structural field types
  - ***object*** Allows to index a sub object
  - ***nested*** Similar to object but allows to conserve the relation between the fields of a sub object
  - ***geo\_point*** or ***geo\_shape*** Allows indexing a geo point or geo\_shape expressed in various ways based on latitude/longitude

# Complex fields | Object type 1/2



- Used to index sub objects in a container object
  - The sub objects are of type **object** and define their own **properties**
  - It is possible to omit the **object** type declaration, Elasticsearch detects the types of objects if a **properties** section is defined as a sub section of another object

Attribute name	Possible value	Description
dynamic	<b>true</b> , false, strict	Switch on/off dynamic mapping. An exception is thrown if you chose strict mode rather than just false
enabled	<b>true</b> , false	Whether the JSON value given for the object field should be parsed and indexed
include_in_all	<b>true</b> , false	Sets the default include_in_all value for all the properties within the object. The object itself is not added to the _all field.
properties		The sub object properties



# Complex fields | Object type 2/2

- An object sample:

```
{ "properties" : {  
  "title" : {  
    "type" : "text"  
  },  
  "authors" : {  
    "type" : "object",  
    "dynamic" : false,  
    "properties" : {  
      "name" : {"type" : "text"},  
      "firstname" : {"type" : "text"}  
    }  
  },  
  "editions" : {  
    "properties" : {  
      "isbn" : {  
        "type" : "keyword"  
      },  
      "publication" : {  
        "type" : "date",  
        "format" : "yyyy"  
      }  
    }  
  }  
}
```

# Complex fields | Multi valued fields or Arrays (1/2)



- Fields of type **object** or **core type** can have multiple values
- Elasticsearch detects automatically multiple values of a field, it is not necessary to declare it in the mapping
  - With this mapping:

```
{ "properties" : {  
    "title" : {  
        "type" : "text"  
    },  
    "authors" : { "type" : "text" },  
    "editions" : {  
        "properties" : {  
            "isbn" : { "type" : "keyword" },  
            "publication" : {  
                "type" : "date",  
                "format" : "yyyy"  
            }  
        }  
    }  
}
```

# Complex fields | Multi valued fields or Arrays (2/2)



- Both of these 2 documents are accepted
  - With multiple values for the fields **authors** and **editions**:

```
{  
  "title" : "Le rouge et le noir",  
  "authors" : [ "Stendhal", "Marie-Henri Beyle" ],  
  "editions" : [  
    {"isbn" : "1234", "publication" : "1830"},  
    {"isbn" : "5678", "publication" : "1903"}  
  ]  
}
```

- With single value for the fields **authors** and **editions**:

```
{  
  "title" : "Le rouge et le noir",  
  "authors" : "Marie-Henri Beyle",  
  "editions" : {  
    "isbn" : "9012",  
    "publication" : "1943"  
  }  
}
```



# Complex fields | Object arrays

- Object arrays can be used for organizing documents but the structure is lost during indexing

```
{  
  "title" : "Le rouge et le noir",  
  "editions" : [  
    {"isbn" : "1234", "publication" : "1830"},  
    {"isbn" : "5678", "publication" : "1903"}  
  ]  
}
```

is internally indexed like this:

```
{  
  "title" : "Le rouge et le noir",  
  "editions.isbn" : ["1234", "5678"],  
  "editions.publication" : ["1830", "1903"]  
}
```

- The following query shows the link between the **1830** publication date and the **1234** isbn is lost:

```
editions.isbn = 1234 AND editions.publication = 1903
```



# Complex fields | Nested types

- Solves the problem of ***object arrays*** regarding link coherence
- Declaration in mapping is similar to ***objects*** but can not be disabled
- Only ***searchable with specific queries***, see the advanced search module
- Nested files internal :
  - Indexed as separate documents in the same data structure. Can lead to ***mapping explosion*** and are then limited
  - Retrieved with the surrounding document



# Complex fields | geo\_point type

- **geo\_point** is used to index and search position information
- A **geo\_point** field can be indexed in multiple ways

```
{"pin" : {"location" : "drm3btev3e86"}},  
{"pin" : {"location" : [-71.34, 41.12]}},  
{"pin" : {"location" : "41.12,-71.34"}},  
{"pin" : {"location" : {"lat" : 41.12,"lon" : -71.34}}}
```



# Complex fields | Other field types

- ***ip*** Used for indexing IPv4 or IPv6 addresses
- ***geo\_shape*** For indexing shapes in GeoJSON format
- ***token\_count*** Counts the numbers of tokens in a field



# Multi field usage

- **fields** allows to index the value of a field in multiple ways from a single input

```
{ "book" : {  
    "properties" : {  
        "title" : {  
            "type" : "text",  
            "fields" : {  
                "keyword" : {  
                    "type" : "keyword"  
                }  
            }  
        }  
    }  
}
```

- Produces multiple fields in Lucene
  - Field **title**: analyzed with the default analyzer
  - Field **title.keyword**: can be analyzed with a **normalizer**, indexed **as a single token**
- Used for instance when it is necessary to sort and aggregate on a field that is also used for search



# Concatenating fields contents

- ***copy\_to*** Copies the content of one or more fields to another field to flatten terms in a chosen field

```
{  
  "book" : {  
    "properties" : {  
      "title" : { "type" : "text", "copy_to" : "description" },  
      "abstract" : { "type" : "text", "copy_to" : "description" },  
      "description" : { "type" : "text" }  
    }  
  }  
}
```

# Root Object Mapping



- Object describing the mapping, contains
  - ***properties***: static field mapping
  - ***dynamic\_date\_formats*, *dynamic\_templates***...: dynamic field mapping
  - ***\_all***...: and other pseudo fields

```
{ "book" : {  
    "dynamic_date_formats" : ["yyyy-MM-dd", "dd-MM-yyyy"],  
    "_all" : {  
        "enabled" : true  
    },  
    "properties" : {  
        "title" : {"type" : "text"},  
        "price" : {  
            "type" : "float",  
            "null_value" : 1.0  
        },  
        "language" : {  
            "type" : "text",  
            "null_value" : "en_US"},  
        ... }  
    } }
```

# Root Object Mapping | Dynamic mapping



- **root object mapping** is used for configuring certain mapping options of a document type

Attribute name	Possible values	Description
dynamic_date_formats	yyyy/MM/dd HH:mm:ss Z yyyy/MM/dd Z	Allows to configure the date format used
date_detection	<b>true</b> false	Allows to deactivate the automatic date detection
numeric_detection	<b>false</b> true	Allows to activate the detection of numeric fields in string values
dynamic_templates		Allows to declare <b>mapping templates</b> for field types

# Root Object Mapping | pseudo field `_all`



- The property `_all` activate/déactivate a catch-all field to include/exclude values from almost all fields in the document field named `_all`
  - Activated by default, includes all fields
  - Works like a generalized `copy_to` field
  - Possible to configure a field `index.query.default_field` that is queried when `_all` is disabled
  - Conserves `boost` values of the fields
- Useful when searching without knowing which fields are there or contain the information(Full text search)

```
{ "type1" : {  
    "_all" : {"enabled" : true},  
    "include_in_all" : false,  
    "properties" : {  
        "title" : {"type" : "text"},  
        "included_field" : {"type" : "long", "include_in_all" : true },  
        "excluded_field" : {"type" : "long", "include_in_all" : false}  
    }  
}
```



# Root Object Mapping | pseudo field \_source

- By default, Elasticsearch stores the JSON content of indexed documents in the field `_source`

```
{ "type1" : {  
    "_source" : {  
        "enabled" : true  
    }  
}
```

- The field is returned for search requests and is used for different functionality:  
***Update API, Highlighting, Reindexing, field extraction, ...***
- It is possible to exclude fields that ***could be indexed*** but ***should not be stored*** in `_source`

```
{ "type1" : {  
    "_source" : {  
        "includes" : ["field1.*", "field2.*"],  
        "excludes" : ["field3.*"]  
    }  
}
```





# Root Object Mapping | pseudo field `_size`

- The property `_size` allows to index the byte size of the `_source` in a field
- Deactivated by default, requires installation of the plugin **`mapper-size`**
- Useful to sort documents by size

```
{ "type1" : {  
    "_size" : {"enabled" : true}  
}  
}
```

- Doesn't store the size of the compressed content but of the JSON original



# Root Object Mapping | pseudo field `_meta`

- The `_meta` field allow to store, additional free format information on the mapping level
- Can be used for:
  - store additonal business information
  - store a meta model of the serialization Document ↔ Java Object
  - and so on...

```
{ "book" : {  
    "_meta" : {  
        "date_last_modified" : "2012-08-13",  
        "model" : {"price" : "java.lang.Float"}  
    }  
}
```

# Root Object Mapping | Other pseudo fields



- These pseudo fields are not configurable but are used in requests, scripts, ...
  - **\_index**: index of document
  - **\_type**: type of document
  - **\_id**: Id of the document (sorting and aggregations not possible)
  - **\_uid**: type + id of the document document, (id used in Lucene)
- Other pseudo fields
  - **\_parent**: defines the type of the parent document (parent child relations)
  - **\_routing**: custom shard routing

With custom `_routing` the document distribution on shards is not necessarily homogenous anymore





# Analyzing and Extracting text



# The basics of text analysis 1/3

- Text analysis is the process that generates from a given text a set of tokens

```
<quote>The straw that broke the camel's  
back</quote>  
→ straw broke camel back
```

- This process is defined via an **Analyzer**
- The generated set of tokens depends of the configuration of the analyzer that is being used to analyze the text



# The basics of text analysis 2/3

- Analyzers are used at two different moments:
  - **Indexing** time: the text of the document is analyzed and the produced tokens are indexed
  - **Search** time: the searched keywords are analyzed and search is performed using the produced tokens
- An Analyzer has a logical name and is composed of a **Tokenizer** and one or more **CharFilter** and/or **TokenFilter**
- Elasticsearch provides several pre-configured analyzers



# The basics of text analysis 3/3

- **Analyzer**
  - Allows to configure an analysis process which can be referenced by a logical name in the mappings
  - Is composed of a Tokenizer and one or more CharFilter and/or TokenFilters
- **CharFilter**
  - First step of the text analysis (optional)
  - Allows to filter/modify the text before the tokenizer phase
- **Tokenizer**
  - Second step of the text analysis
  - Generate tokens from a text
- **TokenFilter**
  - Following steps of the text analysis
  - Allows to modify, delete or add tokens
  - Executed one after the other: the result produced by the 1st TokenFilter is the input of the 2nd TokenFilter, etc.

# CharFilter



**Character filter** that is capable of filtering or modifying the text before it is processed by a Tokenizer.

- **HTML Strip**
  - Remove some HTML tags from a text
- **Mapping**
  - Replace some characters by others:
    - ph → f
    - qu → k
    - & → and

# Tokenizer 1/3



Generate a token list from a text

- **Whitespace**: Generates tokens by splitting the text at every space character encountered

*"That's the real life!"* → "That's", "the", "real", "life!"

- **Letter**: Splits the text at every non alphabetic character
  - Uses `java.lang.Character.isLetter()`

*"That's the real life!"* → "That", "s", "the", "real", "life"

- **Pattern**: Splits the text using a regular expression

*"ISBN:123456, Part:AZS69"* → "123456" "AZS69"



# Tokenizer 2/3

- **Keyword:** Handles the whole text as a single token

"ISBN-123456" → "ISBN-123456"

- **NGram:** Generates tokens of adjacent letters, according to a specified range

- By default min\_gram = 1 and max\_gram = 2

"Bonjour" → "B" "o" "n" "j" "o" "u" "r" "Bo" "on" "nj"  
"jo" "ou" "ur"

- **Edge Ngram:** Similar to NGram but generate tokens from the start

"Bonjour" → "B" "Bo" "Bon" "Bonj" "Bonjo" "Bonjou"  
"Bonjour"



# Tokenizer 3/3

- **Standard:** Splits the text according to the Unicode standard  
<http://unicode.org/reports/tr29/>

- Adapted to western languages

*"C'est la vie!"* → "C'est" "la" "vie"

- **Lowercase:** Tokenizes according to the Letter Tokenizer and converts to lowercase

- **Path Hierarchy:** Creates tokens in a progressive way

*"/home/Zenika/docs"* → "/home" "/home/Zenika"  
"/home/Zenika/docs"

- **UAX Email URL:** Similar to Standard, but keeps e-mail addresses and URLs



# Token Filter 1/4

Add, delete or modify the tokens

- **ASCII Folding:** Replaces accentuated characters with their non accentuated ASCII counterpart

*Généreux à souhait* → *Genereux a souhait*

- **Lowercase:** Transforms the tokens to lowercase
- **Stop:** Deletes the tokens that belong to a list of forbidden words

*The articles to be deleted* → *articles deleted*

- **Length:** Deletes tokens that are too short or too long



# Token Filter 2/4

- **Snowball / Stemmer / KStem**: Removes the plural and conjugate forms of the tokens

*Failing tests → fail test*

- **Synonym**: Allows to inject synonyms at the token level
- **Ngram / Edge Ngram**
  - Similar the tokenizer with the same name
- **Word Delimiter**: Handling for hyphen, camelcase and digits

*Wi-Fi SD500 → Wi Fi WiFi SD 500*



# Token Filter 3/4

- **Phonetic**: Calculates a phonetic representation of each token
  - Several algorithms are available
  - Available as plugins
- **Reverse**: Inverses the order of the letters in the tokens
  - Used with Edge Ngram to generate tokens from the end
- **Truncate**: Truncates tokens from a configurable length
- **Elision**: Removes articles followed by an apostrophe
  - Article list is configurable

*C'est la vie* → *est la vie*



# Token Filter 4/4

- **Unique**: Remove duplicated tokens
- **Pattern Replace**
  - Replace tokens according to a regular expression
- **Shingle**: Build combinations of tokens from several tokens
  - "So let it be" → "So let" "let it" "it be"
- **Compound Word**: Identify compound words and split them
  - "snowboard" → "snow" "board"



# Pre-configured analyzers 1/2

Elasticsearch comes with several pre-configured analyzers

- **Standard**: Based on
  - Tokenizer `standard`
  - TokenFilter `standard`,`lowercase` and `stop` (with an empty list)
- **Simple**: Based on `lowercase` Tokenizer
- **Whitespace**: Based on `whitespace` Tokenizer
- **Stop**: Based on `lowercase` Tokenizer and `stop` TokenFilter



# Pre-configured analyzers 2/2

- Analyzers present by default for a lot of languages
  - french, english, german, greek, armenian, russian, thai ...
- Same base schema :
  - **standard** Tokenizer
  - Lowercasing (TokenFilter **lowercase**)
  - Removal of "stop words" (TokenFilter **stop**)
  - list specific for each language
  - Stemming
- Example of language specifics:
  - removal of '**s**' in English possessive form
  - **elision** to remove French articles
  - normalize special characters (such as ß, æ ...)

# Configuring analyzers



```
PUT /zenika/
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_analyzer": {
          "type": "custom",
          "tokenizer": "my_tokenizer",
          "filter": ["lowercase", "my_token_filter"],
          "char_filter": ["my_char_filter"]}},
      "tokenizer": {
        "my_tokenizer": {
          "type": "whitespace",
          "max_token_length": 255}},
      "filter": {
        "my_token_filter": {
          "type": "stop",
          "stopwords": ["la", "le", "les", "va", "a", "l"]}},
      "char_filter": {
        "my_char_filter": {
          "type": "html_strip",
          "escaped_tags": ["<b>", "<quote>"]}}}}}
```



# Updating analyzers

- Impossible to update analyzers and token filters **when the index is being used** (hot update)
- Procedure :
  - The index must be **closed** via the Close API

```
POST /zenika/_close
```

- **stop read/write operations on the index**
- keeps data on disk but remove them from heap
- alternative to simply deleting an index

- Update the index **analysis** definition using the API PUT Settings
- Open the index with the API Open

```
POST /zenika/_open
```

- **Warning :** if data is not reindexed, there can be invalid results (different analyzers used at index time after the update)

# Using analyzers 1/2



- The analyzers to be used for each field of type text for a document are declared in the mappings
- The **analyzer** property allows to declare an analyzer for indexing and search
- The **search\_analyzer** allows to declare a different analyzer for search
- If nothing is specified the default analyzer is used



# Using analyzers 2/2

- Example of mapping using analyzers

```
PUT /zenika/dvd/_mapping
{"dvd": {
  "properties": {
    "title": {
      "type": "text",
      "analyzer": "my_analyzer", "search_analyzer": "standard"},

    "category": {
      "type": "text",
      "analyzer": "my_analyzer" },

    "author": {
      "type": "text" }}}}
```



# Example



- Let's go back to the example presented at the beginning of this chapter

*<quote>The straw that broke the camel's  
back</quote>*

→ *straw broke camel back*

- Which combination can be used to obtain the expected result?

# REST API Analyze 1/4



- The "Analyze" REST API allows to test the analysis of a text
  - Testing an already configured analyzer
  - Testing by specifying which Tokenizer and TokenFilters to use
  - Testing an analyzer associated to a field of a document
- Very useful when defining analyzers and mappings
- Example testing the "standard" analyzer

```
POST _analyze
{
  "analyzer": "standard",
  "text": ["C'est la vie!"]
}
```

- The JSON response gives the generated tokens

```
{ "tokens": [
    {"token": "c'est", "start_offset": 0, "end_offset": 5, "position": 0, "type": "<ALPHANUM>"},
    {"token": "la", "start_offset": 6, "end_offset": 8, "position": 1, "type": "<ALPHANUM>"},
    {"token": "vie", "start_offset": 9, "end_offset": 12, "position": 2, "type": "<ALPHANUM>"}]
```

# REST API Analyze 2/4



- Example of analysis test by providing Tokenizer and TokenFilters to use in JSON

```
GET /_analyze
{
  "tokenizer": "keyword",
  "filter": ["lowercase", "asciifolding"],
  "text": "Généreux à souhait"
}
```

## Response

```
{"tokens": [
  {"token": "genereux a souhait",
   "start_offset": 0, "end_offset": 18, "type": "word", "position": 0}]}]
```

- Example on a document field analyzer

```
GET /zenika/_analyze
{
  "field": "title",
  "text": "7 years"
}
```

# REST API Analyze 3/4



- Example of analysis testing with a personalized filter

```
GET /_analyze
{
  "tokenizer": "whitespace",
  "filter": ["lowercase", {"type": "stop", "stopwords": ["a", "is", "this"]}],
  "text": "this is a Test"
}
```

Response

```
{"tokens": [
  {
    "token": "test",
    "start_offset": 10,
    "end_offset": 14,
    "type": "word",
    "position": 3
}]}
```



# REST API Analyze 4/4 | Explain

- Example with explanation for each step

```
GET /_analyze
{
  "tokenizer": "keyword",
  "filter": ["lowercase", "asciifolding"],
  "text": "Généreux à souhait",
  "explain": "true"
}
```

Réponse

```
{"detail": {
  "charfilters": [],
  "tokenizer": {
    "name": "keyword",
    "tokens": [{"token": "Généreux à souhait", ... }]},
  "tokenfilters": [
    {
      "name": "lowercase",
      "tokens": [{"token": "généreux à souhait", ... }]},
    {
      "name": "asciifolding",
      "tokens": [{"token": "genereux a souhait", ... }]}]}
```



# Use case | Deleting words



- Some documents contain a lot of words with no real meaning (articles, pronouns, be, have, ...)
- These words can be ignored/deleted
  - Slightly reduce the index size
  - Improve search performances
  - Impact on relevancy!
- The **Stop** filter allows to ignore a set of words
  - Pre-configured set of words ([french\\_stop.txt](#)) is usually good enough
  - Can be completed with business terms



# Use case | Deleting words

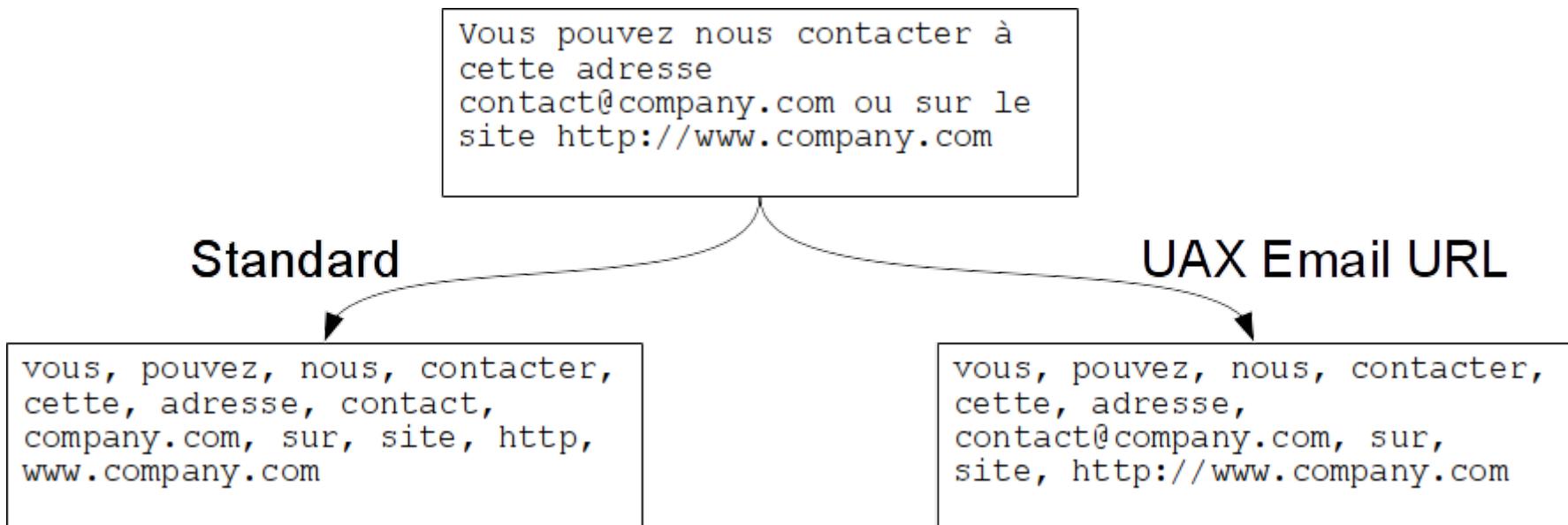
- Example using the **stop** filter

```
PUT /zenika
{
  "analysis": {
    "analyzer": {
      "my_analyzer": {
        "type": "custom",
        "tokenizer": "whitespace",
        "filter": ["filter_stop"]}},
    "filter": {
      "filter_stop": {
        "type": "stop",
        "stopwords": ["_french_"],
        "ignore_case": "true"}}}}
```

# Use case | E-mails & URLs



- **E-mail** addresses and **URLs** are not recognized by standard or letter tokenizers
- The **UAX Email URL** works like a standard tokenizer but preserve e-mail addresses and URLs





# Use case | E-mails & URLs

- Example using the **UAX Email Url** tokenizer

```
PUT /zenika
{
  "analysis": {
    "analyzer": {
      "my_analyzer": {
        "type": "custom",
        "tokenizer": "uax_url_email"
      }
    }
  }
}
```



# Use case | Removing HTML code

- When indexing HTML document, it can be useful to remove the numerous HTML tags
  - Reduce index size
  - Improve relevance
- To remove the HTML tags, a **HTML Strip** CharFilter is used
  - Processing done before tokenizer
  - The filter recognize all the HTML tags

```
{ "analysis": {  
    "analyzer": {  
        "my_analyzer": {  
            "type": "custom",  
            "char_filter": [ "html_strip" ],  
            "tokenizer": "standard",  
            "filter": [ "standard", "lowercase", "stop" ] } } } }
```

# Use case | Stemming



- **Stemming** is the process that transform a word to its root or main part, inflections or formative elements being removed.
- For French there are several alorithms and TokenFilters
  - The default stemmer is the Martin Porter's Snowball
    - Produces numerous stems, but also some errors ("overstemming")
      - <http://snowball.tartarus.org/texts/introduction.html>
      - <http://www.lirmm.fr/~mroche/Recherche/Articles/Porter/porter.pdf>
  - Minimalist stemmer based on an algorithm from Jacques Savoy
    - Uses word dictionaries <http://members.unine.ch/jacques.savoy/papers/frjasis.pdf>
  - Stemmer applicable to several languages (French, Portuguese, Hungarian, German), based on another algorithm from Jacques Savoy
    - Uses statistic methods to produce less stems with better quality
      - [http://doc.rero.ch/lm.php?url=1000,43,4,20091216091830-OY/Savoy\\_Jacques\\_-\\_Light\\_Stemming\\_Approaches\\_for\\_the\\_French\\_20091216.pdf](http://doc.rero.ch/lm.php?url=1000,43,4,20091216091830-OY/Savoy_Jacques_-_Light_Stemming_Approaches_for_the_French_20091216.pdf)

# Use case | Multi-language documents



- Often necessary to manage multiple languages
- Problems:
  - Relevancy is "polluted" (incorrect *idf*)
  - Incorrect stemming
    - rules specific to each language
- Several cases depending of the documents organization:
  - Case 1: one language per document
  - Case 2: one language per field
  - Case 3: several languages per field

# Use case | Multi-language documents (Case 1)



- Case for **single principal language for the document**
- Create an index per language, with same types and mappings
- Simply modify the analyzers being used

```
POST /zenika-fr
{ "mappings": {
  "book": {
    "properties": {
      "title": {"type": "string", "analyzer": "french"}}}}}
```

```
POST /zenika-en
{ "mappings": {
  "book": {
    "properties": {
      "title": {"type": "string", "analyzer": "english"}}}}}
```



# Use case | Multi-language documents (Case 1)

- Simple case to handle:
  - add new languages = create associated index
  - request on a specific language
    - ... or on several languages at the same time
- No relevancy problem

```
GET /zenika-en/book/_search?q=title:spring
```

- ... or on several languages at the same time

```
GET /zenika-*/book/_search?q=title:spring
```

# Use case | Multi-language documents (Case 2)



- Case where **several languages are present separately** in the same document
  - Example: different translations of the same field
- **Solution:** a field per language with the associated analyzer

```
POST /zenika
{
  "mappings": {
    "book": {
      "properties": {
        "title": {"type": "string", "analyzer": "french"},
        "title_en": {"type": "string", "analyzer": "english"},
        "title_de": {"type": "string", "analyzer": "german"}}}}}
```

- Relevancy problem (***idf*** preserved)
- Add a language
  - At least re-index the documents
  - If custom analyzer, downtime is required to:
    - close the index
    - modify index settings
    - reopen the index

# Use case | Multi-language documents (Case 3)



- Case where **several languages are in the same field** (complex)
- Three solutions:
  - Beforehand, separate in several fields:
    - Google **Compact Language Detector**
  - Analyze several times (with different analyzers)

```
POST /zenika
{
  "mappings": {
    "book": {
      "properties": {
        "title": {
          "type": "string",
          "fields": {
            "fr": { "type": "string", "analyzer": "french" },
            "en": { "type": "string", "analyzer": "english" },
            "de": { "type": "string", "analyzer": "german" }}}},
      "analyzer": "standard"
    }
  }
}
```

- Use n-grams
  - low level approach





# Searching documents



# Searching using the REST API 1/2

- The RESTFull **Search API** allows to search documents
  - in one, several, or all the indices (**multi-index**)
  - on one, several, or all the document types (**multi-types**)
  - with one or several search queries (**multi-search**)
- Usage

```
POST /index(s)/type(s)/_search
{
  "query" : {...}
}
```

- Search query body must be written in **JSON**
- The **query** field holds the search query
- <https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html>



# Searching using the REST API 2/2

- There is a shortcut to perform text searches:

```
GET /index(s)/type(s)/_search?q=titre:Misérables
```

- Simpler, useful to quickly test a search
- Accepts several parameters:
  - **q** : search query (of type **query\_string**)
  - **df** : default field for search
  - **explain** : display information about the relevance calculation
  - **from,size,stored\_fields,sort** fields



# Multi-index search

--- Search on an index and a document type

POST /zenika/book/\_search

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

--- Search on several indices and one document type

POST /zenika1,zenika2/book/\_search

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

--- Search on all the indices and one document type

POST /\_all/book/\_search

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



# Multi-index search using wildcards

--- Search on indices that start with "zenika"

```
POST /zenika*/_search
{
  "query": {
    "match_all": {}
  }
}
```

--- Possibility to include/exclude index name patterns

```
POST /+test*,!-test1/_search
{
  "query": {
    "match_all": {}
  }
}
```

- It is also possible to control the expansion behaviour with boolean parameters:

- **ignore\_unavailable**: Do we accept to ignore closed matching indices ?
- **allow\_no\_indices**: Do we accept to ignore no matching index?



# Multi-type search

--- Search on one index and several document types  
POST /zenika/book,dvd/\_search

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

--- Search on several indices and several document types  
POST /zenika1,zenika2/book,dvd/\_search

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

--- Search on all the indices and all the document types  
POST /\_all/\_search

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

# Search results



- The response to a search query is formatted in JSON

```
{ "_shards": { "failed": 0, "successful": 5, "total": 5},
  "hits": {
    "hits": [
      { "_index": "zenika", "_type": "book", "_id": "sIMPUEKOSNqmus6pnXzUag",
        "_score": 1.0,
        "_source": {
          "content": "Les Misérables...", ...}, { ... }]
      "max_score": 1.0,
      "total": 3
    },
    "timed_out": false,
    "took": 1}
```

- \_shards** is the number of queried shards
- took** is the query execution time in ms
- hits.total** is the total number of results without pagination
- hits.max\_score** is the max relevancy
- hits.hits** is the results array
- hits.hits[0].\_score** is one result relative relevancy



# Pagination

- It is possible to control the number of results fetched

```
POST /index(s)/type(s)/_search?from=0&size=100
{
  "query": {...}
}
```

- from** : position of the first result to be returned (0 by default)
- size** : number of results to be returned (10 by default)
- from + size < 10000
- For a very big number of results, prefer the **scroll API**:
  - Similar to the way a database cursor works
  - Request parameter **scroll** = duration to keep the results (e.g 1m)
  - Response contains the first results and a **scroll\_id**
  - To get next results, execute the query with the received **scroll\_id**
  - Faster when search is **\_doc** sorted

# Scroll pagination



- Scroll allows to keep a result for a given duration, **index state is frozen** for that query and no update is taken into account
- It allows to get back to a result without running the query again
- The result is stored **in memory** in the cluster

```
--- Create the scroll
POST /zenika/book/_search?scroll=10m
{
  "size" : 10,
  "query": {"query_string": {"query": "title:hunger"}}
}
--- Result -----
{
  "_scroll_id": "cXVlcnlUaGVuRmV0Y2g7NTsxMDE4MD00...",
  "hits": { ... }
}

--- Next call
GET _search/scroll?scroll=10m&scroll_id=cXVlcnlUaGVuRmV0Y2g7NTsxMDE4MD00...

--- Clear scroll to free cluster resources
DELETE /_search/scroll
{
  "scroll_id" : ["cXVlcnlUaGVuRmV0Y2g7NTsxMDE4MD00..."]
}
```

# Result structure



- It is possible to specify the fields to be returned in the result:

```
--- Use store_fields when fields to retrieve are stored
```

```
POST /zenika/book/_search
```

```
{ "query": {  
    "match_all": {}  
},  
  "stored_fields": ["title", "lang"]  
}
```

```
--- Use _source to extract field values when not stored and _source is enabled
```

```
--- Potentially cpu intensive with large paginations
```

```
POST /zenika/book/_search
```

```
{ "query": {  
    "match_all": {}  
},  
  "_source": ["title", "lang"]  
}
```

Depending on the mapping, specifying fields can reduce search performances (multiple disk accesses)



# Sorting the results 1/2

- By default, the results are sorted by ascending relevancy
- It is possible to sort on one or several fields

```
POST /zenika/book/_search
{
  "query" : {"match_all" : {}},
  "sort" : [
    { "title.keyword" : "desc" },
    { "price" : {"order" : "asc", "missing" : "_last"} },
    { "category" : {"unmapped_type" : "string"} },
    "_score"
  ]
}
```

- The **missing** option specifies how to treat the documents not having the expected field. Put them **\_first** or **\_last**?
- The **unmapped\_type** option prevents errors when searching a **non mapped field**. It gives the data type to use to handle this field default value to proceed with when sorting.



# Sorting the results 2/2

Sorting on a field can consume **a lot** of **memory** because all the field values on the results are loaded in memory. Analyzed fields need to have **fielddata** enabled to be sortable.

- Do not sort on Text fields unless vital ! Behavior may lead to bad performances and strange results...

Warning: Once a query has a sorting section, **\_score** won't appear in the results unless you ask to track scores

```
--- To have non null scores and max score, specify "track_scores": true
{
  "query" : { ... },
  "sort": [{"lang": "asc"}],
  "track_scores": true
}
```

```
--- To control multiple ordering with _score, specify a _score sorting axis
{
  "query" : { ... },
  "sort": [{"lang": "asc"}, {"_score": "asc"}],
  "track_scores": true
}
```



# Counting the results

- The RESTFull **\_count API** has been removed and replaced with the **hits.total** section in the query results

--- If not interested in results by themselves, put a size option to **0**

```
{  
  "query": {  
    "term": {  
      ...  
    }  
  },  
  "size": 0  
}
```

# The Queries





# Search language

- Elasticsearch has its own search request definition language: **Query DSL**
  - This language is written in **JSON**
  - It is passed on the request **query** field
- General structure :
  - The language defines search operations
  - Each operation has its own parameters
  - Some operations allow to combine other operations

```
{ "query": {  
  "search_operation_x": {"search_parameter_x": "value"...},  
  "search_operation_y": {"search_parameter_x": "value"...}...}  
}
```



# Request types

- **match\_all**
- Fulltext: **match**, **match\_phrase**, **multi\_match**, **query\_string**...
- Terms: **term**, **range**, **exists**, **prefix**, **wildcard**, **regexp**, **fuzzy**...
- Composed: **bool**, **not**...
- Join: **nested**, **has\_child**, **has\_parent**
- Spatial: **geo\_distance**, **geo\_bounding\_box**...
- Proximity, sentences: **span\_near**, **span\_term**...
- Scoring: **constant\_score**, **function\_score**, **boosting**...



# Query | match\_all

- Search and returns all the results for selected indexes and types
- This is the default query when no query section is provided

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



# Query | query\_string (1/2)

- Lucene fulltext search

```
{ "query": {  
    "query_string": {  
        "fields": ["content", "title^2", "author.*"],  
        "query": "Victor AND Hugo OR 1862"  
    }  
}
```

- **Strict** syntax
- Options
  - A lot of available options. Take a look at the online [documentation](#)
  - Search is by default performed on the **\_all** field
  - Specify the fields on which the search should be performed using **fields**
  - **Query parameter** is automatically parsed to build a search query from:
    - optional field names,
    - terms,
    - keywords like OR, AND, ...
  - Analyzers are detected and applied at search time





# Query | query\_string (2/2)

- Other valid syntaxes

```
POST library/book/_search
{
  "query": {
    "query_string": {
      "fields": ["content", "title^2", "author.*"],
      "query": "Victor AND Hugo OR 1862"
    }
  }
}
-----
POST library/book/_search
{
  "query": {
    "query_string": {
      "query": "(title:spring OR action) AND authors:dubois"
    }
  }
}
-----
GET library/book/_search?q=title:spring OR guillaume
```





# Query | simple\_query\_string

- simple\_query\_string is similar to query\_string and simpler

```
{ "query": {  
    "simple_query_string": {  
        "fields": ["content", "title^2", "author.*"],  
        "query": "Victor AND Hugo OR 1862"  
    }  
}
```

- Less strict on the syntax, simply **ignore** the tokens that are **invalid**

Not recommended as errors would be hidden



# Query | match

- **Match** is a full-text search

```
{ "query": {  
    "match": {"author": "Victor Hugo"}  
}
```

- Analyzers are **applied**
- Search keywords are **not parsed** and this is very much less error prone
- A lot of available options. Take a look at the online **documentation**
- Preferable to **query\_string**



# Query | match

- By default, a document matches if at least one of the term is present
- Behaviour can be modified by setting the **operator** parameter to **and|or**
- Behaviour can be also modified by setting the **fuzziness** parameter
- Complete syntax:

```
{  
  "query": {  
    "match": {  
      "author": {  
        "query": "Victor Hugo",  
        "operator": "and"  
      }  
    }  
  }  
}
```



# Query | match\_phrase

- **Match\_phrase** is a variation of the **match** query

```
{  
  "query": {  
    "match_phrase": {  
      "author": "Victor Hugo"  
    }  
  }  
}
```

- The **positions** of the terms must be **respected**
- A flexibility can be introduced using the **slope** parameter (**0** by default):
  - Indicates the **number of moves** that are **allowed** for the terms.



# Query | match\_phrase\_prefix

- **Match\_phrase\_prefix** is similar to the **match\_phrase** request

```
{  
  "query": {  
    "match_phrase_prefix": {  
      "author": "Victor Hu"  
    }  
  }  
}
```

- The **last** term is used as a prefix to search among the terms
- Can be used as **quick-and-dirty** autocomplete
  - Pros: nothing to prepare beforehand (no need to reindex)
  - Cons: the least efficient solution



# Query | multi\_match

- **Multi\_match** is the same as a **Match** query on several fields

```
{ "multi_match" : {  
    "query" : "Victor Hugo",  
    "fields" : [ "author", "description" ]  
}  
}
```

- Allows to perform a **match** query on several fields without requiring to use boolean queries
- Uses a Lucene **dis\_max** query by default
  - Returns the document that match at least one of the sub-query
  - The score is the maximum score of the sub-queries
- Several **strategies** are possible:
  - **best\_fields** (default)
  - **most\_fields**
  - **cross\_fields**
  - **phrase**
  - **phrase\_prefix**



# Query | term

- **Term** is an exact matching operator and is not a full-text search

```
{ "query": {  
    "term": {"lang": "fr"}  
}
```

- **No analyzer** is applied
- Options:
  - Possibility to **boost** a result compared to another one
  - Several **term** queries can be combined in a **terms** query



# Query | prefix

- **Prefix** searches documents starting with a term:

```
{ "query": {  
    "prefix": {  
        "content": "rom"  
    }  
}
```

- **No analyzer** is applied

**Slows** search queries to a large degree(autocomplete is a better way)



# Query | wildcard

- **Wildcard** Search for documents containing an expression:

```
{ "query": {  
    "wildcard": {  
        "title": "du?l*"}}}
```

- \* character to match zero, one or several characters
- Question mark ? to match a single character
- **No analyzer is applied**

**Slows** search queries to a large degree(autocomplete is a better way)

**Never** starts the expression with \* or ?



# Query | range 1/3

- **range** query searches documents that are within two boundaries:

```
{  
  "query" : {  
    "range" : {  
      "price" : {  
        "gte" : 50,  
        "lt" : 200  
      }  
    }  
  }  
}
```

- Applicable to dates, numbers and strings
- Boundaries can be exclusive or inclusive:
  - gt (>) / gte ( $\geq$ )
  - lt (<) / lte ( $\leq$ )
- See example above: price between 50 (inclusive) and 200 (exclusive)



# Query | range 2/3

- **range** query is also applicable to dates

```
{ "query": {  
    "range": {  
        "created": {  
            "gte" : "2015-12-15"}}}}  
  
{ "query": {  
    "range": {  
        "created": {  
            "gte" : "now-1d/d",  
            "lt" : "now/d" }}}}
```

- Supported time unites are: **y** (year), **M** (month), **w** (week), **d** (day), **h** (hour), **m** (minute), and **s** (second).
- **+1d** and **-1d** mean plus or minus one unit of type day.
- **/d** mean unit rounded to its lower bound



# Query | range 3/3

- range query is also applicable to the new range fields

```
--- With this document
```

```
{  
  "event_name" : "Special offer",  
  "time_frame" : {  
    "gte" : "2015-10-31 12:00:00",  
    "lte" : "2015-11-01"  
  }  
}
```

```
--- You can make this kind of range request where relation can be set to  
--- 'within', 'contains' or 'intersects'(default)
```

```
{  
  "query" : {  
    "range" : {  
      "time_frame" : {  
        "gte" : "2015-10-31",  
        "lte" : "2015-11-01",  
        "relation" : "within"  
      }  
    }  
  }  
}
```

# Fuzzy search 1/2



- Search by **similarity**
- Calculated using the **Damerau-Levenshtein distance**
  - Minimal number of operations to transform a word **W1** to another **W2** :
    - substitution of one character from **W1** in one character of **W2** (same position)
    - inserting in **M1** a character from **W2**
    - removing a character from **W1**
    - permutation of two **W1** and **W2** adjacent characters
- Example: Levenshtein distance between **avicto** et **victor**

‘avicto’ => ‘victo’: removal of 1st character

‘victo’ => ‘victor’: insertion of 6th character

- As a result: distance = 2



# Fuzzy search 2/2

- **fuzziness** parameter can be used in several query types:  
`match, multi_match, fuzzy ...`
- Possible values:
  - distance between 0 and 2
  - **AUTO** (default value)
    - length < 3: distance = 0
    - length between 3 and 5: distance = 1
    - length > 5: distance = 2

The maximum possible distance is **2**, higher values are ignored

Removed support for fuzzy queries on numeric, date and ip fields, use range queries instead



# Query | fuzzy

- **Fuzzy** query is similar to the **term** query with fuzziness added

```
{ "query": {  
    "fuzzy": {  
        "author": {  
            "value": "vitcor",  
            "fuzziness": 2,  
            "prefix_length": 1  
        }  
    }  
}
```

- **No analyzer** is **applied**
- Parameters having an impact on the performances:
  - **prefix\_length**: numbers of characters to remain unchanged at the begining of the term
  - **max\_expansions** : max number of generated terms
- In general, a **match** query with **fuzziness** is simpler

This query can be very heavy if prefix\_length is set to 0 and if max\_expansions is set to a high number.

# Query | bool 1/2



- **bool** query allows to combine several queries
- Requests are combined by using 4 subsections
  - **must**: the document have to be matched by the query
    - implicit **and** operator between subqueries
  - **filter**: like must clause without scoring. seen later on
    - implicit **and** operator between subqueries
  - **must\_not**: the document shouldn't be matched by the query
    - implicit **or** operator between subqueries
  - **should**: the document may be matched by the query
    - implicit **or** operator between subqueries
    - each matching query gives a `_score` bonus
- If no **must** clause is declared, the document should at least be matched by a **should** clause
- **minimum\_should\_match** defines the number of clauses that have to be respected in order for the document to be kept in the result



# Query | bool 2/2

- Example of boolean query

```
{ "query": {  
    "bool": {  
        "must": [  
            {"match": {"title": "spring in action"}},  
            {"term": {"tag": "spring"}}  
        ],  
        "must_not":  
            {"range": {"year": { "lte": 2010 }}}},  
        "should": [  
            {"term": {"tag": "java"}},  
            {"term": {"categories": "development"}}  
        ]  
    }  
}
```



# Query | exists

- **exists** allows to filter the results according to the existence of a field

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

- To filter according the absence of a field, use a must\_not query

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



# Query | index, type and ids

- **type** allows to select some document types

```
{ "query": {  
    "type": {"value": "book"}  
}  
}
```

- **ids** allows to select some ids

```
{ "query": {  
    "ids" : {  
        "type" : "book",  
        "values" : ["4", "12", "38"]  
    }  
}
```

- A **term** can also be used on \_index, \_type, \_id pseudo fields

```
{ "query": {  
    "term": {  
        "_type": "book"  
    }  
}
```



# Query | filter 1/3



- A **bool** or **constant\_score** query can be used with **filter** clause to restrict document matching

```
{ "query": {  
    "bool": {  
        "must": {  
            "match": {  
                "author" : "victor hugo"  
            },  
            "filter": {  
                "range": { "price": {"gt": 25, "lt": 50} }  
            }  
        }  
    }  
}  
--- Or  
{ "query": {  
    "constant_score": {  
        "filter": {  
            "range": { "price": {"gt": 25, "lt": 50} }  
        }  
    }  
}
```

# Query | filter 2/3



Filters are very interesting from the performance viewpoint

- **No score calculation**

- A filter does not participate to the \_score calculation
- Ususally faster than a classic **query**
- To be preferred if there is no score to calculate
- Works well with operators that do not perform analysis **term**, **range**, **exists**...
- Allows to limit the subset of document on which the score will be calculated



# Query | filter 3/3

- **Caching**
  - Automatically cached in the form of **BitSets** structures
  - A few exceptions do not use caches: **script**, **range** + **now**

Doc Id	A	B	C	D	E	F	G	H
Term	1	0	0	1	1	0	1	0

- **Can be combined**
  - Think to prioritize filters from the least costly to the most costly

Doc Id	A	B	C	D	E	F	G	H
Term	1	0	0	1	1	0	1	0
Range	0	1	0	1	0	1	1	0
And	0	0	0	1	0	0	1	0





# Aggregations



# Aggregations

- Aggregations are used to
  - **Order and analyze** the search results
    - Group the results per type / tab
    - Show the results in diagrams
  - **Improve navigation** in the results
    - Diagrams or filtering options can be clicked
- Several aggregation types are available:
  - Counting
  - Histograms,
  - Statistical data,
  - Geo proximity...
  - Pipelining (experimental)

The idea is to extract **insights** from the data by looking at it under different **facets**.

# Aggregations



VW Das Auto.

Modèles | Offres | Occasions VW | VW Entreprises | Services & Après-vente | La Marque | Think Blue. | Innovations | [Configurateur](#) | [f](#)

Accueil | Contact | Plan du site | Informations

## Modèles

### Nos modèles

**Choisissez:**

Catégories

<input checked="" type="checkbox"/> Berline	<input checked="" type="checkbox"/> Break
<input checked="" type="checkbox"/> Cabriolet	<input checked="" type="checkbox"/> Citadine
<input checked="" type="checkbox"/> Compacte	<input checked="" type="checkbox"/> Coupé
<input checked="" type="checkbox"/> Monospace	<input checked="" type="checkbox"/> SUV
<input checked="" type="checkbox"/> Transport de personnes	

Prix

9 490  57 893

Portes

3 Portes  5 Portes

Motorisation/Transmission

<input checked="" type="checkbox"/> BlueMotion	<input checked="" type="checkbox"/> Diesel
<input checked="" type="checkbox"/> Essence	<input checked="" type="checkbox"/> Automatique
<input checked="" type="checkbox"/> DSG	<input checked="" type="checkbox"/> Manuelle

Emissions

89  298

[Reset](#)

  
up!

  
Polo

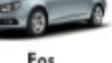
  
Golf

  
Jetta

  
Passat

  
Scirocco

  
Golf Cabriolet

  
Eos

  
Golf SW

  
Passat SW

  
Tiguan

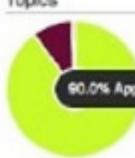
  
Touareg

  
Golf Plus

  
Touran

  
Sharan

  
Categories

  
Topics  
60.0% Apple

  
Sentiment

Top 5 Authors

roznamo : 331 (2.63%)
Zoravý : 122 (0.75%)
tomášp : 100 (0.61%)
Daniel Očkočák : 93 (0.57%)
MačiBog.sk : 84 (0.51%)

[Hide Dashboard](#)

# Syntax



- **Aggregations('aggs')** section is placed at the **query** section level in the whole search query:

```
{ "query" : {...},  
  "aggs" : {  
    "<agg_name>" : {  
      "<agg_type>" : {  
        "<param_1>" : "<param_1_value>", ... }  
      }  
    }  
  }  
--- Response contains the search AND aggregation results.  
--- Set size:0 to have no hit returned if you do not care about search results  
{ "hits" : {...},  
  "aggs" : {  
    "<agg_name>" : { ... }  
  } }
```

- Keyword: **aggs**
- Calculated on documents returned by the query
- If no query defined, aggregation is performed on all the documents



# Aggregation

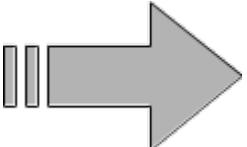
- Let's take a set of documents of type **book**

Title	Authors	Date	Price	Tags
Spring Batch in Action	Arnaud Cogoluègnes, Thierry Templier, Olivier Bazoud	10-2011	59.99	java spring
Lucene in Action	Erik Hatcher, Otis Gospodnetić	01-2004	44.95	java lucene
Spring in Action	Craig Walls	11-2011	44.99	java spring
Spring par la Pratique	Arnaud Cogoluègnes, Thierry Templier, Julien Dubois	07-2009	45.00	java spring
MongoDB The definitive book	Kristina Chodorow	09-2012	31.99	nosql
Programming Android	Zigurd Mednieks	09-2012	49.99	java mobile



# Aggregation Terms 1/2

Tags
java spring
java lucene
java spring
java spring
nosql
java mobile



Tags	Count
java	5
spring	3
lucene	1
nosql	1
mobile	1



# Aggregation Terms 2/2

- A **terms** aggregation returns the most frequent terms in the search results for a given field

```
{ "query" : { "match_all" : { }},  
  "aggs" : {  
    "tags" : {  
      "terms" : {  
        "field" : "tags" }}}}  
}  
--- Result response contains the aggregation with the chosen name  
{ "hits" : {...},  
  "aggs": {  
    "tags": {  
      "buckets": [  
        {"key": "java", "doc_count": 5},  
        {"key": "spring", "doc_count": 3},  
        {"key": "nosql", "doc_count": 1} ]}}}  
}
```

// The aggregation's name  
// Aggregations list  
// "tags" response details  
// Elements of the aggregation

- Options
  - Limit the number of aggregations returned with **size**
  - Order aggregations with **order: \_count, \_term**,



# Range aggregation 1/2

Price
59.99

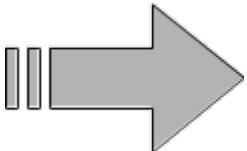
44.95

44.99

45.00

31.99

49.99



Price	Count
Price < 40	1

40 < Price < 50 4

Price > 50 1



# Range aggregation 2/2

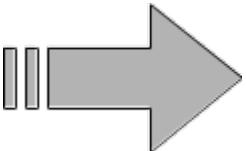
- A **range** aggregation returns how the values of a field are distributed.

```
{ "query": {"match_all": {}},  
  "aggs": {  
    "aggregation_Price": {  
      "range": {  
        "field": "price",  
        "ranges": [  
          { "to": 40 },  
          { "from": 40, "to": 50 },  
          { "from": 50 }]}  
      },  
      "size": 0  
    }  
  }  
--- Result -----  
{ "hits" : {...},  
  "aggs" : {  
    "aggregation_Price" : {  
      "buckets" : [  
        { "to" : 40, "doc_count" : 1},  
        { "from" : 40, "to" : 50, "doc_count" : 4}  
        ...]  
      }  
    }  
}
```

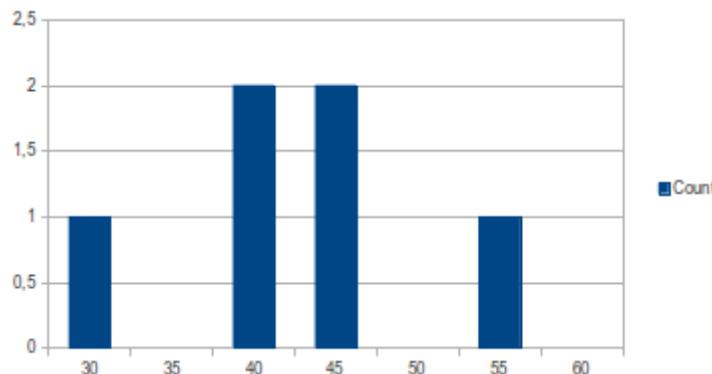


# Histogram Aggregation 1/2

Price
59.99
44.95
44.99
45.00
31.99
49.99



Price	Count
30	1
35	0
40	2
45	2
50	0
55	1
60	0





# Histogram Aggregation 2/2

- histogram aggregation type:

```
{ "query": {"match_all": {}},  
  "aggs": {  
    "histo_aggregation": {  
      "histogram": {  
        "field": "price",  
        "interval": 5}}  
    }  
}  
--- Result -----  
{ "hits" : {...},  
  "aggs" : {  
    "histo_aggregation" : {  
      "buckets" : [  
        {"key" : 30, "doc_count" : 1},  
        {"key" : 40, "doc_count" : 2},  
        {"key" : 45, "doc_count" : 2},  
        {"key" : 55, "doc_count" : 1} ]}}}
```

- Works on numeric fields by rounding their value
- Sortable using the **order** parameter:
  - **\_count**, **\_key**, <field\_name>



# Date histogram aggregation 1/3

Date
10-2011

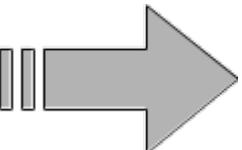
01-2004

11-2011

07-2009

09-2012

09-2012



Per year	Count
2004	1

2009

2011

2012

Per month and year	Count
January 2004	1

July 2009

October 2011

November 2011

September 2012



# Date histogram aggregation 2/3

- **date\_histogram** aggregation type:

```
{  
  "query": {"match_all": {}},  
  "aggs": {  
    "year_aggregation": {  
      "date_histogram": {  
        "field": "publication",  
        "interval": "year"}},  
    "month_aggregation": {  
      "date_histogram": {  
        "field": "publication",  
        "interval": "month"}  
    }  
  }  
}
```



# Date histogram aggregation 3/3

- Result of a **date\_histogram** aggregation:

```
{ "hits" : { ... },
  "aggs" : {
    "year_aggregation" : {
      "_type": "date_histogram",
      "buckets" : [
        { "key" : 1072915200000, "doc_count" : 1 },
        { "key" : 1230768000000, "doc_count" : 1 },
        { "key" : 1293840000000, "doc_count" : 2 },
        { "key" : 1325376000000, "doc_count" : 2 }
      ],
      "month_aggregation": {
        "_type": "date_histogram",
        "entries" : [
          { "key" : 1072915200000, "doc_count" : 1 },
          { "key" : 1246406400000, "doc_count" : 1 },
          { "key" : 1317427200000, "doc_count" : 1 },
          { "key" : 1320105600000, "doc_count" : 1 },
          { "key" : 1346457600000, "doc_count" : 2 }
        ]
      }
    }
  }
}
```

# Filter aggregation



- A **filter** aggregation returns buckets of documents that match a filter criteria

```
{ "query": { "match_all": {}},
  "aggs": {
    "filter_aggregation": {
      "filter": {
        "terms": {
          "title": ["spring", "lucene"]
        }
      },
      "aggs" : {                                     //Imbrication seen later on
        "avg_price" : { "avg" : { "field" : "price" } }
      }
    }
  }
}
--- Result -----
{ "hits" : {...},
  "aggs" : {
    "filter_aggregation" : {"doc_count" : 4}}
}
```

- Often used to narrow down the current aggregation context to a specific set of documents. Here it is used to compute the average price of lucene and spring books.



# Min/Max/Avg Aggregation 1/2

Price
59.99

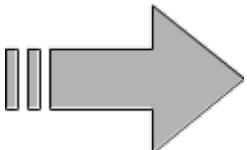
44.95

44.99

45.00

31.99

49.99



Price	Value
max	59.99

Price	Value
min	31.99

Price	Value
avg	46.15



# Min/Max/Avg Aggregation 2/2

- A **min**, **max** or **avg** aggregation respectively returns the minimum, maximum or average value of a field

```
{ "query": {"match_all": {}},  
  "aggs": {  
    "max_price": {  
      "max": {"field": "price"}  
    }  
  }  
}  
--- Result -----  
{ "hits" : {...},  
  "aggs": {  
    "max_price": {"value": 39.99}  
  }  
}
```

- Can be used for numeric and date fields



# Statistical aggregation

- A "statistical aggregation" calculates statistics for a given numeric field.
- There are two possible options, **stats** and **extended\_stats**

```
{ "query": { "match": { "title": "spring" } },
  "aggs": {
    "statistics": {
      "stats": { "field": "price" }
    }
  }
}
--- Result -----
{ "hits" : { ... },
  "aggs": {
    "statistics": {
      "count": 6,
      "min": 31.989999771118164,
      "max": 59.990016784668,
      "avg": 46.15166759490967,
      "sum": 276.910005569458
    }
  }
}
```

- The calculated statistics are
  - total, sum, square sum, average, minimum, maximum, variance,



# Geo Distance aggregation

- The **geo\_distance** aggregation allows to calculate a distribution according to a distance between geographic points.

```
{  
  "query" : {"match_all" : {}},  
  "aggs" : {  
    "geo1" : {  
      "geo_distance" : {  
        "field" : "location",  
        "origin" : "40, -70",  
        "ranges" : [  
          { "to" : 10 },  
          { "from" : 10, "to" : 20 },  
          { "from" : 20, "to" : 100 },  
          { "from" : 100 }]  
      }  
    }  
  }  
}
```

<https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-geodistance-aggregation.html>





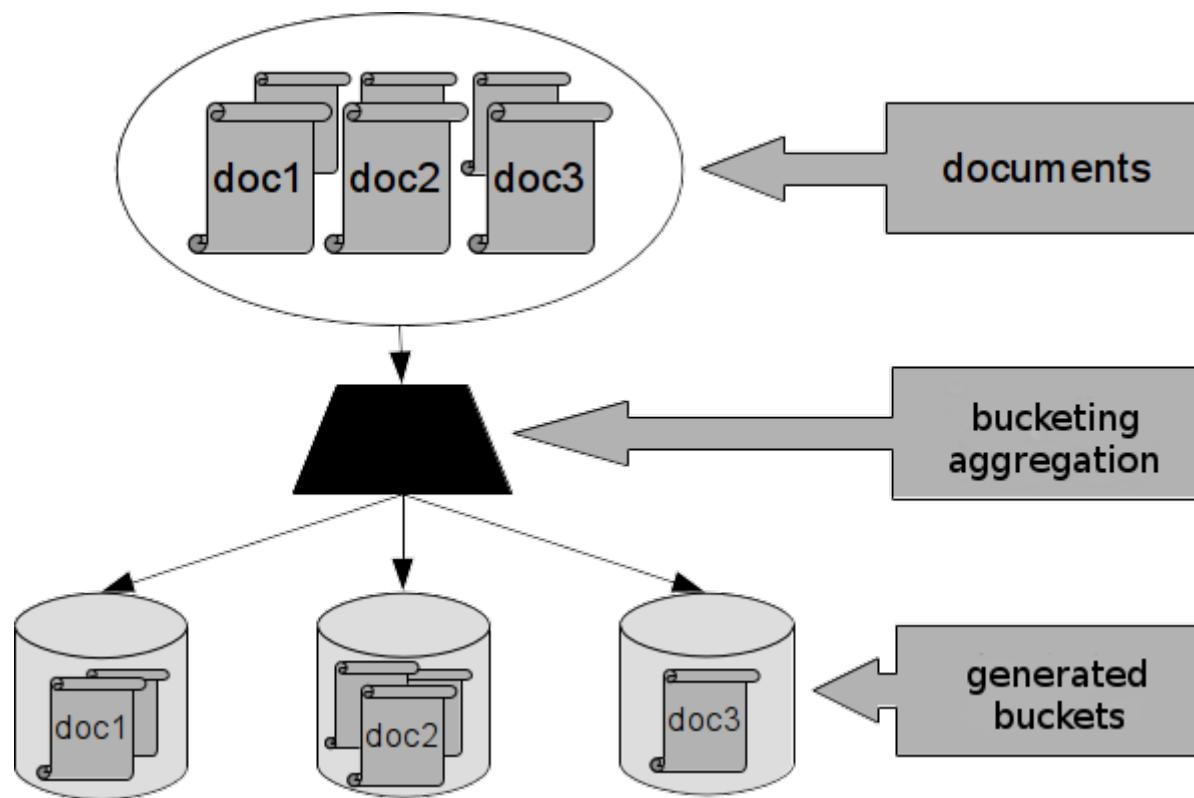
# Aggregation categories 1/3

- **bucket** aggregations separate the documents in one or several subsets (called buckets)
  - **terms**
  - **histogram, date\_histogram**
  - **range, date\_range**
- **metrics** aggregations calculate metrics on the documents
  - **min, max**
  - **sum, avg**
  - **stats**
- **pipeline** aggregations work on the results of other aggregations
  - **min\_bucket, max\_bucket**
  - **sum\_bucket, avg\_bucket**
  - **stats\_bucket**



# Aggregation categories 2/3

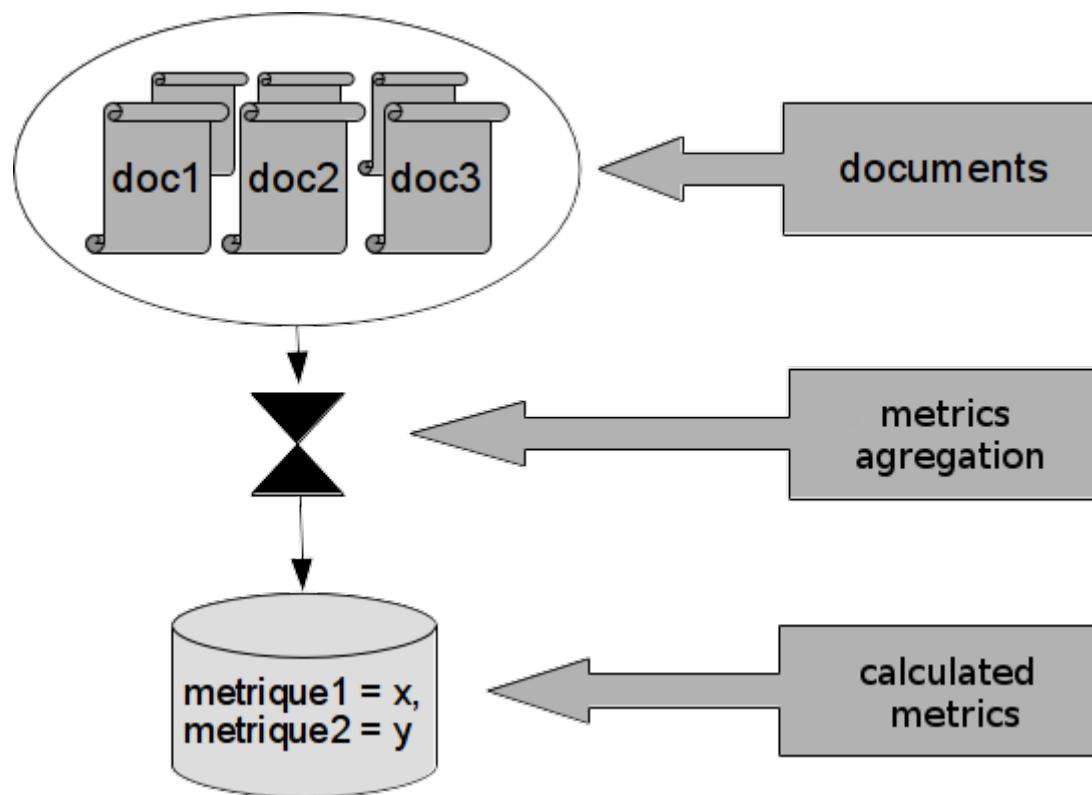
- How a **bucket** aggregation works





# Aggregation categories 3/3

- How a **metrics** aggregation works



# Combining aggregations 1/4



- Powerful mechanism to extract precise information for a set of documents
- Syntax: example with the price by label

```
{ "query" : {...},
  "aggs" : {
    "label" : {
      "terms" : { "field" : "tags" },
      "aggs" : {
        "max_price":{
          "max" : { "field" : "price" }
        }
      }
    }
  }
}
--- Result -----
{ "aggs": {
  "label": {
    "buckets": [
      { "key": "java", "doc_count": 3, "max_price": { "value": 59.99 }},
      { "key": "spring", "doc_count": 2, "max_price": { "value": 59.99 }},
      { "key": "lucene", "doc_count": 1, "max_price": { "value": 44.95 }}]
    }
  }
}
```

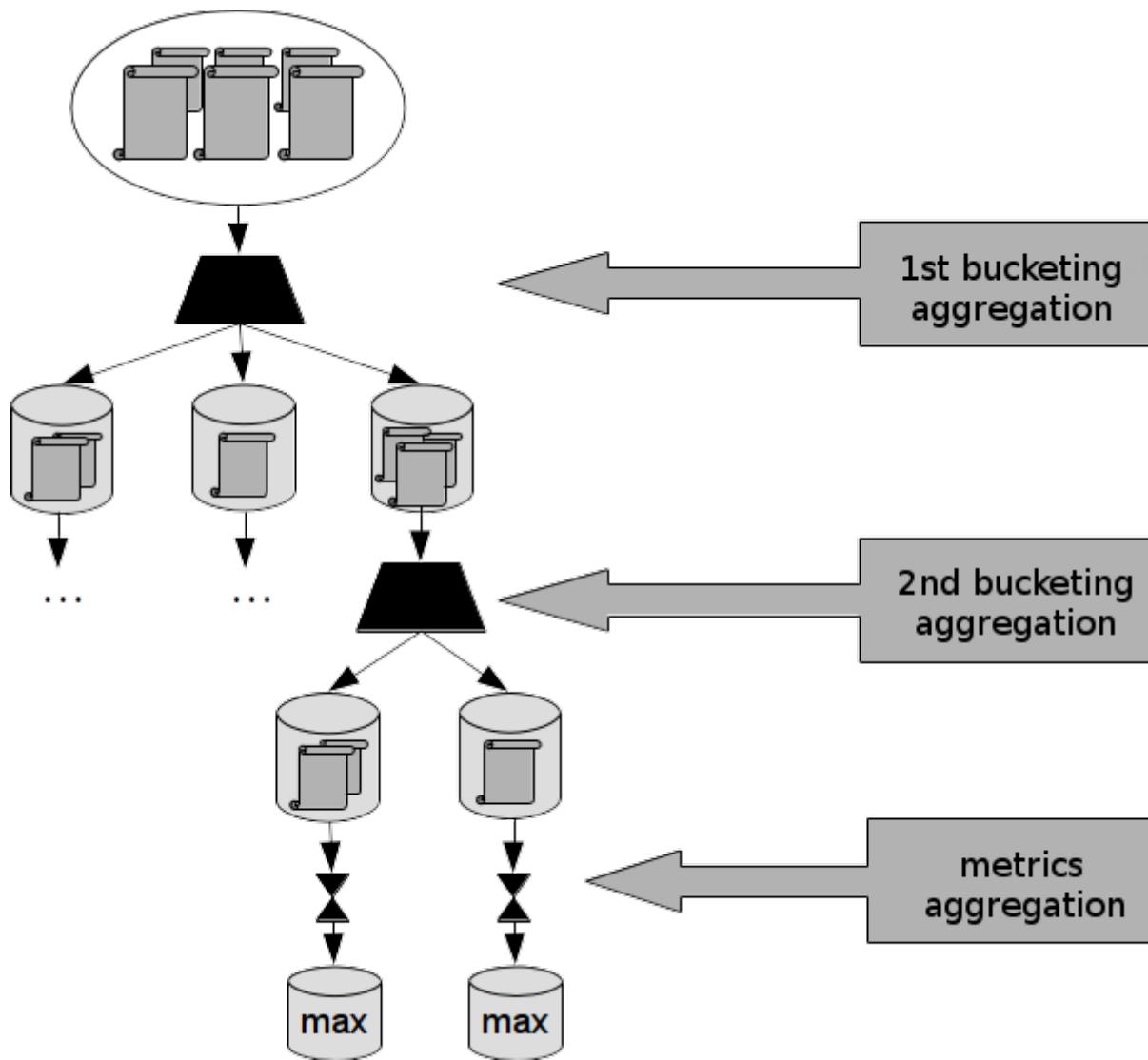


# Combining aggregations 2/4

- Theoretically no limit on maximum number of aggregation levels (depth)
- Only bucket aggregations can have sub aggregations
- It is possible to calculate several sub aggregates per bucket

```
{ "query" : {...},  
  "aggs" : {  
    "label" : {  
      "terms" : { "field" : "tags" },  
      "aggs" : {  
        "max_price":{  
          "max" : { "field" : "price" }},  
        "authors": {  
          "terms" : { "field" : "author" }  
        }  
      }  
    }  
  }  
}
```

# Combining aggregations 3/4





# Combining aggregations 4/4

- A **terms** aggregation can be sorted based on a sub-aggregation
- Syntax: **order** parameter and the sub-category name

```
{ "query" : {...},
  "aggs" : {
    "label" : {
      "terms" : {
        "field" : "tags",
        "order" : {
          "max_price.value" : "asc"
        }
      },
      "aggs" : {
        "max_price":{
          "max" : {
            "field" : "price"
          }
        }
      }
    }
  }
}
```



# Top Hits Aggregation 1/3

- **Top Hits** is used as a metric aggregation and can not have sub-aggregation
- **Top Hits** makes sub query using each containing bucket criterias to find the more relevant documents of this containing bucket:
  - Top hit aggregation allows to access the content of the buckets
  - By default, returns the first 3 results sorted by **\_score** (descending)
  - reading the **top.hits.total** value of each top hits response makes it possible to count the containing bucket documents

```
{  
  "query": {"match": {"title": "action"}},  
  "aggs": {  
    "label": {  
      "terms": {  
        "field": "tags"},  
      "aggs": {  
        "top": {  
          "top_hits": {}  
        }  
      }  
    }  
  }  
}
```



# Top Hits Aggregation 2/3

```
--- Result -----
{ "hits": {...},
  "aggs": {
    "label": {
      "buckets": [
        { "key": "java",
          "doc_count": 3,                                //Even with only three hits by default,
          "top": {                                         //doc_count and top.hits.total could be
            "hits": {                                     //greater with more than 3 java books
              "total": 3,
              "max_score": 0.5,
              "hits": [
                { "_index": "training",
                  "_type": "books",
                  "_id": "<id>",
                  "_score": 0.5,      // Best result first
                  "_source": {
                    "title": "Spring in Action",
                    "auteur": "Craig Walls", ...
                  }
                }
              ]
            }
          }
        ]
      }
    }
  }
}
```



# Top Hits Aggregation 3/3

- Equivalent to:
  - Execute the query a first time
  - Get tag values
  - Re-execute the query for each tag
- Advantages:
  - better performances rather than "manually" proceed to these steps
  - a single query allows to get all the information
- **top\_hits** aggregation supports most of the search features:
  - **size**
  - **sort**
  - **\_source** filtering

# Pipeline aggregations 1/4



- Allows to reprocess aggregation results
- Make a synthesis of the results obtained per bucket:

```
{ "aggs": {  
    "per_publisher": {  
        "terms": { "field": "publisher.code" },  
        "aggs": {  
            "avg_price": {  
                "avg": { "field": "price" }  
            }  
        }  
    },  
    "max_avg_price": {  
        "max_bucket": {  
            "buckets_path": "per_publisher>avg_price" }  
        }  
    }  
}
```

- **buckets\_path** syntax: **agg\_parent>agg\_child.metric\_val**
  - **agg\_xxx**: aggregation name or **\_count**
  - **metric\_val**: name of the value for multi values metrics such as **stats**

# Pipeline aggregations 2/4



- Perform "sliding" calculation on the buckets

```
{ "aggs": {  
    "per_month": {  
        "date_histogram": {  
            "field": "date",  
            "interval": "month" },  
        "aggs": {  
            "count_diff": {  
                "serial_diff": {  
                    "buckets_path": "_count" }}}}}}}}  
--- Result -----  
[...]  
[{"key_as_string": "2015-07-01", "key": ..., "doc_count": 12},  
 {"key_as_string": "2015-08-01", "key": ..., "doc_count": 38,  
 "count_diff": { "value": 26 }},  
 {"key_as_string": "2015-09-01", "key": ..., "doc_count": 24,  
 "count_diff": { "value": -12 }} ...]  
[...]
```

- Requires to have a histogram parent aggregation



# Pipeline aggregations 3/4

- Filters buckets

```
{  
  "aggs": {  
    "par_mois": {  
      "date_histogram": {  
        "field": "date",  
        "interval": "month" },  
      "aggs": {  
        "selector": {  
          "bucket_selector": {  
            "buckets_path": {  
              "month_count": "_count" },  
              "script": "month_count > 100"  
            }  
          }  
        }  
      }  
    }  
}
```



# Pipeline aggregations 4/4



2 types of pipeline aggregations:

- **Sibling:**
  - Works like metrics aggregations, but works on buckets instead of the documents
  - **avg\_bucket, min\_bucket, max\_bucket, stats\_bucket, ...**
  - At the same level as the bucket aggregation on which it operates
- **Parent:**
  - Performs calculations on sliding window of buckets
  - **serial\_diff, derivative, moving\_avg, cumulative\_sum, bucket\_selector...**
  - Inside a bucket type aggregation (usually histogram) on which it will operate



# Aggregations & Filters 1/3

- What is the impact of filters on aggregation calculations?
- For a query **filter**
  - Results are filtered by the filter
  - Aggregations are calculated on the filtered results
- When navigating with aggregations it can be useful to have:
  - a result list filtered according to a given bucket
  - keeping aggregation calculations on all the unfiltered results
- **post\_filter** is specified at the same level as the **query** or **aggs** section
  - Impact on **hits result list** which will be filtered
  - No impact on aggregations



# Aggregations & Filters 2/3

	Hits
1	Spring Batch in action
2	Spring in Action

Aggregation	Count
spring	2
java	2

## Filtered Query

```
{  
  "query": {  
    "bool": {  
      "must": {  
        "match": {"title": "action"}  
      },  
      "filter": {  
        "term": {"title": "spring"}  
      }  
    },  
    "aggs": {  
      "aggregation_terms": {  
        "terms": {"field": "tags"}  
      }  
    }  
  }  
}
```



# Aggregations & Filters 3/3

	Hits
1	Spring Batch in action
2	Spring in Action

Aggregation	Count
java	3
spring	2
lucene	1

## Search Filter

```
{
  "query": {
    "match": {"title": "action"}
  },
  "aggs": {
    "aggregation_terms": {
      "terms": {"field": "tags"}
    }
  },
  "post_filter": {
    "term": {"title": "spring"}
  }
}
```

# Performance



- Aggregations manipulate a lot of data
  - It is important to consider performance
  - Can slow down queries
  - Think before using an aggregation
- Aggregation data is put in memory (RAM) on each shard
  - Node (the JVM) must have enough memory
  - Overflows triggered OutOfMemoryError before docvalues disk storage
- Precautions
  - Size the memory dedicated to the JVM according to the queries
  - Divide the index in shards
  - Filter results to work with less documents
  - Allocate shards to the more powerful nodes



# Performance - doc values

- **Text** type requires setting **fielddata:true** to enable sorting or aggregating on it:
  - it consumes a lot of memory when sorted by or aggregated on its values
  - it can lead to strange results as the field is analyzed
- Most of the other field types use **docvalues** by default:
  - It enables Elasticsearch to store values on disk reducing memory problems to only those where a lot of concurrent requests builds memory data for sorting or aggregation operations





# Advanced search features



# Scoring

- Relevance is calculated using the Lucene scoring system
  - All the queries participate in the relevance calculation
  - Relevance is indicated in the `_score` field in the results
- Relevance calculation can be inspected using the `?explain=true` option

```
{"...": "...",
"hits": {
  "total": 3, "max_score": 1.0114732,
  "hits": [
    {..., "_score": 1.0114732},
    {..., "_score": 0.99248385}
  ]
}
```



# Scoring | Bool Query

- **bool query:**

```
POST library/_search?explain=true
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "title": {
              "query": "spring in action"
            }
          }
        }
      ],
      "should": [
        {
          "match_phrase": {
            "title": {
              "query": "spring in action"
            }
          }
        }
      ]
    }
  }
}
```

- scores for **must** clauses accumulate
- scores for **should** clauses bring a scoring bonus
- **filter** and **must\_not** clauses have no impact on the score



# Scoring | Query Boost

- **boost** in a query

```
POST library/_search
{ "query": {
    "bool": {
        "should": [
            { "match": {
                "title": {
                    "query": "mitterand",
                    "boost": 2 }}},
            { "match": {
                "authors": {
                    "query": "mitterand",
                    "boost": 1 }}}]
    }
}}
```

```
POST library/_search
{ "query": {
    "multi_match": {
        "query": "mitterand",
        "fields": ["title^2", "authors"]}}
```



# Scoring | DisMax Query

- **dis\_max** (disjunction max)

```
POST /zenika/books/_search
{
  "query": {
    "dis_max": {
      "tie_breaker": 0.3,
      "queries": [
        { "term": {
            "author": {
              "term": "Jean Anouilh",
              "boost": 1.5}}}],
        { "match": {
            "title": {
              "query": "ans",
              "boost": 0.5}}}}]]}}
```

- Looks similar to the **bool/should** query, but takes the highest score instead of the sum
- Useful to favor a field to another one
- **tie\_breaker** (between 0 and 1) adds a bonus if several queries match. In the example: max score +  $0.3 \times \sum$  other scores.

# Scoring | Function Score Query



- The **function\_score** allows the introduction of custom scoring calculation

```
{ "query": {  
    "function_score": {  
        "query": {  
            "match": { "title": "spring in action" }},  
        "functions": [  
            { "field_value_factor": { "field": "price", "factor": 0.1 }}],  
        "boost_mode": "multiply" }}}}
```

- Several **functions** types:
  - script\_score**
  - field\_value\_factor**: on a numeric field
  - decreasing function (**gauss**, **linear**, **exp**)
  - weight**: constant
- Limit the application of the functions with **filter** within the function
- boost\_mode**: Mode used to combine the calculated scores: **multiply**, **sum**...
- Use cases: popularity, price, date, geo-distance, in stock



# Suggest API 1/4

- The **Suggest API** allows to suggest terms to the user without specific mapping
  - The suggest section is at the same level as the query section
  - The query section is optional
  - The suggest section can contain multiple user named queries
  - The text query can be set globally for all named suggesters
  - The suggester can have 4 types:
    - **Term suggester**
    - **Phrase suggester**
    - **Completion suggester**
    - **Context suggester**

# Suggest API 2/4



```
POST /zenika/_search
{
  "query": {"match": {"title": "Les aventrues de Tintni"}}, //optional
  "suggest": {
    "text": "Les aventrues de Tintni", //Global text query
    "suggest_1": {
      "term": {"field": "title"} //Term suggester
    },
    "suggest_2": {
      "term": {"field": "summary"} //Term suggester
    }
  }
}
--- Different texts sample
```

```
POST /zenika/_search
{
  "suggest": {
    "suggest_1": {
      "text": "Les aventrues de Tintni", //Specific text query
      "term": {"field": "title"} //Term suggester
    },
    "suggest_2": {
      "text": "Les aventrues de Lucky Luke", //Specific text query
      "term": {"field": "summary"} //Term suggester
    }
}
```



# Suggest API 3/4

- Suggestions are sent in the response under the chosen labels:

```
{ "hits" : { ... },
  "suggest" : {
    "suggest_1" : [
      { "text" : "les",
        "offset" : 0,
        "length" : 3,
        "options" : [ ] },
      { "text" : "aventrues",
        "offset" : 4,
        "length" : 9,
        "options": [ { "text": "aventures", "score": 0.8888889, "freq": 1 } ] },
      { "text" : "de",
        "offset" : 14,
        "length" : 2,
        "options" : [ ]},
      { "text" : "tintni",
        "offset" : 17,
        "length" : 6,
        "options": [ { "text": "tintin", "score": 0.8333333, "freq": 1 } ]
    [...]
```

Score depends of the suggester type being used

# Suggest API 4/4



- Different implementations available
  - Term
    - Based on a "edit" distance
    - Different suggestion modes: missing, popular, always
    - A lot of possible configuration parameters: field, size, analyzer, frequency/relevance sort...
  - Phrase
    - Additional process after the term suggester
    - Prioritize the suggestions when the terms used together
  - Completion
    - Classic autocomplete feature
  - Context
    - A completion suggester with additional context filtering/boosting capacities



# Autocompletion 1/4

- An autocomplete feature can be based on:
  - Search + Edge NGrams analyzers
    - Increase index size
    - Computation at indexing time
  - Completion Suggester
    - More efficient than term/phrase suggesters,
    - Computations made at search time
    - Data structure built at indexing time and stored in memory
- The completion suggester needs a specific mapping

```
PUT /library/book/_mapping
{
  "book" : {
    "properties" : {
      "title" : { "type" : "string" },
      "suggest" : {
        "type" : "completion",
        "analyzer" : "simple",
        "search_analyzer" : "simple"
      }
    }
  }
}
```



# Autocompletion 2/4

--- Indexing the document

```
PUT /library/book/1
{ "title" : "Les Misérables",
  "suggest" : {
    "input": [ "Misérables", "Miserabilli" ],
    "weight" : 10
  }
}
```

--- Indexing the document with different weights for each input

```
PUT /library/book/1
{ "title" : "Les Misérables",
  "suggest" : [ {
    "input": "Misérables",
    "weight" : 10
  },
  {
    "input": "Miserabilli" ,
    "weight" : 3
  }
}
```

- With:

- **input**: indexed text, on which the autocomplete is performed
- **weight**: Weighting, used to prioritize suggestion results



# Autocompletion 3/4

- When searching:

```
POST /library/book/_search
{
  "suggest": {
    "book-suggestion" : {           //suggestion name
      "prefix" : "m",
      "completion" : {
        "field" : "suggest"
      }
    }
  }
}
```

- With:

- **text**: text used for autocompletion (user input)
- **completion**: suggester type
- **field**: document field (see mapping) on which the autocompletion is performed



# Autocompletion 4/4

```
--- Suggest result
{
  "hits" : {...},
  "suggest": {
    "book-suggestion": [
      {
        "text": "m",
        "offset": 0,
        "length": 1,
        "options": [
          {
            "text": "Misérables",
            "_index": "library",
            "_type": "book",
            "_id": "1",
            "_score": 10,
            "_source": { //source is now returned instead of output or payload
              "title": "Les Misérables",
              "suggest": [
                { "input": "Misérables", "weight": 10 },
                { "input": "Miserabilli", "weight": 3 }
              ]
            }
          }
        ]
      }
    ]
  }
}
```

# Context Suggest 1/3



```
--- Mapping defining a category context named 'book_type' where the categories  
--- must be sent with the suggestions.
```

```
PUT /library/book/_mapping
```

```
{ "book" : {  
    "properties" : {  
        "title" : { "type" : "string" },  
        "suggest" : {  
            "type" : "completion",  
            "contexts": [{"name": "book_type", "type": "category"}]}  
    }  
}
```

```
--- Mapping defining a category context named 'book_type' where the categories  
--- will be read from the cat field.
```

```
PUT /library/book/_mapping
```

```
{ "book" : {  
    "properties" : {  
        "title" : { "type" : "string" },  
        "suggest" : {  
            "type" : "completion",  
            "contexts": [{"name": "book_type", "type": "category", "path": "cat"}]}  
    }  
}
```

# Context Suggest 2/3



```
--- Indexing the document
PUT /library/book/1
{
  "title" : "Les Misérables",
  "suggest" : {
    "input": [ "Misérables", "Miserabilli" ],
    "weight" : 10,
    "contexts": {
      "book_type": "Novel"
    }
  }
}
--- Indexing the document with a context path named 'cat'
PUT /library/book/1
{
  "title" : "Les Misérables",
  "suggest" : {
    "input": [ "Misérables", "Miserabilli" ],
    "weight" : 10
  },
  "cat": "Novel"
}
```



# Context Suggest 2/3

- When searching:

```
POST /library/book/_search
{
  "suggest": {
    "book-suggestion" : {           //suggestion name
      "prefix" : "m",
      "completion" : {
        "field" : "suggest",
        "contexts": {
          "book_type": [
            { "context" : "Novel" },
            { "context" : "Comic strip", "boost": 2 }
          ]
        }
      }
    }
  }
}
--- result is similar to completion suggest...
```

# Highlighting 1/2



- **Higligting** allows to highlight words used for search in the results on certain fields.
- The highlighting section is to put at the same level as the query one
- The highlighting result reminds the relative field and the fragment samples
- There are multiple types of Highlighters, each with its advantages and drawbacks:
  - **Plain Highlighter** is the default one but stills the less efficient
  - **Postings Highlighter** is a faster Highlighter selected when the mapping 'index\_options' is set to 'offsets'
  - **Fast vector Highlighter** takes some disk place, can combine multifield matches and works well on large fields. Selected when the mapping 'term\_vector' is set to 'with\_positions\_offsets'
  - **Unified Highlighter** uses one of the other Highlighters depending of the field mapping and the query to highlight. Experimental.
- The chosen fields has to be either **stored** or **\_source** available
- Options
  - Highlighting source (storage or \_source via 'force\_source')
  - Fragment size (extract)
  - Number of fragments
  - Tags surrounding the fragments

# Highlighting 2/2



```
POST /zenika/book/_search
```

```
{ "query": {  
    "match": {"title": "Jack volts"}},  
"highlight": {  
    "fields": {  
        "title": {  
            "fragment_size" : 100, "number_of_fragments" : 5}}}}
```

--- Result -----

```
{ "hits" : {  
    "total" : 2, "max_score" : 0.10848885,  
    "hits" : [  
        { "_index" : "zenika", "_type" : "book", "_id" : "1",  
        "_score" : 0.10848885,  
        "_source" : {...},  
        "highlight" : { //In the results, each result has a "highlight" part  
            "title" : [ "Barnaby <em>Jack</em> a réussi à lui envoyer plusieurs  
                        décharges de 830 <em>volts</em>" ]}}, ... ]}}
```

*Allows to highlight words used for search in the results*

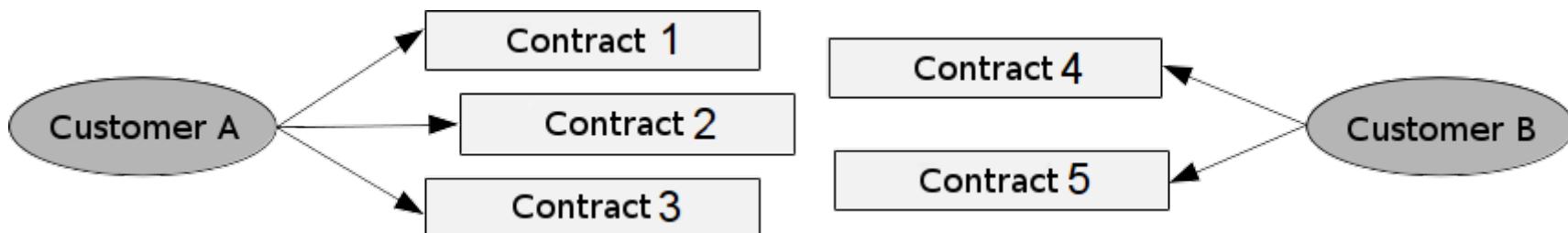


# Searching: Object / Nested / Parent-Child

# Relationship between documents



- When indexing documents, these document may be linked to other documents
- For example, a customer can have several contracts:



- Elasticsearch is document oriented and has a flexible schema. It allows to model relations between document in different ways
- Still, Elasticsearch is not meant to be used like a relational database!



# Denormalized documents

- A first solution is to denormalize the relations by storing all the documents within a single document

```
{"code": "C1", "type": "AV", "deadline": "2012", "name": "Ventura"},  
 {"code": "C2", "type": "PEL", "deadline": "2018", "name": "Ventura"},  
 {"code": "C3", "type": "AD", "deadline": "2017", "name": "Ventura"},  
 {"code": "C4", "type": "AV", "deadline": "2017", "name": "Desproges"},  
 {"code": "C5", "type": "LA", "ceiling": "19500", "name": "Desproges"}
```

- Search queries are easy to write
  - All the necessary data are stored within the documents
- When customer data changes all the contracts have to be reindexed
- Important data duplication



# Type | Object 1/3

- A second solution is to use the **object** mapping type which allows to index sub-documents

```
{ "identifier": "A1", "name": "Ventura", "prename": "Lino",  
  "contracts": [  
    {"code": "C1", "type": "AV", "deadline": "2012"},  
    {"code": "C2", "type": "PEL", "deadline": "2018"},  
    {"code": "C3", "type": "AD", "deadline": "2017"}  
  ]  
}  
  
{ "identifier": "B2", "name": "Desproges", "prename": "Pierre",  
  "contracts": [  
    {"code": "C4", "type": "AV", "deadline": "2017"},  
    {"code": "C5", "type": "LA", "ceiling": "19500"}  
  ]  
}
```





# Type | Object 2/3

- No data duplication
- When data changes the whole document has to be re-indexed
- Dot notation makes building search queries easier
  - Example: Search all the customers having a life insurance contract

```
{ "query": {  
    "term": {  
        "contracts.type": "av"  
    }  
}
```

- This kind of mapping does not allow to search several parameters in the same sub-object as documents are indexed like this:

```
{  
    "identifier": "A1",  
    "name": "Ventura",  
    "prenom": "Lino",  
    "contracts.code": [ "C1", "C2", "C3" ],  
    "contracts.type": [ "AV", "PEL", "AD" ],  
    "contracts.deadline": [ 2012, 2018, 2017 ]  
}
```



# Type | Object 3/3

- Example: Search all the customer having a life insurance with a deadline in 2017

```
{ "query": {  
  "bool": {  
    "must": [{ "match_all": {}}],  
    "filter": [  
      { "term": { "contracts.type" : "av" }},  
      { "term": { "contracts.deadline" : "2017" }}  
    ]  
  }  
}
```



- This query returns the customers having
  - a life insurance contract AND/OR
  - a contract (of any type) with a deadline in 2017



# Type | Nested

- Another solution is to use the **nested** field mapping type which allows to declare nested documents and to keep coherently each subdocument values

```
PUT /customers/customer/_mapping
{
  "customer": {
    "properties": {
      "contracts": {
        "type": "nested"
      }
    }
  }
}
```

# Filter | Nested



- This kind of mapping requires to use a **nested** search filter which allows to filter nested documents

```
PUT /customers/customer/_search
{
  "query": {
    "bool": {
      "filter": {
        "nested": {
          "path": "contracts",
          "query": {
            "bool": {
              "filter": [
                {"term": {"contracts.type": "av"}},
                {"term": {"contracts.deadline": "2017"}}
              ]
            }
          }
        }
      }
    }
  }
}
```

# Parent / Child 1/2



- A last solution is to create different document types with hierachical relationships between them
- The **\_parent** property is used for that relationship

```
PUT /customers/_mapping/contract
{
  "_parent": {"type": "customer"},  

  "properties": {  

    "type": {"type": "keyword"},  

    "deadline": {"type": "date", "format": "YYYY"}  

  }  

}  
  
PUT /customers/_mapping/customer
{  

  "properties" : {  

    "name" : {"type" : "text"},  

    "firstname" : {"type" : "text"}  

  }  

}
```

The Parent/child mapping has some impacts on the documents routing and repartition on the cluster as a child has to be on the same shard as its parent...



# Parent / Child 2/2

- When indexing child documents, the identifier of the parent document is specified

```
POST /customers/customer/2
{"identifier": "B2", "name": "Desproges", "firstname": "Pierre"}
```

```
POST /customers/contract/4?parent=2
{"code" : "C4", "type" : "AV", "deadline" : "2017"}
```

```
POST /customers/contract/5?parent=2
{"code" : "C5", "type" : "LA", "ceiling" : "19500"}
```



# Query | Has Child

- The **has\_child** operator has been created to operate on child/parent relationships
- Returns the parent document which has at least one child document matching the query

```
POST /customers/customer/_search
{
  "query": {
    "has_child": {
      "type": "contract",
      "query": {
        "bool": {
          "filter": [
            {"term": { "type": "AV" }},
            {"term": { "deadline": "2017" }}
          ]
        }
      }
    }
  }
}
```

Performance considerations: Do not forget the join table with parent/child documents identifiers are loaded in memory

# Query | Has Parent



- The **has\_parent** search operator returns the documents whose parent matches the query

```
POST /customers/contract/_search
{
  "query": {
    "has_parent": {
      "parent_type": "customer",
      "query": { "query_string": { "query": "name:deproge~2" } }
    }
  }
}
```

Performance considerations: Do not forget the join table with parent/childs document identifiers are loaded in memory

# Advanced search operators





# Query | More Like This (1/2)

- Documents that look like a given text
  - Very customizable, but difficult to use
  - Warning: the default configuration only suits a few use cases and data

```
POST /library/book/_search
{
  "query": {
    "more_like_this": {
      "fields": ["title", "authors"],
      "like": "spring in action",
      "max_query_terms": 3,
      "min_term_freq": 1,
      "min_doc_freq": 1}}}
```

- Selection of searched terms

<b>name</b>	<b>Default</b>	<b>Description</b>
<b>max_query_terms</b>	25	number of terms in the query
<b>min_term_freq</b>	2	minimum number a term repetition
<b>min_doc_freq</b>	5	minimum number of documents containing the term
<b>max_doc_freq</b>		maximum number of documents containing the term



# Query | More Like This (2/2)

- Example to find similar documents
- Finds all the document that look like a given document

```
POST /library/book/_search
{
  "query": {
    "more_like_this": {
      "fields": ["title", "authors"],
      // The library book of id 3 is the "Spring in action" book
      "like": [{"_index": "library", "_type": "book", "_id": "3"}],
      "max_query_terms": 3,
      "min_term_freq": 1,
      "min_doc_freq": 1,
      "max_doc_freq": 5}}}
```

# Geolocation 1/3



- Geographical positions can be indexed with to mapping types:
  - **geo\_point**
  - **geo\_shape** (Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection, envelope, circle)
- Geo\_points are compatible with those search filter types:
  - **geo\_bbox** : points contained in an box
  - **geo\_distance** : points in a given distance
  - **geo\_distance\_range** : points within a distance range (before 5.0)
  - **geo\_polygon** : points within a polygon
- Geo\_shapes is compatible with the **geo\_shape** query
- There are dedicated aggregations for geo\_points
  - **geo\_distance** aggregation, counts the documents within defined distance ranges
  - **geo\_bounds** aggregation, computes the bounding box containing all geo\_point values for a field in a given bucket
  - **geo\_centroid** aggregation, computes the weighted centroid from all coordinate values for a Geo-point datatype field

Nevertheless, Elasticsearch is not a "GIS" database

# Geolocation 2/3



```
--- Geo_shape sample mapping
```

```
PUT /example
```

```
{  
  "mappings": {  
    "doc": {  
      "properties": {  
        "location": {  
          "type": "geo_shape",  
          "tree": "quadtree",  
          "precision": "1m"  
        }  
      }  
    }  
  }  
}
```

```
--- Indexing a MultilineString
```

```
POST /example/doc
```

```
{  
  "location" : {  
    "type" : "multilinestring",  
    "coordinates" : [  
      [ [102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0] ],  
      [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0] ],  
      [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8] ]  
    ]  
  }  
}
```

# Geolocation 3/3



```
--- Search the multilinestring which is within the envelope
GET /example/_search
{
  "query": {
    "bool": {
      "must": { "match_all": {} },
      "filter": {
        "geo_shape": {
          "location": {
            "shape": {
              "type": "envelope",
              "coordinates" : [[100, 0], [103, 3]]
            },
            "relation": "within"
          }
        }
      }
    }
  }
}
```

*Geo\_shape query can use geo\_shape queries between two different indexes. The one having the documents we are searching for and the other the geo\_shape we want to select with.*





# Clients & Java API



# Client types

- **Client**
  - Technical component allowing to communicate with a node of an Elasticsearch cluster
- Several clients are available for different languages
  - ElasticSearch.NET or NEST (.Net clients)
  - Elastica (PHP client)
  - Pyelasticsearch (Python client)
  - ... and also for Ruby, Erlang, Clojure, Perl, Groovy, Scala
- List of available clients
  - <https://www.elastic.co/guide/en/elasticsearch/client/index.html>

# Integrations



- Integration
  - Technical component which allows to embed Elasticsearch within an existing project or tool
  - Integration can be made as a plugin or a library
- Some example of Elasticsearch integrations:
  - Grails (plugin)
  - Play ! Framework 1.x (module)
  - Drupal 7.x (module)
  - Spring (factory) or Spring Data Elasticsearch
- List of existing integrations
  - <https://www.elastic.co/guide/en/elasticsearch/plugins/current/integrations.html>

# Java API



- Elasticsearch is written in Java and provides a complete API
- Java API
  - covers all the Elasticsearch operations
    - index, search, delete, get, status, refresh, etc
  - used internally to execute operations whatever the incoming protocol is
  - can be used to instantiate a Java node
  - can provide a Java client to communicate with a cluster
- API Reference:  
<https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/index.html>

# Client



- A client is required to perform operations
- Operations are asynchronous, but some accept a **Listener** or return a **Future**
- Mass operations can be grouped in a **Bulk**
- Two ways to get a **Client** :
  - **Transport Client**: Instanciate a **TransportClient** by providing a hostname and a port number
  - **RestClient**: Low level client that uses HTTP to communicate with Elasticsearch
- A client is a thread safe object



# Transport Client

```
Settings settings = Settings.builder()
    .put("cluster.name", "myClusterName").build();
TransportClient client =
    new PreBuiltTransportClient(settings).build()
    .addTransportAddress(
        new InetSocketAddress(
            new InetSocketAddress("host1", 9300)))
    .addTransportAddress(
        new InetSocketAddress(
            new InetSocketAddress("host2", 9300)));
...
// Always close the client
client.close();
```

A **TransportClient** does not join the cluster, it communicates with cluster hosts indicated by the **addTransportAddress()** method in a **round robin** manner





# Admin Client

- An **AdminClient** allows to perform maintenance operations
- Operations on a cluster:
  - Get info or statistics about the nodes,
  - Verify cluster health...
- Operations on indices:
  - Create, delete, verify the existence on a index
  - Optimize, refresh, manage mapping and aliases...

```
// Getting an admin client from a TransportClient instance
AdminClient admin = client.admin();
```

# Index 1/2



- Indexing of documents is performed through the **prepareIndex()**, **execute()** and **actionGet()** methods
- They accept a source in **byte[]**, **String**, **JSON** or **Map** format
- Elasticsearch provides a utility **XcontentFactory.jsonBuilder()** to simplify the creation of JSON objects
- Example - indexing a String source:

```
import org.elasticsearch.action.index.IndexResponse;
//...
// Indexing a String source
IndexResponse response = client.prepareIndex("people", "author", "1")
    .setSource("{\"lastname\":\"Hugo\", \"firstname\":\"Victor\"}",
              XContentType.JSON)
    .execute()
    .actionGet();
```

- **IndexResponse** can be used to get the id of the generated document



# Index 2/2

- Example - indexing a JSON source:

```
import org.elasticsearch.action.index.IndexResponse;
import static org.elasticsearch.common.xcontent.XContentFactory.jsonBuilder;
...
// Indexing a JSON source without specifying its ID
IndexResponse response = client.prepareIndex("people", "author")
.setSource(jsonBuilder()
.startObject()
.field("lastname", "Beyle")
.field("firstname", "Marie-Henri")
.endObject() )
.execute().actionGet();
// Get the generated identifier
// ex: EHxd_aejS9K835A8vD-Y0Q
String id = response.id();
```



# Index Management 1/2

- Example - create an index:

```
import org.elasticsearch.action.admin.indices.create.CreateIndexResponse;  
  
// Create the "people" index  
CreateIndexResponse response = client.admin()  
    .indices()  
    .prepareCreate("people")  
    .execute()  
    .actionGet();
```

- Example - delete an index:

```
import org.elasticsearch.action.admin.indices.delete.DeleteIndexResponse;  
  
// Delete the "people" index  
DeleteIndexResponse response = client.admin()  
    .indices()  
    .prepareDelete("people")  
    .execute()  
    .actionGet();
```



# Index Management 2/2

- Verify that an index exists:

```
// Verify the existence of the "people" index
boolean indexExists = client.admin()
    .indices()
    .prepareExists("people")
    .execute()
    .actionGet()
    .isExists();
```

# Mapping 1/2



- Adding a mapping can be done at index creation time or later on, using the Mapping API
  - The mapping have to be provided in String, JSON or Map format
- When creating the index, with the **addMapping()** method

```
// Index creation
CreateIndexResponse response = client.admin()
    .indices()
    .prepareCreate("other")
    .addMapping("author", mapping)
    .execute()
    .actionGet();
```

- When the index already exists using the **putMapping()** method

```
// Adding a mapping the document type and the index
client.admin()
    .indices()
    .preparePutMapping("people", "other")
    .setType("author")
    .setSource(mapping)
    .execute()
    .actionGet();

// Throw an exception if the mapping does not exists
// org.elasticsearch.indices.IndexMissingException
```

# Mapping 2/2



- To get the mapping of an index in Java, use the Cluster State API

```
// Cluster State filtered for the index
ClusterStateResponse response = client.admin()
    .cluster()
    .prepareState()
    .setFilterIndices("people")
    .execute()
    .actionGet();

// Get metadata
MetaData metaData = response.getState().getMetaData();

// Get index mappings
ImmutableMap<String, MappingMetaData> mappings =
    metaData.index("people").mappings();

MappingMetaData mapping = mappings.values().iterator().next();
```

# Get



- Retrieval of a document is performed using the **prepareGet()**, **execute()** and **actionGet()** methods
- Document id should be known

```
import org.elasticsearch.action.get.GetResponse;

// Get a document of identifier 2
GetResponse response = client.prepareGet("people", "author", "2")
    .execute()
    .actionGet();

// True if the document has been found
boolean found = response.exists();

// Contains the document source
response.getSourceAsString();
response.getSourceAsMap();
```

- By default, **GetResponse** contains the document source
- Possibility to limit the returned fields with the **setFields()** method

# MultiGet



- Several documents can be retrieved in a single request
- Indices and document types can be different
- Use the **prepareMultiGet()**, **add()**, **execute()** and **actionGet()** methods
- Document id should be known

```
import org.elasticsearch.action.get.MultiGetResponse;

// Multiple documents retrieval
MultiGetResponse response = client.prepareMultiGet()
    .add("people", "author", "1", "2")
    .add("people", "other", "3")
    .execute()
    .actionGet();

assertEquals(3, response.responses().length);
```

# QueryBuilder



- The Elasticsearch Java API provides utility classes to build queries and search filters
  - **QueryBuilders**
  - **FilterBuilders**
- The queries and filters built with this API can be used in the Search and Count APIs
- All the query types can be built with a **QueryBuilder**

```
import static org.elasticsearch.index.query.QueryBuilders.*;  
  
// Instantiate a QueryBuilder for term search  
TermQueryBuilder queryBuilder = termQuery("lastname", "hugo");  
  
// Instantiate a QueryBuilder for boolean search  
QueryBuilder queryBuilder = boolQuery()  
    .should(termQuery("lastname", "hugo"))  
    .should(termQuery("lastname", "beyle")));
```



# Search 1/6

- Searching for documents and getting back the results is performed using the Search API
- Searches can be executed on one or several indices and document types
- Search example:

```
import static org.elasticsearch.index.query.QueryBuilders.*;  
  
// Instantiate a QueryBuilder to create a term search  
TermQueryBuilder queryBuilder = termQuery("lastname", "hugo");  
  
// Execute search  
SearchResponse response = client.prepareSearch(indexName)  
.setQuery(queryBuilder)  
.execute()  
.actionGet();
```





# Search 2/6 : Getting fields

- Document fields to be retrieved have to be specified
  - with the **addField()** or **addFields()** methods
  - if no field is specified the document's source is returned
- **setFrom()** and **setSize()**
  - Allows to return only a subset of the results
  - Useful to implement results pagination

```
// Execute search
SearchResponse response = client.prepareSearch(indexName)
    .setQuery(termQuery("lastname", "hugo"))
    .addField("age")
    .addFields("lastname", "firstname")
    .setFrom(0)
    .setSize(10)
    .execute()
    .actionGet();
```



# Search 3/6 : Sorting results

- By default, results are sorted by score
- The **addSort()**method allows to sort according to one or several fields:

```
// Perform search and sort results by name
SearchResponse response = client.prepareSearch(indexName)
    .setQuery(termQuery("lastname", "hugo"))
    .addSort("lastname", SortOrder.ASC)
    .addSort("firstname", SortOrder.ASC)
    .setFrom(0)
    .setSize(10)
    .execute()
    .actionGet();

//Sort can also be define like this:
SortBuilders.fieldSort("lastname").order(SortOrder.ASC)
```

- Sorting can lead to memory problems
  - Fields used for sorting shouldn't be analyzed
  - When sorting, values are put in memory. Each node should have enough memory to contain all the values to sort



# Search 4/6 : Extras

- Other methods
  - **addAggregation()** add aggregation to the results
  - **setPostFilter()** defines filter on results after the aggregation have been calculated
  - **setExplain(true)** get information about the relevance of each search result
  - **addHighlightedField()** add a field whose keywords matching the search will be highlighted in the results (highlighting requires extra information such as the position of the keyword within the text)
  - **addIndexBoost()** allows to "boost" results from a specific index (useful for searches in multiple indices)





# Search 5/6 : Results

- The **SearchResponse** object contains the search results:

- The **SearchHits**:

```
List<SearchHits> hits = response.hits();
```

- The total number of results:

```
long total = response.hits().getTotalHits();
```

- The score and identifier of each result:

```
for (SearchHit hit : response.hits()) {  
    float score = hit.score();  
    String id = hit.id();  
}
```



# Search 6/6 : Results

- The **SearchResponse** object contains the search results:
  - Fields value (if they have been specified in the search query)

```
for (SearchHit hit : response.hits()) {  
    SearchHitField hitField = hit.field("lastname");  
    if(hitField != null){  
        hitField.values();  
    }  
}
```



# Scrolling 1/3

- If an important number of results have to be retrieved, it is preferable to use scrolling rather than performing successive queries
- Reminder: during scrolling, results are kept for a given amount of time by Elasticsearch
  - Faster to navigate through the results
  - Similar to the way cursors work in databases
- In order to use scrolling:
  - Specify the **scroll=** parameter
  - The first query returns the results, plus a **scroll\_id**
  - In the subsequent queries, reuse the **scroll** and **scroll\_id** parameters



# Scrolling 2/3

- Scrolling example:

```
import static org.elasticsearch.index.query.FilterBuilders.*;
import static org.elasticsearch.index.query.QueryBuilders.*;

// Search using scroll
// scrollResponse will contain a scrollId
SearchResponse scrollResponse = client.prepareSearch(test)
    .setSearchType(SearchType.SCAN)
    .setScroll(new TimeValue(60000))
    .setQuery(termQuery("lastname", "hugo"))
    .setSize(100)
    .execute()
    .actionGet();
```





# Scrolling 3/3

- Scrolling example:

```
// Iterate over the results
while (true) {
    scrollResponse = client.prepareSearchScroll(scrollResponse.getScrollId())
        .setScroll(new TimeValue(60000))
        .execute()
        .actionGet();

    // Using the results...
    for (SearchHit hit : scrollResponse.getHits()) {
        ...
    }

    // Stop condition: no more result is returned
    if (scrollResponse.hits().hits().length == 0) {
        break;
    }
}
```



# Bulk Request



- Bulk request allow to group the execution of several index or delete requests within a single request

```
// A BulkRequest will contain other requests
BulkRequestBuilder bulkRequest = client.prepareBulk();

// The build queries the client#prepare method can be used,
// or the RequestBuilders
bulkRequest.add(
    client.prepareIndex("people", "author", "1")
        .setSource(jsonBuilder()
            .startObject()
                .field("lastname", "Hugo")
                .field("firstname", "Victor")
            .endObject()));
}

// Add other requests to be performed...
bulkRequest.add(...);

// Execute the bulk request
BulkResponse bulkResponse = bulkRequest.execute().actionGet();
if (bulkResponse.hasFailures()) {
    // In case of error, iterate over bulkResponse.items()
}
```

# Refresh



- A refresh allows to make documents searchable that have been indexed since the last refresh
- Refreshes are automatically triggered by Elasticsearch
- it can sometimes be useful to manually trigger a refresh
  - After a massive indexing process
  - If a document has to be available immediately
- Several mechanisms to refresh an index

```
//Triggering an index refresh
client.admin()
    .indices()
    .prepareRefresh("people")
    .execute()
    .actionGet();

// Forcing a refresh after indexing a document
client.prepareIndex("people", "author")
    .setSource(source, XContentType.JSON)
    .setRefresh(true)
    .execute()
    .actionGet();
```

Use with care for performance reasons

# Needed Maven dependencies



- Add Elasticsearch as Maven dependency:

```
<!-- Needs Log4J 2 as dependencies or SLF4J log4J bridges-->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.8.2</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.8.2</version>
</dependency>

<!-- Transport client -->
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>transport</artifactId>
</dependency>

<!-- alternative: -->
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>rest</artifactId>
</dependency>
```





# Integration in an existing application

- There are several ways to integrate Elasticsearch to an application
- **Embedded** mode:
  - Instantiate a local Node capable of containing data
  - Only visible in the application JVM
  - Starting and stopping of the node have to be properly managed
  - Not recommended by elastic
- **Client** mode:
  - Instantiate a **TransportClient** (requires elasticsearch dependency)
  - **RestClient**

# Integration in an existing application



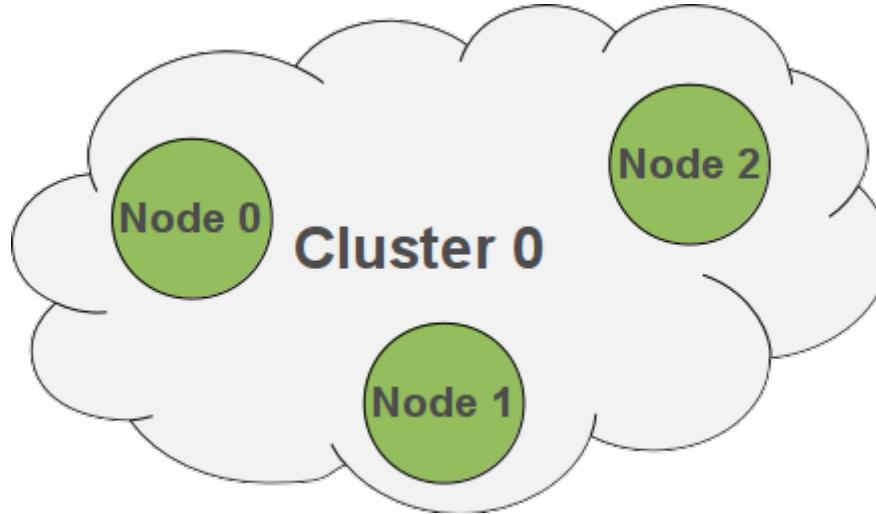
- Web Application
  - Favor the "client" mode
  - Do not perform long / massive operations
  - The **Client** object can be used in multi thread applications
- Long operations
  - Massive index/delete operations should be performed in autonomous batches
- SpringBatch is an excellent framework for batch processing:
  - <http://static.springsource.org/spring-batch/>
  - <https://github.com/acogoluegnes/Spring-Batch-Workshop/>





# Cloud & Cluster

# Elasticsearch Cluster 1/2



- A **cluster** is
  - Composed of one or several nodes
  - Identified by a name used by nodes to recognize other nodes of the cluster
  - Nodes can be added to or removed from the cluster while it is running
- A **node** is
  - An instance of an Elasticsearch Java process
  - Identified by a name that is automatically generated by default

# Elasticsearch Cluster 2/2



- The cluster and node names are configured in the **config/elasticsearch.yml** file

```
# ----- Cluster -----
#
# Use a descriptive name for your cluster:

cluster.name: prod-es

# ----- Node -----
#
# Use a descriptive name for the node:

node.name: prod-es-1
```

Configure the cluster and node names

- In the cluster all the nodes communicate together



# Communication modes

- By default Nodes listen on local loopback interface **127.0.0.1** or **local** and you shall change it to go to production environments

```
# Change the bind address to a specific IP (IPv4 or IPv6):  
network.host: 192.168.0.1  
  
#you could also separate http.host and transport.host values if you wish  
  
#listen on port 9300 for binary communication between nodes or Java API  
transport.tcp.port: 9300  
  
#listen on port 9200 for HTTP communications  
http.port: 9200  
  
# Nodes communicate using Unicast  
# Each node has to be configured with addresses and ports of the nodes it should connect to  
discovery.zen.ping.unicast.hosts: ["host1", "host2:port", "host3[portX-portY]"]
```

- There are several plugins for communication in cloud AWS, Azure, GCE
- ECE is using docker containers to parameterize their discovery process

# Elasticsearch Node 1/3



- A node can have several roles
  - **Master**: the master coordinates the cluster
  - **Data**: the node store data (shards and replica) and participates in the execution of the search queries
  - **Ingest**: the node can parse incoming documents
  - **Http** : All nodes have the HTTP requests responding role
- A node can be specialized by disabling some of these features

# Elasticsearch Node 2/3



- A cluster only has a single **master** node
  - Master is automatically elected
- Using "Zen Discovery" module
  - The nodes ping each other
  - If no response or takes too long to respond
    - The node is unreferenced
    - If it is the master, a new master is elected

# Elasticsearch Node 3/3



- In order to specialize a node, use the following settings
  - `node.master`: true/false, the node can be master or not
  - `node.data`: true/false, the node can hold data or not
  - `node.ingest`: true/false, the node can process incoming documents with a processors pipeline

*A node which is not a master node nor an ingest or data node is called a coordinating only node*

Role	Master	Data	Ingest
Lighthead master	true	false	false
Search Load Balancer	false	false	true
Workhorse	false	true	false
Coordinating only node	false	false	false



# Split Brain

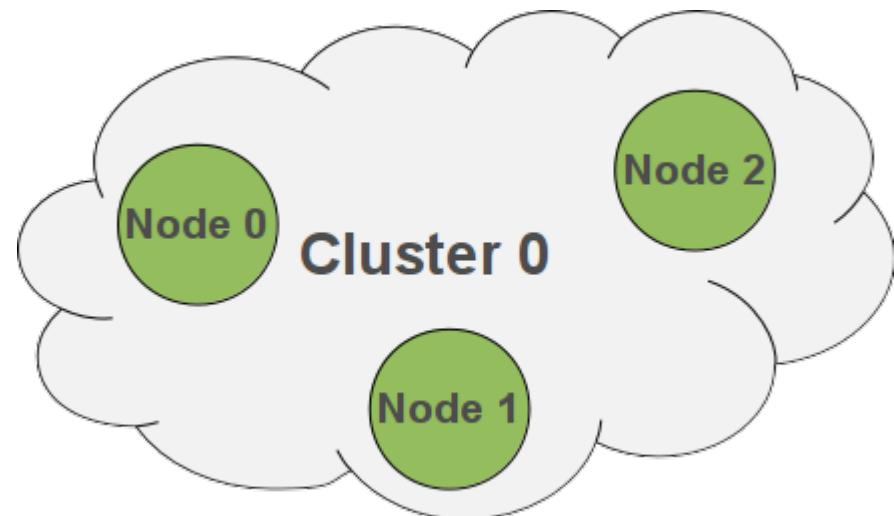
- A **Split Brain** occurs when nodes of a cluster get isolated from the other nodes
  - They may elect a new master
  - It can lead to having two masters in parallel
- To avoid this situation, it is mandatory to configure the **discovery.zen.minimum\_master\_nodes** property to be equal to

(number of eligible master nodes in the cluster / 2) + 1

# Index partitioning



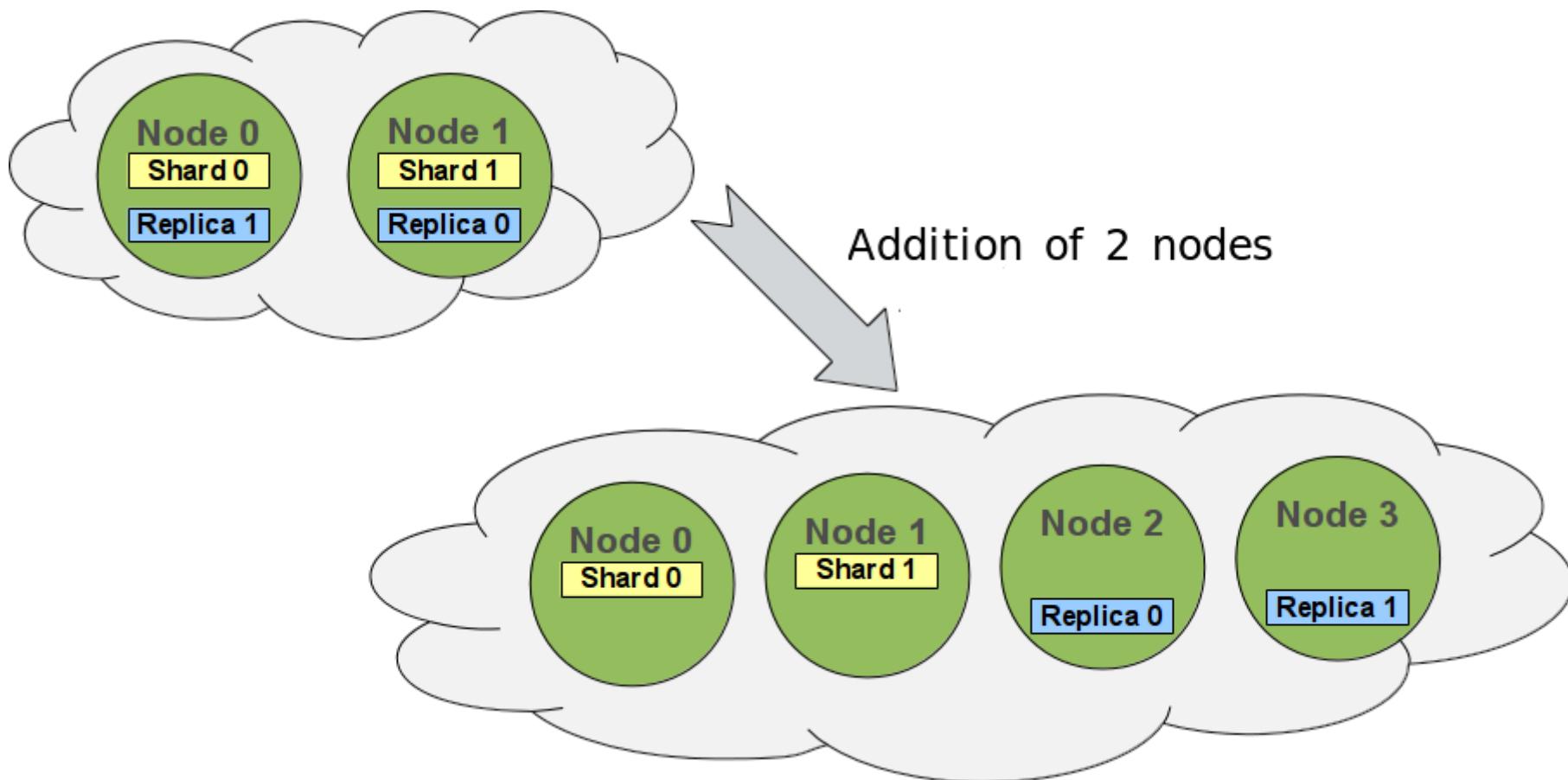
- Indices are **distributed** in the cluster
  - Each index is divided in one or several partitions (shard)
  - Each partition can be replicated several times (replica)
- Shards and replicas are automatically distributed in the cluster to ensure fault tolerance
- Example of cluster with
  - 2 nodes
  - 1 index
    - Shard = 2
    - Replica = 1





# Shard distribution 1/2

- A redistribution of the shards/replicas is automatically performed when the number of nodes in the cluster changes





# Shard distribution 2/2

- Some distribution commands

```
--- Freezing the shard distribution during restart
cluster.routing.allocation.enable: none # ou all
cluster.routing.rebalance.enable: none

--- Controlling shard allocation according to a node zone with a first configuration
node.name :node1
node.zone :zone1
--- On node 2
node.name :node2
node.zone :zone2

--- Then setting an index to prefer a zone
PUT /zenika/_settings
{
  "index.routing.allocation.include.zone" : "zone1",
  "index.routing.allocation.exclude.zone" : "zone2"
}

--- Controlling shard allocation according to the IP address
PUT /_cluster/settings
{
  "persistent" : {
    "cluster.routing.allocation.exclude._ip" : "192.168.0.11"
  }
}
```

- These commands can be configured at index or cluster level
- These commands can be executed while running



# Number of shards per node

- It is recommended to set a maximum number of shards per node
  - Can be specified at the index level
  - Hot reloading

```
PUT /zenika/_settings
{
  "index.routing.allocation.total_shards_per_node" : 2
}
```



# Shard allocation and disk space

- Elasticsearch adjusts shard allocation according to disk space occupation

```
PUT /_cluster/settings
{
  "transient" : {
    "cluster.routing.allocation.disk.threshold_enabled" : true,
    "cluster.routing.allocation.disk.watermark.low" : "60%",
    "cluster.routing.allocation.disk.watermark.high" : "10gb"
  }
}
```

- **watermark.low** (85% by default): the nodes won't accept any new shard
- **watermark.high** (90% by default): shards will be moved to other nodes

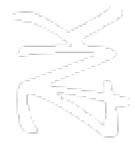
# Cluster Reroute API



- The Cluster Reroute REST API allows to move and reallocate shards

```
POST /_cluster/reroute
{
  "commands" : [
    { "move" : {
        "index" : "test", "shard" : 0,
        "from_node" : "node1", "to_node" : "node2"
      }
    },
    { "allocate" : {
        "index" : "test", "shard" : 1,
        "node" : "node3"
      }
    }
  ]
}
```

# Supervision





# Cluster health 1/2

- Elasticsearch provides a "Health" REST API which exposes the cluster health

```
GET /_cluster/health
{
  "cluster_name" : "dev-1.0",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 5,
  "active_shards" : 5,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 5}
```

- The **status** field indicates the cluster health
  - red**: indicates that a shard has not been allocated on the cluster
  - yellow**: indicates that a shard is allocated but not its replicas
  - green**: everything is ok



# Cluster health 2/2

--- It is possible to get the health status for a particular index

```
GET /_cluster/health/index1, index2
```

--- Get the detail per shard for an index

```
GET /_cluster/health/index1?level=shards
```

--- Waiting for the cluster to be in a particular status

```
GET /_cluster/health?wait_for_status=yellow&timeout=120s
```

# Cluster state



- Elasticsearch provides a "Cluster State" REST API which provides a global overview of the cluster static state

- Details for nodes, shards, templates, running allocations, version, ...

```
GET /_cluster/state
```

- Filter out the response for **metrics**: `version`, `master_node`, `nodes`, `routing_table`, `metadata`, `blocks`

```
GET /_cluster/state/{metrics}/{indices}
```



# Cluster statistics

- The "Cluster Stats" API provides global statistics about the cluster dynamic state

```
GET /_cluster/stats?human
```

- Available statistics:

- Number of indices
- Number of indexed and deleted documents
- Size of indices on disks
- Filter and field caches size
- Total number of segments
- Node details (CPU, Memory, File descriptors...)

# Pending tasks



- The "Pending Tasks" API provides information about tasks that haven't been performed yet

```
GET /_cluster/pending_tasks

--- Result
{
  "tasks": [
    {
      "insert_order": 101,
      "priority": "URGENT",
      "source": "create-index [foo_9], cause [api]",
      "time_in_queue_millis": 86,
      "time_in_queue": "86ms"
    },
    {
      "insert_order": 46,
      "priority": "HIGH",
      "source": "shard-started ([foo_2][1], node[tMTocMvQQgGCKj7QDHl30A], [P], s[INITIALIZING]),
      reason [after recovery from gateway]..."
    }
  ]
}
```

# Nodes information



- The "Node Info" REST API provides information about the cluster nodes static states.

```
GET /_nodes
```

--- Filtering metrics

```
GET /_nodes/node1,node2/_jvm,process
```

- Available information

- JVM
- Operating System, CPU, Memory, Swap...
- Processes
- Thread pool
- Network
- Plugins



# Nodes statistics

- The "Node Stats" REST API provides statistics on the nodes dynamic states

```
GET /_nodes/stats
```

- Collected statistics
  - File system
  - Network
  - Index
  - JVM (with Garbage Collector and memory)
  - Operating system
  - Processes
  - Thread pool



# Hot Threads

- The "Hot Threads" API allows to identify the most resource consuming threads
- Usage:

```
GET /_nodes/_hot_threads
```

# Supervision plugin | Monitoring (1/2)



- Monitoring supervision plugin for Elasticsearch and Kibana is free with **Elastic Stack basic license** (registration needed)
  1. On each node, install the **XPack module** and accept the java security manager changes
  2. Restart each node
  3. Install x-pack on **kibana** and then restart it
  4. Eventually install x-pack on **Logstash** and then restart it
  5. Install the **basic licence**
  6. Enter the default password '**changeme**'



# Supervision plugin | Monitoring (2/2)

--- Elasticsearch

```
bin/elasticsearch-plugin install x-pack
```

--- Kibana

```
bin/kibana-plugin install x-pack
```

--- Logstash

```
bin/logstash-plugin install x-pack
```

--- License activation

```
curl -XPUT -u elastic 'http://localhost:9200/\_xpack/license' -H "Content-Type: application/json" -d @user-21c7d4d8-643f-465e-8991-d8407314cf3b-v5.json
```

--- License Acknowledgment

```
curl -XPUT -u elastic 'http://localhost:9200/\_xpack/license?acknowledge=true' -H "Content-Type: application/json" -d @user-19c7d4d8-643f-465e-8991-d8407314cf3b-v5.json
```



# Log



- Log level is configured in the **config/logging.yml** file
  - Use the Java log4j 2 library
- Configuration hot reloading

```
PUT /_cluster/settings
{
  "transient" : {
    "logger.discovery" : "DEBUG"
  }
}
```

# Slow Log



- Index Slow Log
  - Allows to log the slowest request in a dedicated file
  - Logs request on each shard
  - Does not reflect the overall performance of a query

```
index.search.slowlog.threshold.query.warn: 10s
index.search.slowlog.threshold.query.info: 5s
index.search.slowlog.threshold.query.debug: 2s
index.search.slowlog.threshold.query.trace: 500ms
index.search.slowlog.threshold.fetch.warn: 1s
index.indexing.slowlog.threshold.index.info: 5s
```

- Configuration hot reloading

```
PUT /library/_settings
{
  "index": {
    "search.slowlog.level": "TRACE",
    "search.slowlog.threshold.query.trace": "2s"
  }
}

PUT /_cluster/settings
{
  "transient" : {
    "logger.index.search.slowlog" : "DEBUG",
    "logger.index.indexing.slowlog" : "WARN"
  }
}
```



# Cat API



- The "CAT" facilitates the use of Elasticsearch in command line
- Provides access to numerous features:
  - List nodes
  - List indices
  - Cluster health
  - Indices details
  - Shards details
  - Allocations
  - ...

# Cat API



- The "CAT" API is used in CLI:

```
GET /_cat/master
```

```
QQ7p922DSb-YwRPYpSqqaw zenika 127.0.1.1 Armageddon
```

- All the commands accept a **v** parameter to activate Verbose mode

```
GET /_cat/master?v
```

```
id          host    ip      node  
QQ7p922DSb-YwRPYpSqqaw zenika 127.0.1.1 Armageddon
```

- All the commands accept a "help" parameter to get details about the response

```
GET /_cat/master?help
```

id		node id
host	h	host name
ip		ip address
node	n	node name

# Cat API



- All the commands accept a "h" parameter to specify the headers to be returned in the response

```
GET /_cat/master?h=id
```

```
QQ7p922DSb-YwRPYpSqqaw
```

- For example to return the number of documents per index and shard:

```
GET /_cat/shards?h=index, shard, prirep, docs
```

```
library      4 r    1
library      4 p    1
library      1 r    2
library      1 p    2
...
```

# Cat API (1/2)



- List of `_cat` commands:

Command	Description
aliases	Information about index aliases
allocation	Details about shard allocations
count	Number of documents
fielddata	heap memory usage on each data node
health	Cluster health
indices	Information about the indices
master	Name and address of the master node
nodeattr	Shows custom nodes attributes
nodes	Information about the cluster nodes
pending_tasks	Details about the pending tasks
plugins	Details about the nodes running plugins with their versions



# Cat API (2/2)

Command	Description
recovery	Details about shards in "recovery"
repositories	Shows the snapshot repositories registered in the cluster
thread_pool	Information about threads pool
shards	Shards details
segments	Shows low level information about shards segments of an index
snapshot	Shows all snapshots that belong to a specific repository
templates	Provides information about existing index templates

- List all available commands with a GET request on the `_cat` API

# Snapshot & Restore



- Backup (**snapshot**) is performed on **repositories** and can be restored (**restore**)
- Elasticsearch supports several type of repositories:
  - Shared File System
  - AWS Cloud Plugin
  - Azure Cloud Plugin
  - HDFS Plugin

# Repository 1/2



- A **repository** is a storage space for future **snapshot**
- Before saving, declare the repository:

--- At the configuration level, we have to specify the path to repository (security reasons)  
path.repo: "/mount/backup"

--- Then you can create the logical repository named 'my\_snapshots'

```
PUT /_snapshot/my_snapshots
{
  "type": "fs",
  "settings": {
    "location": "/mount/backup/my_snapshots",
    "compress": true
  }
}
```

--- Once created, the repository configuration can be accessed like this

```
GET /_snapshot/my_snapshots
```



# Repository 2/2

- The "Snapshot" API allows to list the existing repositories:

```
GET /_snapshot
```

```
GET /_snapshot/_all
```



# Shared File System Repository

- The default repository type is "Shared File System"

```
PUT /_snapshot/my_snapshots
{
  "type": "fs",
  "settings": {
    "location": "/mount/backup/my_snapshots",
    "compress": true,
    "concurrent_streams": 5,
    "chunk_size": "10m",
    "max_restore_bytes_per_sec": "20mb",
    "max_snapshot_bytes_per_sec": "20mb"
  }
}
```

- Options

- location:** directory or root mount point for backups. Must be accessible by all the nodes and below the configured **repo.path**
- concurrent\_streams:** number of processes (per node) performing the backup
- chunk\_size:** allows to split the biggest files in fragments
- max\_xxx\_bytes\_per\_sec:** throttle the bandwidth for backups and restorations



# "Read Only" URL Repository

- A repository can be defined with an URL

```
PUT /_snapshot/snapshots_1
{
  "type": "url",
  "settings": {
    "location": "http://www.domain.com/snapshots_1",
    "concurrent_streams": 5
  }
}
```

- The repository will be read only, and will only allow restore operations
- Options
  - **location** : root URL pointing to a repository. Must be accessible from all the nodes.
  - **concurrent\_streams** : number of processes (per node) performing the backup



# Snapshot 1/3

- Each repository can contain several snapshots

```
--- Creates a backup  
PUT /_snapshot/sauvegardes/snapshot_1?wait_for_completion=true  
  
--- Result -----  
{ "snapshot":{  
    "snapshot":"snapshot_1",  
    "indices":["zenika","clients",".marvel-2014.02.10"],  
    "state":"SUCCESS",  
    "start_time":"2014-02-10T20:46:32.242Z",  
    "start_time_in_millis":1392583592242,  
    "end_time":"2014-02-10T20:46:36.764Z",  
    "end_time_in_millis":1392583596764,  
    "duration_in_millis":4522,  
    "failures":[],  
    "shards":{"total":11, "failed":0, "successful":11 }  
}  
}
```



# Snapshot 2/3

- By default all the started and opened indices are saved
- The list of indices to save can be restricted:

```
PUT /_snapshot/sauvegardes/snapshot_1
{
  "indices": "zenika_1,test*",
  "ignore_unavailable": "true",
  "include_global_state": false
}
```

- By default, each backup is
  - incremental
    - Only files since the last backup are saved
  - include the cluster state
    - **include\_global\_state** prevent from storing it every time
  - non blocking
    - limited impact a requests being executed



# Snapshot 3/3

- However, when a backup is running it blocks
  - the shards allocation
  - the execution of other backups

--- Views the details of a snapshot  
GET `/snapshot/sauvegardes/snapshot_1`

--- Deletes a snapshot  
DELETE `/snapshot/sauvegardes/snapshot_1`

# Restore



- Restore operation of a backup:

```
POST /_snapshot/sauvegardes/snapshot_1/_restore
```

- It is possible to limit the indices to be restored:

```
POST /_snapshot/sauvegardes/snapshot_1/_restore
{
  "indices": "zenika,clients",
  "ignore_unavailable": "true",
  "include_global_state": false,
  "rename_pattern": "zenika(.+)",
  "rename_replacement": "restored_index_$1"
}
```

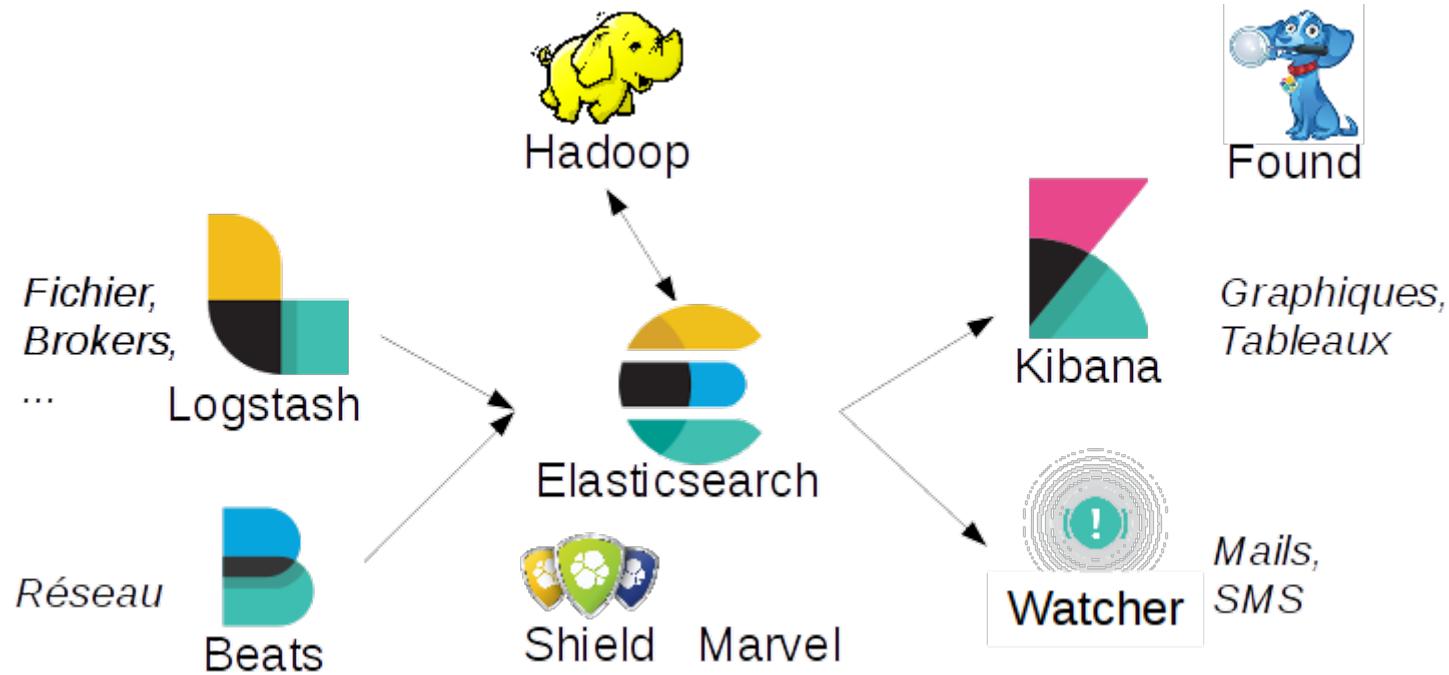
- Note :

- restored indices are automatically reopened
- an existing index can be restored only if it has been closed beforehand



# Advanced functionality

# Elasticsearch ecosystem



# Plugins 1/3



- Analysis
  - ICU Analysis
  - Phonetic
  - Analysis for simplified Chinese
  - Analysis for Japanese, Russian
- Scripting
  - Javascript
  - Python

# Plugins 2/3



- Discovery (Cloud)
  - AWS
  - Azure
  - GCE
  - Multicast ...
- Security
  - Shield
  - **Searchguard**
    - Searchguard free, Shield commercial
    - Security via ACL
    - HTTPS communication
    - Plugins for logstash/kibana



# Plugins 3/3



- Other :
  - **knapsack**
    - Import/export for Elasticsearch
    - HTTP API or Java
    - Archive formats (zip, tar...)
  - **reindex**
    - endpoint `_reindex`
    - Reindex from an existing to a new index



# Data import



- **Elasticsearch JDBC Importer**
  - Descendent of the JDBC River, transformed to standalone
  - Allows to import data from a relational database
- **Logstash**
  - log oriented: access logs, metrics
  - Many connectors: Files, brokers, ...
  - Transformation and parsing
- **Beats**
  - Agent for sending data to other systems
  - Filebeat for logfiles, PacketBeat, MetricBeat

# Scripting



- Elasticsearch can interpret different scripting languages using plugins
- Possible to write scripts for
  - Calculate a field when returning results
  - Modify the score
  - Modify the value of a field on partial update
  - Execute math functions (max, min, random, sin etc)
- Supported languages
  - Native : Painless, Groovy
- The script files can be placed in a folder config/scripts

# Percolate 1/5



- Elasticsearch provides a Rest API « Percolate » that can be used for
  - Register search queries for an index
  - Execute percolate requests that include a document
  - Retrieve the queries that match the document
- Opposite of the Elasticsearch default
  - Standard :
    - Index documents → Query the indexed documents
  - Percolation
    - Re却ister queries → Retrieve the matching queries for a document

# Percolate 2/5



```
--- Mapping for a simple book document type and a percolator document with percolator field
PUT /library
{
  "mappings": {
    "book": {
      "properties": {
        "title": { "type": "text" }
      }
    },
    "percolator-query": {
      "properties": {
        "query": { "type": "percolator" }
      }
    }
  }
}
--- Indexing a book
PUT library/book/1
{
  "title": "Les aventures de Tintin"
}
--- indexing a percolator query, e.g someone is searching for a comic strip with Tintin
PUT library/percolator-query/1
{
  "query": {
    "match": {
      "title": "aventures tintin"
    }
  }
}
```

# Percolate 3/5



```
--- Matching a document before its insertion
GET /library/_search
{
  "query" : {
    "percolate" : {
      "field" : "query",
      "document_type" : "book",
      "document" : {
        "title" : "Les aventures du capitaine Haddock"
      }
    }
  }
}
--- Result -----
{ [...],
  "hits": [
    {
      "_index": "library",
      "_type": "percolator-query",
      "_id": "1",
      "_score": 0.2824934,
      "_source": {
        "query": {
          "match": {
            "title": "aventures tintin"
          }
        }
      }
    }
  ]
}
```

# Percolate 4/5



--- It is also possible to percolate on indexed documents but it needs some tuning

--- Mapping a first library's book document type

PUT /library

```
{  
  "mappings": {  
    "book": {  
      "properties": {  
        "title": { "type": "text" }  
      }  
    }  
  }  
}
```

--- indexing a first book

PUT library/book/1

```
{ "title": "Les aventures de Tintin" }
```

--- Mapping percolator query in a dedicated index(requires a doctype with same title mapping)

PUT percolator

```
{  
  "mappings": {  
    "unused": {  
      "properties": {  
        "title": { "type": "text" }  
      }  
    },  
    "percolator-query": {  
      "properties": {  
        "query": { "type": "percolator" }  
      }  
    }  
  }  
}
```

# Percolate 5/5



```
--- Index a percolate query
PUT percolator/percolator-query/1
{
  "query": {
    "match": { "title": "aventures tintin" }
  }
}

--- Percolate
GET /percolator/_search
{
  "query" : {
    "percolate" : {
      "field": "query",
      "document_type" : "percolator-query",
      "index" : "library",
      "type" : "book",
      "id" : "1"
    }
  }
}
```

# Quelques exemples d'utilisation d'ElasticSearch



- Uses Open Data of New York on accidents
  - <https://www.elastic.co/blog/byodemos-new-york-city-traffic-incidents>
  - Uses Kibana to extract information
- One Million Tweets Map
  - Uses geo functionality
  - <http://onemilliontweetmap.com/>

