Introducing Elastic Search, Logstash and Kibana

David THIBAU – 2022

david.thibau@gmail.com

Agenda

- Introduction
  - ELK proposition and its use cases
  - Components of the stack
  - Distributions : Community versus X-Pack
  - Built-in features
  - Deployment alternatives
  - Core concepts of clustering
- Data ingestion
  - Beats concepts, provided beats by Elastic,
  - Logstash Concepts: pipelines, input, filters and output plugins
  - Elastic search pipelines
- Search engine capabilities
  - Inverted index and distributed search
  - Supported Data Types
  - Analyzers, predined and custom
  - DSL syntax for queries
  - Full text and relevance score
  - Advanced search
  - Aggregations or faceting
  - Geo queries
- Data Visualization with Kibana
  - What is Nearly-Real-time analysis?
  - Discover, Visualize and Dashboards
  - Basic visualisation
  - Time series
  - Location analysis
  - Data relationships with graph
  - Machine Learning

Introduction

ELK proposition and use cases

Distributions : Community vs X-Pack

Built-in features

Deployment alternatives

Core concepts of clustering

Introduction

The elastic company provides the ***ElasticStack suite*** in 2 editions (OpenSource and Enterprise)

The suite addresses 2 aspects, which van be applied in a BigData environment :

- **Near Real-time search**
  Permanently indexed data is available for search, the latency due to indexing is negligible (Near Real Time)
- **Near Real-time data analysis.**
  Data is aggregated in real time to provide visualizations and dashboards to extract insights

The core component of this stack is ***ElasticSearch***

ElasticSearch

ElasticSearch is a server offering a REST API[1] :

- To store and index data (Office documents, tweets, log files, performance metrics, ...)
- To perform search queries (structured, full-text, natural langage, geolocation)
- Aggregate data in order to calculate metrics

1. There is no User Interface embeded in the product Elastic Search

Features

- Massively **distributed and scalable** architecture in volume and load
  => Big Data, remarkable performance
- **High availability** through replication
- **Muti-tenancy** or multi-index. The document base contains several indexes whose life cycles are completely independent
- Based on the **Lucene** reference library
  => Full-text search, consideration of languages, natural language, etc.
- **Flexibility** : Structured storage of documents but without prior schema, possibility of adding fields to an existing schema
- Very complete **RESTFul API**
- **Open Source**

Apache Lucene

- Apache project Started in 2000, used in many products
- The "low layer" of ELS: a Java library for writing and searching **index files**:
  - An index contains ***documents***
  - A document contains ***fields***
  - A field consists of ***terms***

ELS brings scalability, a REST API, simplicity and aggregation features

http://lucene.apache.org/

ELS vs SolR

The Apache Foundation offers ***SolR*** which is very close to ELS in terms of functionality

	SOLR	ELS
Installation & Configuration	Not simple, Very detailed documentation	Simple and intuitive
Indexation/ Search	Natural langage oriented	Text and other data types for aggregations
Scalability	Clustering via ZooKeeper and SolRCloud	Natively cluster
Community	Important but stagnant	Has exploded
Documentation	Very complete but very technical	Very complete, many examples

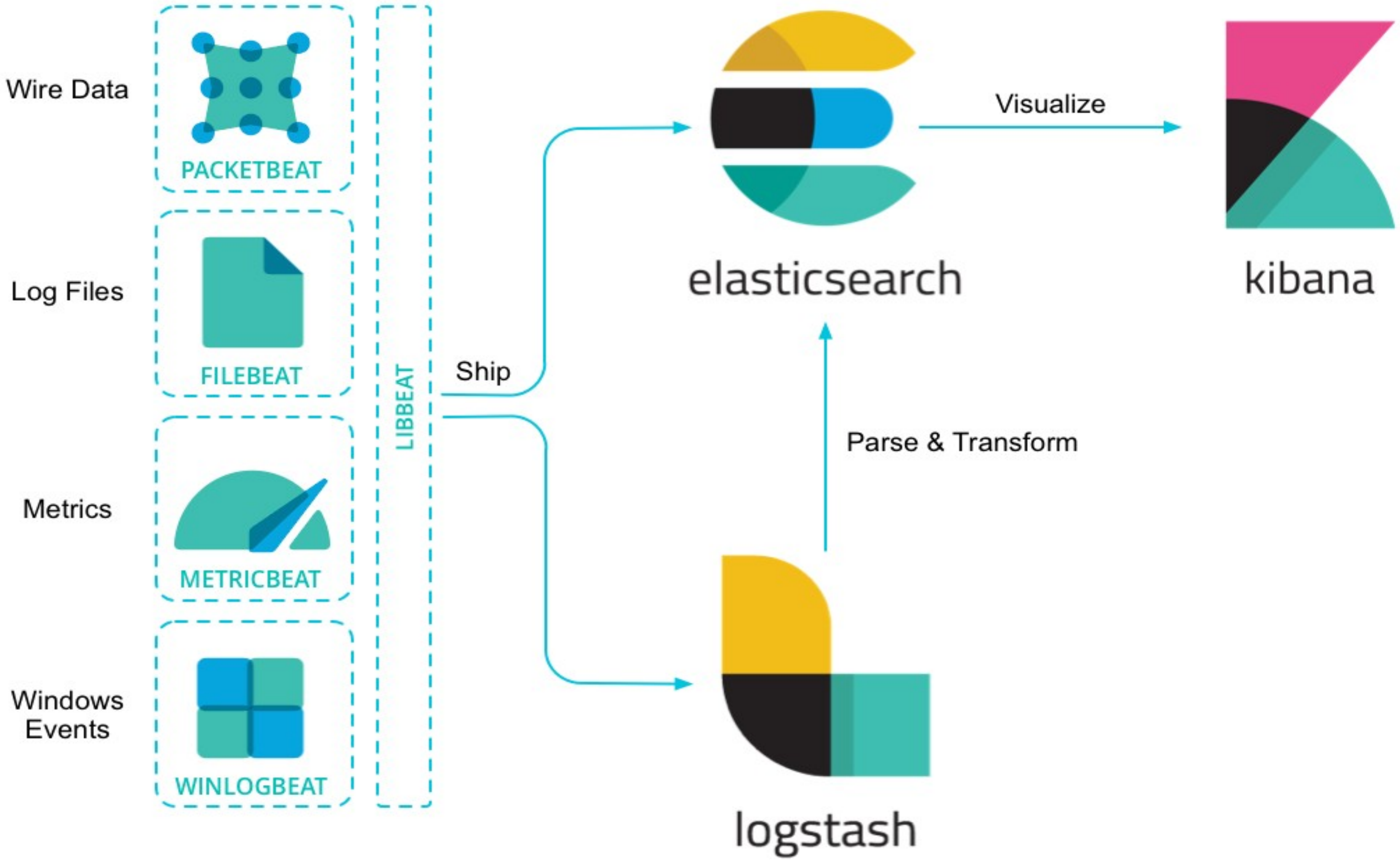
Elastic Stack

The suite Elastic Stack is dedicated to Real Time Analysis in the context of BigData.

It consist of :

- ***Logstash / Beat :*** Ingestion of data. Extract and transform
- ***Elastic Search*** : Storage, Indexing and Search
- ***Kibana*** : Data visualization

Architecture



Beats

- Beats are little programs installed on target devices of server
- They periodically sends data :
  – Directly to ElasticSearch
  – Or to a logstash pipelines

Logstash

Logstash is a tool which takes as input events which can come from different sources

It transform, enrich filter these events

Then, it writes the transformed event in different destinations but mainly in ElasticSearc indexes

Clients Alternatives for ElasticSearch

- **Classical REST clients**:
curl, Postman, SOAPUI, ...
- **Libraries** provided by Elastic :
Java, Javascript, Python, Groovy, Php, .NET, Perl
Developers can easily develop programs which integrates ElasticSearch
- **Kibana** is a generic client for managing cluster, provide dashboards, develop queries and many more

Kibana

Kibana is an application for :

- Data managers
- Operators of the stack
- Developers of application which integrates ElasticSearch

It has lot of menus and also lot of marketing contents

Introduction

ELK proposition and use cases
Distributions : Community vs X-Pack
Built-in features
Deployment alternatives
Core concepts of clustering

Releases

Releases are available for download in different formats :

- Archive ZIP, TAR,
- Packages DEB or RPM
- Docker Images

Versions

- 8.0.0 : February 2022
- 7.x : Avril 2019 - Continuing
- 6.x : November 2017 to December 2019
- 5.0.0 : October 2016
- 2.4.1 : September 2016
- 2.4.0 : August 2016

Related Offers

The company Elastic also offers:

- ***X-Pack*** : commercial and enterprise features
  - Security, Monitoring (before release 8.x)
  - Alerts, Machine Learning
  - Application Performance Management
  - Reporting and scheduling
- ***Elastic Cloud*** : Delegate operations of the cluster to the elastic company or deploy the solution in a private cloud

Introduction

ELK proposition and use cases
Distributions : Community vs X-Pack
Built-in features
Deployment alternatives
Core concepts of clustering

Solutions out-of-the-box

Enterprise Search : Define sources of content (web, other..), index content and provides search capabilities

Elastic Observability : Monitor CPU and memory consumption, agregate logs, inspect network traffic, Application Performance Management .
Machine Learning to detect anomalies and anticipate future

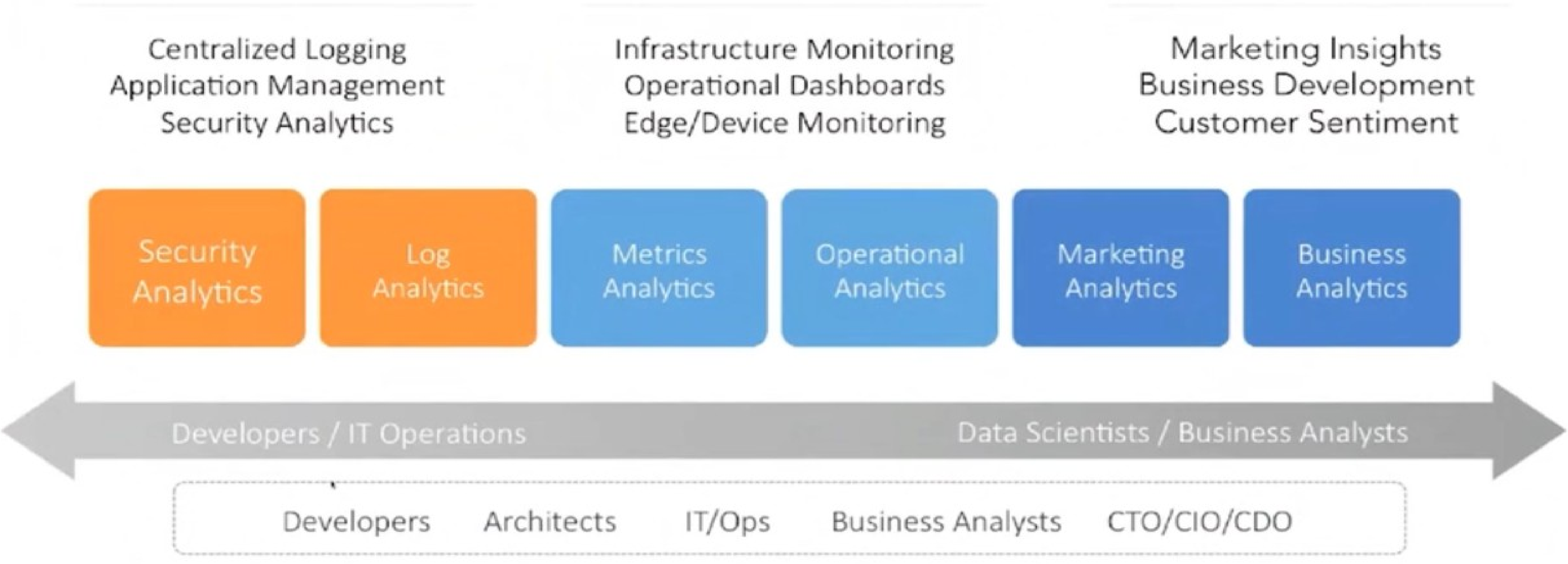
Elastic Security : Detect and respond to threats

Other Use Cases

- The stack can have very different usages
  - Business Analytics : Real-time analysis of KPI
  - Marketing : User behavior, frequentation analysis, marketing segmentation
  - Business risk management and fraud detection
  - IoT (domotic or health sensors ...)
  - Big Data in general

https://www.elastic.co/customers/

Elastic Use cases



Introduction

ELK proposition and use cases
Distributions : Community vs X-Pack
Built-in features
Deployment alternatives
Core concepts of clustering

Sizing of the cluster

Depending on the use case, we can deploy different kind of cluster.

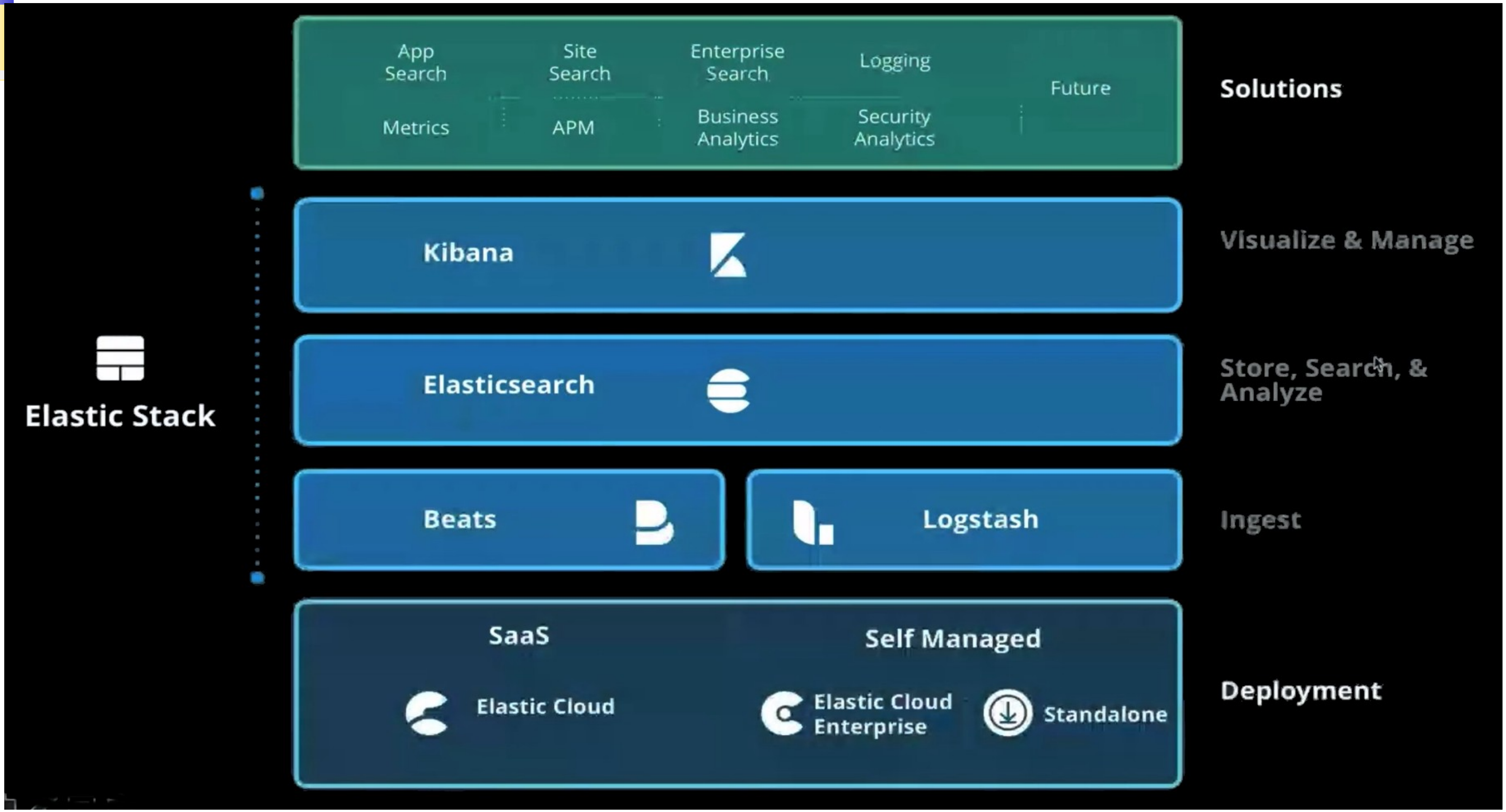
- Simple Search Engine with a reasonable amount of data :

  A one node cluster may be sufficient
- Large amount of data and/or different indices
  A 3 nodes cluster and all the nodes have the same configuration
- Big Data :
  Cluster with different kind of specialized nodes (ingestion, queries, machine learning jobs, …)
- Big Data and global company
  A cluster spread on different data centers

Where to deploy ?

- Standalone bare-metal or VMS operate by your own
- Private cloud (Kubernetes, OpenShift, OpenStack) operate by your own
- Elastic Cloud provided by elastic company
- Other public clouds also proposed ELK services

In summary



App Search
Site Search
Enterprise Search
Logging
Future
Metrics
APM
Business Analytics
Security Analytics
Solutions
Kibana
Visualize & Manage
Elastic Stack
Elasticsearch
Store, Search, & Analyze
Beats
Logstash
Ingest
SaaS
Self Managed
Elastic Cloud
Elastic Cloud Enterprise
Standalone
Deployment

Introduction

ELK proposition and use cases
Distributions : Community vs X-Pack
Built-in features
Deployment alternatives
Core concepts of clustering

Cluster

A **cluster** is a set of servers (nodes) that contains all the data and offers search capabilities on the different nodes

- It has a unique name in the local network (default : "*elasticsearch*").

=> A cluster can consist of only one node

=> A node cannot belong to 2 separate clusters

Node

A node is a single server (Java process) that is part of a cluster.

A node generally stores data, and participates in the indexing and search functionalities of the cluster.

A node is also identified by a unique name (generated automatically if not specified)

The number of nodes in a cluster is not limited

Master node

In a cluster, a node is elected as the **master node**, it is in charge of managing the configuration of the cluster and the creation of indexes.

For all document operations (indexing, searching), each data node is interchangeable and a client can address any of the nodes

In large architectures, nodes can be specialized for data ingestion, Machine Learning processes, and in this case, they do not participate in search functionalities.

Index

- An index is a collection of documents that have similar characteristics
  - For example an index for customer data, another for the product catalog and yet another for orders
- An index is identified by a name (in lower case)
  - The name is used for indexing and search operations
- In a cluster, you can define as many indexes as you want

Document

A **document** is the basic unit of information that can be indexed.

- document has a set of fields (key/value) expressed with JSON format

Inside an index, you can store as many documents as you want

Sample

```
{
  "name":  "John Smith",      // String (may be analayzed or not)
  "age": 42,                  // Nombre
  "confirmed": true,          // Booléen
  "join_date": "2014-06-01",  // Date
  "home": {                   // Imbrication
    "lat": 51.5,
    "lon": 0.1
  },
  "accounts": [               // tableau de données
    {
    "type": "facebook",
    "id": "johnsmith"
    }, {
    "type": "twitter",
    "id": "johnsmith"
    }
  ]
}
```

Metadata

Metadata is also associated with each document.

Main metadata are :

- ***_index*** : The location where the document is stored
- ***_id*** : The unique identifier
- ***_version*** : The version number of the document
- ...

Shard

- An index can store a very large amount of documents that may exceed the limits of a single node.
  - To overcome this problem, ELS allows you to subdivide an index into several parts called ***shards***
- When creating the index, it is possible to define the number of shards
- Each shard is an independent index that can be hosted on one of the cluster nodes

Benefits of sharding

Sharding allows :

- To scale the volume of content
- To **distribute** and parallelize operations => improve performance

Replica

To tolerate failures, it is recommended to use fail-over mechanisms in the event that a node fails (Hardware failures or upgrade of a system)

ELS allows you to set up copies of shards: **replicas** in order to

- **High-availability :** A replica is hosted on a different node than the primary shard
- To **scale** . Requests can be load balanced on every replica .

Data Ingestion

Beats

Logstash Concepts: pipelines, input, filters and output plugins

Elastic search pipelines

Introduction to Beats

Beats convey data

They are installed as agents on the different targets (servers) and periodically send their data

- Directly to *ElasticSearch*
- Or, if the data need to be transformed, to *Logstash*

Types of beats

ELK provides :

- ***Packetbeat*** : All the network packets seen by a specific host
- ***Filebeat*** : All the lines written in a log file .
- ***Metricbeat*** : Performance metrics on CPU, Memory, Disk, IO, etc
- ***Winlogbeat*** : Windows log events

They all produce data which can be directly ingested in ElasticSearch

They are also distributed with Kibana dashboard which can easily be imported and used

Other beats provided by third party are also available

Community beats

amazonbeat : Reads data from a specified Amazon product.

apachebeat, ***nginxbeat*** : Reads status from HTTPD server-status.

browserbeat : Reads and ships browser history (Chrome, Firefox, & Safari) to an Elastic output.

githubbeat : Easily monitors GitHub repository activity.

httpbeat : Polls multiple HTTP(S) endpoints and sends the data to Logstash or Elasticsearch.

kafkabeat : Reads data from Kafka topics.

prometheusbeat : Send Prometheus metrics to Elasticsearch

twitterbeat : Reads tweets for specified screen names.

...

Demo : MetricBeats and its dashboards

Data Ingestion

Beats

Logstash Concepts: pipelines, input, filters and output plugins

Elastic search pipelines

Introduction

- Logstash is a **real-time data collection** tool able to unify and normalize data from different sources
- Originally focused on log files, it has evolved to handle very diverse events
- Logstash is based on the notion of processing **pipelines**.
- Extensible via plugins, it can read/write from/to very different sources and targets

Sources of events

Traces and metrics : Server logs, syslog, Windows Event, JMX, ...

Web : HTTP request, Twitter, Hook for GitHub, JIRA, Polling d'endpoint HTTP

PersistenceStore : JDBC, NoSQL, *MQ

Miscellaneous sensors : Mobile phones, Connected Home or Vehicles, Health sensors

Data enrichment

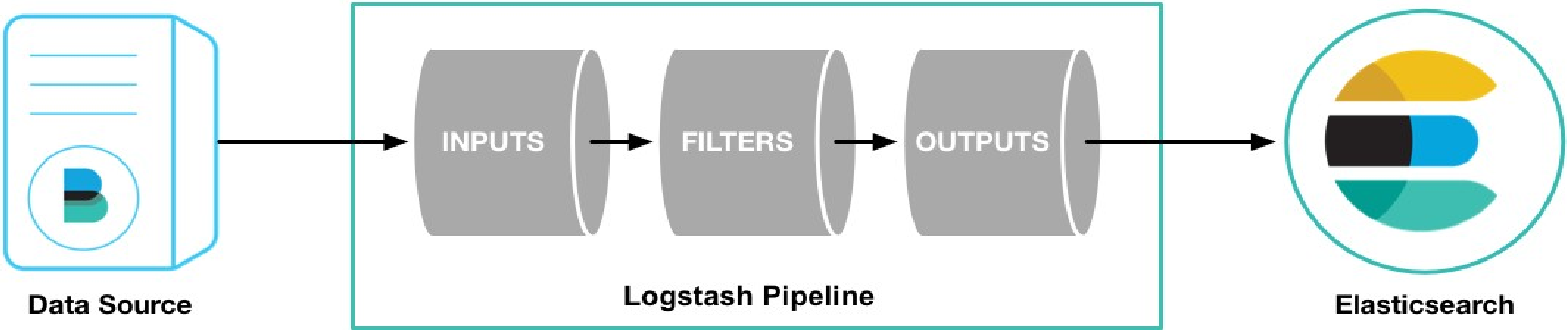
Logstash also allows you to enrich the input data.

- Geo-location from an IP address. (lookup to a service)
- Data encoding/decoding
- Date normalization
- Elastic Search queries for enrichment
- Anonymize sensitive information

Pipeline Logstash

A logstash **pipeline** defines :

- An input plugin to read data from a source
- An output plugin to write data to a target (typically ElasticSearch)
- Optional filters to perform transformations



Inputs

Some inputs :

- ***beats*** : Events produced by a beat
- ***elasticsearch*** : ElasticSearch Query response
- ***exec*** : Shell command output
- ***file*** : Reading lines of files
- ***jdbc*** : Records from database
- ***http-poll***, ***http***, ***STOMP*** : Events received via HTTP
- ***Imap***, ***Irc***, ***rss*** : Read mails, irc, rss, twitter
- ***Kafka***, ***redis***, ***rabbitmq***, ***jms*** : Messages in message brokers
- ***Tcp***, ***udp***, ***unix***, ***syslog*** : Low level socket events ...

Some filters

geoip : Ajoute des informations latitude/longitude à partir d'une IP

grok : Divise un champ en d'autres champs via des expressions régulières

mutate : Transforme un champ

prune : Filtre des événements à partir d'une liste rouge ou blanche

range : Vérifie que la taille ou longueur d'un champs est dans un intervalle

translate : Remplace les valeurs des champs à partir d'une table de hash ou fichier YAML

truncate : Tronque les champ supérieur à une certaine longueur

urldecode : Décode les champs URL-encoded

useragent : Parse les chaînes user agent

xml : Parse le format XML

Output plugins

Some outputs

- ***csv*** : Write CSV files
- ***elasticsearch*** : Index into Elasticsearch
- ***email*** : Send the event via mail
- ***file*** : Writes into a file
- ***stdout*** : Writes in the standard output
- ***pipe*** : Send to standard input of another program
- ***http*** : Send to an HTTP or HTTPS endpoint
- ***kafka***, ***redis***, ***rabbitMQ*** : Messaging
- ...

Elasticsearch Output

Main options of elasticsearch outpus are :

- ***hosts*** : Data nodes of the cluster
- ***index*** : The name of the index. Example : logstash-%{+YYYY.MM.dd}.
- ***template*** : A path to a template to create the index
- ***template_name*** : The tempate's name present in ElasticSearch
- ***document_id*** : The id of the ElasticSearch document (allow us updates)
- ***pipeline*** : The ElasticSearch pipeline to execute before indexing
- ***routing*** : To specify the shard to use

Data Ingestion

Beats
Logstash Concepts: pipelines, input, filters and output plugins
Elastic search pipelines

Introduction

Without logstash, it is possible to perform common transformations on the data before indexing by using the **ingest pipelines of ElasticSearch**

Features are similar to logstash but less powerful

ELS pipelines can be used to remove fields, extract values from text, and enrich your data.

It consists of a series of configurable tasks called ***processors***. Each processor runs sequentially, making specific changes to incoming documents

Pipeline

A **pipeline** is defiined via :

- A name
- A description
- And a list of *processors*

Processors are predefined

They will ve executed in sequence

Usage

```
#Create a pipeline named attachment
PUT _ingest/pipeline/attachment
{
  "description" : "Extract attachment information",
  "processors" : [
    {  "attachment" : { "field" : "data" } }
  ]
}
# Using a pipeline during indexation of a document.
PUT my_index/my_type/my_id?pipeline=attachment
{
  "data":
"e1xydGYxXGFuc2kNCkxvcmVtIGlwc3VtIGRvbG9yIHNpdCBhbWV0DQpccGFy
IH0="
}
```

Result

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

Available processors

Append : Add a field

Convert : Data type conversion

CSV : CSV format

Date : Create a @timestamp field

Drop : Drop the document on some condition

Enrich : Entich data from other index

Fingerprint : Compute a fingerprint to detect doublon

GeoIP : Determine geo coordinates from IP Adress

HTML strip : Remove HTML tags

Lowercase : Transform to lowercase som efields

Remove : Remove some fields

Rename : Rename a field

User agent : Split the user agent string

ELS plugins

The functionalities of ELS can be increased via the notion of plugin

Plugins contain JAR files, but also scripts and configuration files.
They add new features, for example new pipeline's processor

They can be installed easily with

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

Ingest plugins

A particular ingest plugins allow to index office documents (Word, PDF, XLS, ...) :

- ***Attachment Plugin*** : Extracts text data from attachments in different formats (PPT, XLS, PDF, etc.). It uses the Apache Tika library.

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

Search Engine Capabilities

Inverted index and distributed search

Supported Data Types

Analyzers, predined and custom

DSL syntax for queries

Full text and relevance score

Advanced search

Aggregations or faceting

Geo queries

Elastic Search

2 techniques allows ElasticSearch to obtain amazing performance :

- The inverted index (Lucene)
- The distributed search among the nodes of the cluster

Inverted index

In order to speed up searches on text fields, ELS uses a data structure called ***inverted index***

This consists of a unique word list where each word is associated with the documents in which it appears.

The method list is created after an analysis phase of the original text field.

Example

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

Distributed search

Research requires a complex execution model, with documents scattered across shards

ElasticSearch must consult a copy of each shard of the index or indexes requested

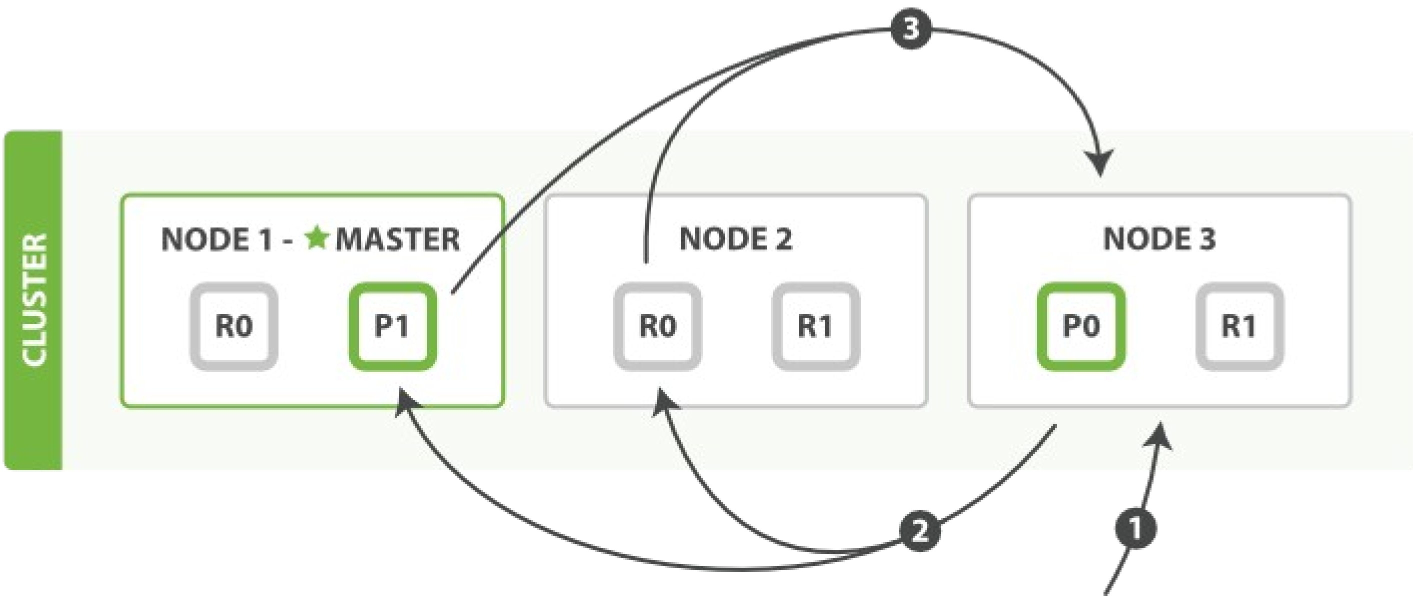
When found; results from different shards must be combined into a single list so that the API can return a page of results

The search is therefore executed in 2 phases:

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

Query phase

During the 1st phase, the request is broadcast to a copy (primary or replica) of all shards. Each shard performs the search and builds a priority queue of matching documents



Query phase

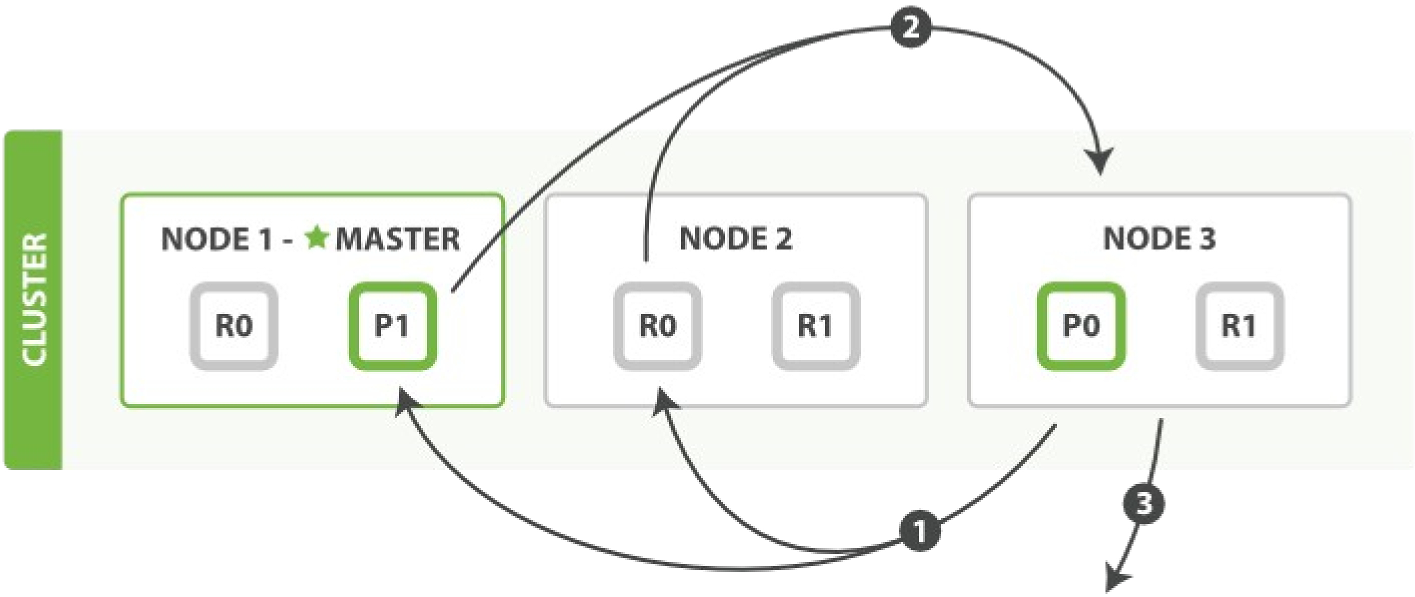
1. The client sends a query to node 3 which creates an empty priority queue of size = *from* + *size* .

2. Node 3 forwards the request to a copy of each shard in the index. Each shard executes the query locally and adds the result to a local priority queue of size = *from* + *size* .

3. Each shard returns the IDs and sort values of all documents in the queue to node 3 which merges these values into its priority queue

Phase fetch

The fetch phase consists of retrieving the documents present in the priority queue.



Fetch phase

1. The coordinator identifies which documents need to be retrieved and issues a multiple GET request to the shards.

2. Each shard loads the documents, enriches them if necessary (highlighting for example) and returns them to the coordinator node

3. When all documents have been retrieved, the coordinator returns the results to the client.

Search Engine Capabilities

Inverted index and distributed search

Supported Data Types

Analyzers, predined and custom

DSL syntax for queries

Full text and relevance score

Advanced search

Aggregations or faceting

Geo queries

Introduction

Each field of a document has an associated datatype.

The metadata which specifies the differents fields and their associated types is called the **mapping**

The mapping API is used to define, visualize, update the mapping of an index

Supported simples data types

ELS supports the following simple data types :

- 2 kinds of string : *text* (analyzed) or *keyword* (not analyzed)
- Numbers : *byte , short , integer , long, float , double, token_count*
- Booleans : *boolean*
- Dates : *date*
- Bytes : *binary*
- Range *: integer_range, float_range, long_range, double_range, date_range*
- IP adress *: IPV4* ou *IPV6*
- Geolocation *: geo_point, geo_shape*
- A query (JSON structure) *: percolator*

Complex data types

In addition to simple types, ELS supports

- **arrays** : There is no special mapping for arrays. Each field can contain 0, 1 or n values
  `{ "tag": [ "search", "nosql" ]}`
  - Array values must be of the same type
  - A null field is treated as an empty array
- **objects** : It is a data structure embedded in a field
  - Each sub-field is accessed with *object.property*

Exact value or full-text

String stored by ELS can be of 2 data types :

- **keyword** : The value is taken as is, filter type operators can be used when searching
  `Foo!= foo`
- **text** : The value is analyzed and split into terms or tokens. Full-text search operators can be used when searching. This concerns data in natural language

Mapping

- ELS is able to dynamically generate the mapping during indexation
  - It guesses the type of fields
  - For string, it uses the value as text and keyword datatypes
- Even if this feature is intersting for a proof or concept, in a real project you have define the datatype of your index and to choose how the text string will be analyzed by the search Engine.
- These decisions are important because you cannote change a mapping after indexing documents

Search Engine Capabilities

Inverted index and distributed search
Supported Data Types
Analyzers, predined and custom
DSL syntax for queries
Full text and relevance score
Advanced search
Aggregations or faceting
Geo queries

Tokenization and Normalization

To build the inverted index, ELS needs to separate a text into words (***tokenization***) and then ***normalize*** the words so that searching for the term "sending" for example returns documents containing: "sending", "Sending", "sendings", "send", …

Tokenization and normalization are called **analysis**.

The analysis applies to the documents during indexing **AND** during the search on the search terms.

Analysis steps

Analyzers transform a text into a stream of “tokens”. It is a combination of:

- ***Character filters*** prepare text by performing character replacement (& becomes *and*) or deletion (removal of HTML tags)
- **Tokenizer .** It splits a text into a series of lexical units: tokens (only 1 per analyzer)
- **Filters** take a token stream as input and transform it into another token stream

Predefined analyzers

ELS offers directly usable analyzers:

- ***Standard Analyzer*** : This is the default analyzer. The best choice when the text is in various languages. It consists of :
  - Split text into words
  - Remove punctuation
  - Lowercase all words
  - Supports stop words
- ***Simple analyzer*** : Separates text based on character different than a letter, changes to lowercase
- ***Stop analyzer*** : Idem as simple but supports stop words
- ***WhiteSpace Analyzer***: Separates text based on spaces
- ***Pattern analyzer*** : Based on regular expression
- ***Language analyzer***: These are language-specific parsers. They include "stop words" (remove most common words) and extract the root of a word. This is the best choice if the index is in one langage
- ***Fingerprint analyzer*** : Use for duplicate detection

Testing analyzers

Every analyzers can be tested via the API :

```
GET /_analyze?analyzer=standard
Text to analyze
```

<u>Réponse :</u>

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

Demo : Testing predefined Analyzers

Specify mapping at index creation

```
PUT /office2
{
  "mappings": {
    "dynamic": false,
      "properties": {
        "author": { "type": "text" },
        "content": {
          "type": "text",
          "analyzer": "english",
          "fields": {
            "fr": {
              "type": "text",
              "analyzer": "french"
            }
          }
        },
        "content_length": {  "type": "long"  },
        "date": {  "type": "date" },
        "description": { "type": "text" },
        "language": { "type": "keyword" },
        "title": {  "type": "text" }
      } } }  } }
```

2 fields are available for full-text searching : *content* and *content.fr*

Custom analyzers

- It is possible to define yours custom analyzers
  - Either by defining precisely the tokenizer, character filters and filters to use
  - Either by overriding an existing analyzer

Example

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

Predefined filters

ELS provides many filters for building custom analyzers. For example :

- ***Length Token*** : Deleting words that are too short or too long
- ***N-Gram*** and ***Edge N-Gram*** : Analyzer to speed up search suggestions
- ***Stemming filters*** : Algorithm to extract the root of a word
- ***Phonetic filters*** : Phonetic representation of words
- ***Synonym*** : Word Match
- ***Keep Word*** : The opposite of stop words
- ***Limit Token Count*** : Limit the number of tokens associated with a document
- ***Elison Token*** : Handling of apostrophes (exemple : French)

Synonym filter

Synonyms can be used to merge words that have nearly the same meaning.

Ex : pretty, cute, beautiful

They can also be used to make a word more generic.

- For example, bird can be used as a synonym for pigeon, sparrow, …

Example

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

Search Engine Capabilities

Inverted index and distributed search

Supported Data Types

Analyzers, predined and custom

DSL syntax for queries

Full text and relevance score

Advanced search

Aggregations or faceting

Geo queries

Introduction

Searches with a request body offer more functionalities than the search lite mechanism.

In particular, they allow you to :

- Combine query clauses more easily
- To influence the score
- Highlight parts of the result
- Aggregate subsets of results
- Return suggestions to the user

Combination example

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

Ex: Combination of combinations

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

Distinction between query and filter

Distinction between query and filter

- ***Filters*** are used for fields with exact values.
  Their result is of Boolean type, i.e. a document satisfies a filter or not
  They are very efficient

- ***Queries*** calculate a relevance score for each document found.
  The result is sorted by the relevance score by default
  More computation but more relevant

Types of operators

Operators can be classified in 2 families :

- Operators that do not perform analysis and operate on a single term (*term, fuzzy*).

- The operators that apply the analyzer to the search terms corresponding to the searched field (*match, query_string*) .

Operators without analysis

term : Used to filter exact values :

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

terms : Allows multiple values to be specified :

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

range : Allows you to specify a date or numeric range:

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

exists and ***missing*** : Allows you to test whether or not a document contains a field

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

match

The ***match*** operator is the standard operator to execute full-text search.

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

It is a Boolean query which by default analyzes the words passed as parameters and builds an OR type query.

multi_match

The ***multi_match*** operator allows you to run the same query on multiple fields:

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

Wildcards can be used for fields

Fields can be boosted with notation ^

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

query_string

query_string operator uses a parser to understand the query string. (the same as for lite search)

It has many parameters (default_field, analyzer, ...)

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

simple_query_string

The ***simple_query_string*** operator avoids causing a parsing error in the search query.

It can be used directly with what a user has entered in a search field

It understands the simplified syntax:

- + for AND
- | for OR
- - for negation
- ...

Search Engine Capabilities

Inverted index and distributed search
Supported Data Types
Analyzers, predined and custom
DSL syntax for queries
Full text and relevance score
Advanced search
Aggregations or faceting
Geo queries

Sorting

During filter-type queries, it may be interesting to set the sorting criterion.

This is done via the ***sort*** parameter

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

Response

In the response, the *_score* field is not calculated and the value of the sort field is specified for each document


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

Other sorting features

It is possible to combine a sorting criterion with another criterion or the score. Please note the order is important.

It is also possible to use an aggregate function to sort multivalued fields (*min*, *max*, *sum*, *avg*, ...)

Relevance

The score of each document is represented by a floating point number (_score). The higher this number, the more relevant is the document.

The algorithm used by ELS is called ***term frequency/inverse document frequency***, or ***TF/IDF.***

It takes into account the following factors:

- The **frequency of the term** : how many times the term appears in the field
- The **inverse document frequency** : How many times does the term appear in the index? The more the term appears, the less weight it has.
- The **normalized length of the field** : The longer the field, the less relevant the terms

Depending on the type of query (fuzzy query, boost factor …) other factors can influence the score

Explanation of relevance

With the ***explain*** parameter, ELS provides an explanation of the score

```
GET /_search?explain
{ "query"
: { "match" : { "tweet" : "honeymoon" }}
}
```

The *explain* parameter can also be used to understand why a document matches or not.

```
GET /us/12/_explain
{
"query" : {
  "bool" : {
    "filter" : { "term" : { "user_id" : 2 }},
    "must" : { "match" : { "tweet" : "honeymoon" }}
} } }
```

Response


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

multi_match and relevance score

With the operator *multi_match*, the ***type*** parameter specifies the way the score is calculated.

- ***best_fields*** (default) : Find documents that match one of the fields but use the best field to assign the score
- ***most_fields*** : Combines the score of each field
- ***cross_fields*** : Concatenates all fields and uses the same parser
- ***phrase*** : Use a *match_phrase* operator on each field and combine each field's score
- ***phrase_prefix*** : Use a *match_phrase_prefix* operator on each field and combine each field's score

bool and relevance score

```
GET /my_index/my_type/_search
{ "query": {
  "bool": {
    "must": { "match": { "title": "quick" }},
    "must_not": { "match": { "title": "lazy" }},
    "should": [
      { "match": { "title": "brown" }},
      { "match": { "title": "dog" }}
] } } }
```

The calculation of relevance is done by adding the score of each *must* or *should* clause and dividing by 3

Boosting clause

It is possible to give more weight to a particular clause by using the ***boost*** parameter

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

boosting operator

The ***boosting*** operator allows you to specify a clause that reduces the score of matching documents

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

Function score query

The function_score allows you to modify the score of documents that are retrieved by a query.

Different functions to evaluate the score can be used :

- **script_score** : customize the scoring of with a computation derived from other numeric field values in the doc using a script expression
- **weight** : allows you to multiply the score by the provided weight
- **random_score** generates scores that are uniformly distributed
- **field_value_factor** : function allows you to use a field from a document to influence the score. It’s similar to using the *script_score*

Example : script_score

```
GET /_search
{
  "query": {
    "function_score": {
      "query": {
        "match": { "message": "elasticsearch" }
      },
      "script_score": {
        "script": {
          "source": "Math.log(2 + doc['views'].value)"
        }
      }
    }
  }
}
```

Example *Field Value factor*

```
GET /_search
{
  "query": {
    "function_score": {
      "field_value_factor": {
        "field": "my-int",
        "factor": 1.2,
        "modifier": "sqrt",
        "missing": 1
      }
    }
  }
}
```

Search Engine Capabilities

Inverted index and distributed search
Supported Data Types
Analyzers, predined and custom
DSL syntax for queries
Full text and relevance score
Advanced search
Aggregations or faceting
Geo queries

Partial Matching

Partial matching allows users to specify a portion of the term they are looking for

Common use cases are:

- Matching postal codes, serial numbers or other not_analyzed values that start with a particular prefix or even a regular expression
- On-the-fly search: searches performed on each character typed to make suggestions to the user
- Matching in languages like German that contain long compound nouns

prefix filter

The *prefix* filter is a search performed on the term. It does not parse the search string and assumes that the correct prefix has been provided.

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

Wildcard and regexp

wildcard or ***regexp*** searches are similar to *prefix* but allow the use of wildcards or regular expressions

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

Indexing for auto-completion

The principle is to index the beginnings of words of each term

This can be done through a particular ***edge_ngram*** filter

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

For each term, it creates n tokens of minimum size 1 and maximum 20. The tokens are the different prefixes of the term.

match_phrase

The ***match_phrase*** operator parses the search string to produce a list of terms but only keeps documents that contain all of the terms in the same position.

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

Proximity and *slop* parameter

It is possible to introduce flexibility to phrase matching by using the slop parameter which indicates how far apart the terms can be.

Documents with these closest terms will have better relevance.

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

match_phrase_prefix

The ***match_phrase_prefix*** operator behaves like *match_phrase*, except it treats the last word as a prefix

It is possible to limit the number of expansions by setting the *max_expansions* parameter

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

Fuzzy query

Fuzzy query returns documents that contain terms similar to the search term, as measured by a Levenshtein edit distance.

An edit distance is the number of one-character changes needed to turn one term into another. These changes can include:

- Changing a character (box → fox)
- Removing a character (black → lack)
- Inserting a character (sic → sick)
- Transposing two adjacent characters (act → cat)

Most typos or misspelling errors have a distance of 1.

=> So 80% of errors could be fixed with single character editing

Fuzzy query

It is possible during a request to position the ***fuzziness*** parameter which therefore gives the Levenshtein distance tolerance.

This parameter can also be set to AUTO:

- 0 for strings of 1 or 2 characters
- 1 for strings of 3, 4 or 5 characters
- 2 for strings longer than 5 characters

<u>Be careful:</u> No stemmer with fuzzy queries!

Example

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

Highlighting

Highlighting consists of highlighting the matched term in the original content that has been indexed.

This is possible thanks to the metadata stored during indexing

The original field must be stored by ELS

Example

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

Response


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

Tags used in the response

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

Fragments

It is possible to control the highlighted fragments for each field:

- ***fragment_size*** : gives the max size of the highlighted fragment
- ***number_of fragments*** : The maximum number of fragments in this field

Search Engine Capabilities

Inverted index and distributed search
Supported Data Types
Analyzers, predined and custom
DSL syntax for queries
Full text and relevance score
Advanced search
Aggregations or faceting
Geo queries

Introduction

Aggregations are extremely powerful for reporting and dashboards

Using the Elastic, Logstash, and Kibana stack demonstrates just how much you can do with aggregations.

Kibana dashboard

DSL Syntax

An aggregation can be seen as a unit of work that builds analytical information on a set of documents.

Depending on its position in the DSL tree, it applies to all search results or to subsets

In DSL syntax, an aggregation block uses the keyword ***aggs***

```
// The max of the price field in all documents
POST /sales/_search?size=0
{
    "aggs" : {
        "max_price" : { "max" : { "field" : "price" } }
    }
}
```

Types of aggregations

Several concepts relate to aggregations:

- **Groups or Buckets** : Set of documents that have a field with the same value or share the same criteria.
  Groups can be nested. ELS provides syntaxes for defining groups and counting the number of documents in each category
- **Metrics**: Metric calculations on a group of documents (min, max, avg, ..)
- **Pipeline** : Aggregations that take input from other aggregations instead of documents or fields.

Example Bucket

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

Response

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

Example metric

```
GET /cars/transactions/_search
{
  "size": 0,
  "aggs" : {
    "avg_price" : {
            "avg" : {"field" : "price"}
        }
  }
}
```

Response

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

Bucket/Metrics juxtaposition

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

Response

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

Nesting aggregations

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

Response

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

Aggregation and search

In general, an aggregation is combined with a search. The buckets are then deduced from the documents that match.

```
GET /cars/transactions/_search
{
"query" : {
  "match" : { "make" : "ford" }
}, "aggs" : {
   "colors" : {
      "terms" : { "field" : "color" }
} } }
```

Sorting buckets

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

Pipeline

Pipeline aggregations work on the outputs produced from other aggregations

They reference the aggregations used to perform their computation by using the ***buckets_path*** parameter

Example

```
POST /_search
{
  "aggs": {
    "my_date_histo": {
      "date_histogram": {
        "field": "timestamp",
        "calendar_interval": "day"
      },
      "aggs": {
        "the_sum": {
          "sum": { "field": "lemmings" }
        },
        "the_deriv": {
          "derivative": { "buckets_path": "the_sum" }
        }
      } } } }
```

Types of buckets

ELS offers different ways of grouping data :

- By term: Requires field tokenization
- Histogram : By range of values
- Date Histogram : By date range
- By IP range
- By absence/presence of a field
- Significant terms
- By geo-location
- ..

Histogram

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

Date histogram

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

significant_terms

The ***significant_terms*** aggregation is more subtle but can give interesting results (anomaly detection).

This consists of analyzing the returned data and finding the terms that appear at an abnormally higher frequency

Abnormally means: relative to the frequency for all documents

=> These statistical anomalies generally reveal interesting things

Mechanism

significant_terms takes a search result and performs an aggregation

It then starts from the set of documents and performs the same aggregation

It then compares the results of the first search that are "abnormal" against the overall search

With this type of operation, you can:

- People who liked... also liked...
- Customers who had questionable credit card transactions all went to such and such a merchant
- Every Thursday evening, the page is much more consulted

Example

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

Response

```
"aggregations" : {
        "significantCrimeTypes" : {
            "doc_count": 47347, // Total query result
            "buckets" : [
                {
                    "key": "Bicycle theft",
                    "doc_count": 3640, // Number docs for the query result
                    "score": 0.371235374214817,
                    "bg_count": 66799 // Number for all documents
       }
                ...
            ]
        }
```

=> Bike theft rate unusually high for « British Transport Police »

Available metrics

ELS offers many metrics:

- ***avg,min, max, sum***
- ***value_count, cardinality*** : Distinct value count
- ***top_hit*** : The top documents
- ***extended_stats*** : Statistical metrics (count, sum, variance, …)
- ***percentiles*** : percentiles

Search Engine Capabilities

Inverted index and distributed search
Supported Data Types
Analyzers, predined and custom
DSL syntax for queries
Full text and relevance score
Advanced search
Aggregations or faceting
Geo queries

Introduction

ELS allows you to combine geo-location with full-text, structured searches and aggregations

ELS has 2 data types to represent geolocation data

- ***geo_point*** : represents a latitude-longitude couple. This mainly allows the calculation of distance
- ***geo_shape*** : defines an area. This allows you to know if 2 areas have an intersection

Geo-point

Geo-points cannot be automatically detected by ELS. They must be explicitly specified in the mapping:

```
PUT /attractions
{ "mappings": {
    "properties": {
      "name": { "type": "string" },
      "location": { "type": "geo_point" }
     }
} }
----
PUT /attractions/1
{"name": "Chipotle Mexican Grill", "location": "40.715, -74.011" }
PUT /attractions/2
{ "name": "Pala Pizza", "location": { "lat":40.722,"lon": -73.989 } }
PUT /attractions/3
{ "name": "Mini Munchies Pizza","location": [ -73.983, 40.719 ] }
```

Filters

4 filters can be used to include or exclude documents with respect to their *geo-point*:

- ***geo_bounding_box***: The geo-points included in the provided rectangle
- ***geo_distance***: Distance from a center point below a boundary.
  Sorting and scoring can be relative to distance
- ***geo_distance_range***: Distance in a range
- ***geo_polygon***: Geo-points include in a polygon

Example

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

Aggregation

3 types of aggregation on geo-points are possible

- ***geo_distance*** (bucket): Groups documents in concentric circles around a central point
- ***geohash_grid*** (bucket): Group documents by cells (*geohash_cell*, google maps squares) for display on a map
- ***geo_bounds*** (metrics): returns the coordinates of a rectangle area that would encompass all geo-points. Useful for choosing the right zoom level

Example

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

Geo-shape

Like *geo_point* , ***geo-shape*** fields must be mapped explicitly:

```
PUT /attractions
{
"mappings": { "landmark": {
"properties": {
  "name": { "type": "string" },
   "location": { "type": "geo_shape" }
} } } }
---
PUT /attractions/landmark/dam_square
{
"name" : "Dam Square, Amsterdam",
"location" : {
  "type" : "polygon",
  "coordinates" : [[ [ 4.89218, 52.37356 ], [ 4.89205, 52.37276 ], [ 4.89301, 52.37274 ],
[ 4.89392, 52.37250 ], [ 4.89218, 52.37356 ] ]
} }
```

Query example

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

Data Visualization with Kibana

What is Nearly-Real-time analysis?

Discover, Visualize and Dashboards

Basic visualisation

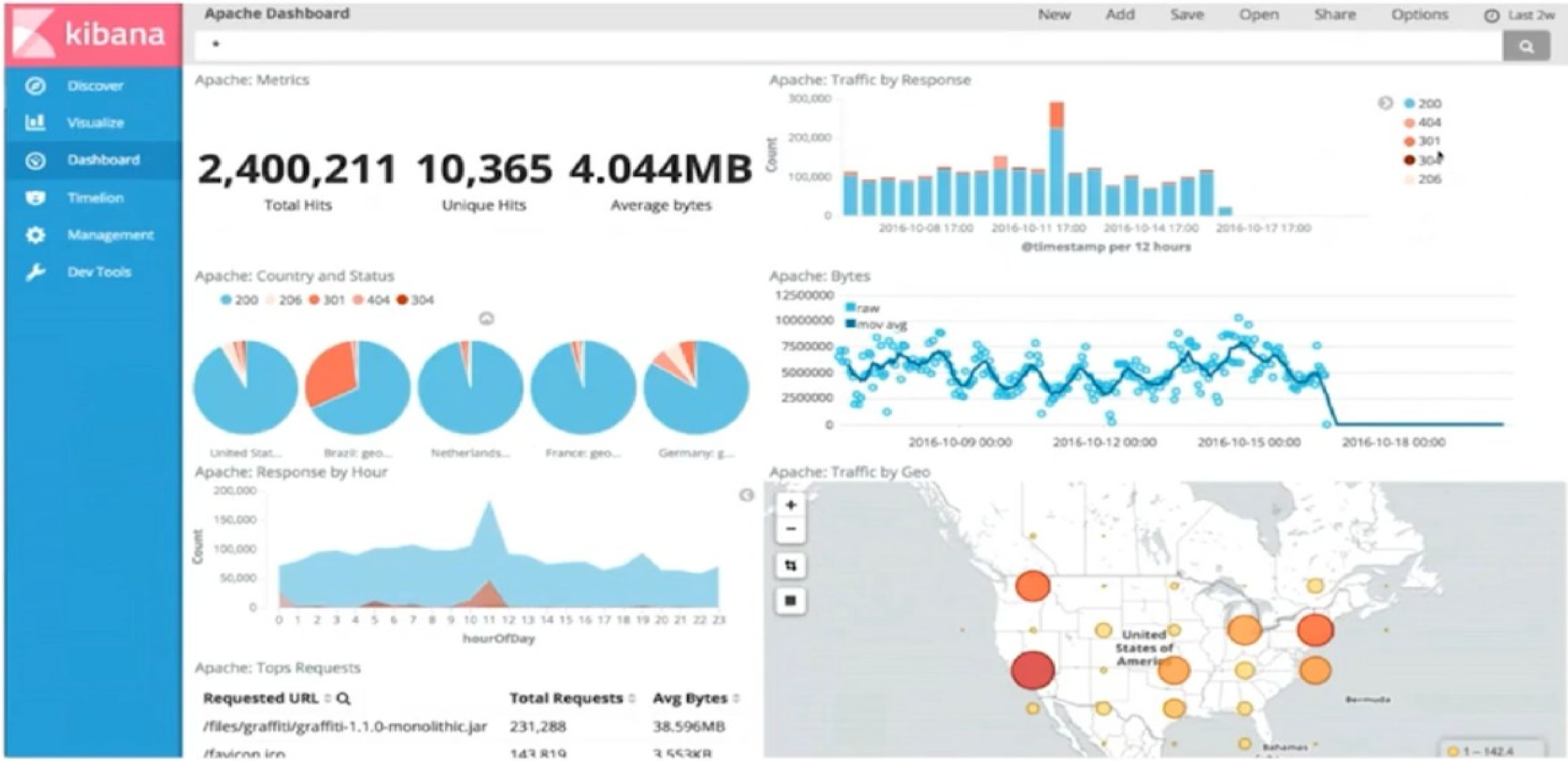
Time series

Machine Learning

Introduction

- Kibana is an analytics and visualization platform powered by ElasticSearch
- It is able to search data stored in ElasticSearch indexes
- It offers a web interface for creating dynamic dashboards displaying the results of queries in real time
- It runs on Node.js

Dashboard Kibana



Final users and use cases

Kibana to destinations of all profiles: Business, Operations, Developer

Once data is ingested into ElasticSearch indexes, it allows:

- Explore the data to better understand it
- Create visualizations and include them in dashboards
- Create presentations to facilitate meetings
- Share via email, web page or PDF documents
- Apply Machine Learning models to detect anomalies or anticipate the future
- Generate graphs visualizing the relationships between your data
- Manage ElasticSearch indexes
- Generate alerts about your data
- Organize Kibana assets into "Space" related to ES security and permissions model

Data Visualization with Kibana

What is Nearly-Real-time analysis?

Discover, Visualize and Dashboards

Basic visualisation

Time series

Location analysis

Data relationships with graph

Machine Learning

Analyze and visualize your data

In the context of the Search Engine features of ELK, Kibana can be used :

- Adjust our ELS queries with the ***Dev Console***
- Understand your Data with the menu ***Discover***
- Analyze your data with ***Visualizations*** and ***Dashboards***

Steps of developing a dashboard

1. Defining an ***Index Pattern/DataView*** and its timestamp field
2. Explore data and optionally save queries
3. Create visualizations, aggregations, times, maps
4. Arrange them on a dashboard
5. Share the Dashboard URL

Index Pattern creation

To exploit data, it is necessary to define ***index pattern***[1], i.e. collection of indexes respecting a naming rule:

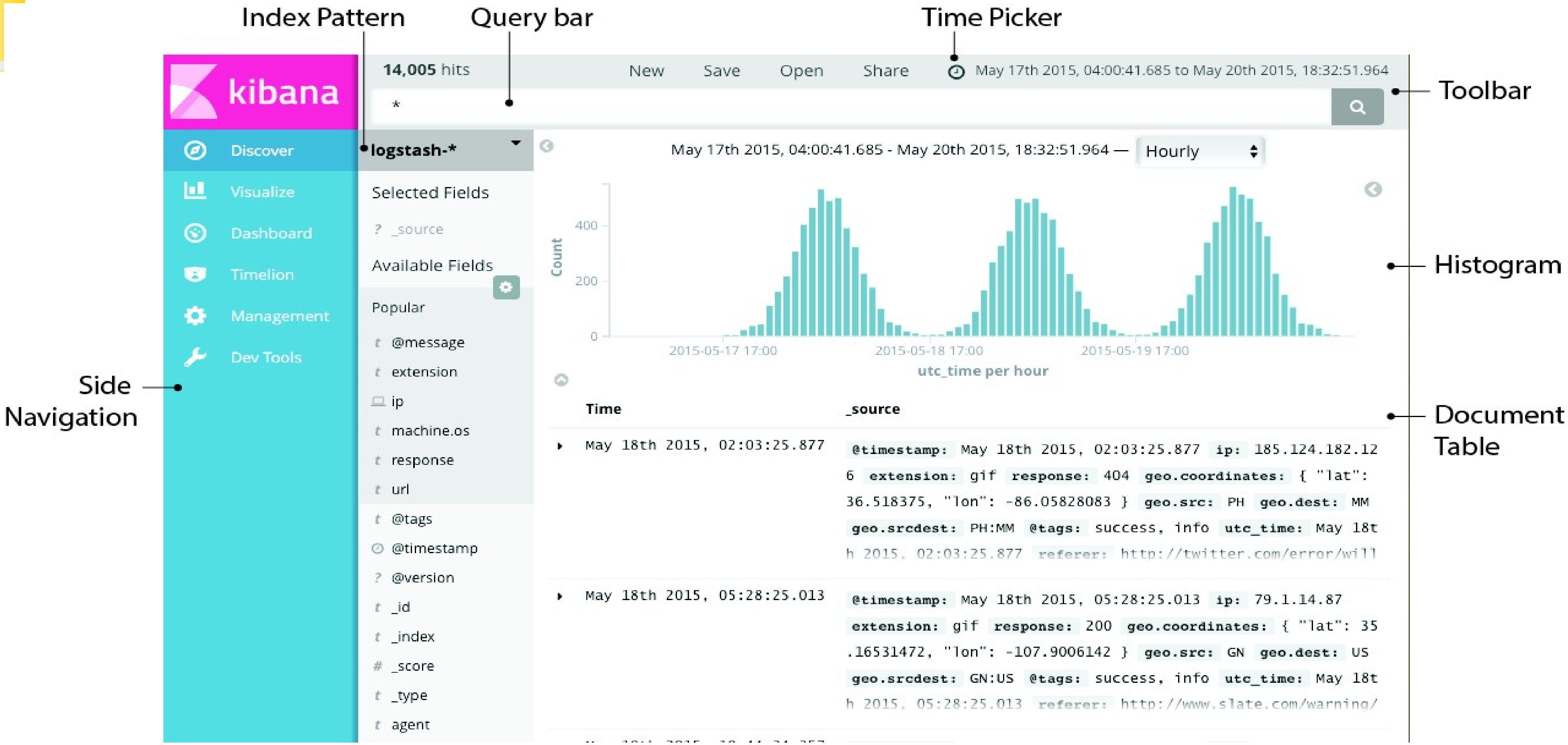
– ex: logstash-*

– You must also indicate the *timestamp* field of these indexes

Then, Kibana loads the mapping information of these indexes and clearly indicates their type, whether they can be used for search or for aggregation.

1. Renamed DataView in ELK 8.x

Discover Menu



Time window

- If a timestamp field exists in the index:
  - The top of the page displays the distribution of documents over a period (by default 15 min)
  - Time window can be changed
- This time window is present in the majority of Kibana pages

Search

Search criteria can be entered in the query bar. It could be:

- A simple text
- A Lucene query
- A DSL request with JSON.

The result updates the whole page and only the first 500 documents are returned in reverse chronological order

- Field filters can be specified
- Search can be saved with a name
- The index pattern can be modified
- Search can automatically refresh every X times

Filters

Several buttons are available on filters:

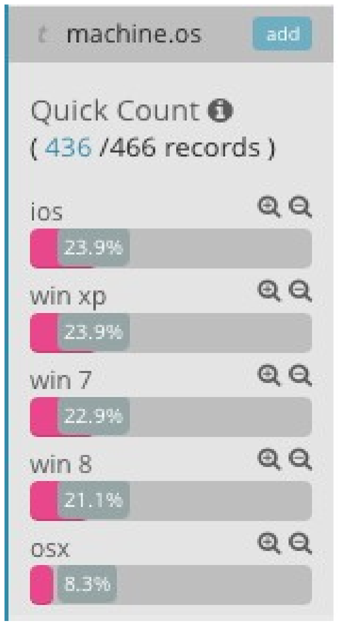
- Activation deactivation
- Pin: Filter is retained even on a context switch
- Toggle: Positive or Negative Filter
- Deletion
- Edit: One can then work with DSL syntax and create filter combinations

Document viewing

- The list displays by default 500 documents (property discover:sampleSize)
- It is possible to choose the sorting criterion
- Add/Remove Fields
- Statistics are also available

Field value statistics

Field value statistics show how many documents have a particular value in a field (facetting)



Data Visualization with Kibana

What is Nearly-Real-time analysis?
Discover, Visualize and Dashboards
Basic visualisation
Time series
Machine Learning

Visualizations

The visualizations are based on the aggregation capabilities of ELS, we can create graphs that show trends, peaks, ...

Visualizations can then be grouped into dashboards

Different editors for visualization are available:

- Lens: Drag and Drop
- Classical Visualizations based on aggregations
- Maps: Geolocation
- TSVB et Timelion: Suitable for time series
- Custom visualizations: Vega

Classical visualization

Area : View total contributions from multiple series

Table de données : Tabular display of aggregated data

Line : Compare different series

Metric: Display a single metric

Gauge/***Goal***: A single value with ranges of good/bad values relative to a target

Pie : Pie .

Heat map : Heat map

Nuage de tags : The most important tags have the largest font

Horizontal / Vertical bar : Histogram allowing the comparison of series

Steps of creation

With the classical editor steps are

1. Choose a chart type
2. Specify search criteria or use a saved search
3. Choose the aggregation calculation for the Y axis (count, average, sum, min, …)
4. Choose the data grouping criterion (bucket expression) (Histogram of date, Interval, Terms, Filters, Significant terms, …)
5. Optionally define sub-aggregations

- Lens allows you to create your visualization more intuitively via Drag & Drop

Bucket expressions

Histogramme Date : Un histogramme de date est construit sur un champ entier pour lequel on a précisé une fenêtre temporel

Intervalle : Entier, date ou IPV4

Terme : Permet de présente les n meilleurs (ou plus basse) pertinence ordonné par la valeur agrégée

Filtres : Il est possible d'ajouter des filtres de données à ce niveau

Significant Terms : Agrégation *significant terms*

Geohash : Agrégation sur les coordonnées géographiques (Data Table et carte)

Options

Lors de plusieurs agrégations en Y, il est possible de spécifier leur mode d'affichage

- ***Stacked*** : Empile les agrégations les unes sur les autres.
- ***Overlap*** : Superposition avec transparence
- ***Wiggle*** : Ombrage
- ***Percentage*** : Chaque agrégation comme une portion du total
- ***Silhouette*** : Chaque agrégation comme variance d'une ligne centrale
- ***Grouped*** : Groupe les résultats horizontalement par les sous-agrégations (Grapique barre) .

Dashboard

- A dashboard groups a set of saved visualizations into a web page
- It is possible to rearrange and resize visualizations
- It is possible to share the dashboards by a simple link

Data Visualization with Kibana

What is Nearly-Real-time analysis?
Discover, Visualize and Dashboards
Basic visualisation
Time series
Machine Learning

Introduction

Timelion is a visualizer dedicated to temporal data that allows to combine completely independent data sources in the same graph.

It is driven by a simple expression language to retrieve data and perform calculations.

With timelion, it is possible to answer this type of question:

What percentage of the Japanese population has visited my site?

Éléments de syntaxe

The timelion interface consists of:

- An expression input box
- A viewing area

Each timelion expression starts with a function identifying a data source. For ElasticSearch:

es(*)

2 data sources can be drawn side by side using comma (,);.

- *.es(*),.es(metric=cardinality:user)*

They can be combined via functions:

- *.es(*).divide(es(metric=cardinality:user))*

Other data sources can be used (Worldbank's Data API for example)

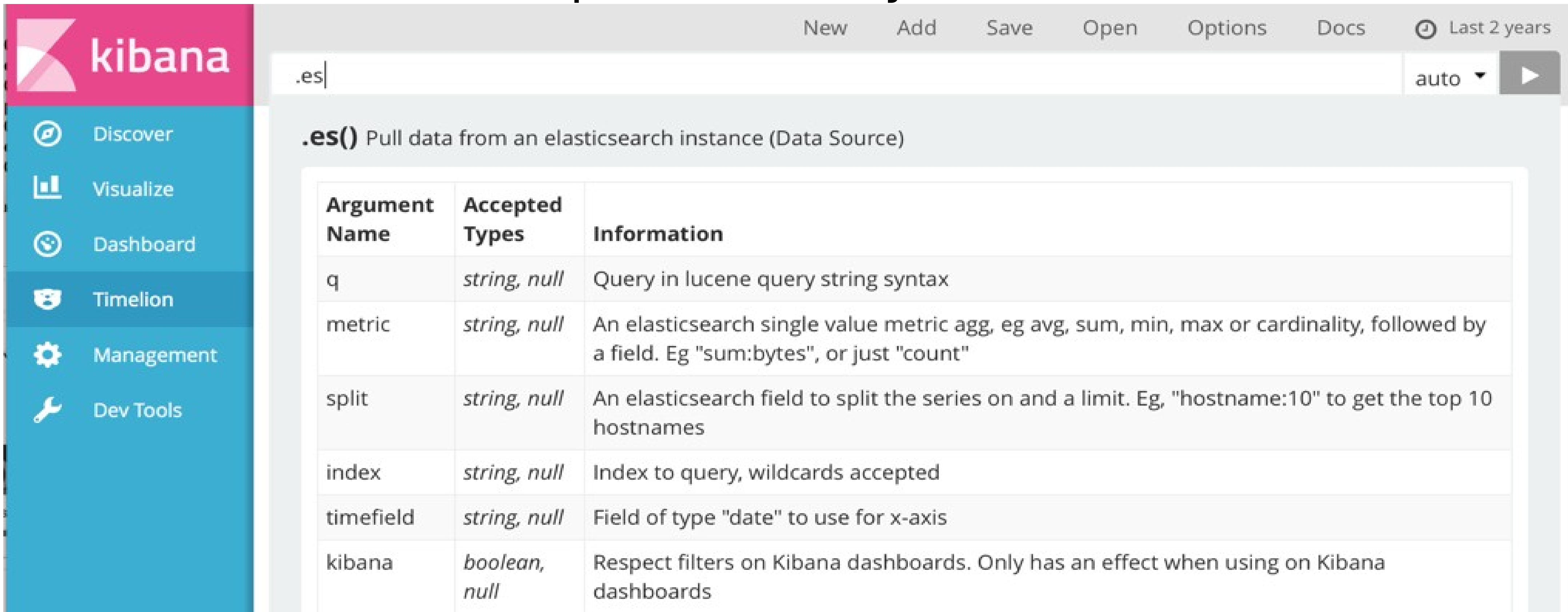
.wbi(MZ) // Population of Mozambique

Online help and documentation

Documentation is part of timelion

It is available through the Docs link

In addition, the completion is very efficient



Examples

```
# Real-time display of the average of the
# user CPU usage percentage from
# of an elastic search metricbeats index
.es(index=metricbeat-*, timefield='@timestamp',
metric='avg:system.cpu.user.pct')

# The same thing by adding a series displaying
# the data for an offset of 1h
.es(index=metricbeat-*, timefield='@timestamp',
metric='avg:system.cpu.user.pct'), .es(offset=-
1h,index=metricbeat-*, timefield='@timestamp',
metric='avg:system.cpu.user.pct')
```

Data Visualization with Kibana

What is Nearly-Real-time analysis?
Discover, Visualize and Dashboards
Basic visualisation
Time series
Location analysis
Machine Learning

Introduction

Elastic's ML offering, although applicable to many areas, focuses on IT monitoring:

- Monitoring: Detection of failure, malfunction
- Security: Intrusion, exfiltration detection

In IT monitoring:

We generate a lot of data but we don't have time to look at it and analyze it

=> Automation of these analyzes by systems that can learn on their own

Anomalies

The majority of monitoring requirements are variations on the theme:

Find something that is different from the ordinary

The analysis processes then attempt to detect anomalies in relation to the observation of the history

=> This requires the history to be large enough so that the ML can refine its model

Objectives of a ML solution

Speed: Notification of a failure, a breach or any other significant anomaly proactively and as quickly as possible in order to mitigate it.

Scalability: Algorithms must be able to scale linearly with data.

Performance: Using modest hardware rather than supercomputers

Applicability: Consideration of the diversity of data in computing environments.

Adaptability: Constantly changing computing environments can quickly make a static algorithm fragile.

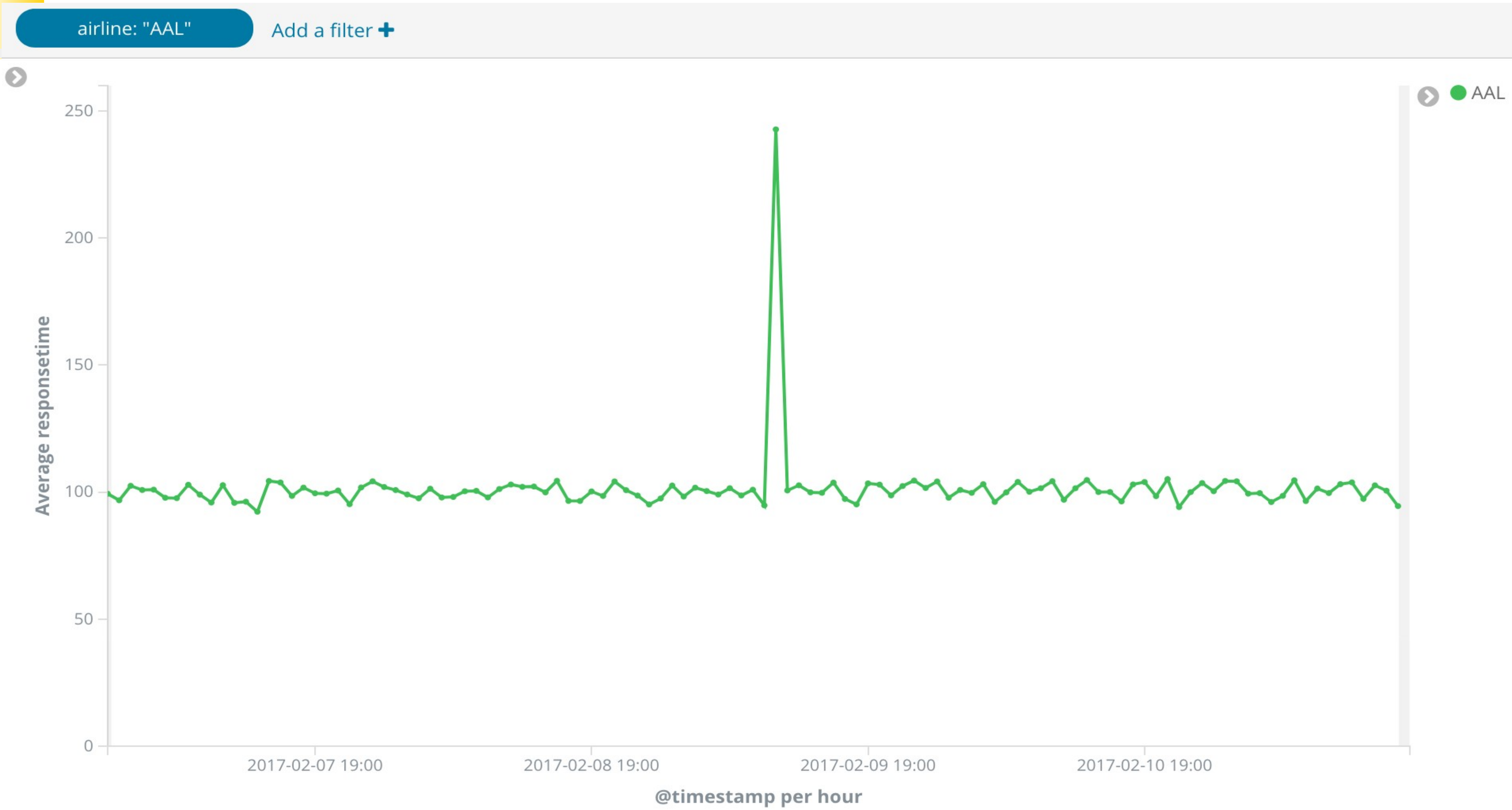
Accuracy: Do not generate false alarms

What is an anomaly ?

2 definitions of an anomaly:

- Something is unusual if its behavior has deviated significantly from an established pattern based on its past history
- Something is unusual if some of its characteristics are significantly different from the same characteristics of other members of a set or population

Ex : Past history



Ex : Population

Unsupervised learning

- In ML, unsupervised learning requires no human intervention for setting up and refining the model.
  - => The model is built, refined, modified according to the data presented to it
  - => The more data there is, the more accurate the model becomes
- Elastic Stack offers 2 types of unsupervised analysis
    - Anomaly detection: Based on temporal data
    - Abnormal value detection: Based on value densities

Supervised learning

Supervised learning requires training data.

ELK-ML offers supervised analysis types:

- **Classification**: Event Classification
  Ex: Normal or malicious requests
- **Regression**: Predicting Numerical Values
  Ex: response for a web request.

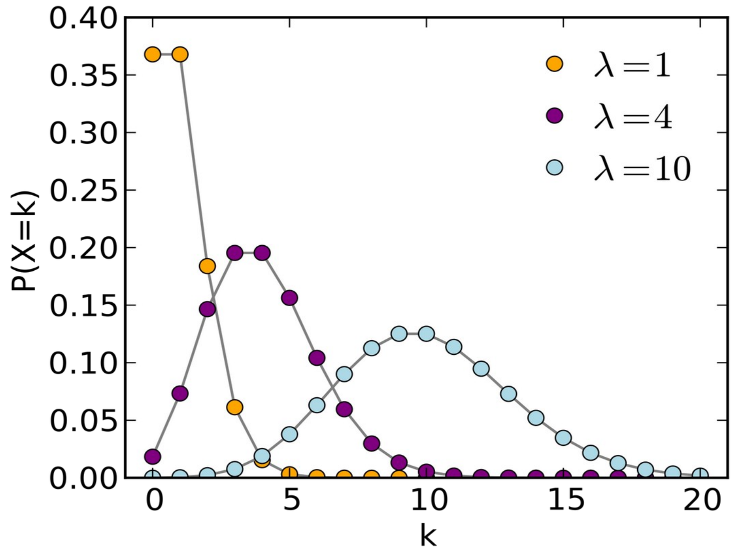
Ex : Poisson Distribution

Statistical models are used to model the behavior of a system.

For example, the Poisson distribution makes it possible to model events such as

- The number of meteorites over a meter in diameter that hit Earth each year
- The number of patients arriving at an emergency room between 10:00 a.m. and 11:00 p.m.
- The number of photons hitting a detector in a particular time interval

$$p(k) = \mathbb{P}(X = k) = \frac{\lambda^k}{k!}e^{-\lambda}$$



If a certain type of event occurs on average 4 times per minute,
to study the number of events occurring in a 10-minute time frame,
we choose as a model a Poisson distribution with parameter λ = 10×4 = 40

Modelisation process

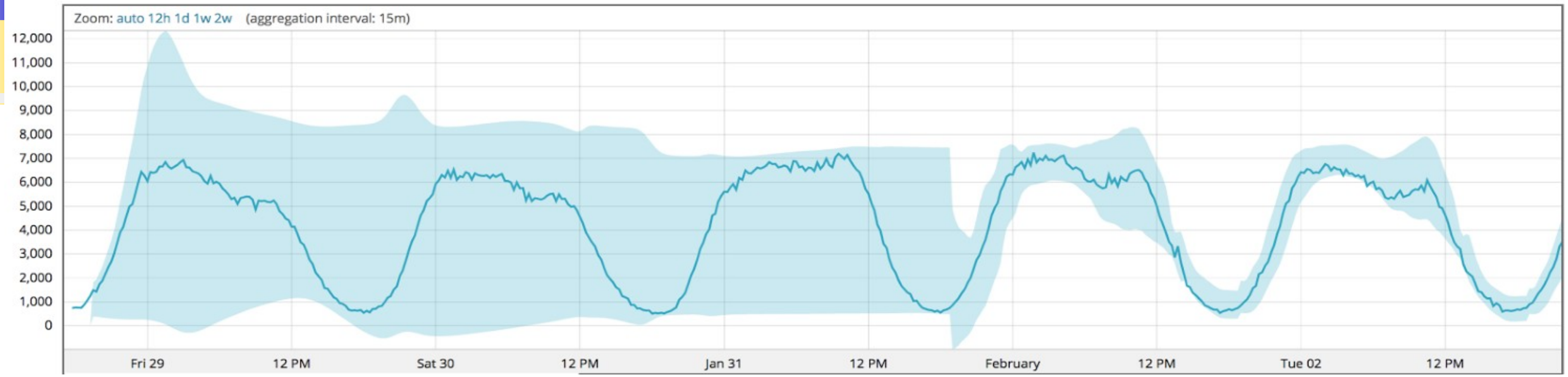
- The machine learning process selects from its observations the appropriate probability model (Poisson, Gaussian, log-normal, etc.) and its coefficients
- Bayesian techniques are used to assess the probabilities of model values given the observed data set and allow the results to be tempered based on the amount of information visualized.
- The modeling performed is continuous, so new information is considered along with old, with an exponential weighting of fresher information
- Modeling can be stopped and then restarted later, hence the need to persist the model.

Cycles

Another important aspect of modeling real data is to take into account the important harmonic trends that occur naturally.

ML automatically finds salient trends in the data (linear growth, cyclic harmonics, etc.) and factors them

Example



The periodic daily cycle is learned and then factored.

The model prediction adjusts after the automatic detection of three successive iterations of this cycle.

Anomaly Score

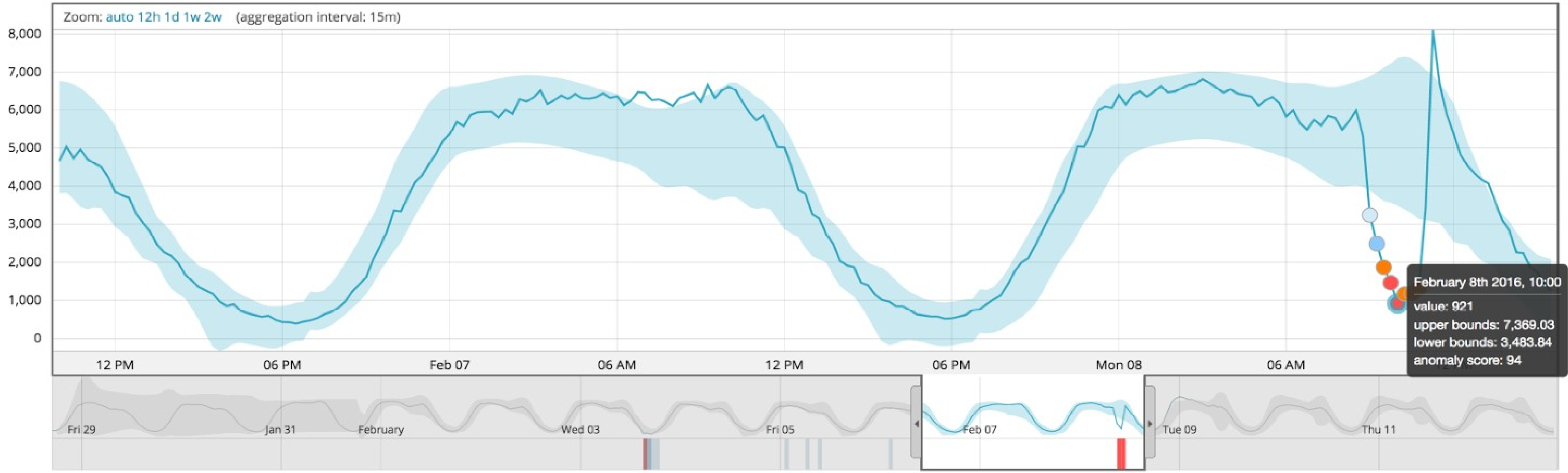
- Probability models are used to calculate the probability rate that a measurement takes on a certain value.
- Probability rates oscillating between 0 and 1 are normalized to **anomaly scores** that oscillate between 0 and 100
- Severity levels, alerts are associated with anomaly score thresholds

Example

The probability of the value 921 was 6.3634e-7

This gives an anomaly score of 94

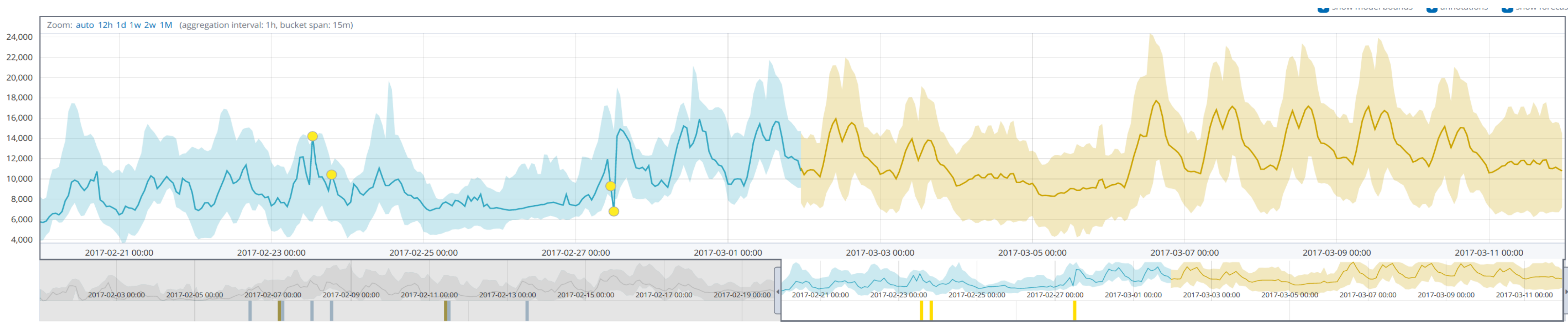
Which gives a critical anomaly (> 75)



Forecasts

- Once ML has created its behavioral model, it uses it to extrapolate future behavior:
  - Estimate a value at a future date
  - Probability that a value reaches a threshold
- Each forecast has a unique id, duration, expiry time
- Forecasts cannot be made on all types of values
- They are limited to 3 for the same job
- They require a lot of memory

Example



Zoom: auto 12h 1d 1w 2w 1M (aggregation interval: 1h, bucket span: 15m)
24,000
22,000
20,000
18,000
16,000
14,000
12,000
10,000
8,000
6,000
4,000
2017-02-21 00:00
2017-02-23 00:00
2017-02-25 00:00
2017-02-27 00:00
2017-03-01 00:00
2017-03-03 00:00
2017-03-05 00:00
2017-03-07 00:00
2017-03-09 00:00
2017-03-11 00:00
2017-02-03 00:00
2017-02-05 00:00
2017-02-07 00:00
2017-02-09 00:00
2017-02-11 00:00
2017-02-13 00:00
2017-02-15 00:00
2017-02-17 00:00
2017-02-19 00:00