



January, 2019

Optimization of the search engine Elasticsearch

Quentin Coviaux

Advisor: Marta Arias (UPC –
Computer Science department)

Supervisor: Georges Soto-Romero
(Ecole d'ingénieurs d'Informatique
et Système d'Information en Santé)

MASTER IN INNOVATION AND
RESEARCH IN INFORMATICS

Data Science specialty

FACULTAT D'INFORMÀTICA DE
BARCELONA (FIB)

UNIVERSITAT POLITÈCNICA DE
CATALUNYA (UPC) - BarcelonaTech

The master thesis has been developed at Orange - France

Abstract

The matter of information retrieval is one that has gained importance ever since the beginning of the digitalization era. With volume of data getting bigger and bigger, it has become essential to be able to retrieve information efficiently. As technology keeps improving, users tend to forget information itself but remember the path to retrieve it.

To retrieve information in such big volume of data, we cannot use traditional key-value pairs matching. Search engines are required to work with unstructured or semi-structured data, such as text or images. Finding useful information is much harder in high volume of data since information can get lost in all the chaos and pollution of the unstructuredness. To sort useful information from the rest, we will see that statistics can help find tendencies in data and sort out common and meaningless information from rare and meaningful data.

This thesis will be divided in two main parts, the moving of large amount of data and the use of these data. In the first part, we will see how to gather data coming from different sources and how to transform it to suit the need of everyone through the use of an ETL tool. We will come to understand that one of the main aspects of having a good system of information retrieval is the preprocessing of the data.

In the second part, we will talk about the fine tuning of relevance that is a major factor for users. Indeed, while it is one thing to be able to retrieve information in large volume of data, users can not be expected to go through every bit of data returned to find the one piece of information they are seeking. That is why the ordering of documents is decisive when talking about the performance of a search engine. To complicate even further the matter, the relevance that is already hard to tune for a given user is also subjective for that user. Two users may not expect the same kind of results for the same query. We will present examples of this in this thesis. All the tuning that we will see will be about the search engine Elasticsearch but the principles are the same for other search engines.

Most of the work of this thesis has been done in the course of the work at Orange and the presented use cases come from use cases encountered during this period.

Acknowledgments

I would like to express my gratitude to all the teams at Orange that welcomed me in the company and that gave me a chance to express myself and my ideas. I would like to thank particularly the SOD and Polaris teams that I joined during this thesis. I am grateful to Karim Naamani for taking me in. I am looking forward to keep working with the different teams at Orange

I would like to address my personal appreciation for all the help and time of Gaël Braconier and Arnaud Lanaspeze, and also their patience during my never-ending questions. I wish to thank Maxime Normand and Sébastien Negele for their help upon my arrival in the company, for taking the time to explain to me the project.

In Alten, I would like to thank Carole Brusson and Jonathan Quidet for their trust and their availability during the conduct of the thesis.

I would like to express my gratitude to Marta Arias Vicente for her help and guidance provided during the thesis. I would like to thank Georges Soto-Romero for the support provided and help during the seeking of the thesis. I would also like to thank both administrative and teaching teams of the Polytechnic University of Catalonia and ISIS engineering school. A special thanks to Bernard Rigaud, former director of ISIS for making the double degree possible.

Index

| | |
|---|----|
| Abstract | 0 |
| Acknowledgments | 0 |
| Chapter 1: Introduction | 2 |
| 1.1 Context | 2 |
| 1.2 Thesis objectives | 3 |
| Chapter 2: Background | 4 |
| 2.1 Clients | 4 |
| 2.1.1 Polaris Video | 4 |
| 2.1.2 OneReco | 5 |
| 2.1.3 Webvideofinder | 6 |
| 2.1.4 Adserver | 6 |
| 2.2 Architecture | 7 |
| Chapter 3: Developments | 13 |
| 3.1 ETL | 13 |
| 3.1.1 Extract | 13 |
| 3.1.2 Transform | 14 |
| 3.1.3 Load | 16 |
| 3.1.4 Results | 20 |
| 3.2 Querying | 21 |
| 3.2.1 Elasticsearch's basic score | 21 |
| 3.2.2 Webvideofinder's relevance | 24 |
| 3.2.3 Study of relevance | 35 |
| 3.2.4 Proxy-query | 38 |
| 3.3 Data analysis | 40 |
| Chapter 4: Further research | 45 |
| 4.1 Elasticsearch use cases | 45 |
| 4.2 Web search systems | 45 |
| 4.3 Elasticsearch and Solr | 46 |
| Conclusion | 49 |
| Bibliography | 50 |

Chapter 1: Introduction

1.1 Context

Orange is a French company and one of the leaders in telecommunication in Europe and in the world, delivering products from internet to mobile to the TV. Present in as many as 28 countries around the world, they reported having over 260 million customers in 2018¹. With around 150.000 employees worldwide, they had sales for 41€ billion revenue in 2017.

The information system it possesses goes from presenting the news on the front-page of the website to managing the content on the TV. Among all those services, it is essential for lots of them to have a search engine to help the users to find what they are looking for easily. Indeed, if the search system is inefficient, whether in time it takes to provide answers or simply in poor relevance, users will not bother and will go look somewhere. In this context, with its many needs for search varying from looking for movie on the TV, or for a video on the website, or for information about a product in a store, the need to be able to have an efficient querying system is paramount.

Behind all those services, a few things come into play, the need to gather data and the ability to retrieve it in an orderly fashion. Some data come from Orange of course, such as the products they propose, but others come from different partners of the company, for instance the TV program that comes from the channels directly. In any case, the data first need to be collected, organized and loaded onto a database prior to making it accessible. The gathering is done by an Extract-Transform-Load (ETL) tool and after this there needs to be an easy access for the different clients. For the need of the project, we can suppose that there were mainly two options as a database, either a relational one or a non-relational one. The volume of data and queries depend on the clients of course, going from a wild range to a few thousands documents for some to a few millions. While there was not hundreds of millions of documents from any service, the volume isn't enough to make a decision on which option would be the most viable. In addition to this, the number of queries for a big service, the broadcast media service, which provides movies, broadcasts, TV show and such, has the following volume: around 17 queries per minute on the smartphone application and 120 queries per minute on TV. Finally, and perhaps, the most important factor here is the fact that a traditional database, while it could hold this volume, is not really appropriate for the need of textual search. With documents moving rapidly and changing types often, a non-relational database could be an option, and in this field, MongoDB is the reference but remains the problem of the full-text search capabilities. A search engine was in order and so Elasticsearch, with its non-relational database, was used. Elasticsearch is an open-source search engine developed by the Elastic company based on the Lucene library from the Apache Software Foundation. It is considered as the most popular enterprise search engine². To better understand what enterprise search engine are, we need to look into Information Retrieval.

To understand Information Retrieval, we need to get a good definition for it. According to the book "*Introduction to Information Retrieval*" [1], we get the following definition:

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

While this is quite broad, we can already see some ideas in it. The unstructured nature comes from the fact that documents are not just a succession of key-value pairs anymore, they can contain very long fields of data that may require a deep analysis to retrieve. Large collections

¹ <https://orange.jobs/site/get-to-know-us-better/Get-to-know-us.htm>

² <https://db-engines.com/en/ranking/search+engine>

further insist on the fact that it isn't just about retrieving the documents. IR is also the need to order those documents. When using a search engine, the number of documents returned is not the only concern because a user will never go through millions of documents matching the search. This is why a very important factor is the ordering of said documents by whatever relevance is most appropriate.

IR systems can be separated in 3 categories, according to their scale. On the lowest end, there is the *personal information retrieval*, which can be found in most personal Operating System nowadays, they include searches on a computer or on an Email account. Overall, the collections of documents are small. Above this category, there is the *enterprise search*, which includes larger collections, with searches on articles or books for instance. Data is usually stored in a centralized system. Finally at the bigger scale we find the *web search*, which crawls the web, indexing billions of documents.

In this thesis, the use case corresponds to the enterprise search given the volume of documents, the types of documents and the query volume.

1.2 Thesis objectives

In this context, we will follow the work done within the Search On Demand (SOD) team and the optimization done for the search engine Elasticsearch.

Elasticsearch, unlike traditional database, is a full search engine enabling full-text search. This kind of technology is necessary when one doesn't need to retrieve documents with key-value pairs but need to retrieve information within unstructured data. These data are opposite to classical information systems used to maintain inventories or other such structured data. Unstructured data comes from the fact that data can't really be divided and stored following the relational databases, and thus they need a deeper analysis to understand them. Such data can be images or videos, which require an analysis to understand, classify and efficiently retrieve them. Perhaps the most obvious example of unstructured data is text, books for instance, because they contain collections of words that cannot be easily sorted. Retrieving information in those can be difficult and consume lots of resources. While in this document we will talk mostly about semi-structured data, the need for analysis is the same. Truly unstructured data for text isn't really common, most texts will follow the structure of a language or have a semblance of structure, with headers and other such additional information.

As part of the optimization, 2 aspects come into play. The first is the improvement of the data processing time. Prior to loading data in the search engine, we need to gather data, process it and this step can take a long time, depending on the kind of data we are trying to access or the processing we execute. We will see ways to improve this time and also how important is the mapping in Elasticsearch.

The second optimization will be on the relevance of Elasticsearch. Relevance, as we will see, is a difficult matter that is specific to each individual. The tuning of the many parameters is a long work and we will see how everything works together.

Finally, a part of the thesis could be seen as consultancy for clients. Indeed, while Elasticsearch is a more and more popular solution for searching in data, it is easy for anyone to just decide to use the hot new technologies. We will come to see that, even if Elasticsearch is a powerful tool, it is not appropriate for everyone.

The structure of the thesis will go as follows. First we will look at the background of the project, some clients and their needs, followed by the architecture of the project with an introduction how Elasticsearch works. In the second part, we will dive deeper into a more technical approach where we will see the way to supply data in the search engine with the means of an ETL tool and the ways to query Elasticsearch. Finally, we will take a step back and study different aspects of a search engine and its ecosystem.

Chapter 2: Background

In this chapter, we will present the state of the art of the project, the different clients that rely on the search engine and how they use it. Then, we will present the architecture of the project, with explaining how Elasticsearch works. Finally, we will look at how we gather data from several sources and the transformation ensuing prior to loading it in the search engine.

2.1 Clients

As stated before, Orange has many services that need a search engine. The team at the center of this architecture is called Search On Demand, abbreviated SOD, and it manages the search engine for over 20 services, that we will refer as clients. The documents that are stored and that needs to be retrieved don't have much in common from a client to another. They go from TV metadata content to ads information. Overall, before getting in the specification of clients, a precision is needed. The documents that we will be talking about are in the JSON format, which derives from the programming language Javascript. While we will not be talking about Javascript, JSON objects are constituted of an ensemble of key-value pairs. As we can guess from this, keys are unique within an object, and they can hold a simple value such as a string of characters, an integer or a Boolean, or they can hold more complicated objects and arrays. Right below, we can find an example of a simple JSON file.

```
{  
  "name": "Bob",  
  "age": 42,  
  "married": true,  
  "pet": {  
    "name": "Clafoutis",  
    "species": "dog"  
  }  
}
```

Figure 1, object JSON example

This short JSON object illustrates the construction of such objects. The keys are on the left side of each line. To each key is associated a value, this way we understand that for the key "name", the value is "Bob". The key "pet" is not associated to a single value but to another object. And now that we understand how this format works, we can get into the presentation of the different clients.

2.1.1 Polaris Video

The first client we are going to look at is Polaris Video. This is the client that manages broadcast media content on the TV and on mobile. The documents they need to retrieve is composed of many fields, mostly about metadata. There exist 4 types of documents that they use: movies, broadcasts, TV shows episodes and seasons. The distinction between the 4 types is quite important because even though most of their fields are the same, there are some exceptions. For instance, the season number or the number of associated episodes are clearly for the season type. Furthermore the need to separate the 4 types is rooted from the final users. Indeed, a user for instance might be looking for a movie to watch on TV and if this is the case, we wouldn't want to suggest broadcast to this user. Having these separate types enable faster filtering when the query arrives in Elasticsearch. We will learn later on how Elasticsearch stores

data but for now the only thing we need to know is that the structure on which Polaris Video documents are stored is called Broadcastmedia2

The main object of the documents contains basic metadata information on said documents, such as the actors that perform, the release date, the genre and other information of the same type. These information however do not provide the ways to consult the content on TV. This is why we add what we call events to the main document. Events can be of 2 types, either on live broadcast on the TV or on On Demand services.

If we take the same short example as above, in this case events would take the place of the object “pet”, except that it would not be a single object, but an array of objects. In addition to this, they are not simple objects but nested objects. Without the specification, objects would not have an atomic value. Let’s explain this with an example, consider the following object being a movie with its title and actors. Suppose we run a query where we would want to match a movie with an actor that has the first name “John” and the last name “Watson”.

```
{  
    "title": "My movie",  
    "actors": [  
        {  
            "firstname": "John",  
            "lastname": "Cena"  
        },  
        {  
            "firstname": "Georges",  
            "lastname": "Watson"  
        }  
    ]  
}
```

Figure 2, JSON array object

If the objects weren’t nested here, we would get a match, which is not the expected behavior. The problem here is that it doesn’t consider the actors as separate objects. The document would match because both requirements would be fulfilled, but on different objects. If the objects were nested, the query wouldn’t have a match because the objects would be considered properly different, it would search for a “John Watson” in each object.

A search from a user of this client can be on the main object to know more about the meta-information or if it is present in the catalog, or on the associated events to know where to watch the content. We need to ensure that in both cases, the user gets what he wants so the search has to be efficient for both use cases.

Prior to sending the query to the search engine Elasticsearch, Polaris Video carries out a semantic analysis. Without getting into the details, this analysis helps translate the search from a user to a specific type of documents. Indeed, because the TV proposes a single search, before sending it to the search engine, it is important to understand the intention behind the search. For instance, if a user types “Thriller” in the search bar, is he looking for a movie with “Thriller” in the name or is he looking for a movie in the “Thriller” genre. While it is possible to run the search on both fields, the ordering would be quite hard to predict, in which case should we give more importance to the genre or to the name. This semantic analysis helps by trying to find the most likely case.

Finally, Polaris Video uses Elasticsearch in 2 ways, to filter documents with key-value pairs and to search text fields.

2.1.2 OneReco

The second client is OneReco. They have mostly the same needs as Polaris Video. When the service was first created, being that they used the same content as present on

Broadcastmedia2, they plugged themselves on it. They provide a recommendation system for the users on the TV, proposing TV shows, movies, or anything than seems in concordance with each person's usage. It uses many parameters to provide to most adapted content but we won't go any further as to how this system works, we will stay on the data from SOD they need and how they get it.

As this client kept evolving, their need went apart from the one of Polaris Video. Many fields were added in Broadcastmedia2 that only one or the other client was using and in the end, the documents were getting bigger and bigger while only a small portion was used by each. The bigger documents meant a higher time to analyze them and to retrieve them. As time went on, we decided it was in the best interest of both to separate the structure, leaving Broadcastmedia2 entirely to Polaris Video and OneReco would get its own data structure, rightly named OneReco.

In addition to the fields being more and more different, OneReco had higher expectation concerning the cleaning of data. Indeed, as we will see later on, OneReco doesn't need to retrieve documents without events, they only need to present documents where a user can get to the broadcasting. Polaris Video on the other hand doesn't have as a requirement to have events on the documents since users may only to see if a movie is currently in the theaters. While it would have been possible to filter for every query the documents without events, it would have taken an unnecessary time since it is possible to make this filtering prior to the query, during the filling of the database.

OneReco uses Elasticsearch without the need to use analysis, simply filtering documents with key-value pairs.

2.1.3 Webvideofinder

The third client we will be talking about is Webvideofinder, the service for finding videos on Orange's website. Using the eponymous structure, it contains millions of documents concerning videos coming from different partners, such as Dailymotion. The search on these videos is essentially textual search and this is where Elasticsearch gets the most useful. Being a search engine and not a traditional database, one of its advantages is to analyze fields of unstructured or semi-structured data. Those analysis enable in-depth search on those fields and this is something quite used by this client since most of the fields are textual.

Webvideofinder uses Elasticsearch's deep analysis capabilities, the search is done essentially on text fields, with the exception of trying to find documents with a recent date. But more on that later.

2.1.4 Adserver

Finally the last client that we will only briefly mention is Adserver. This client's data indexes advertisements that will be displayed on their products. The structure of these documents is quite simple, containing links, date and small texts, but most importantly geographical coordinates. While they don't use any analysis on their textual fields since they do not use textual search, they use another feature from Elasticsearch, the possibility to aggregate per location and to provide documents that are within a certain range quite easily.

Overall, the global architecture of the project looks as follows.

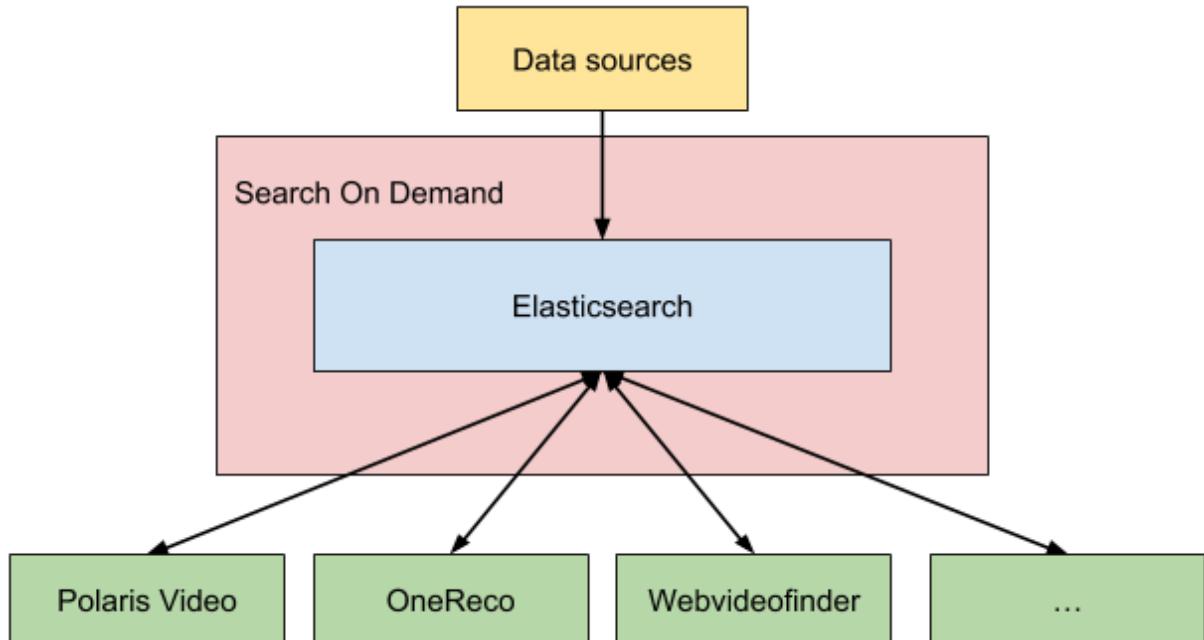


Figure 3, Search On Demand architecture and clients

Here, everything in the red rectangle is managed by the SOD team, including the arrows. The gathering of the data is done by a hand-coded ETL tool and then put in the Elasticsearch cluster. After this, the team makes the queries available for each client.

2.2 Architecture

To understand the architecture of the project, we must talk about how Elasticsearch works. The most fundamental concepts of the search engine are how it orders and stores the data. This is the hierarchy:

- Document : in JSON, the unity in the data we store
- Type : the type of documents we want for a given index
- Index : the equivalent of a database, can have one type
- Cluster : Can be constituted of several index

Documents are the object of the search, they are constituted of many fields, are stored in an index, and have a unique identifier. If we take the case of the client Polaris Video, a document can be a movie or a broadcast, and is composed of several fields of different datatypes. In the same example, the types are whether the documents are movies or broadcasts. Finally the index is the whole collection of the different types. These indices are the structures that hold data from the different clients.

The type of the index allows setting a specific mapping for the data that will be loaded in it. The mapping is describing the datatype of every field that we have in our documents. We can either set it ourselves or let Elasticsearch do it. It is usually better to map the type ourselves because in addition to the datatype, it is possible to specify the kind of analysis we want to provide on a given field, which will be very useful for the queries. It is to note that the default mapping done by Elasticsearch cannot guess what kind of treatment we want to do on the fields and this could affect the search. Furthermore, the mapping can be set in dynamic or static. A dynamic mapping means that if we add documents with fields that are not expected, Elasticsearch will update the mapping automatically. If we want to reject documents with a

different mapping than we predicted, the mapping has to be set strict. Presently, Elasticsearch is in its 6.5.4 version, and many things have changed from the beginning. In this thesis, the clients are on Elasticsearch 2.4 version. While in the first few versions, it was possible to have multiple types for one index, which is the case for Polaris Video, this is no longer possible in the newer versions for 2 main reasons. The first reason is that people used to compare the structure of the index with a relational database. While comparing an index to a database and a document to a tuple seems sound, a type cannot be compared to a table because tables are independent, types are not. For instance, if we created an index with 2 or more types, we could set a mapping different for each type, however if a field is named the same in two mappings, then it has to be the same datatype. It is not possible to have the field "name" be a String in a mapping and an Integer in the other. The other reason has to do with how the cluster stores the data, which we are going to see.

The cluster has the construction pictured below:

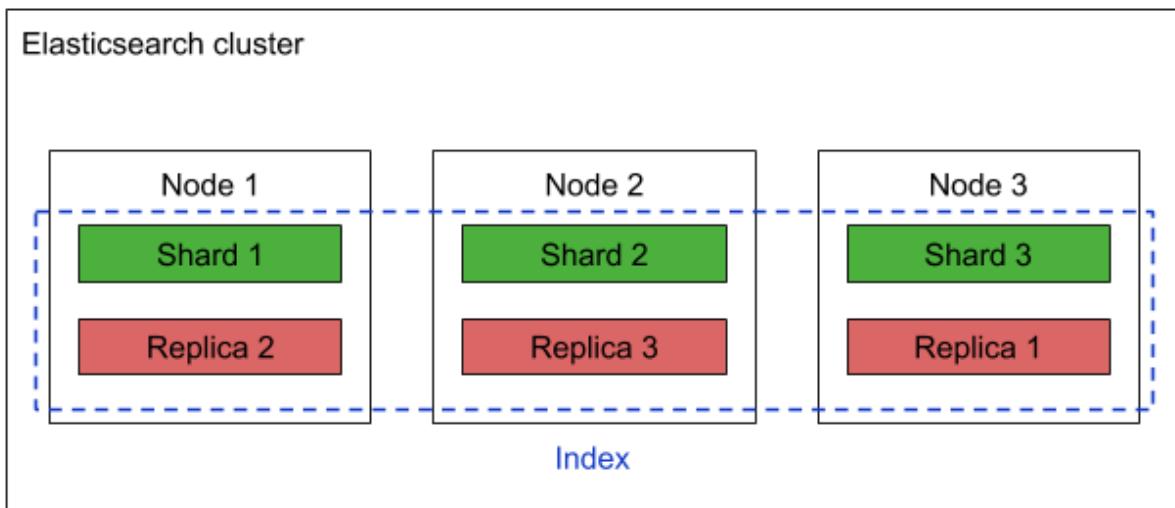


Figure 4, Elasticsearch architecture

- A cluster is composed of one or several nodes
- Each node works independently and each node can have 0,1 or many primary shards, the same thing for replicas
- Primary shards and replicas are Lucene inverted indexes
- Replicas are copies of the primary shards
- An index is made of 1 or several primary shards. Each primary shards can have 0 or more replicas

Elasticsearch is based on Lucene, an Apache library that allows creating inverted index that are very useful for fast-query. An inverted index or segment, is a data structure that maps a token (a word, a number, etc.) to the document it comes from. Let's suppose that we have the following corpus:

- Document 1: "The Hitch Hikers Guide to the galaxy"
- Document 2: "Guardians of the Galaxy"

The resulting inverted index would be:

| Term ordinal | Terms list | Postings list |
|--------------|------------|---------------|
| 0 | galaxy | 1,2 |
| 1 | guide | 1 |
| 2 | guardians | 2 |
| 3 | hikers | 1 |

| | | |
|---|-------|-----|
| 4 | hitch | 1 |
| 5 | of | 2 |
| 6 | to | 1 |
| 7 | the | 1,2 |

Here we see each word number of appearance and the documents they are in. It is also possible to go a level deeper and have the position of the terms in the documents. This example is fairly simple, the documents only have one field. When indexing real-world documents, this is rarely the case. To create the inverted indices, Elasticsearch separates the fields. This is the second reason for the deprecation of having several types per index. A quick example to see how Elasticsearch parses the documents and their fields. Suppose the 2 following documents:

- Document 1:
 - Name: John
 - Age: 25
- Document 2:
 - Name: Jack
 - City: New York

These 2 documents have a field in common and that's all. From this, Elasticsearch would create the following table:

| | Document 1 | Document 2 |
|------|------------|------------|
| Name | John | Jack |
| Age | 25 | null |
| City | null | New York |

The problem here is the *null* values. If a document has a different mapping than what has been set for the index, Elasticsearch still tries to fill the void by setting it to *null*. In small cases as this one, it is not a big issue, however if the index has several types with hundreds of fields with little in common, there will be a lot of empty spaces. Not only will that take storing space, it will also take computation and compression time.

For those 2 reasons, types are now limited to 1 per index. It is possible to bypass this restriction by simply setting the type as a field of the mapping and not as a different mapping. However this would negate the point to not have many types by index.

When inserting data in our cluster, it is done in a round-robin fashion, meaning that the first document will get in the shard 1, the second document will get in shard 2 and so on, each shard getting a document in its turn. Every second on every shard, an inverted index is made for the new documents inserted during that period. Once created, the inverted index is immutable, it cannot be modified, meaning we cannot add or delete terms, and it is stored in the shard with the documents. If more documents are added to the node, it will be analyzed and computed in another segment. If we decide to remove documents from our node, a new type of segment is created where each document to be deleted is marked. At this point, the documents are still in the cluster but they are no longer accessible. When Elasticsearch is less under stress, whether in writing or in reading, it will decide to read this segment and to actually delete the documents. When doing so, it will recreate a new inverted index with the documents remaining. Furthermore, if a segment is too small or by request of the user, segments can be merged.

Now that we have seen the basics of Lucene, we have to understand why Elasticsearch organizes itself in such fashion.

First, the round-robin when inserting documents. When creating a cluster, each node is positioned on a machine or server. Having 3 machines means that you have 3 nodes for instance. Distributing the load of data is a way of not overwhelming a machine. Furthermore, splitting the data equivalently has two advantages:

- It is safer should a node be disconnected from the cluster, we would lose less data
- When querying time comes, it is better to not have a giant shard next to small ones in terms of performance

The first point is strongly connected to replicas. Indeed, as stated before, replicas are copies of a primary shard, and we can have as many as we want for a primary shard. They have 2 main uses:

- As they are never positioned of the same node as the primary, it ensures that if a node fails, we do not lose access to the data
- When querying time comes, it is the fastest answer of the shard and its replicas that will be taken into account

If we take as example the same cluster as pictured before, with 3 nodes, 3 primary shards and 1 replica per shard, let's suppose that the node 3 gets disconnected from the cluster. As the primary shard on it had one third of the data of the index, it would be pretty bad if there was no backup. Luckily we set the replica of the primary in the node 2, ensuring that the data is in fact not lost. Of course, if we lose node 2, it gets more problematic but we could have set 2 replicas per primary shards, then node 1 would have had the shard 1, and the replica 2 and 3.

In addition to this, Elasticsearch by default dispatches the shards dynamically. This means that if we create an index with 3 shards, Elasticsearch will decide on which node to set the shards. Most usually, and thankfully, the cluster spreads the shards on different nodes, according to some parameters, for instance the size of the hard-drive of the server and the space left on the hard-drive. In our 3 node cluster, say the node 1 is full however, the indices with 3 shards would be spread on the 2 remaining nodes, and this would defeat the advantages of having multiple shards. However when some space would be made available on node 1, Elasticsearch is free to move one shard to it.

The second point is about the way Elasticsearch deals with queries. When a user asks for something, the request is sent to each node. Then each primary shard and replica executes the query on the data they contain. When they are done, they send the response, merging their answer with the other nodes. If we suppose that shard 1 has 75% of the data, it will take a much longer time to retrieve the documents and to compute the score for a given query as there are more documents and Lucene segments to analyze.

When talking about a term, we usually understand a word, but it can be more than that, which is why we must associate a term with a token instead. In the example seen before, we had standard tokens, one word is equal one token but that is not always the case. As we saw earlier, when creating the mapping for an index, we can specify the datatype of every field. While there are many types available, going from a String to a Geo-IP, we will focus on the Text datatype for now. Elasticsearch enables us to add multiple analyzers for a field. For instance, if we have a field that is the description of a movie, we can choose how we want the tokens to be created. The most classic way is to create a token every time we encounter a white space. But that might not be enough, we may want a deeper analysis, depending of the language of the documents or the information stored. For instance we may want to separate words if there is a capital letter in it ("ElasticSearch" would become "Elastic" and "Search" in the Lucene segment) or if there is a special character ("Elastic-search" would become "Elastic" and "search"). All these options make the search engine really flexible.

Now that we have a better understanding of how exactly information is stored in an Elasticsearch cluster, we are going to dive in the architecture in which the search engine is used in the company. We have seen how the cluster behaves when it gets data, but we haven't talked about the way data is getting there. As mentioned before, the documents have to be in JSON format, meaning that the documents are constituted of key-value pairs and can contain long strings of characters. In real-life, the documents indexed in a search engine can come from all kind of sources and have all kind of formats. So we know that we cannot put directly the data in the cluster, we need to add a step and this is what ETL tools are for. ETL stands for Extract-Transform-Load, their goal is to gather data from different places, to transform them according to the need and to load them onto the database of our choosing, in our case Elasticsearch. This is

not the only brick we add to the cluster so let's have a look at the graph below to see how everything works together.

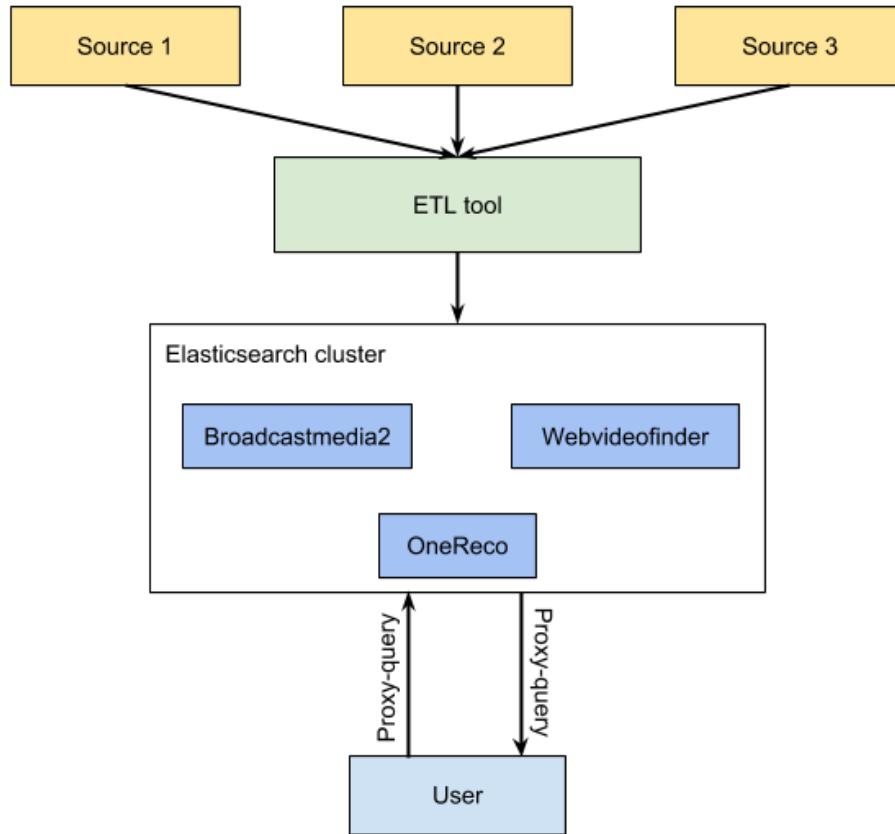


Figure 5, Search On Demand architecture

From the top to the bottom, we first have the different sources from which we gather the data. These sources come from the different clients that need to have a full-text search capability. In some cases, the data is sent by another company and when Orange gets it, a filtering happens with a first ETL tool, but this is not pictured above. After this, the data are loaded in different formats and whether it is a relational database, a XML file or anything else, the second ETL tool collects it then cleans it, adds intelligence to it and gives it to the cluster. The cluster, in the manner explained above, stores the data for each index. Finally, when a user sends a query, it is run by a hand-coded tool called Proxy-query. It is used mainly for 2 reasons. First the query is sent as an URL link but this is not valid for Elasticsearch, JSON is used as the querying language. Proxy-query translates the parameters sent to a valid query. Second, when a document is a match for a query, it is not possible to filter the fields dynamically, the whole document is sent to the user, and this tool enables us to add further filtering. We will talk more deeply about Proxy-query later on.

Elasticsearch has 2 ways to connect the ETL tool with the cluster. At the beginning, we were using the TransportClient. This client, developed by the Elastic company, is connected via the port 9300, which is the same port used for the internal communication between the nodes of the cluster. This means that the ETL tool can basically be assimilated to a node of the cluster, and given that Elasticsearch is developed in Java, everything is set for it. However, when Elasticsearch does an update, the Java version usually changes, rendering the TransportClient version obsolete as well. In the case where the ETL is hand-coded, the TransportClient is part of it and if one wants to do the update, the client won't work fully. This is a problem because it can be quite time consuming to update the ETL tool. Since Orange wanted to update its Elasticsearch version, we needed to rework our tool. Now the second way is to use the REST client, exposing the cluster via web services and HTTP. This way, it doesn't matter the programming language one uses, as long as there is a REST client available. As this client doesn't have a dependence

on the Java version, it is much more stable, and so we decided to change our ETL tool to use the REST API.

When one inserts data in a search engine, a question arises: when do we add the new data? Ideally, we would always want the freshest data. Say we had a search engine for news, we would want everything available instantly. However it is not always that easy. If we are taking data from files, how do we know the data that were removed or the ones that have been added. We could do a file for the documents that we need to add and another one for those that need to be removed. But we may want a backup file with every document if we want to start from scratch. For us, one of the sources from which we gather data is another Elasticsearch index. In this case, we can manage to detect the new documents by looking at the time they were added, and if this time is more recent than the last document added in our index, then we take it. However how can we detect the documents that were removed.

Let's take the index Webvideofinder that was presented earlier. The data come from another index that takes its data from different partners. But the documents here are videos coming from website such as Dailymotion, and as such, videos can be done by anyone. If a video violates the images right, it needs to be taken down. In this case, first the previous index deletes this video, but then how our index detects this deletion. One solution is to set an Enterprise Service Bus (ESB), such as RabbitMQ. RabbitMQ is a message broker tool that allows sending messages to whomever. It works by creating different queues, in which messages are sent. Then any user can subscribe to a queue and get the message in real time. Those messages can be anything, for instance a document identification with a CRUD instruction. CRUD stands for Create, Read, Update, Delete, and describes the main functions of storage. So upon received the notification to delete a document, people managing the upstream index could also transfer the information in a queue. Then we would only need to subscribe to it and deal with the instructions one at a time. However this works for simple documents easily but not so much for other indices we have, such as Broadcastmedia2. Indeed, the documents of this index are constituted of several documents that are merged during the transformation step. Trying to update sub-documents seemed quite difficult at the time, so instead we decided to work by re-indexing the data when needed. This means than for example, for Webvideofinder, a new index is created every 2 hours and we start the indexation from 0. Other indices that don't require such new data can be indexed once a day or even less often to avoid overloading the cluster.

To prevent any kind of problem where a client would find 0 documents in its index, Elasticsearch has a simple solution. Indices created require a unique name. Unfortunately, names change and using those names for any clients isn't a viable option since they would have to update their configuration every so often. Instead, it is possible to set an alias on indices. Clients would then use the alias to query an index instead of the name. This way, it is possible to create a new index and take as much time as we want to fill it, and when it is finally over, the switch of alias is instantaneous. As an example, suppose we have an index named "adserver_v1". For some reason, a new indexation is necessary and so we create the index "adserver_v2". The alias "adserver" is set on the version 1 while the indexation happens. Once it is complete, the alias is removed from "adserver_v1" and set to "adserver_v2" and this is done as atomic operation. Either both go through or none. Furthermore it is possible for an alias to be set on several indices, and while we will not go more in details, it may be useful if we want to query several indices that possess the same fields.

Finally, Elasticsearch can be combined with others components, Kibana and Logstash to form the ELK stack. Each of these components has a specific purpose. Elasticsearch is storing, indexing and making data accessible. Around it, Kibana is the data visualization tool directly plugged to Elasticsearch, and Logstash is the ETL tool made available by the company Elastic to gather the data, transform it and load it in the cluster.

Now that we have seen the architecture of the project, let's go deeper in Elasticsearch.

Chapter 3: Developments

3.1 ETL

Now that we have seen the architecture of the project, we will get a more practical point of view. We just saw that Elasticsearch had an ETL tool designed right for it, so why decide to go back to square 1 and do one ourselves. Before we get into the how it works, we have to understand what it is. An ETL is a process to move data, add some transformations and to load it in a database. There are 3 main steps for it, as the name suggests. Extract to gather data from their sources, Transform to manipulate data in any way wanted and Load, which is self-explanatory. Hand-coding such a tool, while making sure the mastery of the tool is kept, can be very long. And even when it is done, it might still require more debugging.

On the other hand, ETL tools are ready-to-use, most of them have an interface for users to design their pipelines, which provides a better visual idea of the logic. Many websites^{3,4} tend to recommend these tools for different reasons. ETL tools are resilient, they are easily pluggable from an API to another, whereas hand-coded solutions could need rethinking and/or rebuilding when changing APIs. Another point that may not seem much is for the documentation. Producing documentation can be long and tiresome, while ETL tools are quite documented. This is a problem for hand-coding it, especially when the communication with other teams is necessary. Teams will need to know which transformations are done on data and without documentation or visual flow, it could be difficult to dive in the code of the application.

A point in the disfavor of ETL tools is that they are usually for simple transformations, substitutions or data cleaning, and for this project we had some more advanced need, such as the documents requiring merging several documents.

Overall, it is usually simpler to go for an existing ETL tool since it is already developed, users only need to implement it. But of course the final say in the matter depends on each need and environment.

Prior to the beginning of the thesis, Orange had already a hand-coded ETL, however there were several problems with. The speed for the whole pipeline was quite slow and the whole tool was quite unstable, crashing at times. When developing the new tool, we also changed programming, going from Java to NodeJS. As it turned out, Java was used in a manner that created lots of connection to Elasticsearch and often these connections would not close. This resulted many times in Elasticsearch being full and closing all connections. Because of all the issues, we decided that it was easier to restart from scratch than rather to try and fix the Java version. Now we are going to see in details the different parts of an ETL.

3.1.1 Extract

An interesting thing to note is that unlike a search engine such as Google, Elasticsearch doesn't have a crawler. Google works basically by trying to connect websites, constructing a map of the web. Elasticsearch on the other hand will not go and try to gather data, we need to directly give it. This strengthens the need for an ETL tool. And the first step for it is the extraction part.

As we discussed before, we saw in our architecture we have several sources from which we take data. Overall, we have four types of source, from a SQL database, from CSV files, from XML files and finally from another Elasticsearch index.

There is not much to be said about the files, they are set on a server and then when we need them, we take them. Most of our data come from an Elasticsearch index named "content". This index is divided in many types of documents that for the most part have nothing in common. The data from this index come from different partners of Orange that send their content, videos,

³ <https://www.etltool.com/the-7-biggest-benefits-of-etl-tools/>

⁴ <https://www.matillion.com/matillion-etl/stop-hand-coding/>

news or ads. This is quite strange given what we know about types and indices, however this was done years ago, on a very early Elasticsearch version, and changing it now would take great efforts. Furthermore, this index is not used for querying, it is only used to feed our indices, so users cannot be affected by it. It plays a role of buffer between the data coming from the exterior and our indices.

To read the data from an index quickly, Elasticsearch uses a technology called scan and scroll. The scroll part is a way to retrieve a larger number of documents with a cursor, making it so we could get the first one thousand documents and giving the id of the scroll, we could get the next thousand and so on. It is important to note that the scroll does a snapshot at the beginning of the chain. Any documents that would come after the beginning would not be taken into account for this scroll. Now the scan is going to be of more interest to us. When doing the scan and scroll, we have to specify a query in order to know which documents we need. And it is possible to sort the documents if we want to. However this poses a major problem, the time it takes to order a very large number of documents. With the ETL hand-coded tool we had, a sort was done and the time wasn't satisfactory. With our new tool, we realized that it wasn't important to sort the documents and that when the scan condition comes in play. By adding it to the query, Elasticsearch understands that the response doesn't need to be sorted.

3.1.2 Transform

Once we have extracted the data we need, it is submitted to the transform pipeline. Here, every kind of transformation can be applied. It depends mostly on the need of the clients. First of all, there is usually a big data cleaning part. That is a necessity in many cases.

For pipelines that have a file as source, there is no query specifying the documents we should keep or we should remove. When we get the whole file, we need to be able to keep only the documents that are important. The transform part enables us to remove the documents that don't have value as we would like.

For pipelines that come from another index, a first filter is done by the query we use to retrieve documents. However that might not be enough. When querying an index, we usually retrieve all fields but we may want to filter the fields we keep in accordance to some condition.

When the project OneReco emerged, it used the same index than Polaris Video, Broadcastmedia2. Both clients had mostly the same needs, the same documents and the same query, so the decision was to use only one index. However, as time went on, both needs diverged from each other. We had to add fields in the documents that were only used by one client, or to transform the same field differently, meaning we had to duplicate fields. This was adding to the weight of the documents and thus degrading performance while making it more and more difficult to maintain as both clients needed to watch over carefully for any changes. Furthermore, we realized that OneReco needed a much bigger validation part than Polaris Video. Finally, the decision was to create a brand new index for OneReco.

The validation part was a big issue for OneReco. Until then, they used to do it on their side, which drew massive performance issues. At the time being, the load was less than a 100 queries per second on the index, but the project was to evolve, first to reach 250 queries per second then above 900 queries per second. Additionally, the time response had to be less than 400 milliseconds. For each query, the need was to retrieve the 100 first documents that matched. However, on the 100 documents, after the validation part on the client side, only 20 documents could be validated while the rest was discarded. So another request was sent to retrieve the next 100 documents that were then going through the validation part, so on and so forth. Overall, each query needed a few back and forth between the client and the cluster to have a reasonable result and that influenced badly the performance and the response time. As this was unacceptable, the validation part was transferred upstream, during the transform part of the indexation pipeline.

In the index Broadcastmedia2, there were around 900.000 documents, but only about 150.000 of them had broadcasting event. For the new index OneReco, we removed the 750.000 documents without any events any it wasn't of any interest to the client. Then according to the specifications, we managed to reduce the number of documents to around 120.000. By removing all those documents, we made sure that there was no need for the back and forth queries to get a decent result. In addition to removing invalid documents, we also greatly reduced the size of each documents. Many fields that weren't used anymore since OneReco had their own cluster were removed. On some documents we removed up to 60% of the size of it.

The benchmark here will present the progress made. Below is a graph of the time response.

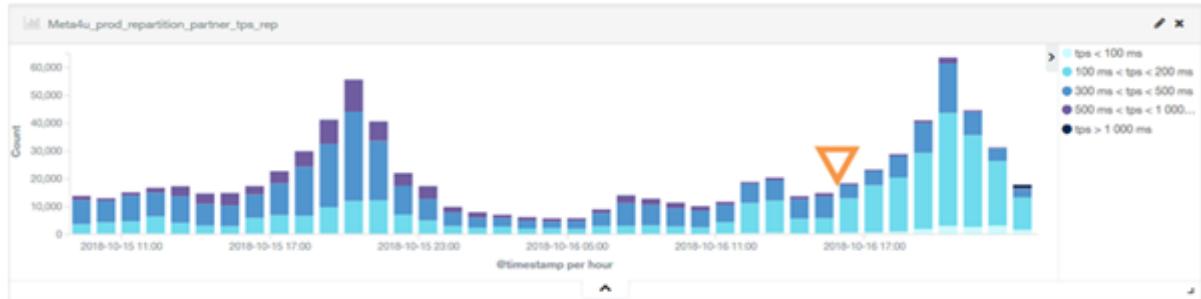


Figure 6, answer time OnceReco

Here, the arrow is when we switched from the index Broadcastmedia2 to OneReco. Before the switch, we see that most of the requests were between 300ms and 500ms. Once we had done all the changes, we see that for the most part of the requests, we get a time response between 100 ms and 200 ms. In summary the changes to get a much better performance were to have a better preprocessing of the data upstream the search engine, by removing more than 80% of the size of the index and by making sure the data was good to use as-is.

The second part of the transformation step is to add intelligence to the data. As we presented earlier, the documents in the index Broadcastmedia2 are composed of many documents. Indeed, at the lowest level, we have the “parent” document that contains most of the metadata. In a movie, it would be for instance the release date or the actors. Then we need to add the “child” documents, which are the broadcast events, for instance when and on what channel will the next broadcasting happen. Unfortunately, doing so reduces the purpose of having a non-relational database, since we are forcing relation between the documents. Furthermore, since the “child” documents do not come from the same type in the index “content”, during the transformation part, we need to multiply the requests to the index, soliciting even further the network and the cluster. Another example of processing is merging fields. For instance if we have a field first name and another last name for an actor, we might want to bring those 2 together and have only field called name. A more interesting case is with the taxonomy array for the OneReco index. This array, which defines the genre of a movie for instance, according to several standards and it can be used as a classifier during a query. This is what it looks like.

```

"taxonomy": [
  {
    "name": [
      "magazine",
      "society",
      "mag. society"
    ],
    "weight": 0,
    "id": "genre-orange:magazine:society:mag--society",
    "type": "Genre Orange"
  },
  {
    "name": [
      "broadcast",
      "magazine"
    ],
    "weight": 25,
    "id": "categories:broadcast:magazine",
    "type": "categories"
  },
  {
    "name": [
      "education"
    ],
    "weight": 30,
    "id": "subjects:education",
    "type": "subjects"
  }
]

```

Figure 7, taxonomy field of OneReco

Here we see 3 types of classification, the genre from Orange, the category and the subject and each value is different. Furthermore, a key “weight” is set, adding to score of the document if matched during a query. The taxonomy is first defined in each broadcasting event. Then it is merged as a union during the transformation part with the other events and brought in the main section of the documents. However by keeping every taxonomy, we may end up polluting our document. Let’s take for example the TV show “Barbie” and suppose the show has 20 episodes with one broadcasting event. Most taxonomy of the events would have information such as “animation”. But if one episode has the character doing tennis, it could get classified as “sport”. This one taxonomy would then get in the main core of the document and a person looking for a sport show could find “Barbie”, less likely to be of any interest. A few solutions exist to deal with this. The first idea is that we could do an intersection of the taxonomy instead of a union, but that would mean potentially lose precious information. The second idea is to set a threshold for which every occurrence of a term below would not be kept. If we keep our example, having 1 in 20 events with a taxonomy “sport”. If we set our threshold at 10%, it means that we keep only the taxonomies that appear in 10% or more of the events. “Sport” would be present in only 5% of the events, meaning it would not be kept for the final document.

3.1.3 Load

The final step of the pipeline is loading in the database.

Several things come into play here. We are first going to look back at something we saw previously in this document. We talked about the fact that while writing data in the cluster, it is done in a round-robin fashion between the primary shards of an index. If the first document will go to the shard 1, the second document will go to shard 2 and so on. This is very useful for 2 reasons, it distributes the load of a node writing-wise and reading-wise. Spreading the documents on several servers will reduce the load in each one and as we have seen before, the shards answer independently to a query, making it so that there are fewer documents to analyze for each shard for each query. We have several parameters to look at if we want to make sure we are using the cluster as we really want to. The first parameter is of course the number of primary shards that we want for our index. To determine this, we have several factors to take into account:

- The number of nodes the cluster have. It is useless to have more shard than node because then several primary shards of the same index would get on the same node and that would be counter-productive.
- The size of the index. The more an index will have documents, the better it is to spread it to not overload a machine and to have faster responses for queries.

The second parameter is the number of replicas. As a reminder, the replicas are copies of the shards, used to have a redundancy and a faster response. A cluster with 3 primary shards and 1 replica per shard would mean that we have 6 shards overall. Replicas are great to have, however the rules to choose the number differ a bit than for the shards. While it still depends on the number of nodes of the cluster, it is probably not a good idea to have too many replicas. Having a replica per shard means that writing has to be done on 2 shards at the same time and that takes computing time and resources. Having even more replicas per primary is great when users send queries, the first shard having the response will send it to the cluster, but on the other hand it is quite bad during the indexation. An interesting note is that, while doing the indexation on a primary shard plus the number of replicas chosen takes some time, duplicating an existing shard is quite fast. A solution would be to do the full indexation on a number of primary shards without any replicas. Then once all the writing is done and the index is full, we could duplicate each shard as much as we would want with a minimum cost. This way, indexation time would be low and during that time the queries would go slower. After this by creating replicas we would get better answer time. While this is a good workaround to improve both conditions, it is not perfect. During the indexation, the risk of having 0 replicas could be a problem. If a node with a primary shard gets separated from the cluster, there would be no backup for those data.

In our project, we decided not to do this solution because several times in a period varying from weeks to months, nodes would get disconnected from the cluster.

Once the concerns about the shards are solved, there other points to look at to load the data. A small issue but interesting nonetheless is the id of the documents. Indeed, documents that are loaded in an index are set with an id, either given by the user or left up to Elasticsearch. The important note here is that if we decide to set the id ourselves, the id should not be randomized because Elasticsearch will try to find patterns in the id, and if they have nothing in common, it will only make its job harder.

Finally, and maybe the most important part in Elasticsearch architecture is the mapping of the documents. We mentioned the different datatypes that could be used and the way tokens work but there is much more to it. As a reminder, a token is a fragment defined by an analyzer. It most usually is a word, but we need to know what words are and how to separate them. The most basic way to separate tokens in a string of character is by cutting the string at each whitespace. The tokens created are then the atomic units used during the query, the unit must be matched by the terms of the query. If we don't specify any analyzers for the fields of our documents, Elasticsearch will use the one by default, so it will create a token each time there is a whitespace. However the real added value of the mapping is to be able to set analyzers ourselves, which is essential to have a good relevance. The creation and use of analyzers is split in several parts.

The first part is creating filters. These filters can have different functions, such as the way to create tokens or setting synonyms. Note that it is possible to have as many filters as we want for a given index. Let's have a look at the filter example below.

```

"filter": {
  "custom_filter_word_delimiter_1": {
    "catenate_all": false,
    "catenate_numbers": false,
    "catenate_words": false,
    "generate_number_parts": true,
    "generate_word_parts": true,
    "preserve_original": false,
    "split_on_case_change": false,
    "split_on_numerics": false,
    "stem_english_possessive": false,
    "type": "word_delimiter"
  },
  "plurals": {
    "type": "synonym",
    "synonyms_path": "dicos/common/plurals.txt"
  },
  "my_stopwords_newshop_filter": {
    "stopwords_path": "dicos/newshop/stopwords.txt",
    "type": "stop"
  }
}

```

Figure 8, Elasticsearch custom filters

This is quite heavy, we are going to take it apart. At the top, the “filter” clause is used to define filters, everything in this object will be considered as a filter and right below we find the name of our first filter. The filter “custom_filter_word_delimiter” has a lot of options. We will not go into what each and every one means, but as we said previously, it allows us to separate terms on characters like apostrophes. It goes even further as to being to separate words if they contain a character uppercase. The more interesting thing here is the type. For this filter, it is set to “word_delimiter”, being quite clear about what it has to do, but there are other types.

The second filter is called “plurals”, we can guess what it does. Here we see that the type is “synonym”. This is very useful if we need to associate terms together. A plural synonym means that for a given word, let’s say “dog”, it will be associated with “dogs”. During the indexation, each time this word will appear on a field with this filter, Elasticsearch will create an inverted index with both token. While without the synonym a search “dogs” on a field “the dog is happy” would not match because the most fundamental unit doesn’t matches, with the synonym filter it will match. In a search engine, synonyms can be used for much more than singular-plural association, for instance for acronyms. Indeed, if we put ourselves in the shoes of a user, the fewer words we write, the better, because writing is exhausting. Writing a full name of an organization would be near unthinkable. We would never write “World Health Organization”, we would only write “WHO” in the search bar. The synonyms allow us to do this connection, making it easier for everyone.

The last filter type we are going to see is concerning the stopwords and to do so we need to understand what stopwords are. They are words that do not carry much meaning because they are really frequent in the corpus. This frequency doesn’t allow to extract any meaningful information about a document because every documents will have more or less the same occurrences. We will look more closely later at how the frequency plays a part in the relevance work of Elasticsearch, but for now we can intuitively understand that words like “the” or “a” are less interesting to retrieve than the word “dog”. The filter type “stop” takes a list of stopwords defined by us and every time the words are indexed, Elasticsearch prevents them from being inserted in the inverted index. This means that we won’t be able to search for them, because they will be virtually nonexistent in Lucene.

The second part is creating the analyzers and using our filters. Analyzers can be composed of filters, tokenizers and character filters. Right below is an example of an analyzer using the filters that we just saw.

```

"analyzer": {
  "custom_analyzer_default_tokenized": {
    "filter": [
      "custom_filter_word_delimiter_1",
      "my_synonyms_newshop_filter",
      "my_stopwords_newshop_filter",
      "plurals"
    ],
    "tokenizer": "whitespace",
    "char_filter": [
      "%26_charfilter"
    ]
  }
}

```

Figure 9, Elasticsearch custom analyzer

The same way we can define several filters, we may create several analyzers, depending on the need of every field and the kind of query we may expect. The analyzer that we created is called “custom_analyzer_default_tokenized” and the first thing we see is that we put all the filters we saw, plus another one for synonyms. When we set this analyzer on a field, all the filters will be applied. Below the filters is the tokenizer, the standard way to separate the words in tokens and in this case we have chosen the whitespace delimiter. Finally the character filter is a way to apply a further processing character by character. It can be used for instance to avoid trouble with encoding characters. For instance, in HTML the character “&” is known as the entity “%26”. If we index data coming from an HTML documents, we may not want to have the entity but rather have the proper character.

Once we have created our filters, we need to use them. The implementation happens in the mapping of the index and this is what it looks like.

```

"description": {
  "type": "string",
  "index": "no",
  "fields": {
    "block": {
      "analyzer": "custom_analyzer_default_keyword",
      "type": "string"
    },
    "tokenized": {
      "analyzer": "custom_analyzer_default_tokenized",
      "type": "string"
    },
    "filtered": {
      "analyzer": "custom_analyzer_default_tokenized",
      "type": "string"
    },
    "elementary": {
      "analyzer": "custom_analyzer_elementary",
      "type": "string"
    }
  }
}

```

Figure 10, mapping of field 'Description'

Here we are looking at the mapping of the field “description”. We saw previously how Elasticsearch indexes documents, by creating Lucene’s inverted index, and we also saw that the tokenization could change. Prior to analyzers, 3 options exist to tell the cluster what we would like

to do with each field. These options concern the parameter “index” that we can see in the mapping above. The default value is “indexed”, it means that it takes the input field and it indexes it in the way we explained, by taking each word apart. The second value is “not_analyzed”, meaning that the input is indexed as one token, no analysis is done and yet the field is still searchable. The search however has to be exactly matched, since it is not analyzed. The third and final option is “no” and it means that the field is not indexed, much in the way of the stopwords, so no query can be done on the field. We can still see the field if the document is retrieved by another field. Preventing fields to be indexed is a good way to save space and computing power if we know that no queries will be done on it.

In the example above, we see that the default “description” is not indexed, meaning we can’t query it. However, the interesting thing is to create subfields with analyzers and that is what we see. Here we set 4 subfields of “description”, each having an analyzer different and among them we see the one we created earlier. Those subfields will only be for Elasticsearch, during the creation of the inverted indices and for the query, the transformation resulting of the analysis will not be returned to the users. Having the simple “description” not indexed is useful because the default analysis may not have any value in some cases, so not indexing it and creating subfields make more sense.

3.1.4 Results

Now that we have seen how the ETL works, we will see the results and the improvement of the different optimization that we have done.

Earlier we talked about how we chose to create new version of the indices for each new full indexation, but prior to recoding the ETL tool, that wasn’t the case. For most cases, we used to create a new index only when the mapping or the settings were different. This had several implications for our different clients. For Webvideofinder, it meant that we weren’t removing documents from the index. Webvideofinder takes its data from the index “content” and the ids are not random, each document has a stable id, so when doing an indexation, we would simply overwrite the corresponding documents, while the documents that were removed were not accounted for. While there isn’t usually lots of removal, on the long run, if not many change occur on the mapping, then the index would not remove those documents for a long time. For Broadcastmedia2, it followed the same principle than Webvideofinder to write updated or new documents. In addition to this, every 15 minutes, the ETL would find every broadcasting event with a past broadcasting date and remove it. While this is not perfect, it was allowing a removal on events that were no longer relevant.

The fact that we didn’t remove documents also means that indices would get bigger and bigger by accumulating Lucene segments. As we said before, the merging of segments is done when the cluster has enough resources to make them. However the indexation were almost constantly running and followed this timing: at each quarter of an hour, if an indexation wasn’t running for a given index, it would start. This meant that at maximum, indices were not written into for 15 minutes. Even if Elasticsearch is built to handle dynamic data, if the index keeps getting written into, the merging of the segments is more difficult.

As stated in “*An introduction to Information Retrieval*” [1], reconstructing an index from scratch is “*a good solution if the number of changes over time is small and a delay in making new documents searchable is acceptable - and if enough resources are available to construct a new index while the old one is still available for querying*”. In our case, since our indices may contain only up to a few million documents, the number of changes are not tremendous. In addition to this, a simple remedy is implemented to prevent delaying access to data. For Webvideofinder, when a full indexation happens on a new index, an update is done every minute on the operational index, importing the new documents.

Overall, 2 results were quite interesting with implementing all that we talked about. The first one is the size of the indices. As an example, Broadcastmedia2, with a little less than a million documents used to be around 6 Gigabytes. With creating a new index around every hour,

the size diminished to around 2.5 Gigabytes. While this means more space on the hard-drive, it also means that the queries have less data to run through and thus we get improved answer time.

The second result is the indexation time. As an example, for our index Webvideofinder, to index 6 million documents for a total of 7 Gigabytes, the old ETL tool used to take up to 20 hours, which is way too important. Different improvements, including the extracting part no longer sorting documents and a better use of Elasticsearch, we improved the indexation time to less than 30 minutes.

To further understand what kind of improvements made this possible, we are going to look in detail to the shards management. The cluster on which we experimented is composed of 6 nodes. From different testing, we have the following results:

| Primary shards/replicas | Indexation time | Average time to process query |
|-------------------------|-----------------|-------------------------------|
| 1 primary/0 replica | 27 minutes | 50 milliseconds |
| 4 primary/0 replica | 20 minutes | 25 milliseconds |
| 5 primary/ 1 replica | 22 minutes | 20 milliseconds |
| 5 primary/3 replicas | 24 minutes | < 20 milliseconds |

Having 1 primary shard means that all documents are written on the same node, so all the load goes to one place. The associated replica is usually on another node and is getting written the same documents at the same time. When doing the indexation with 5 primary shards and 1 replica per shard, the time dropped to around 22 minutes which can be explained because the millions of documents are being separated in 5 and thus it lessens the load of each node. If we set 3 replicas per shard with 5 primary shards, the time goes back up to around 24 minutes, which seems normal because documents have to be written in 4 shards every time. However, with a given data being on 4 shards, we get very good processing time for the query since we only need the response from the fastest of the 4 shards. On the other hand, the worst processing time is when we only have one shard, because when a query needs to be processed, the one shard needs to run through all the data.

A note of warning still needs to be drawn from managing the number of shards that we haven't seen. If we have a lot of primary shards and replicas, it means that data is split a lot. While this can be great for indexation and querying time, it can be source of problem for the relevance. Indeed, the score of a query is computed independently on each shard, and then the top documents are returned to the cluster with the score computed for each shard. The tops from each shard are then merged and sent to the user with the score that was computed at shards level. The problem with this is the way Elasticsearch computes the basic score of a query concerns shards individually, having many shards means different basic score for each while having one shard means one basic score for all queries.

Overall, tuning the number of shards depends of the number of nodes on the cluster and the load of data that is expected. Some general guidance can be found online⁵.

3.2 Querying

3.2.1 Elasticsearch's basic score

Once we have our data inside the database, we want to be able to retrieve it. One of the main aspects of Elasticsearch is that it allows full-text search, enabling to compute score based on how well the documents match the query.

Elasticsearch uses the Lucene library for the search and Lucene uses the Boolean model. When a user sends a query, it is split in a set of terms and then the search is done with the Boolean logic to retrieve documents that match all or some of the terms. It is possible to specify whether a term should be contained in a document or if it has to be absent. The terms are

⁵ <https://qbox.io/blog/optimizing-elasticsearch-how-many-shards-per-index>

connected with the usual operators of the Boolean logic: “and”, “or” and “not”. When we have the documents satisfying the logic, a score can be applied.

During earlier versions of Elasticsearch, to compute the basic score of a request, the *TFIDF* (term frequency-inverse document frequency) was used (described in [2]). It is a statistic to determine how important a term is in a document part of a bigger collection. It works as followed: the term frequency gives a higher score if a term is more present in a document, and the inverse document frequency gives a higher score if a term is less present in the whole corpus. The term frequency is a nice indicator in itself to note if we have a recurring term in the document, however it is not enough as common words that do not carry much information would be ranked very high. That is why we use the inverse document frequency in top of that. The result is that we rank high the terms that are frequent in one document but rare in the corpus. Those terms usually carry the most meaning. Finally, Lucene adds another measure to compute the final score, it is the *field-length Norm*. As the name suggests, it takes into account the length of the field which is really important when computing the relevance of a term in a document. Indeed, if a document with thousands of pages contains once a term, it should mean that the term is not the main focus of the text, whereas a title only a few terms-long containing a term should have great meaning.

Let's take a closer look at the formulas behind it.

$$tf = \sqrt{\text{Number of times the term is in a given document}}$$

$$idf = 1 + \log \left(\frac{\text{Total number of documents}}{\text{Number of documents that contain the term} + 1} \right)$$

$$\text{fieldlength norm} = \frac{1}{\sqrt{\text{Number of words of the field}}}$$

The *field-length norm* is really important while doing a full-text search because, whereas the classic *TFIDF* looks at the whole document, the norm allows us to specify the search to a field in particular.

Now, suppose we have a corpus of 5 documents:

| Document Id | Text |
|-------------|---|
| 1 | Space is great. I'm in space. I love space. |
| 2 | The dog is running there. |
| 3 | My car is parked in double lane. |
| 4 | That restaurant is the best. |
| 5 | That train was just really late. |

If we had a search looking for the term “space”, we see that in the first document, it has a high term frequency and in the whole corpus a rather low frequency. This means that the document 1 should be the document with the highest score.

| | Document 1 | Document 2 | Document 3 | Document 4 | Document 5 |
|----------------------------|---|---|---|---|---|
| Term frequency | $\sqrt{3}$ | $\sqrt{0}$ | $\sqrt{0}$ | $\sqrt{0}$ | $\sqrt{0}$ |
| Inverse document frequency | $1 + \log \left(\frac{5}{1+1} \right)$ |
| Field-length norm | $\frac{1}{\sqrt{9}}$ | $\frac{1}{\sqrt{5}}$ | $\frac{1}{\sqrt{7}}$ | $\frac{1}{\sqrt{5}}$ | $\frac{1}{\sqrt{6}}$ |
| TFIDF | 0.807 | 0 | 0 | 0 | 0 |

Indeed, since the other documents don't even have the term, they have a score of 0. The same way, if we were looking to score documents according to the term "is", document 5 would get a score of 0. Furthermore, before doing the computation, we can guess that the best score would be attributed to the documents 2 and 4, they both have the term and they have a smaller field-length, which should give them an advantage.

| | Document 1 | Document 2 | Document 3 | Document 4 | Document 5 |
|----------------------------|--------------------------------------|--------------------------------------|--------------------------------------|--------------------------------------|--------------------------------------|
| Term frequency | $\sqrt{1}$ | $\sqrt{1}$ | $\sqrt{1}$ | $\sqrt{1}$ | $\sqrt{0}$ |
| Inverse document frequency | $1 + \log\left(\frac{5}{4+1}\right)$ |
| Field-length norm | $\frac{1}{\sqrt{9}}$ | $\frac{1}{\sqrt{5}}$ | $\frac{1}{\sqrt{7}}$ | $\frac{1}{\sqrt{5}}$ | $\frac{1}{\sqrt{6}}$ |
| TFIDF | 0.333 | 0.447 | 0.378 | 0.447 | 0 |

Our guess was right, documents 2 and 4 get the highest score. Note that in our example, the number of terms in the field and the number of terms in the document are the same because our document only have one field. This isn't usually the case. Furthermore, note that the scores computed here are smaller than in the previous example. The best score here is 0.447 where it was 0.807 before. This is due to the fact that the term "space" is rare in the document, unlike "is". In cases like these, if we see that a term is present in a lot of document and we know that it does not carry much meaning, we can consider this term as a stopword, removing it from the search.

In newer versions of Elasticsearch, *TFIDF* is not the default scoring method anymore. Now Lucene uses *BM25*⁶, for *Best Matching 25*, which is an improvement of the *TFIDF*. Both methods use the term frequency, but for the *TFIDF*, there is no maximum. If the term is in the document 1 million times, the *tf* will keep growing. This is due to simple limitations in terms of technology when *TFIDF* was first invented. Back then, it was better if the stopwords were simply removed, clearing memory. Nowadays, even if the stopwords don't carry much meaning, they still have some. So instead of removing them, the *BM25*'s term frequency has a maximum that never get reached. Elasticsearch gives the following graph.

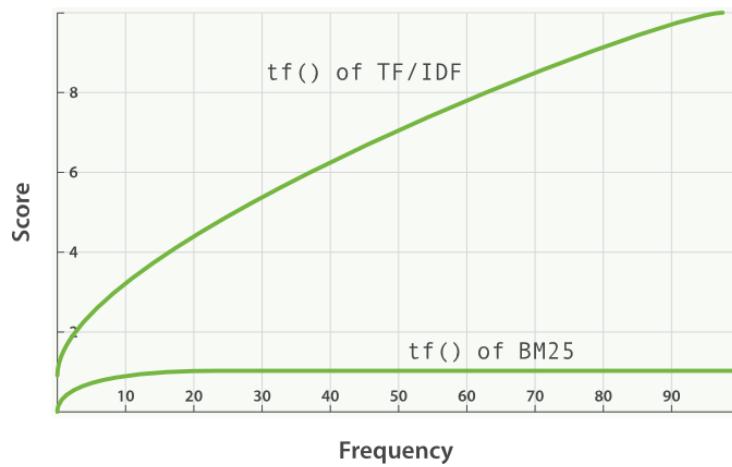


Figure 11, term frequency of TFIDF vs BM25⁷

⁶ <https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevancy/>

⁷ <https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables>

Here, we clearly see that the term frequency of the *TFIDF* grows without limitation, as long as a term is more and more present, the score will be higher. Inversely, *BM25*'s term frequency grows quickly at first but then stops increasing. This is due to 2 new parameters that come into play, giving the following formula:

$$TF = \left(\frac{(k_1 + 1) * tf}{k_1 * (1 - b + b * \left(\frac{\text{Number of words in the document}}{\text{Average number of words for document collection}} \right)) + tf} \right)$$

First, we have the parameter k_1 . It is a way to control the time it takes to reach the saturation. The default value is 1.2. Increasing it would mean that it would take a longer time to reach the saturation, stretching the difference of score for the term frequency in high frequency and low frequency terms. Decreasing it would have the opposite effect.

Second, the parameter b enables us to tune the field-length normalization. A value of 0 disables the normalization while 1 normalizes fully. The default value is 0.75.

For the *IDF* part of *BM25*, the formula changes as well, giving the following:

$$IDF = \log \left(1 + \frac{\text{Total number of documents} - \text{Number of documents that contain the term} + 0.5}{\text{Number of documents that contain the term} + 0.5} \right)$$

The traditional *BM25* *IDF* doesn't have the addition of the 1 in the log, it was added by Lucene to avoid getting a negative score.

Finally, to compute the whole *BM25* score, we remove the field-length norm since it's already taken into account in the *TF*.

Let's try this new computation on our previous example, given the same corpus and the query for the term "is". Suppose we keep the default value for k_1 and b .

| | Document 1 | Document 2 | Document 3 | Document 4 | Document 5 |
|----------------------------|---|---|---|---|---|
| Term frequency | $\frac{2.2 * \sqrt{1}}{4.566}$ | $\frac{2.2 * \sqrt{1}}{3.239}$ | $\frac{2.2 * \sqrt{1}}{3.930}$ | $\frac{2.2 * \sqrt{1}}{3.239}$ | $\frac{2.2 * \sqrt{0}}{3.593}$ |
| Inverse document frequency | $\log \left(1 + \frac{1.5}{4.5} \right)$ |
| TFIDF | 6.020×10^{-2} | 8.486×10^{-2} | 6.994×10^{-2} | 8.486×10^{-2} | 0 |

Once all the computation is done, we have the results above. The score is inferior to what we had before, which could be expected given the new iteration of the term frequency.

3.2.2 Webvideofinder's relevance

Now that we have seen how Elasticsearch computes the basic score for a query, let's dive deeper. In the same way that data is stored, the querying syntax for Elasticsearch is in JSON. There are lots of conditions that one can use in order to tune the relevance, the use of them will depend on the need and will result usually of a big trial-and-error part. It is possible to find more information on Elasticsearch's parameters on its website⁸.

Here is an example of a simple query:

⁸ <https://www.elastic.co/guide/index.html>

```
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "TITLEPAGE": "Emmanuel Macron"
          }
        }
      ],
      "should": [
        {
          "range": {
            "DATEVIDEO": {
              "gte": "now-3d"
            }
          }
        }
      ]
    }
  }
}
```

Figure 12, Elasticsearch simple query

The first term “query” is to define the conditions of the request. In here, it is possible to specify what kind of search we want to do. In our case, at first we just want a simple match with documents that have “Emmanuel Macron” in the title. A very important difference with SQL is, whereas in SQL we “must” absolutely match the “WHERE” clause, in Elasticsearch, you have the possibility to either specify that you “must” match the query or you “should” match the query, or both. The score is computed by each parameter we enter. In this case, the “must” clause is similar to that of SQL meaning that we will only get the documents that contain the 2 terms of our query, and we add to this the “should” that allows us to give a better score to the documents that match this clause, in our case if the documents are recent (within 3 days ago and now). Here, given that the “should” clause is only on the date, the number of documents returned doesn’t change, only the ordering of the result varies, putting fresher documents at the beginning.

There are different parameters that play a role in the score computed, in addition to Lucene’s *TFIDF* score (or *BM25* for more recent version). We can add boosts for certain criteria, we can modify the score according the position of the words in relation to each other, and many other things to tune relevance as we want.

Let’s take an example of a query that we are actually tuning to improve relevance. This was the original query for the video service on Orange’s website, Webvideofinder.

```

    "must" : {
        "function_score": {
            "functions": [
                {
                    "exp": {
                        "DATEVIDEO": {
                            "scale": "1d",
                            "decay": 0.2
                        }
                    }
                }
            ],
            "query": {
                "match": {
                    "TITLEPAGE.no_stopwords": {
                        "query": "Emmanuel Macron",
                        "operator": "or"
                    }
                }
            },
            "score_mode": "multiply",
            "boost_mode": "multiply"
        }
    },
    "should" : {
        "function_score": {
            "functions": [
                {
                    "exp": {
                        "DATEVIDEO": {
                            "scale": "1d",
                            "decay": 0.2
                        }
                    }
                }
            ],
            "query": {
                "match": {
                    "TITLEPAGE": {
                        "query": "Emmanuel Macron",
                        "operator": "and"
                    }
                }
            }
        }
    }
}

```

Figure 13, first iteration Webvideofinder's query

Here we see several things and the query is getting complete than the previous one. First, let's have a look at the "must" conditions. We see that it is divided in 2 parts. On one side we have the query, similar to what we saw just above, with the difference that we added the parameter "operator" with "or". As seen above, Elasticsearch uses the Boolean logic for the query and this parameter is to manipulate the Boolean logic between the set of terms. In this case, we specify that we want at least one term matching in the set. In addition to the match on a textual field, we add a decay on the date of the document. This decay is a way to punish documents that

are older, since we want documents that are recent, but we will come back on the decay in a bit. If we go back to the field “TITLEPAGE”, we see a variation of the previous query. As we saw in the previous part about the mapping of the index, it is possible to set one or more analyzers on a given field. For this field on this index, we have the following mapping.

```
"TITLEPAGE":{  
    "type": "string",  
    "fields": {  
        "block": {  
            "type": "string",  
            "analyzer": "custom_analyzer_default_keyword"  
        },  
        "no_stopwords": {  
            "type": "string",  
            "analyzer": "stopwords_do_not_count"  
        }  
    },  
    "analyzer": "index_analyzer"  
}
```

Figure 14, TITLEPAGE's analyzers

Here we see that “TITLEPAGE” has the “index_analyzer” as the default analyzer and then we create another analyzer that removes the stopwords. In the query, it is possible to say if we want to use the default analyzer or one we created. In our case, we set the “must”, the minimum requirement, to not take into account the stopwords. The other analyzer, block, concatenates all terms, removes special characters, and transforms letters to lower case, creating effectively only one token for the field. This is useful when one wants a query only in exact match but as soon as the query has more terms than the title (or the other way around), the match will not happen. As an example, we can take the string “The rabbit is caught by the cat”. With this analyzer, instead of having a token per word in the inverted index, we would get the token “therabbitiscaughtbythecat”. If we were to run a query “The rabbit is caught” with the same analyzer, the resulting token would be “therabbitiscaught” and thus the tokens would be different.

Thus far in this thesis, we have seen that analyzers are created and set on fields during the mapping. They are then used for text in the index and in the query to be of the same format. However, it is possible to use an analyzer in the index but not in the query. In other words, the index’s fields will be analyzed while the query will not. This is not very useful if we try to have the same format, but it can be used for autocomplete for instance. This is something we have not talked so far, but it is possible to propose a completion with Elasticsearch. While there are different possibilities to do so, we will only present the analyzer “edge_ngram”. If set on a field, it will divide the text into words and then will create N-gram of the text. To better understand this, we need to understand what N-grams are. N-grams are a continuous combination of n items, usually letters from a text. On the word “test”, the 2-grams will be “te”, “es” and “st”. While it may not be quite clear how it could be useful in our case, Elasticsearch adds the fact that the N-grams have to be anchored at the beginning of tokens.

In other word, if we index the text “Great Scott” with this analyzer, we get the following tokens: “G”, “Gr”, “Gre”, “Grea”, “Great”, “S”, “Sc”, “Sco”, “Scot”, “Scott”. These tokens will get in an inverted index. Now, if we want this to be used as autocomplete, the only remaining thing to do is to send a query on the analyzed field for each letter typed by the user in the search. This way, if the user types “Grea”, our indexed field will be a match and it will be returned.

Let’s go back to the query. Just above the match part, we see the parameter “function”. This function enables us to affect to the score computed by the match by adding or multiplying the function score. This function score is ranged between 1 and 0. Here, we made a function that would decrease the score further and further as the date of the video is ancient. When setting a decay function, we have 3 types of decay, as pictured below.

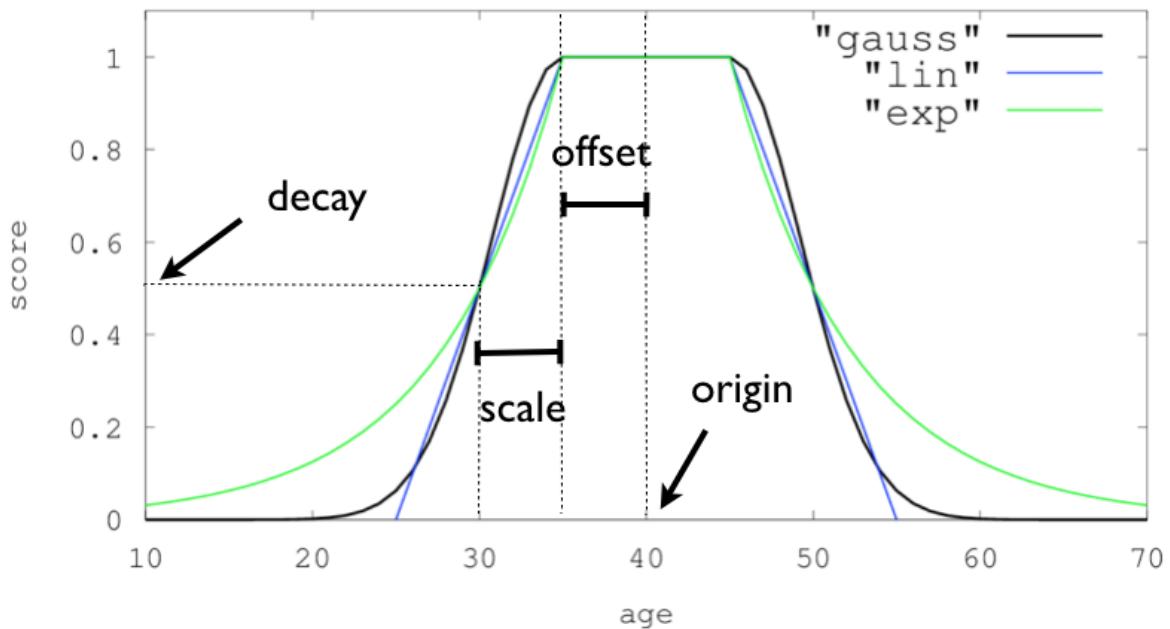


Figure 15, decay function curves

This illustration, available on Elasticsearch's documentation⁹, uses an age variable for the decay function but it works the same for any decay. First, we have to choose the type of decay we want among the 3 possibilities: a Gaussian, a linear or an exponential function. We must choose according to what we need of course. The linear will decay steadily until it reaches a score of 0, the Gaussian will decay slowly at first, then will speed up and then will slow down near 0, and finally the exponential will decay very fast at first and will slow down progressively. Once we have the global curve of the decay we want to apply, there are several parameters to tune. The "origin" is the starting point, by default it is set to now for dates. We can select an "offset" that will prevent values within the offset to be affected by the function. The decay will only be applied to the values that are further apart from the origin than the offset. Then we have 2 final parameters that come together. The "decay" will indicate the score when the "scale" is reached. So in our case, every videos that are between now and one day ago will go from a score of 1 to a score of 0.2. Past this scale, the score will continue to go down following the trend set.

Let's dive deeper in the functions. The following are the formulas to compute the score for each decay:

- Linear:

$$\text{Score(doc)} = \max\left(\frac{s - \max(0, |\text{fieldvalue} - \text{origin}| - \text{offset})}{s}, 0\right), \text{ where } s = \frac{\text{scale}}{1 - \text{decay}}$$

- Exponential:

$$\text{Score(doc)} = \exp(\lambda - \max(0, |\text{fieldvalue} - \text{origin}| - \text{offset})), \text{ where } \lambda = \frac{\ln(\text{decay})}{\text{scale}}$$

⁹ <https://www.elastic.co/guide/en/elasticsearch/guide/current/decay-functions.html>

- Gaussian:

$$\text{Score(doc)} = \exp\left(-\frac{\max(0, |\text{fieldvalue} - \text{origin}| - \text{offset})^2}{2\sigma^2}\right), \text{ where } \sigma^2 = -\frac{\text{scale}^2}{(2*\ln(\text{decay}))}$$

Now this may seems a bit dense, let's take an example. We suppose that we have the following values: origin = 40; offset = 5; scale = 5; decay = 0.5; fieldvalue = 25

If we compute the linear decay, we have:

$$\begin{aligned}\text{Score(doc)} &= \max\left(\frac{10 - \max(0, |25 - 40| - 5)}{10}, 0\right) \\ &= \max\left(\frac{10 - \max(0, 10)}{10}, 0\right) \\ &= 0\end{aligned}$$

This result means that for every age below or equal to 25, the score of the decay function is 0. This is great if we want to remove the documents for which the age is not between 25 and 55 for instance. Let's see how the exponential decay goes.

$$\begin{aligned}\text{Score(doc)} &= \exp(-0.139 * \max(0, |25 - 40| - 5)) \\ &= \exp(-0.139 * 10) \\ &= 0.249\end{aligned}$$

The score returned at 25 for the exponential decay 0.249. The exponential function is a way to avoid getting 0 as the score for a longer time than with the linear decay. This is particularly useful when the score is computed on the score and we don't want to remove the documents for which the date is too old, but we will talk about that later. Finally, we have the Gaussian decay function.

$$\begin{aligned}\text{Score(doc)} &= \exp\left(-\frac{\max(0, |25 - 40| - 5)^2}{2 * 18.038}\right) \\ &= \exp\left(-\frac{10^2}{36.076}\right) \\ &= \exp(-2.772) \\ &= 0.062\end{aligned}$$

Similarly to the exponential decay function, the Gaussian is softer at the edges, meaning it takes a longer while to reach a score of 0.

After we get the score computed, several actions are possible. It is possible to have multiple functions and if we do, we must choose what relation we want between them with the parameter "score_mode". We can add them, multiply them, choose the highest score or the lowest, or make an average. When we get the functions score, we are faced with the same choice to merge it with the score of the match with the parameter "boost_mode". In our case, we choose to use the multiplier, this way as the documents get older and older, their score greatly decrease.

On the second side of the query, the "should" gives us to possibility to add to the score of the documents that satisfies the first part. We have pretty much the same thing than in the "must", the same decay function and the same type of match with one distinction. The operator is set to

“and”, meaning that now that we have all documents that have any of the terms, we give a better score to those that contain all the terms.

However, as one knows, the tuning of relevance is a long work and this first iteration was not perfect. Indeed as we saw, the “match” type of query can either try to match all the terms of the query or only some of them but the relations between the terms are not taken into account as proximity doesn’t matter. We can have a term at the beginning of the field and the other one at the end of it without any relation to each other, and we lose information, getting documents that have nothing to do with our query. In order to change that, we can use the “match_phrase” parameter, giving the following query.

```
"must" : {
    "function_score": {
        "functions": [
            {
                "exp": {
                    "DATEVIDEO": {
                        "scale": "1d",
                        "decay": 0.2
                    }
                }
            }
        ],
        "query": {
            "match_phrase": {
                "TITLEPAGE.no_stopwords": {
                    "query": "Emmanuel Macron"
                }
            }
        },
        "score_mode": "multiply",
        "boost_mode": "multiply"
    }
},
"should" : {
    "function_score": {
        "functions": [
            {
                "exp": {
                    "DATEVIDEO": {
                        "scale": "1d",
                        "decay": 0.2
                    }
                }
            }
        ],
        "query": {
            "match_phrase": {
                "TITLEPAGE": {
                    "query": "Emmanuel Macron"
                }
            }
        }
    }
}
```

Figure 16, second iteration Webvideofinder’s query

This new iteration has the same structure than the first one, except for the type of query. Before, if we were to look for the humorist “Eric Antoine” for instance, the first results we had were documents with titles such as “Eric Nolleau [...] Antoine Griezmann” that had nothing to do with our search. With the term proximity in the query the results for “Eric Antoine” were very good as we only got the documents with “Eric Antoine” in the title.

Yet again, the tuning of the different parameters for relevance is quite sensible, because when finding an example of a query that does not respond the way you would like, we get so focused on improving this result, we may deteriorate the rest. And this is exactly what happened for this example. By using the exact match only, we lost many results, the filtering was too hard. Luckily for us, a parameter is made right for this kind of issue, and the query changed again to the one below.

```
{
  "query": {
    "bool": {
      "must": {
        "function_score": {
          "functions": [
            {
              "exp": {
                "DATEVIDEO": {
                  "scale": "1d",
                  "decay": 0.2
                }
              }
            }
          ],
          "query": {
            "match": {
              "TITLEPAGE.no_stopwords": {
                "query": "Emmanuel Macron"
              }
            }
          },
          "score_mode": "multiply",
          "boost_mode": "multiply"
        }
      }
    }
  }
},
```

Figure 17, third iteration Webvideofinder's query

```

    "should": [
      {
        "function_score": {
          "functions": [
            {
              "linear": {
                "DATEVIDEO": {
                  "scale": "1d",
                  "decay": 0.2
                }
              }
            }
          ],
          "query": {
            "match_phrase": {
              "TITLEPAGE": {
                "query": "Emmanuel Macron",
                "slop": 5
              }
            }
          },
          "score_mode": "sum",
          "boost_mode": "sum"
        }
      }
    ],
  }
}

```

Figure 18, third iteration Webvideofinder's query

```

{
  "function_score": {
    "functions": [
      {
        "exp": {
          "DATEVIDEO": {
            "scale": "1d",
            "decay": 0.2
          }
        }
      }
    ],
    "query": {
      "match": {
        "TITLEPAGE": {
          "query": "__param_q1",
          "minimum_should_match": "2<75%",
          "boost": 0.005
        }
      }
    },
    "score_mode": "sum",
    "boost_mode": "sum"
  }
}

```

Figure 19, third iteration Webvideofinder's query

An example of result we lost is when looking for “Nicolas Hulot végétarien”. The only documents we had in the database with a title similar to this were “Nicolas Hulot veut devenir végétarien”, and since this is not exactly matched, we didn’t return it. However this is the kind of results the user might be interested in. Luckily, there is a parameter in Elasticsearch that could help us with that, the “slop” parameter. It takes an integer and basically enables us to switch the position of the words the number of times wanted. By putting 1 in the slop, we wouldn’t get any results as there are two words between the terms of our search in the title we want to match. So in this example we would need a slop of 2 to match the video. However, it draws the question how to correctly set the slop parameter. In this case, 2 is enough, but in any other case it might not be. A good point with the slop is that the closer the terms are to each other, the higher the score will be. So one might think that as long as the terms are present in the title, we could put a slop at 50 for instance. However by doing this we would probably deteriorate other results, such as “Eric Antoine”. In this case, if the documents were quite old, the decay function would diminish the score heavily, and we would get the results with a bad relevance but a high freshness. Instead of taking the risk to deteriorate the results, we decided to add another “should” clause.

On this other “should”, we go back to a classic “match” with proximity between words. Here, it is the parameter “minimum_should_match” that will interest us. As the name suggests, we may specify the minimum number of terms that we want match in the title. So here we can specifically say that we want a minimum of 3 terms matched. But as always, this could be great for some cases, but if the user is typing only two words in the search bar, then we are in exact match. What we do is if we have 2 words or less in the search, then we should exactly match it, but if it is above, then we want to match only 75% of the terms.

But with this query, we still had problems, 2 use cases that we had trouble satisfying as they were contradictory.

- First use case: When looking for the videos “Les Recettes Pompettes”, we want to retrieve the documents that match exactly, even if they are old.
- Second use case: When looking for the videos “Emmanuel Macron”, we want to retrieve the most recent documents.

The 2 solutions we had at this point were either to put a higher weight for matching exactly the title or to put a higher weight on the decay function to favor the recent documents. If we decided to go for the first solution, we would get the right documents for the first use case, however the second use case wouldn’t have been satisfied. We have a lot of documents that have only “Emmanuel Macron” as the title with an old date, while the more recent documents have more terms in the title, and here the old documents would come up first. Inversely, if we decided to do the second solution, we would get the fresher documents for the second use case but the first use case would not be satisfied. Indeed we have documents that contain the same word as the first use case with recent dates and the decay function was too aggressive. The documents we want to retrieve are too old, and with a decay that fast, where we go from 1 to 0.2 of score in a day, a document even a year old would get a function score of 0. Keeping in mind that boost mode that we chose, multiplying the function score by the query score, we would get a score of 0 for the documents satisfying the first use case. Even by changing the boost mode, either the average or the sum, the results were good in some cases, but disappointing in others. It is however possible to take care of the multiplication by 0 issue. As we have seen above, the score mode is relative to the different functions and there are different methods. Given that our problem was the lower boundary of the function score, we decided to sum the score of the decay with 1. With this, if the decay gives a score of 0, it is in fact 1 when merging it with the query. By moving the function score range from [0:1] to [1:2], we no longer had score with 0 due to the date and we improved our results, but this still wasn’t good enough.

Now there are websites that choose to offer the possibility to the user to filter however they choose, if they want more matching documents or more recent ones. In our case, that wasn’t an option, so we managed to make it work for both use cases with one query.

```
{
  "query": {
    "bool": {
      "must": [
        {
          "function_score": {
            "functions": [
              {
                "exp": {
                  "DATEVIDEO": {
                    "scale": "5d",
                    "decay": 0.2
                  }
                }
              }
            ],
            "query": {
              "multi_match": {
                "query": "Emmanuel Macron",
                "type": "cross_fields",
                "fields": [
                  "TITLEPAGE",
                  "DESCVIDEO"
                ],
                "minimum_should_match": "3<75%"
              }
            },
            "score_mode": "multiply",
            "boost_mode": "multiply"
          }
        }
      ],
      "should": [
        {
          "function_score": {
            "functions": [
              {
                "exp": {
                  "DATEVIDEO": {
                    "scale": "5d",
                    "decay": 0.2
                  }
                }
              }
            ],
            "query": {
              "match_phrase": {
                "TITLEPAGE": {
                  "query": "Emmanuel Macron"
                }
              }
            },
            "boost": 2,
            "score_mode": "multiply",
            "boost_mode": "multiply"
          }
        }
      ]
    }
  }
}
```

Figure 20, fourth iteration Webvideofinder's query

```
"should": [
  {
    "function_score": {
      "functions": [
        {
          "exp": {
            "DATEVIDEO": {
              "scale": "5d",
              "decay": 0.2
            }
          }
        }
      ],
      "query": {
        "match_phrase": {
          "TITLEPAGE": {
            "query": "Emmanuel Macron"
          }
        }
      },
      "boost": 2,
      "score_mode": "multiply",
      "boost_mode": "multiply"
    }
  }
]
```

Figure 21, fourth iteration Webvideofinder's query

This is the final iteration of the query. Let's start with the "should". It is pretty similar to what we have seen so far, with 2 differences. First, we decided to go easier on the decay function. We raised the scale to 5 days. So instead of taking a day to get a function score of 0.2, it takes 5 days. Second, we add a boost of 2 that will be used during the multiplication of the score of the query by the function score. The "match" clause is a bit different from what we have seen so far. It is possible with Elasticsearch to query multiple fields at once with the same query. In this case, we add to the query the possibility to look in the description of the documents. The type of the query enables us to choose how the score should be computed between the different fields. We can either choose the best matching field for the search or do a "match_phrase" search type and the best score will be taken. Here, we use the "cross_fields", the search will be done on all the fields in a way that, a term can be in the title and the other can be in the description. With this query, although it is far from perfect, we managed to get good results for most use cases.

Trying to fit every use cases in one query is tedious. Even the smallest change of a parameter and it may produce disastrous side effects. An alternative to this can be to make several queries for a search, each one being less strict than the previous. For instance, for one specific client, we decided to split a search in 4 queries. When typing a search on the front-end application, we send a first query, a multi-match with an exact match type query on just a few fields. Then if we don't have enough documents returned (10 in our case), then we send a more lenient query, a multi-match with a cross fields type query with more fields. If we still haven't matched 10 documents minimum with the first 2 queries, then we have a third and a fourth one, with less restrictive matching.

This kind of workflow has 2 main advantages. First, it allows for an easier tuning of the relevance, each query can remain small and manageable. The side effects are limited to one query. The second aspect is the ordering. In Elasticsearch, it is possible to sort the results of a query however we want. We can choose to sort on the title of a document, on the date, anything. By default, the sort is done on the score of the documents. In case several documents have the same score, it is also feasible to add a sort on the date for instance. If only one query is done with many parameters such as the ones we have seen above, the sort on the score or anything else might not be enough. In some instance, we may want to get the results of an exact match query above the rest, but due to other parameters, we might get another ordering. If we have multiple queries, since the list of results will be separated, we can decide to present the list given by the exact match first, even though the score is lesser than the list done by the following query.

However of course, this has to be done carefully, in compliance to the usage of the index, the number of query, the size of documents and the number of documents expected. In the case presented here, it is fitting, the number of query is quite low, and the size of the documents is merely a few kilo octets. Furthermore, the minimum number of document to return is 10, so if the first query doesn't return 10 or more documents, we do the second one, and so on. In our case it is good because we want to somewhat limit the number of documents and pages. However if we wanted to remove this limit, we could get way too many documents. Additionally, this workflow requires doing a few back and forth with the cluster if we don't get enough documents, which may be unwanted depending on the infrastructure. Overall, this way of proceeding can be very beneficial but it also really depends on many criteria.

3.2.3 Study of relevance

We have discussed so far the tuning of parameters to achieve a good relevance, but as mentioned, it is a tricky problem. The reason for this is because relevance is very relative to the person doing it or judging it. One person may be more focused on getting documents recent while the other one may want more accurate but less recent results. In information retrieval, there are many measures to compute the relevance of a query, we are going to see the 2 most famous. The first one is the recall and the second one is the precision.

Let's suppose that our set of documents is D . When a user runs a query, R is the number of documents that are relevant to that query while the set of documents returned is A . We can already draw some comments. First, in a real life dataset, R is not something we know. For one

query, a user may expect a certain set of documents while another user could want something else. So not only is it something hard to determine, but it is also specific to a person. Second, in an ideal world, we would like something as $A = R$, that is we only return the documents perfectly fitting the query.

The recall is measured with the following computation:

$$\text{Recall} = \frac{|R \cap A|}{|R|}$$

We compute the number of relevant documents in the returned set divided by the overall number of relevant documents in the dataset. This gives us a ratio for the number of relevant documents correctly classified.

The precision is measured with the following computation:

$$\text{Precision} = \frac{|R \cap A|}{|A|}$$

We compute the number of relevant documents in the returned set divided by the set of documents returned. This is could be equivalent to the correctness of the result, we measure the ratio of correct documents returned.

Now that we have the 2 measures, we want to maximize them as much as possible.

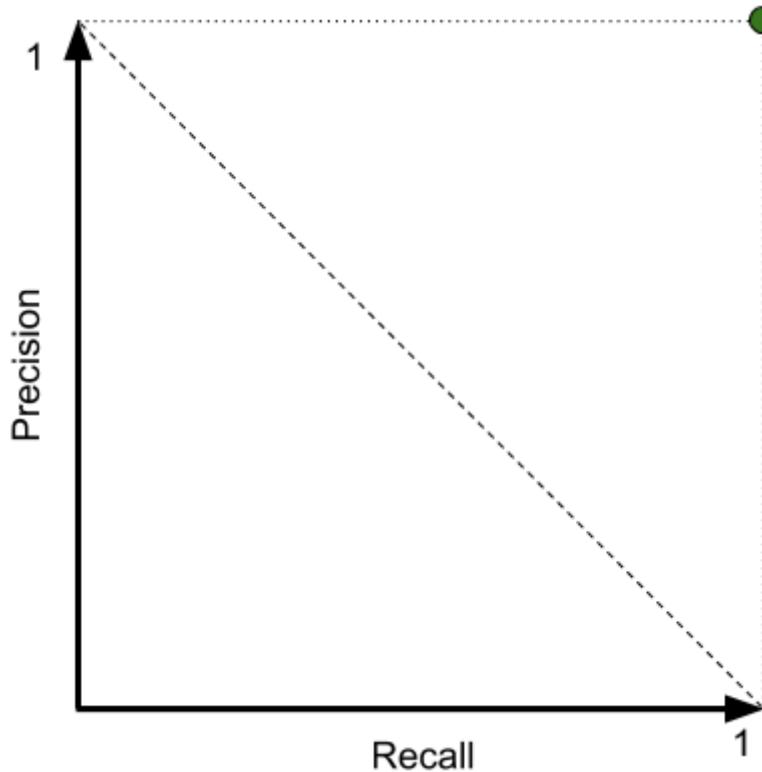


Figure 22, precision vs recall measures

If we look at the figure above, our objective would be to match the green dot, where the precision is 1 and the recall is 1 as well. On one side, the precision equal to 1 would mean that all the documents retrieved for a given query are relevant. On the other side, the recall equal to 1 would mean that all the relevant documents for a query have been retrieved. However in a real world dataset, it can be really difficult, even impossible to achieve that. Indeed, the recall and the precision are not independent, there's a tradeoff between the two. For instance, if we wanted to get a recall of 1 absolutely, we could return all the documents in the dataset, in which case the relevant documents would be returned, along with all the non-relevant. Usually, one must be

prioritized over the other. In the case of a search engine, returning all the documents of the set for each query would be a terrible idea. Precision is usually the go-to for search engines, when a user does a query, he expects to have the documents relevant, and not to be polluted by other results. By sending more documents, we would increase the recall at the expense of the precision. However in other cases, we might want a better recall.

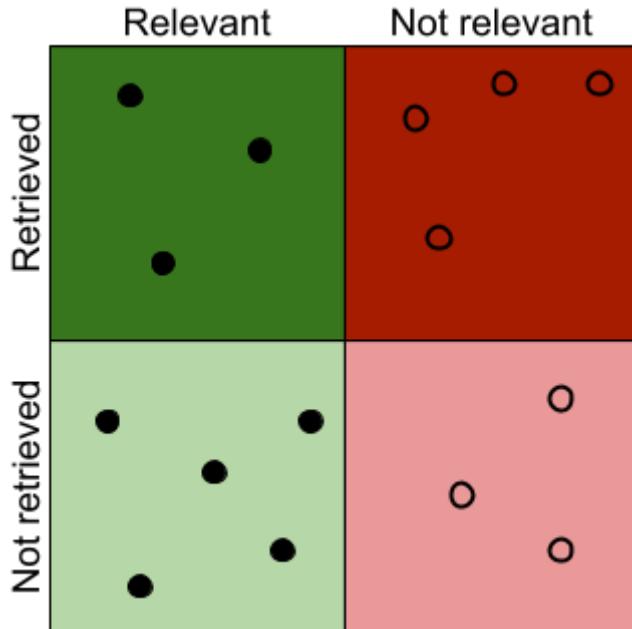


Figure 23, matrix of relevant documents vs retrieved documents

If we look at the figure above, it will help us understand when to prefer the recall. As a reminder, the recall is measured by dividing the green top left square by the 2 left squares. An example in which one should favor the recall is in healthcare. Let's say we are trying to diagnosis a disease on some people. In our sample, we have 15 persons, 7 of which are not affected. On the 8 remaining, our test found that only 3 people had the disease. The precision and the recall are:

$$\text{Recall} = \frac{3}{8} = 0.375$$

$$\text{Precision} = \frac{3}{7} = 0.429$$

So here we have a better precision, which could be great in some cases, but here the recall has to be higher. In our example, the precision predicted that 4 people had the disease when in fact there are safe. However, the much bigger issue is the people that have the disease but have not been diagnosed. The goal if possible would be to reduce the not retrieved cases to get a recall of 1, even if it means having a low precision. There are other measures that one can use to work on relevance, such as the F1 score that tries to blend the precision and the recall, but we won't talk about it here.

For a search engine, computing a good relevance even with those measures is no easy feat. It is a long work and has requirements. The dataset needs to be mastered, one needs to know what is "the truth" for the query tested, what is expected to be retrieved. For that end, the dataset must also be finite, it shouldn't be updated during the work. Even then, it will take a long time to get a good relevance.

Finally, another way to improve the relevance is that ask a feedback from the user. For each query, we ask the user to rate the documents returned, whether they are relevant or not. The next time the query is ran, we show the documents that have the highest rating first and we ask the user to rate as well. This way we can get a mean of the documents we should return.

However in this case, we are strongly dependent to whether the user takes the time to make a rating. If asking for a feedback isn't an option, it is also possible to measure the number of time a document has been seen, the user's history, the time spent looking at an item, etc.

3.2.4 Proxy-query

Earlier in this document, we talked about a tool called Proxy-query. It has 2 uses, one for querying and the other for post-processing.

In a real world environment, we have a choice to make when we want to query the search engine: create the query in the front-end application or on the server side. Creating this processing on the front-end may not be the best option because it will then take more resources on the user's part. On the server side, Elasticsearch has a way to create query templates. These templates are stored in the cluster and they are linked to one or many indices. When a query arrives, the template reconstruct the query from the parameters sent and then run it by the corresponding index. However these templates were not fully furnished in the first versions of Elasticsearch and the project Proxy-query was created. It works the same way by creating a template for the queries, parameter by parameter. We associate a customizable parameter to a query part and we interlock the different parts in a bigger query with the other parameters. This may sound a bit tricky, so let's have a look at an example. If we suppose we are searching on the Broadcastmedia2 index, the url will look something like this: "<http://my-ip/?servicecode=VOD>". The parameters sent in the url will be matched by the Proxy-query template, which looks like this.

```
{
  "enable": "servicecode",
  "replace": [
    {
      "content": [
        {
          "query": {
            "match": {
              "onDemandEvent.serviceCode": {
                "query": "__raw_servicecode",
                "analyzer": "broadcastmedia_pattern_uppercase_block_analyzer"
              }
            }
          }
        ],
        "pattern": "__dynamic_filter_servicecode_ondemandevent__"
      }
    ],
    "type": "string"
  }
}
```

Figure 24, dynamic query for servicecode field of Broadcastmedia2 with Proxy-query

Several things will interest us in here. The parameter "servicecode" is allowed in the url thanks to the "enable" in Proxy-query. It is possible to enable as many parameters as we want, and for each one we can create a different query. Below the "enable" parameter, we find "replace" which is composed of 2 keys, "content" and "pattern". The pattern is similar to a variable in Python for instance, we will put the JSON query that is constructed in "content" in it. And speaking of "content", it is there that we have the query part that we associate with the parameter "servicecode", in which the value sent in the url will replace the "__raw_servicecode" variable. In this case we do a simple match query on the field "serviceCode" and we use a specific analyzer for the query.

Once we have the query in the pattern, it is compiled with the eventual other parameters in the final query.

```
{
  "conditional": ["onlyevent=onDemandEvent||!onlyevent"],
  "content": [
    {
      "nested": {
        "path": "onDemandEvent",
        "filter": {
          "bool": {
            "must": [
              "_dynamic_filter_servicecode_ondemandevent_",
              "_dynamic_filter_dates_ondemandevent_",
              "_dynamic_filter_channellanguage_ondemandevent_",
              "_dynamic_filter_channelname_ondemandevent_"
            ]
          }
        }
      }
    }
  ]
}
```

Figure 25, dynamic query of Broadcastmedia2 with Proxy-query

In the final query, the pattern will be replaced by the query and then sent to Elasticsearch. In this case we have the pattern for the “servicecode” but we have other possibilities. Had we sent a parameter specifying a channel name, it would get incorporated in the query but since we didn’t, the empty patterns will be removed prior to the Elasticsearch interrogation. Otherwise, the query constructed is quite simple, with a “must” clause like we have seen before and the “nested” parameter is to get in the event. Indeed in Elasticsearch, nested object are considered separate entities and thus to be able to query on them, we have specify where to look for the fields of the object in the main document. Finally, another very useful feature of Proxy-query, something that is not present in ES templates, is the possibility to add a condition. The conditional here is telling us that to execute this query, we need to either send a parameter “onlyevent=onDemandEvent” in the url or to not send it. In the case where “onlyevent” is present but with another value, this query will not be executed.

As mentioned, the second use of Proxy-query is the post-processing. After the query is done on Elasticsearch and that we have the documents ordered, we may want to add some post-treatment. This is case for indices with nested objects such as Broadcastmedia2 and OneReco. Indeed, as we presented before, a movie document from Broadcastmedia2 can have a big number of events. But maybe all those events do not interest us. Suppose a movie has 1 event concerning the Video On Demand service and 9 events on replay and we are interested only in the VOD event. When we send the query in Elasticsearch, we set the “servicecode=VOD”, and the way the cluster works, since even 1 event in 10 matches, the whole document will be retrieved. However we do not want to pollute the front-end application with 90% on events useless for the user. Unfortunately, Elasticsearch is not able to filter nested objects dynamically, and this is the second purpose of Proxy-query. The post-filters look like this.

```
{
  "conditional": [ "servicecode" ],
  "foreach": "onDemandEvent",
  "find": "serviceCode",
  "cgi": "servicecode"
}
```

Figure 26, additional filter of Proxy-query

The “conditional” looks at whether the parameter was used during the query, and the “cgi” will take its value. Once we have the value chosen by the user, we get in each event, and “foreach” event we have to “find” have match the field. In the case where the “cgi” and the value

of the field of the nested object don't match, then the event is removed, reducing the pollution and the size of the document before sending it to the user.

3.3 Data analysis

Now that we have talked about Elasticsearch and its query possibilities, we are going to have a look at another tool of the ELK stack, Kibana. Kibana is the data visualization tool built by the same company and it is plugged directly in Elasticsearch. With lots of feature built within, Kibana is a powerful tool, enabling to do simple dashboards with histograms to geographical aggregations on a map. In addition to being able to visualize data from the cluster, it is also possible to visualize logs, which can be really useful. The kind of logs that can be used depends on the need, from which query had 0 result or which documents are the most requested. This kind of data is a good way to improve the relevance or the monitor the state of the cluster.

In this part, we will focus specifically on studying data from the index Broadcastmedia2 and its clients, the service Polaris Video. And first we are going to have a look at the logs of a full day.

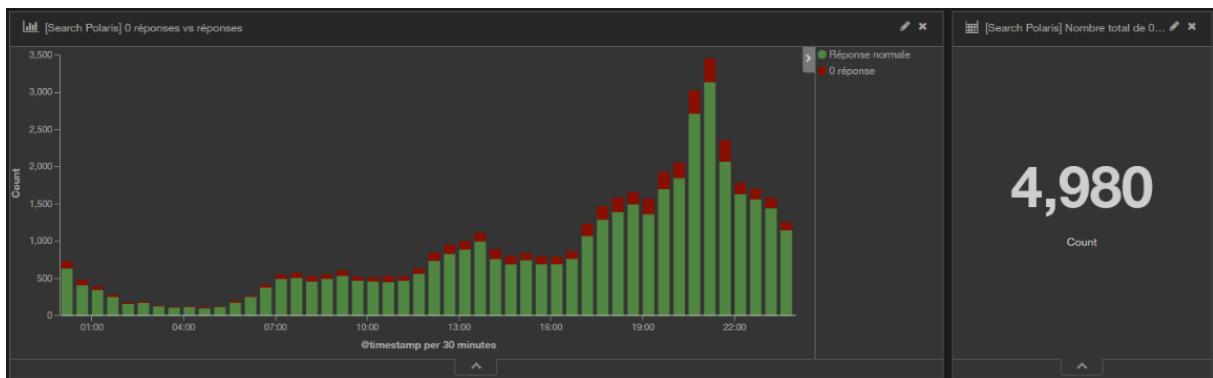


Figure 27, number of queries in 24 hours for Polaris Video

In the dashboard above, we see 2 things. First, the histogram is a count aggregation on the number of queries that were run in last 24 hours, separated in 2 types, in green the ones that yield results and in red the ones that matched 0 documents. Overall, there were 5000 queries with 0 matches but it seems like a small percentage of the total number of queries. In addition, the distribution of the graph gives us a pretty good idea of what is going on. On the left, we see that the request number is quite low, which seems to be normal since it's night time. On the right, we reach the peak numbers, which can be explained by the fact that it is during the prime time, during the evening. From this, a simple conclusion is that people tend to watch more content during the evening than during the rest of the day. A further inquiry in the 0 result queries could be interesting.

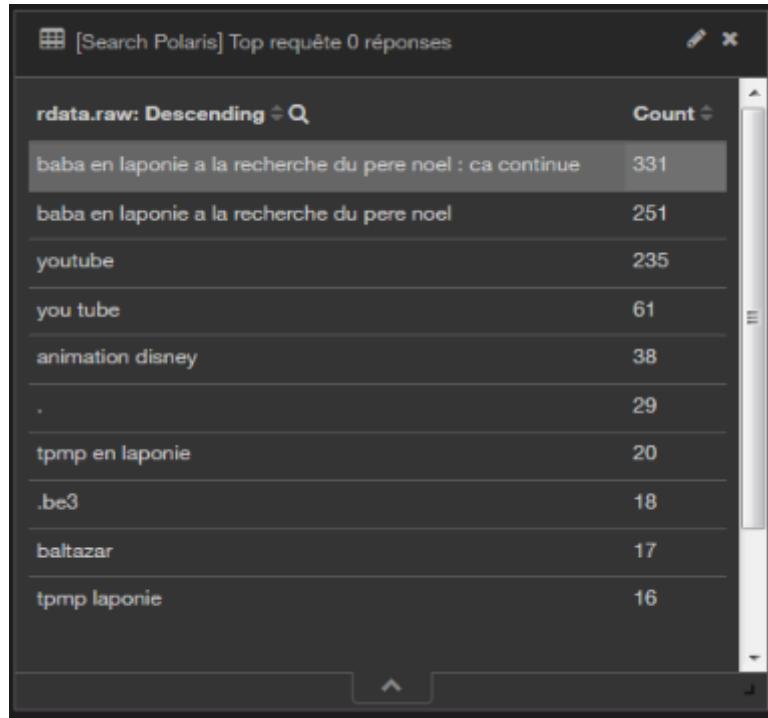


Figure 28, most searched documents for Polaris Video with 0 results

Right above are those queries, ordered by the number of times they were run. The first 2 appear to be quite similar and ran a lot. People were interested in this content, however it seems that the cluster didn't retrieve it, either because of a relevance problem or more likely due to the simple fact that this content wasn't indexed. As stated before, we get content from partners and sometimes broadcasts get on the replay service few days after the first diffusion, which could explain why we didn't match anything. The second 2 results are for the famous video-sharing website Youtube. As it happens, some people may try to get on the platform from their TV. Should the number of request for this service continue to grow, it might be interesting for the user to offer a link towards it.

Now that we know a bit more about the queries left unanswered, we are going to look at the more requested movies and broadcasts.

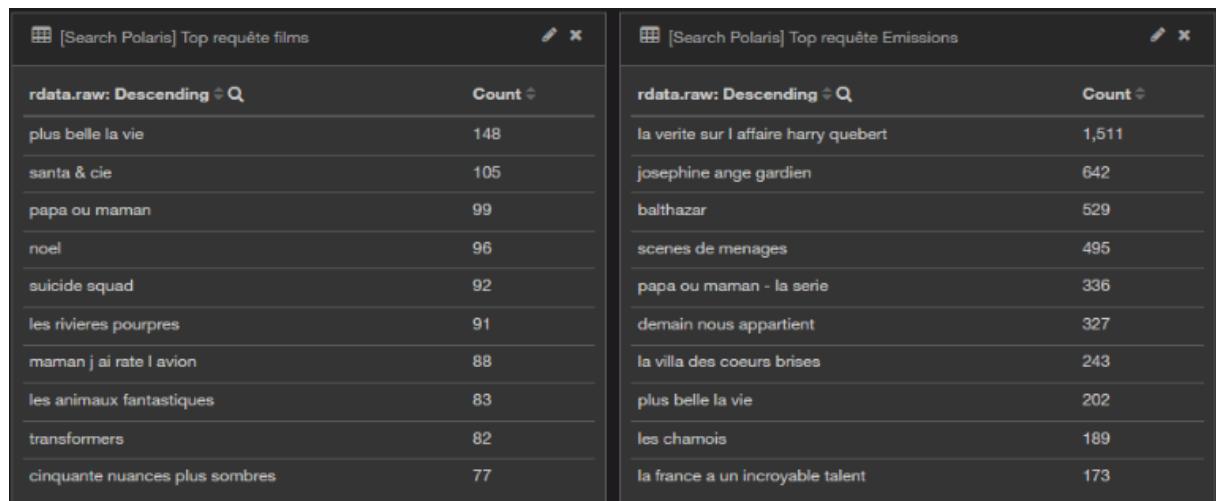


Figure 29, most requested documents for Polaris Video

On the left we have the top requested movies and on the right the broadcasts. A strange thing to note is that the most asked query for movies is “Plus belle la vie” which is in fact a TV show. This could be due to several things, either the document is wrongly classified in the cluster or the semantic analysis prior to sending the query to Elasticsearch is wrong. In the first case, we can always change the classifier in the search engine, however in the second case, it would have to be transferred to the team handling the semantic analysis. For the most requested broadcast, we can see it is quite clearly above the rest, it could be interesting to put the broadcast in the front page of the TV, this way it would be easier to users to access the content.

It is also possible to present data in a pie chart.

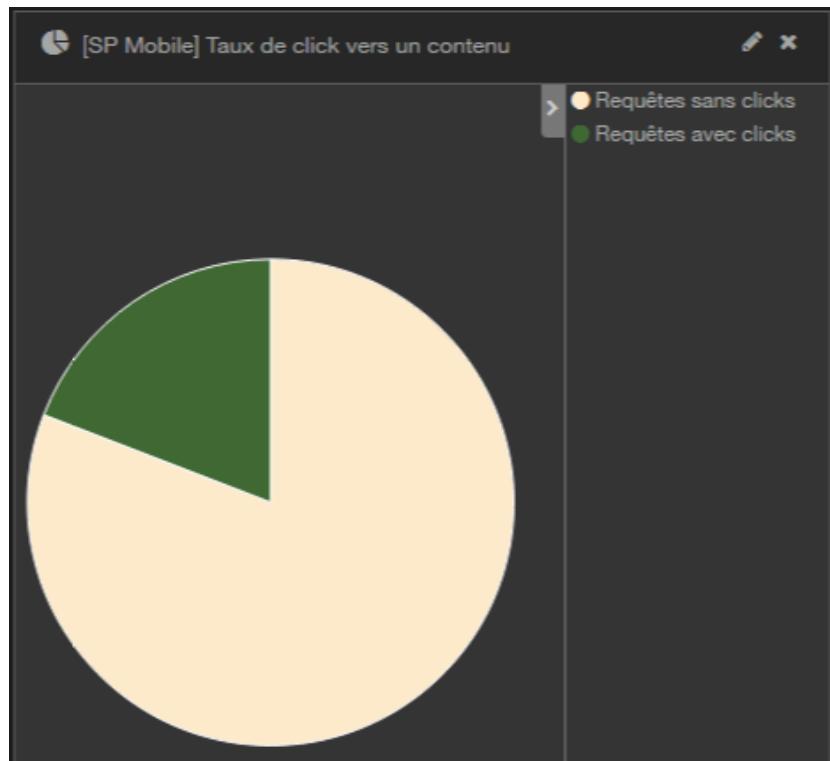


Figure 30, pie chart for requests followed by a click from the user for Polaris Video

Above is a graph representing the proportion of the queries leading to a click from the user and those who aren't for the search on smartphones. In white are the requests with clicks and in green are the queries with clicks. We see that the queries without clicks in the last day on mobile represent more than three quarters of the total queries. Having this kind of information is useful because we can draw 2 conclusions, either users are just looking to see if the content is in the cluster or the results presented are not satisfactory, in which case a makeover of the relevance could be in order.

If we stay in data from the past 24 hours, it is interesting to have a look at the efficiency of the search completion.

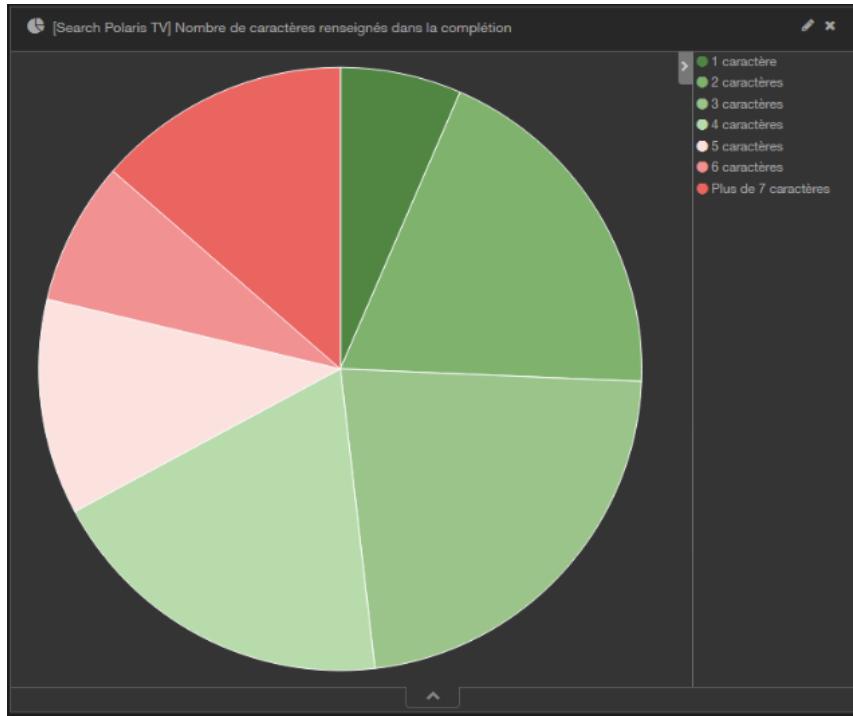


Figure 31, pie chart for number of characters in completion of Polaris Video

This pie chart shows the number of characters that had to put in the search bar before the users could find what they were looking for in the search completion. The greener, the less characters were filled and the redder the most characters. We see that for two thirds of the pie, the users had to type a maximum of 4 characters which is good. The more concerning thing is those who had to type in more than 7 characters, which appears to be a sizable proportion. This is a problem because users will not bother typing long title and if we can't detect what they want fast, they will get bored and they will stop looking. To avoid that, we must improve the suggestion system.

A cross-matching can be done between the few visualization that we have seen so far. Indeed if we cross-check previous graphs, if we find the queries that had more than 7 characters in the completion system to the top requested documents, we could draw interesting conclusions.

| [Search Polaris TV] Recherches cliquées avec plus de 7 caractères | |
|---|----|
| le pere | 16 |
| youtube | 15 |
| mystere | 12 |
| les feu | 11 |
| michael jackson | 10 |
| affaire | 9 |
| demain n | 9 |
| game of | 8 |

Export: [Raw](#) [Formatted](#)

1 2 3 4 5 ...117

Figure 32, documents clicked with more than 7 characters in completion for Polaris Video

This table contains the count of the queries that were finished by the completion after 7 characters were typed in. The penultimate result could be of great value. “Demain n” is a broadcast that was in the top 6 most requested broadcasts. Having it in the table with more than 7 characters for the completion could be bad for one of the broadcast the most asked for. The solutions could be to either give this document an additional weight in the completion or to put a direct link on the TV’s home page instead of having to search for it.

Another use for monitoring the queries is for anomaly detection.

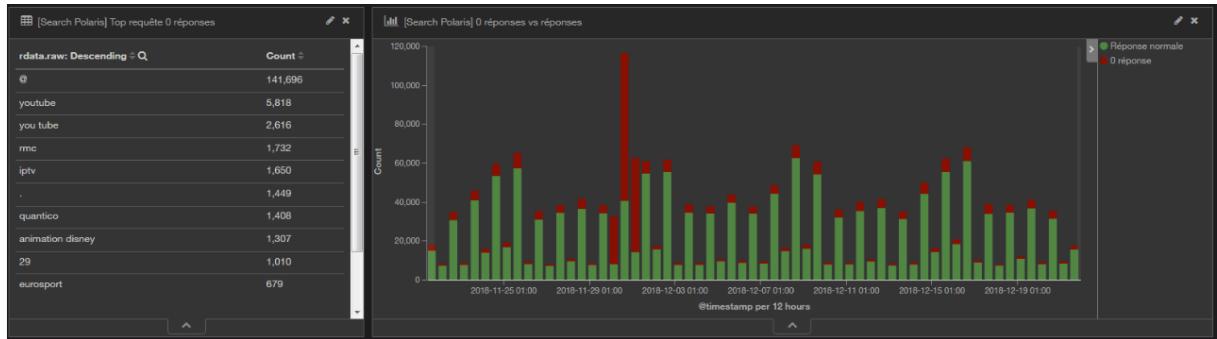


Figure 33, number of queries in a month for Polaris Video

This is the same histogram as before but instead of looking at data from the last 24 hours, we have data from the past month. Each day is divided in roughly 2 bars, and for most of it the same analysis can be done as earlier, queries tend to be more numerous during the second part of the day. The thing that is going to interest us however is the huge spike of 0 result requests. On the left table, we see that the most queried thing is simply the character “@” with over 140.000 queries, and for the most part it appears to be within a day and a half. Now this is even stranger, given the usual volume of query we receive. We see that for the whole month, the second query is for “youtube”, being asked barely 4% as the first one. Kibana allows us to zoom in on the date, which we are going to do to review this closer.

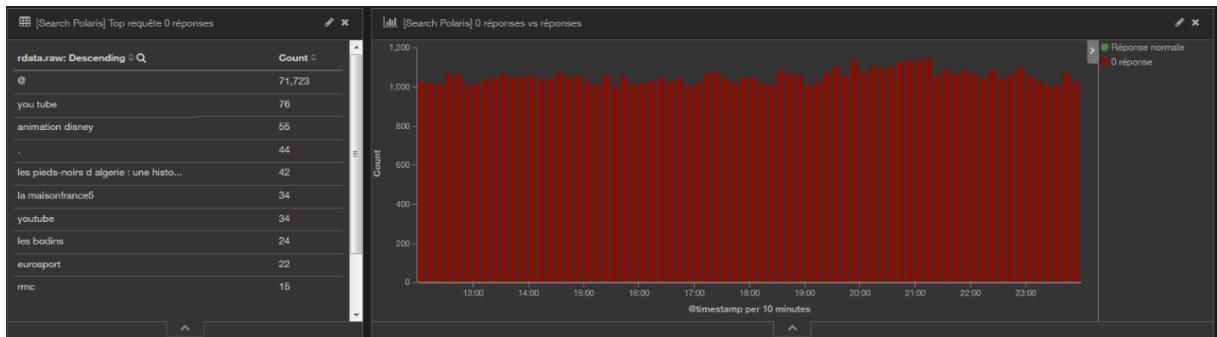


Figure 34, number of 0 response requests in 12 hours for Polaris Video

This is the aggregation for the second part of the day and in red are the 0 result queries. The proportion is so one-sided we don’t even see the queries with results, and within 12 hours, the “@” was queried over 70.000 times. Overall, the query had to be sent every 1.6 seconds on average during those 12 hours, which is unlikely to come from a human. Most likely it was coming from a machine, but we are not sure what the purpose was. Still, if this kind of behavior continues, it is possible to block queries like that on the client side, preventing it to overload unnecessarily the network.

Data visualization from logs of the cluster can give us good insight in what could be interesting to put forward in terms of ads for instance. It is also a good way to monitor the search engine health.

Chapter 4: Further research

In this final chapter, we are going to see some further research or observations done during the thesis.

4.1 Elasticsearch use cases

In the course of this document, we talked about different clients that were using Elasticsearch, but it would be interesting to see if this tool suits their need. We can ask ourselves to whom should Elasticsearch be recommended.

Starting with one of our biggest clients, Polaris Video. We talked about the kind of documents they needed to access, complex JSON objects. Most of their searches include simple key-value pairing, as an example the “serviceCode” parameter that we talked briefly before. This field can have only one value, no analysis is required, it either matches the parameter or it doesn’t. In addition to these simple pairing, they also use fields of unstructured data, such as the title of a movie. Polaris Video could use another tool if there was only queries about strict values, however since there are some textual searches, Elasticsearch is necessary and allows a deep manipulation of the scoring of documents returned.

Moving on to OneReco, as mentioned earlier the only kind of search they do through Elasticsearch is with key-value pairs. Since they don’t offer the possibility for the user to search for a content, there is no text to send to the search engine. The queries they send are created according to users’ preference, what they usually watch, what could please them. And among those information, there is nothing about unstructured or semi-structured data. The filters they use are for example on the genre of a TV show or the channel of broadcasting. While Elasticsearch is completely able to do this type of search, it loses its main added-value. One of the most valuable thing in the search engine is the mapping with the possibility to analyzer fields, but this is not done for these kind of searches. A few other potential clients of the Search On Demand team are looking into Elasticsearch, however their need usually do not fit with this tool. For key-value pairing, it is better to prefer more traditional database, either relational or non-relational. Studies are actually ongoing in Orange to find what would be the best solution for OneReco, in hopes of replacing Elasticsearch.

Now, if we take a step back from what we have seen for Webvideofinder, we can already guess if Elasticsearch is adapted. The documents of this index is mostly semi-structured, meaning that even if there are mostly textual fields, those fields complete each other, we have a title and a description for example. The queries of Webvideofinder that we presented are using a lot of functionalities of the search engine, from using analyzers to computing the score from different parameters as we would like. This is the kind of work that would not be possible without a deep analysis of the fields, and this is precisely what Elasticsearch can do.

4.2 Web search systems

In the beginning of this thesis, we briefly talked about Information Retrieval and the fact that it could be divided in 3 categories, and while mostly talked about the *enterprise search*, we are now going to talk a bit about *web search* based systems. These systems have different needs and ways of gathering data. Indeed, since it is a scale above the enterprise search, the numbers of documents needing to be indexed are in the billions. Furthermore, anyone can at any time create a new website, the addition of content is not something that can be controlled. Documents

usually do not go through an ETL tool, the indexation is done through the means of a crawler. This crawler takes a few websites at the beginning and then, with the exploitation of hyperlinks, it crawls through the web. This gives search engine a good idea of the map of the Internet.

As we said in the presentation of Elasticsearch, it is not possible to do any kind of crawling, so it cannot go from documents to documents using links. This is significant for the relevance of documents. We described how Elasticsearch relevance is constructed and we saw some ways to improve it, using for instance users' feedback.

We briefly talked about Google previously in this document and from what we now know, we understand that it is a *web search* system. We may wonder how is it that Google, above all else has risen to be the most popular search engine. At the beginning of the century, Google tried to bring something more on the relevance: the importance of connection and popularity of websites. This was done by the simply adding 2 things in the equation. How many websites a page refers to and how many websites refer to a page. We can quickly look at the figure below.

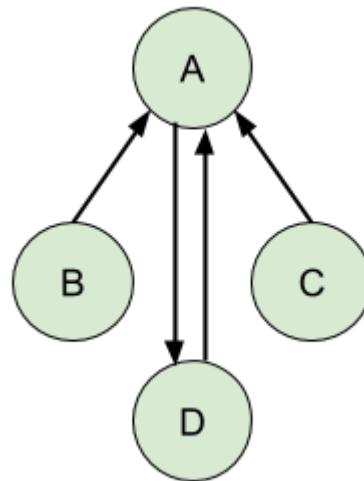


Figure 35, connection between pages

Without going in further computation, we can take time to understand this. Let's suppose each node is a webpage and arrows are links towards a page. Every node here points to one page. Page A has 3 pages that refers to it, page B and C don't have any pages that refer to them and page D has one page that points to it. According to the PageRank algorithm, the more a page is referred to, the more weight it gains. If a popular page points to another page, the weight of this link is bigger than the weight of links coming from smaller node. This mapping allows to find pages that are widely popular, then when a user asks a query, the weight of the pages are taken into account, pages with higher weight will be returned in higher position.

It is possible to find more information about the PageRank in the paper of Sergey Brin and Lawrence Page [3].

4.3 Elasticsearch and Solr

All throughout this thesis we talked about the search engine Elasticsearch but there are many other. As mentioned early, Elasticsearch is based on the Lucene library developed by Apache and before Elasticsearch was created, Apache had its own search engine, Solr. Now those 2 seem to be the most popular and are based on the same library. While we will not do a comparison point by point, it might be interesting to see at the main differences between them. The book *Relevant Search* [4], summarized by the book's authors¹⁰ offers good insights on the

¹⁰ <https://opensourceconnections.com/blog/2015/12/15/solr-vs-elasticsearch-relevance-part-one/>

differences of the 2 tools and it is possible to find more information on the adequate tool for different use cases on many websites¹¹.

An important distinction between the 2 search engines is the use of analyzers. In Elasticsearch, it is much easier to custom analyzers, and even when to use them. We talked about the advantage of using analyzers at indexation time or query time only, with the possibility to add synonyms to search terms or to offer completion. However it is not that simple for Solr. First it divides the text search on whitespace, meaning that it is not possible to add multi-words synonyms to a search for instance. Indeed, when the search "not happy" comes up, Solr creates 2 tokens prior to try to analyze it with synonyms, so it is not possible to associate the token "unhappy" without the need of additional plugins¹². The second point is that in Elasticsearch, as we saw previously, it is possible to specify different analyzers on a field, and then we can use them as we wish. Solr doesn't let us set several analyzers on a field, if we want an analyzer that removes stopwords and another that adds synonyms, we need to duplicate the field.

As stated before, Elasticsearch's query language is JSON, just like its documents' structure. While this makes that it is quite easy to construct and read queries, factorization is hardly possible, which makes it that code has to be repeated sometimes. Without going too much into specifics, Solr queries are more compact. Let's compare an Elasticsearch query to a Solr query.

```
{  
  "query":{  
    "bool":{  
      "must":{  
        "match":{  
          "name": "John Watson",  
          "operator": "and"  
        }  
      }  
    }  
  }  
}
```

Figure 376, Elasticsearch simple query

```
{!lucene q.op=AND}name:(John Watson)
```

Figure 367, Solr simple query

On the left, we have Elasticsearch's query, which is easily comprehensible. We are trying to match a name and we want both terms to be found. On the right, we have Solr's query which looks a bit denser. Elasticsearch's query, as we have seen, will always be this structured and more explicit and there will never be local or global parameters, only the current JSON object. Solr, while enabling to compute score between different parameters of the query, will be more complex to decipher and to create, but it will open a world of local and global parameters, making it really possible to manipulate queries efficiently. The main advantage of Elasticsearch is that it is easy to take in hand and it is difficult to have surprises. Solr, while more complicated at first, offers a deeper tuning of the relevance score.

A few things to note between the two solutions. Elasticsearch is quite autonomous, it is easy to start a cluster and it will be able to handle itself pretty nice, allocating shards on its own, creating a default mapping and such. On the other hand, Solr requires more attention to begin with, shard are not dynamically allowed, and the cluster needs more assistance to expand to other nodes¹³.

¹¹ <https://www.searchtechnologies.com/blog/solr-elasticsearch-cognitive-search>

¹² <https://opensourceconnections.com/blog/2013/10/27/why-is-multi-term-synonyms-so-hard-in-solr/>

¹³ <https://opensourceconnections.com/blog/2016/01/22/solr-vs-elasticsearch-relevance-part-two/>

A good thing from Solr is that it is possible for anyone to add contribution to the search engine, usually in the form of plugins. Elasticsearch however, while the source code is available can only be updated by the company Elastic, which can be the cause of problems if the direction undertaken doesn't suit the need of users.

Conclusion

As a conclusion of this thesis, a lot of tasks have been carried out to understand the work necessary to set a data pipeline and the information retrieval difficulties. At the beginning of the thesis, the search engine Elasticsearch at the center of many searches of Orange lacked optimization from getting data to retrieving it.

The Extract-Transform-Load hand-coded tool was not made to handle such complex and high volumes of data, making it so that the pipeline would fail and indexation would stop at random. It was decided that trying and fix this tool would be more expensive than starting from scratch. Unfortunately, due to different constraint of life in a company, we weren't able to perform a deep benchmark of different solutions for ETL tools, and instead of trying to find a tool that could fit our need, we decided to rewrite it.

This rewriting, while time and resources consuming, allowed us to regain grasp of what is going on in the pipeline, which is essential to be able to maintain it efficiently. In doing so, researches have also been made to optimize Elasticsearch cluster itself and the way data are stored. Based on Apache's Lucene, Elasticsearch offers many solutions to control the flow of data and where it should go. As we saw, the tuning of numbers of shards and replicas could play a lot in the indexation time and the querying time. These are many more concerns that need to be addressed when creating an Elasticsearch cluster.

In addition to this, we have seen that Elasticsearch's stack could be very useful. Monitoring data from the cluster and the logs is very handy, and allows users to really be masters of their data. Meaningful insights can be drawn from logs of a search engine to ease the life of a user, such as suggest documents if they are asked a lot or adding synonyms to common words. The visualization tool is a good way to monitor the cluster's health, to know if there are some points to be concerned by.

One of the most challenging part of course is the tuning of every parameters for the relevance of a search engine. We have seen how is computed the basic score and how different parameters come into play, yet managing a good relevance is no easy fit. The first reason is because even if it is clear what each parameter does, by adding more parameters, it gets more and more difficult to keep track of everything. The second reason is that the set of documents is not mastered. Indeed, training a good relevance usually takes a finite set where the *truth* is known and where it is possible to fine tune every parameter to match this *truth*. In real life however, data keeps changing, which makes it ever more difficult to find a good ranking system. The third reason and perhaps the most important is because relevance is quite dependent on each user. There is in fact no one *truth* in a relevance on a dataset. A user might expect a certain ranking for a search, but it might differ from what expects another user. In such cases, users might be of great value to improve relevance. It is possible through diverse means to get feedbacks and to incorporate them in the scoring of the relevance.

Bibliography

- [1] Manning C., Raghavan P., Schütze H, 2008, <https://nlp.stanford.edu/IR-book/>, Cambridge University Press
- [2] Turnbull D., Berryman J., 2016, *Relevant Search*, Manning, pp.67-69.
- [3] Brin S., Page L., 1998, <http://infolab.stanford.edu/pub/papers/google.pdf>
- [4] Turnbull D., Berryman J., 2016, *Relevant Search*, Manning