



Advanced Databases

Search Engines and Elasticsearch

Submitted to:

Prof. Esteban Zimanyi
INFO-H-415 Advanced Databases
(2018-2019)

Submitted by:

Ioannis Prapas (000473813)
Sokratis Papadopoulos (000476296)

December 2018

CONTENTS

1	Introduction	4
1.1	What are search engines?	4
1.2	How search engines differ from relational databases?	5
1.3	Other search engine DBs	7
1.3.1	Splunk	7
1.3.2	Solr	8
1.4	Rest of the report	9
2	The Elastic Stack	10
2.1	Elasticsearch	11
2.1.1	Key concepts	13
2.1.2	CRUD operations	15
2.1.3	Querying plan	16
2.2	Beats	17
2.3	Logstash	18
2.4	Kibana	19
2.5	Industry use cases	24
3	Practical Example	25
3.1	Installation Guide	25
3.1.1	Installing Elasticsearch	25
3.1.2	Installing Kibana	26
3.1.3	Installing Logstash	26
3.1.4	Installing Filebeat	27
3.2	Importing the data	28
3.3	Building Visualizations in Kibana	31
3.3.1	Accidents map	31
3.3.2	Other Visualizations	32
3.4	Querying Elasticsearch	32
3.4.1	URI Search	33
3.4.2	Request Body Search	35
4	Conclusion	36

Figure 1: DB-Engines ranking of search engine DBMS according to their popularity	4
Figure 2 Splunk use case for healthcare	8
Figure 3 The elastic stack architecture	10
Figure 4 Sample documents and resulting inverted index	11
Figure 5 Example of a json document.....	12
Figure 7 Elasticsearch core elements	14
Figure 6 How Elasticsearch works.....	17
Figure 8 The family of Beats data shippers.....	17
Figure 9 Logstash pipeline structure.....	18
Figure 10 Kibana Discover interface	19
Figure 11 Kibana Visualize interface, simple bar chart.....	20
Figure 12 Kibana Dashboard interface	20
Figure 13 Kibana Timelion interface for time series visualization.....	21
Figure 14 Kibana Canvas interface for web traffic.....	21
Figure 15 Kibana Canvas interface for e-commerce.....	21
Figure 16 Kibana Dev Tools: Console	22
Figure 17 Constructing query using Auto-complete in Kibana query bar	23
Figure 18 Adjusted visualizations upon result of the query on Kibana dashboard	23
Figure 19 Adjusted documents list upon result of the query on Kibana discover	24

1 INTRODUCTION

With “google” being an official verb in English dictionary and “googling” occurring more than 3.5 billion times per day, search engines’ importance hits the top of our online world. In this project we are looking into search engine databases and dive deeper into the most popular of them (Figure 1), Elasticsearch. It is worth mentioning that Elasticsearch is now the 8th most popular database management system across all different database models and being on a rising path ever since its creation.

Rank			DBMS	Database Model	Score		
Dec 2018	Nov 2018	Dec 2017			Dec 2018	Nov 2018	Dec 2017
1.	1.	1.	Elasticsearch	Search engine	144.70	+1.24	+24.92
2.	2.	3.	Splunk	Search engine	82.18	+1.81	+18.39
3.	3.	2.	Solr	Search engine	61.35	+0.47	-4.95
4.	4.	4.	MarkLogic	Multi-model	14.28	+0.81	+3.13
5.	5.	5.	Sphinx	Search engine	7.82	+0.46	+1.79
6.	6.	6.	Microsoft Azure Search	Search engine	5.67	+0.36	+1.56
7.	7.	7.	Algolia	Search engine	3.62	-0.13	+0.57
8.	8.	9.	Google Search Appliance	Search engine	2.93	+0.03	+0.20
9.	9.	8.	Amazon CloudSearch	Search engine	2.64	-0.11	-0.09
10.	10.	10.	Xapian	Search engine	0.60	-0.08	-0.02
11.	11.	11.	CrateDB	Multi-model	0.44	-0.06	-0.15
12.	12.	12.	SearchBlox	Search engine	0.24	0.00	+0.01
13.	13.	14.	searchxml	Multi-model	0.09	-0.01	+0.09
14.	14.	13.	DBSight	Search engine	0.06	+0.00	+0.05
15.	15.	14.	Manticore Search	Search engine	0.04	-0.01	+0.04
16.			FinchDB	Multi-model	0.03		
17.	16.	14.	Exorbyte	Search engine	0.01	+0.01	+0.01
18.	16.	14.	Indica	Search engine	0.00	±0.00	±0.00

Figure 1: DB-Engines ranking of search engine DBMS according to their popularity¹

1.1 What are search engines?

Search engines are NoSQL database management systems dedicated to the search for data content.² Typically, search engines offer the following features:

- Full text search
- Stemming (reducing inflected words to their stem)
- Faceting, Highlighting, Fuzzy matching
- Support for complex search expressions
- Ranking and grouping of search results
- Creation of dashboards to visualize and analyze results
- Geospatial search
- Distributed search for high scalability
- Support alerting/events notification when a specific criterion is met

Search engines allow benefiting from the analysis of machine data, coming in unstructured waves of different sources (sensors, IoT, mobiles, website logs, etc.), extracting useful insights

¹ <https://db-engines.com/en/ranking/search+engine>

² <https://db-engines.com/en/article/Search+Engines>

from them. Potential applications are covering a wide spectrum; from data-driven decision making, to network security and failures management.

Full-text search engines evolved much later than traditional database engines, as corporations and governments found themselves with more and more unstructured textual data in electronic format. These new text documents didn't fit well into the old table-style strictly defined databases and querying in that format is hard, so the need for unstructured full-text searching was apparent.

Since it was developed later, search engine technology borrowed heavily from the database world, and many search engines still employ some type of traditional table structures in their underlying architecture. As traditional relational databases had dominated the spectrum of databases being so well established for many decades, many of the key RDBMS paradigms have also migrated into search engine technology, though often renamed or recast.

Search engines may use other document stores as secondary database model. Document stores, also called document-oriented database systems, are characterized by their more flexible schema and organization of data, with most popular of them being MongoDB.

1.2 How search engines differ from relational databases?

Relational Databases are neatly organized collections of data, based on a strictly defined schema storing information into specific fields that offer consistency and reliability. You can retrieve all kind of stored information by searching on specific keywords, titles, headings and more specific fields. Results are always relevant and an exact reflection of your request. If you are looking for a consistent, carefully organized way to store and retrieve information, relational databases will perform just fine.

Moving to an unstructured online world, relational databases tend to struggle to keep up with the user demands in a wide variety of cases. Let's now focus on searching and study examples.

A relational database stores data by splitting into different fields and tables. Most users would not like to select a specific field before performing a query. Indeed, databases could still try to answer user's query searching on all possible fields, joining all possible tables, however this leads to the construction of complex queries that will result in very slow performance. In another perspective, search engines come into place and let the searching of smart indexes instead of the full text. They manage to be better at the specific field of searching by relaxing the requirement for ACID transactions, in favor of fast lookup times.

As proven, relational DBs tend to work better for exact matches. Search engines facilitate searching in a way that is intuitive for users, by using smart indexing and prefix trees that help with additional tricks like fuzzy matching e.g. spelling mistakes, lowercase, ä->a or ae, prefix matches, n-gram matches, just to name a few. Search engines are NoSQL Databases featuring no relations, constraints, or any transactional behavior and this imply much easier scalability.

Let us assume that you search documents for the word “cats”. Even if you logically expect to also get results with the word “cat” (or even “Cats”) this will not be the case with relational databases, as the keyword should exactly match what is stored on the relevant field. What is more, you cannot search for “cat family” and get results for “family of cats”, or search for “cafe” and get results for “café”, “Cafe”, or “coffee”.

As you understand by now, the more we experiment with text, the more trouble will arise for relational databases to keep up with the demands. And there come search engines.

On the one hand, a relational database offers primarily consistent data storage and retrieval capabilities, the second of which can be improved by indexes. A search engine primarily focuses on indexing data to look them up fast. Databases are good for storing data neatly into tables with static format and doing ACID transactions, while search engines are good at indexing data for searching.

Considering size, relational databases perform well, when data is at the Gigabyte scale, but today it is a big data world and you have to deal with Petabytes of data. The more the data grows, the more we are shifting away from relational databases and search engines is a very good alternative for a batch of cases. But let us showcase how these two technologies differ at core under the hood.

A core difference is that in traditional relational databases we use a fully predefined schema, whereas search engines implement a more unstructured way of storing, called mapping. On the Table 1 we display how different elements (hierarchically top-to-bottom) are defined in each technology along with their equivalents.

RDBMS Terminology	Search Engines Terminology
DB Instance	Node
DB Cluster	Cluster
Database	Index
Table	Type
Schema	Mapping
Physical Partition	Shard
Logical Partition	Route
Row	Document
Column	Field
SQL	For ES: DSL (Query DSL)

Table 1: Terminology differences of RDBMs and Search engines³

³ <https://www.oreilly.com/library/view/data-lake-for/9781787281349/5e06ee29-742f-4b6d-bbad-b59bb0987901.xhtml>

Plus, some further differences between RDBMS and Search Engines on different categories.

Category	RDBMS	Search engine
Transaction Capability	Supports ACID (Atomicity, Consistency, Isolation, Durability) properties	Very less support or no support for ACID
Partitioning	Horizontal partitioning and sharding	Supports only sharding
Consistency	Immediate consistency	Eventual consistency
Keys	Support primary and foreign key	Supports only primary key
Models	Relational model	Document store

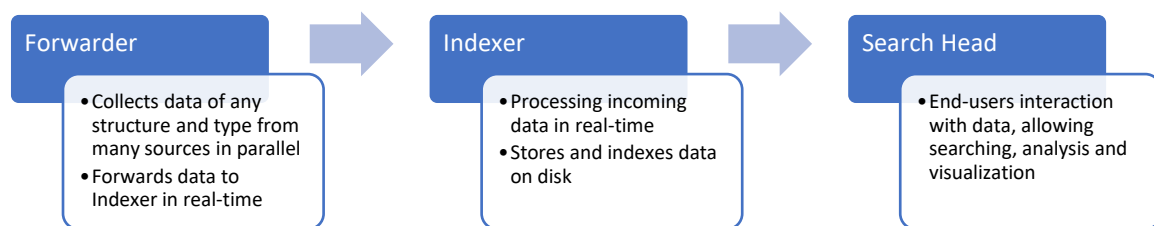
Table 2: Differences of RDBMs and Search engines⁴

1.3 Other search engine DBs

The search engine database field has been dominated by Elasticsearch from the moment it came out, because of its ease of usage and the fast, near real-time performance. However, in recent years big players of the technology arena are starting to catch up. An exhaustive list can be found in the dbengine site⁵. Let's see briefly what the most popular of them (Splunk, Solr) has to offer, showcasing the power of this technology.

1.3.1 Splunk

Splunk has been named the “Google for log files”. It was the first log analysis tool, released in 2003, identifying itself as a data collection, indexing, and visualization engine for operational intelligence, commonly used for analyzing logs and machine data. Splunk accepts any data type and structure from multiple sources in parallel and manages to analyze data and provide results in real-time, sending alerts or notifications when needed. Below you can find the general structure of how Splunk works.



Here we present a small Splunk use case for a better understanding of the tool. This is how Bosch used Splunk for data analytics.

- **Forwarder** collected the healthcare data from the remotely located patients using IoT devices (sensors), in real-time.
- **Indexer** indexed and stored the input accordingly in real-time.

⁴ <https://dzone.com/articles/search-engine-solr-vs-relational-database>

⁵ <https://db-engines.com/en/ranking/search+engine>

- **Search Head** analyzed data and any abnormal activity (based on thresholds previously set) would be reported to the doctor and patient via the patient interface.

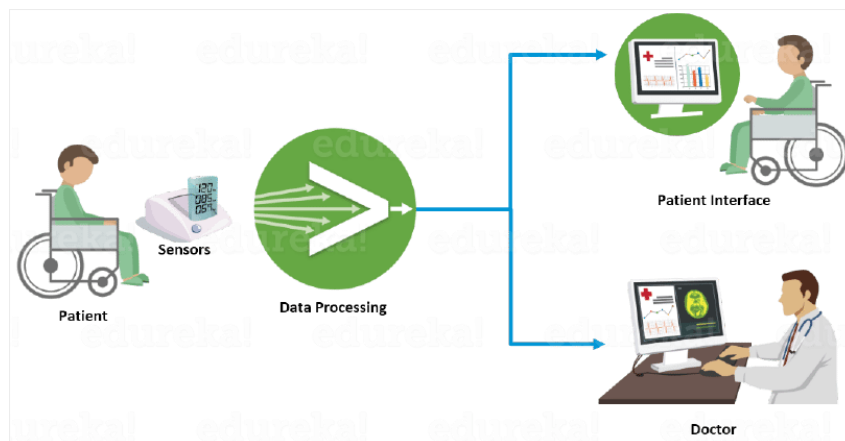


Figure 2 Splunk use case for healthcare⁶

Using Splunk Bosch could not only delve deeper into patients' health record analyzing patterns, but also guarantee a real-time monitoring of patients' health conditions, alarming both doctors and patients when patients' health degrades.

1.3.2 Solr

Apache Solr, as Elasticsearch is a Lucene-based search engine and platform for fast and scalable search⁷. Solr stands for "*Search on Lucene and Resine*", it is primarily programmed in Java and it was released on 2004. Its major features include full-text search, hit highlighting, faceted search, dynamic clustering, database integration and rich document handling. Instead of searching the text directly, Solr searches an inverted index. This is like retrieving pages in a book related to a keyword by scanning the index at the back of a book, as opposed to searching every word of every page of the book. In Solr, a Document is the unit of search and index. The schema is defined before documents are added and it is represented in a file called schema.xml. The schema declares the existing fields, the primary keys, if fields are required and how to index and search each field.

```
<fieldtype name="phonetic" stored="false" indexed="true" class="solr.TextField" >
</fieldtype>
```

Then, for each field we need to declare name, type, if it is indexed, if original value is stored and if it is multivalued. Below you can find an example⁸ of a field definition.

```
<field name="id" type="text" indexed="true" stored="true" multiValued="true"/>
```

When data is added to Solr, it goes through a series of transformations (lowercase, stemming, etc.) before being added to the index, this is called the analysis phase, which finishes with the output tokens being added to the index. So, when queries are performed, Solr searches based

⁶ <https://www.edureka.co/blog/what-is-splunk>

⁷ <https://www.edureka.co/blog/solr30thoct>

⁸ <http://www.solrtutorial.com/basic-solr-concepts.html>

on these tokens and not the original text. Only the fields that we specify as indexed are the fields which undergo the analysis phase and are finally added to the index. If a field is not indexed, it cannot be searched on, but it can be displayed in the search results if its stored variable is set to true. The reason why it is not advisable to store all fields is because the more the storing fields the more size Solr needs to store the index. And the larger the index the slower the search because of more I/O's needed to fetch results.

A typical example of Solr use case is text analytics for HR⁹. Solr can be the favorite tool of hiring managers as it can provide a much faster alternative than going through all these big piles of resumes that reached their company, in order to analyze and filter only the ones they are interested in. Solr can be fed resumes in any document type like PDF, Word, XML or plain text and integrate those into its index. With its development as a search engine, it can easily process the unstructured text. It can extract key words and phrases, perform language detection and transparently deal with differing word forms. After a hire, periodic reviews can be combined with keywords, key phrases and other metadata extracted from the source resume to form a predictive model, which can then be used in later hiring processes.

1.4 Rest of the report

In this report we are focusing more on how Elasticsearch and the Elastic Stack fit in the search engine database domain. In the next chapter, we give a brief overview of the different parts that comprise the Elastic Stack and then go through the basic concepts of Elasticsearch. In Chapter 3 we dive into a concrete example from products installation to end visualization with real data and show some representative queries. Lastly, Chapter 4 concludes the report.

⁹ <https://www.blue-granite.com/blog/apache-solr-3-analytic-use-cases>

2 THE ELASTIC STACK

Elasticsearch is the base of a wide range of products that construct the elastic stack. And while you can do everything with Elasticsearch's API, there have been built some tools around it that offer even easier ways to interact with it. In this chapter we go through them. Let us start by showcasing the structure of the whole elastic stack products.

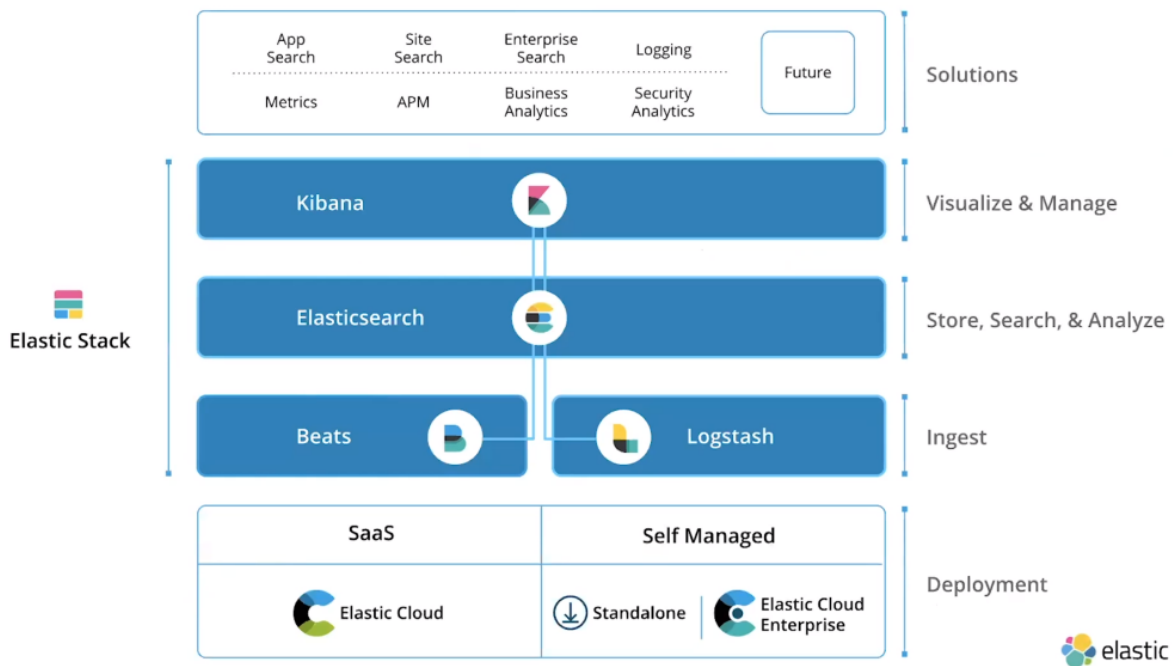


Figure 3 The elastic stack architecture¹⁰

Reading Figure 3 bottom-up, this product is offered both in cloud and as self-managed standalone deployment. Setting it up in cloud is very straight forward as everything is orchestrated together and maintained by the Elastic Stack team, though setting it up locally is also a very easy process. From our own experience, we set it up on our local machines (Windows & Linux) within minutes, without facing any obstacles.

At the bottom of the elastic stack lies the ingest products, responsible to bring data into the game. While Beats and Logstash are the main products performing this task, there are a lot of plugins that take care of seamlessly import many kinds of data input. Beats are lightweight data shippers and thus they are used to import specific types of data, having a small footprint and using fewer system resources than its brother Logstash. Logstash is the standard product of loading data into Elasticsearch, having a larger footprint but providing many alternatives for input, filter and output. As more and more plugins are added, Logstash becomes a robust point of reference for any kind of data collection and transformation from a variety of sources.

Elasticsearch is the core product, being responsible for storing, searching and analyzing data, being at the moment (December 2018) at top of the market.

¹⁰ <https://www.elastic.co/products>

Kibana is the data visualization tool, managing all the data that have been inserted into Elasticsearch by Logstash/Beats. It offers a wide variety of charts and data analysis tools, featuring a user-friendly interface and query language (Query DSL).

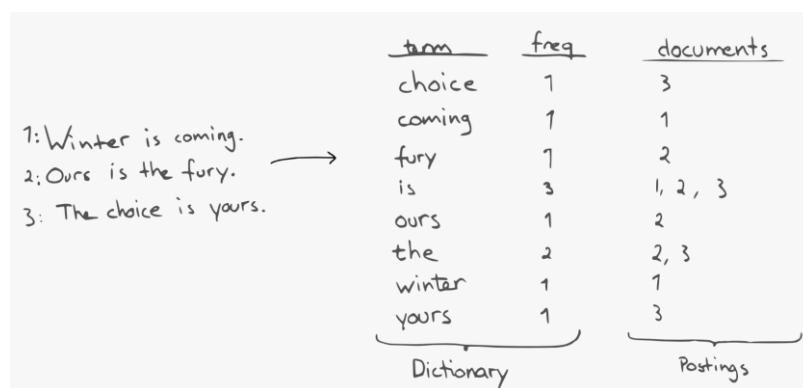
Finally, the presented model (Elastic Stack) can support a wide variety of applications, featuring cutting edge full text search capabilities and analytics. Considering the current dominance of elastic products in the market along with its large community, it is safe to assume that it is developing in fast pace and the community is anticipating the future developments.

2.1 Elasticsearch

Elasticsearch is the heart of elastic stack. It is an open source, Lucene-based¹¹, distributed, scalable, highly available, document-oriented, RESTful, full text search engine with real time search and analytics capabilities¹². It is built to handle huge amounts of data volume with very high availability and to distribute itself across many machines, to be fault- tolerant and horizontally-scalable; all while maintaining a simple but powerful API that allows applications from any language or framework access to the database. It delivers a full-featured search experience across big amounts and different structures of data. It is written in Java and therefore it is a cross-platform product. It is designed to take data from any source and make it searchable.

How does it make all data searchable?

Elastic uses immutable Lucene indexes which follow the paradigm of inverted indexes. An integral part of Elasticsearch is also text processing that transforms text into the vector space and allows for highly efficient similarity comparison against a query. While creating the inverting index, Elastic search offers a wide variety of text processing like tokenization, lower-casing, stopwords removal and stemming just to name few basic ones.



term	freq	documents
choice	1	3
coming	1	1
fury	1	2
is	3	1, 2, 3
ours	1	2
the	2	2, 3
winter	1	1
yours	1	3

Dictionary
Postings

Figure 4 Sample documents and resulting inverted index¹³

¹¹ <http://lucene.apache.org/>

¹² <https://www.elastic.co/videos/speed-is-key-elasticsearch-under-the-hood>

¹³ <https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up>

Immutability of indexes

Indexes in Elasticsearch are immutable; that is they cannot be changed. This comes with important benefits with respect to lookup time:

- No need for locking
- Once in memory or cache, it remains valid.
- Large indexes can be constructed, fact which can help keep them compact.

On the downside, updates on documents will cause a reconstruction of the index and deletes don't really happen in-place, but are marked so in a bitmap, until an index refresh.

Communicating with Elasticsearch is done through an HTTP REST API. Just like No-SQL databases it stores schema-less JSON documents (mapping), so you do not have to define fields and data types before adding data unlike traditional relational databases. It is near real-time, meaning that any document modification is propagated throughout the entire cluster within one or two seconds.

Elasticsearch uses JavaScript Object Notation, or JSON, as the serialization format for documents and queries. JSON serialization is supported by most programming languages, and has become the standard format used by the NoSQL movement. It is simple, concise, and easy to read, even though rows of files may seem big for the amount of data or actions they carry.

```
{
  "email":      "john@smith.com",
  "first_name": "John",
  "last_name":  "Smith",
  "info": {
    "bio":      "Eco-warrior and defender of the weak",
    "age":      25,
    "interests": [ "dolphins", "whales" ]
  },
  "join_date": "2014/05/01"
}
```

Figure 5 Example of a json document

Elasticsearch as a primary store?

While it is very possible to use Elasticsearch as a primary database store, it is generally recommended (and a common practice) to use Elasticsearch capabilities on top of any persistent No-SQL or SQL storage like MongoDB or relational databases. First thing to consider is that updates are slow in Elasticsearch as they require index reconstruction. Deletes are also not performed in place, but marked for later removal. Using another persistent data store you ensure correctness and robustness, while data can also be pushed to Elasticsearch for performing all kind of fancy searching and analytics. Although Elasticsearch is extremely good for indexing and searching big datasets, it is not a general purpose database like MongoDB.

Concepts like replication, integrity, consistency, robustness also exist in Elasticsearch. There are as many replicas of your data as user specifies (by default 1) in different nodes, making it fault-tolerant and highly available. Just like relational database user can specify constraints to define consistency, like referential integrity and uniqueness.

2.1.1 Key concepts

To better understand Elasticsearch we are diving into each of its core components presenting the key concepts following a bottom-up approach:

- **Field** – The smallest individual unit. Each field comes with two values: data type and data value. Data types can be simple (full-text, double, date, boolean) or complex (object, nested, multi-fields, geo information).
- **Document** – The base unit of storage, it is a collection of fields of a specific schema defined in JSON format. Every document is associated with a type and it is stored within an index. Documents also contain some metadata to declare the index and type it belongs to, as well as a unique identifier.
- **Type [Deprecated]** – Types are logical collections of documents sharing a set of common fields present in the same index. Since version 6, types are deprecated and while it was possible to define many types for one mapping this is no longer possible, and types will be completely vanished in a later version¹⁴.
- **Mapping** – It is the definition of how a document and its fields are stored and indexed. Mappings are providing the schema, stating which is the data type of each field in the documents of the index.
- **Index** – It is a collection of documents and document properties that somehow portray similar characteristics. In other words, it is a data organization mechanism, partitioning the data in a certain logical way, equivalent of a database on the relational world. Indexes use the concept of shards to improve the performance. During indexing, a concept called analyzer is used, in order to break down phrases or expressions into terms constructing the inverted index. Analyzers consist of a tokenizer and any number of token filters. The default analyzer removes stopwords, most punctuation and lowercases terms. There is a wide variety of built-in tokenizers offered by Elasticsearch, and you can also create and use a custom tokenizer.
- **Shard & Replicas** – As data scale you reach a point where they do not fit in a single space, or the query response time is getting slower. Then indexes are horizontally subdivided into

¹⁴ <https://www.elastic.co/guide/en/elasticsearch/reference/current/removal-of-types.html>

smaller pieces, called shards. Each shard contains all the properties of document but a smaller number of JSON objects than index. The horizontal separation makes shard an independent unit, which can be stored in any node, ensuring scalability as it enables distribution and parallelization of operations across many shards. Primary shard is the original horizontal part of an index and then these primary shards are replicated into replica shards. Replica shards are the result of data replication. They are stored in a different node (to provide high availability in case original node fails) and allow scaling the search volume as searches can be executed on all replicas in parallel.

- Node** – It refers to a single running instance of Elasticsearch carrying through the crucial task of storing and indexing data. There are different kind of nodes for different kind of jobs: Data nodes (for storing data and executing search operations on them), master nodes (handling the whole cluster management and configuration for actions like adding a new node in the cluster), client nodes (load balancers, forwarding cluster requests to master node and data-related requests to data nodes), tribe nodes (acting like client nodes, but used to perform operations against all clusters interconnected through config file), ingestion nodes (used to preprocess documents before indexing). Every node is by default a data node and within a cluster master node is by default randomly selected, though this can change in the occurrence of a failure.
- Cluster** – A Cluster is comprised from one or more nodes. Its job is to hold all your data together and provide indexing and search capabilities across all the nodes for entire data. It includes one master node (as defined above). As cluster grows it can reorganize itself, in order to spread data across its nodes, assuring balanced processing load and faster query times.

Putting all core elements of Elasticsearch we build the below diagram.

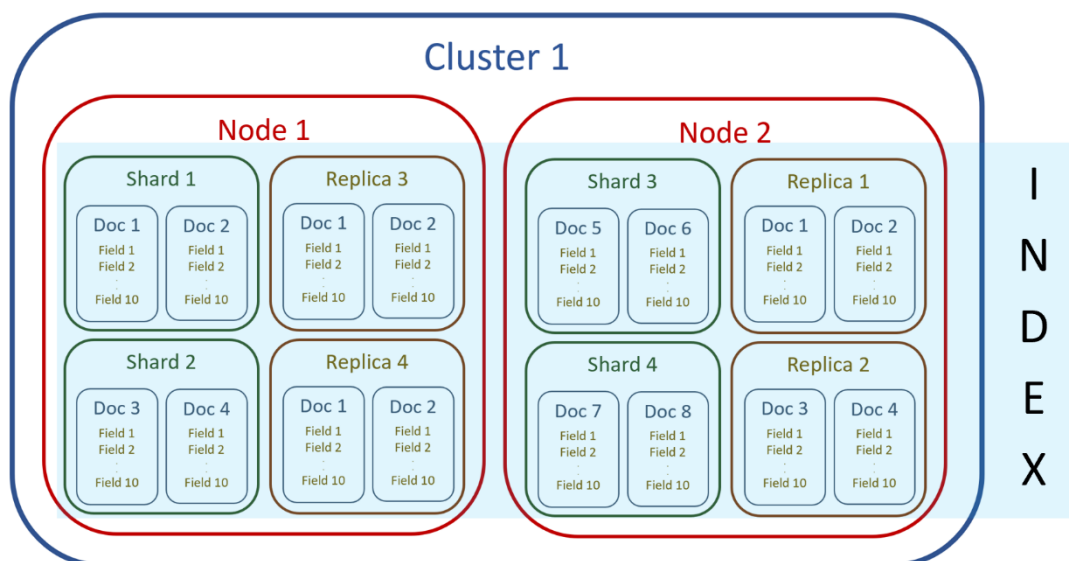


Figure 6 Elasticsearch core elements

2.1.2 CRUD operations

Taking into account elements defined above, in this section we will briefly showcase by means of an example how the basic CRUD (Create Read Update Delete) operations happen within Elasticsearch through Kibana > Dev Tools > Console interface (explained on section 2.4)

To do that let us first create an index named *book_index* along with a type named *book* that will define the schema of the documents we will later load. Note that the creation of type is optional as we can simply let Elasticsearch guess the data types.

```

1. PUT book_index
2. {
3.   "mappings": {
4.     "book": {
5.       "properties": {
6.         "title": { "type": "text" },
7.         "author": { "type": "text" },
8.         "pages": { "type": "integer" },
9.         "published_at": { "type": "date" }
10.      }
11.    }
12.  }
13. }
```

Now we **create** our document with a type of *book* previously defined and 1 as document id. If book mapping was not defined, Elasticsearch would define it on the spot. Note that we can simply insert two authors (as an array of strings) as long as both values match the type of the field (in this case: "text"). On the left it is our command and on the right the result returned.

<pre> 1. PUT book_index/book/1 2. { 3. "title": "Data Warehouses", 4. "author": ["Esteban Zimányi", "Alejandro Vaisman"], 5. "pages": "625", 6. "published_at": "2016-08-23" 7. }</pre>	<pre> 1. { 2. "_index" : "book_index", 3. "_type" : "book", 4. "_id" : "1", 5. "_version" : 1, 6. "result" : "created", 7. "_shards" : { 8. "total" : 2, 9. "successful" : 1, 10. "failed" : 0 11. }, 12. "_seq_no" : 0, 13. "_primary_term" : 1 14. }</pre>
---	--

In this stage, let us perform a get request to **read** our recently created document by its id:

<pre> 1. GET book_index/book/1</pre>	<pre> 1. { 2. "_index" : "book_index", 3. "_type" : "book", 4. "_id" : "1", 5. "_version" : 1, 6. "found" : true, 7. "_source" : { 8. "title" : "Data Warehouses", 9. "author" : [10. "Esteban Zimányi", 11. "Alejandro Vaisman" 12.], 13. "pages" : "625", 14. "published_at" : "2016-08-23" 15. } 16. }</pre>
--------------------------------------	--

Deleting the document is just a simple command:

```
1. DELETE book_index/book/1
```

```
1. {
2.   "_index" : "book_index",
3.   "_type" : "book",
4.   "_id" : "1",
5.   "_version" : 2,
6.   "result" : "deleted",
7.   "_shards" : {
8.     "total" : 2,
9.     "successful" : 1,
10.    "failed" : 0
11.  },
12.   "_seq_no" : 1,
13.   "_primary_term" : 1
14. }
```

If we instead want to **update** the document we can do this :

```
1. PUT book_index/book/1
2. {
3.   "title": "Data Warehouses - Design and implementation",
4.   "author": ["Esteban Zimányi", "Alejandro Vaisman"],
5.   "pages": "625",
6.   "published_at": "2016-08-23"
7. }
```

```
1. {
2.   "_index" : "book_index",
3.   "_type" : "book",
4.   "_id" : "1",
5.   "_version" : 5,
6.   "result" : "updated",
7.   "_shards" : {
8.     "total" : 2,
9.     "successful" : 1,
10.    "failed" : 0
11.  },
12.   "_seq_no" : 6,
13.   "_primary_term" : 1
14. }
```

2.1.3 Querying plan

As Elasticsearch is distributed, the search process is split into two parts:

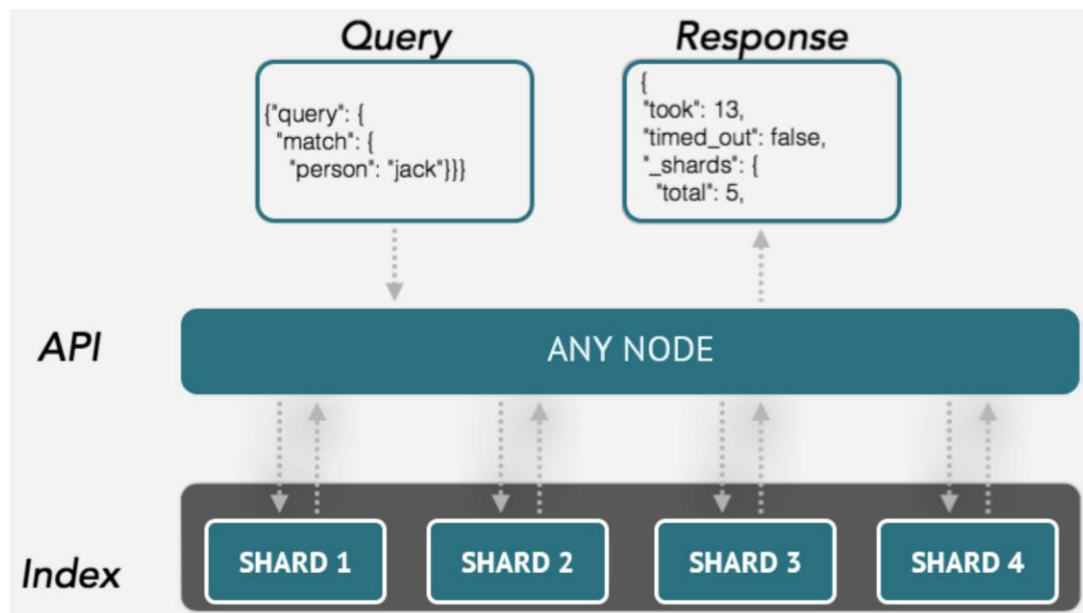
1. The query phase

Let us take as an example a top 10 query. The query is sent out to all shards (primary or replicas) via the coordinating node within the index that is being searched over. Every single shard performs the query locally and sends back its top 10 results according to the relevance score.

2. The fetch phase

Coordinating node gathers the top 10 results from each shard and identifies the final top 10 documents according to their relevancy score and issues a GET request to the relevant shards they belong. Shards returns the requested documents and coordinate node returns results to the user.

Below there is a visualization of the above process.

Figure 7 How Elasticsearch works¹⁵

2.2 Beats

Beats is a family of single-purpose data shippers. They send data from hundreds or thousands of machines and systems to Logstash or Elasticsearch. You can see the available shippers on the figure below, along with the kind of data they manipulate.

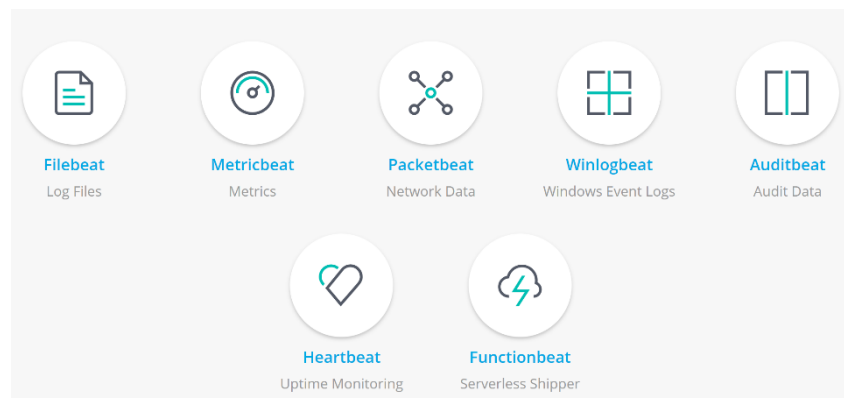


Figure 8 The family of Beats data shippers

As an example, Filebeat is a robust lightweight way to forward and centralize logs and files. It worths mentioning that it implements a backpressure-sensitive protocol when sending data to Logstash or Elasticsearch. If Logstash is struggling to keep up with Filebeat's pace, it lets Filebeat know to slow down its read. When Logstash becomes loose again, Filebeat will build back up to its original pace and keep on shipping.

¹⁵ <https://dzone.com/articles/what-is-elasticsearch-and-how-it-can-be-useful>

Another example, Heartbeat generates uptime and response time data. For a given list of URLs, Heartbeat performs a ping, gathers response information and sends them to Logstash or Elasticsearch for further analysis, visualized with Kibana.

2.3 Logstash

Logstash is a data processing pipeline, fetching data from multiple sources simultaneously (files, logs, sockets, tpc, udp, etc.) in a continuous streaming fashion, processes it (adding timestamp, IP-based geoinformation, etc.) and ships it to the destination stash. As expected, Logstash works seamlessly with Elasticsearch as stash, as it is part of the elastic stack family, though it can be applied on other kind of stashes too. Similar as with Beats, if Elasticsearch is struggling to keep up with Logstash's pace of incoming data, Logstash is notified to slow down and it resumes when Elasticsearch health is back to green.

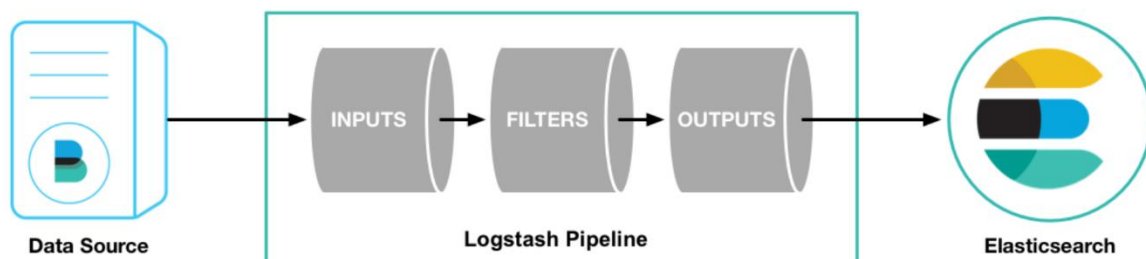


Figure 9 Logstash pipeline structure

As data travels from source to store (Elasticsearch), Logstash filters parse each object. Given the wide variety and richness of filters available, we can confidently assume that any kind of data transformation can happen. Finally, the filtered input is outputted to a stash. We hereby present the most typical filters used with Logstash when it dynamically parses data:

- Grok: As data is commonly unstructured, Logstash parses data into fields
- Geolp: Adds geographical information derived by the IP address
- Fingerprint: Anonymizes data by replacing values with a consistent hash
- Date: Parsing the date field of a data object, it constructs a timestamp
- Mutate: Performs mutations on fields
- JSON: Parses JSON events
- Xml: Parses XLM into fields
- Range: Checks data object values against thresholds of acceptable values range
- i18n: Removes special characters from the data fields
- Drop: completely drops one data object/event
- Clone: makes a copy of a data object/event possibly adding or removing fields

Overall, Logstash is a well-established powerful tool in the market, being very popular at its category, as it is also used outside elastic family.

2.4 Kibana

Kibana is an open source analytics and visualization platform, completing the elastic products family. Searching, viewing, querying and generally interacting with data stored in Elasticsearch is taken care from the very user-friendly environment of Kibana. It offers a wide variety of visualization elements like charts, tables, heatmaps and coordinate maps just to name a few. In addition, advanced data analysis techniques like machine learning are supported. Its simple browser-based interface enables you to create and share dynamic dashboards that display changes to Elasticsearch queries in real time. That means that Kibana instantly displays every data record that is coming into Elasticsearch through Logstash for example. Setting it up is simple as possible and it is up and running in seconds both in Windows and in Linux, always connecting seamlessly to the local Elasticsearch installation.

Kibana interface consists of the following elements:

- **Discover:** interactively explores data, querying using Query DSL syntax, filter results, view data. When you submit a search request, all the elements of the page (fields, documents, graph) are updated to reflect the search results. The total number of hits (matching documents) is shown in the toolbar.

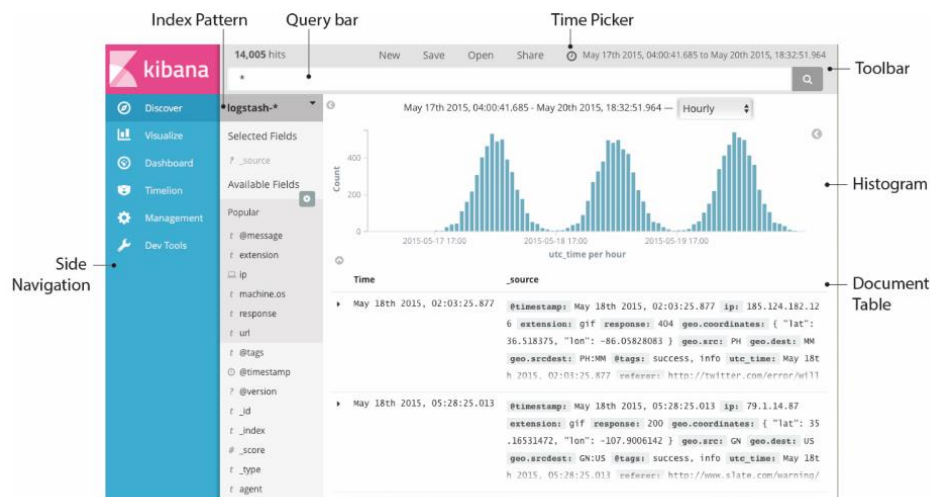


Figure 10 Kibana Discover interface¹⁶

- **Visualize:** Using a saved query performed on Discover, or creating a new one from scratch, visualize interface offers creation of different kind of visualizations with all kinds of metric aggregations. Currently, Kibana supports:
 - basic charts: line, area, bar charts, heat maps, pie charts
 - data: data table, metric, goal and gauge
 - maps: coordinate map, region map
 - time series: timelion, time series visual builder
 - other: controls, markdown widget, tag cloud, vega graph
 - interactive input controls: dropdown list and range.

¹⁶ <https://www.elastic.co/guide/en/kibana/current/discover.html>

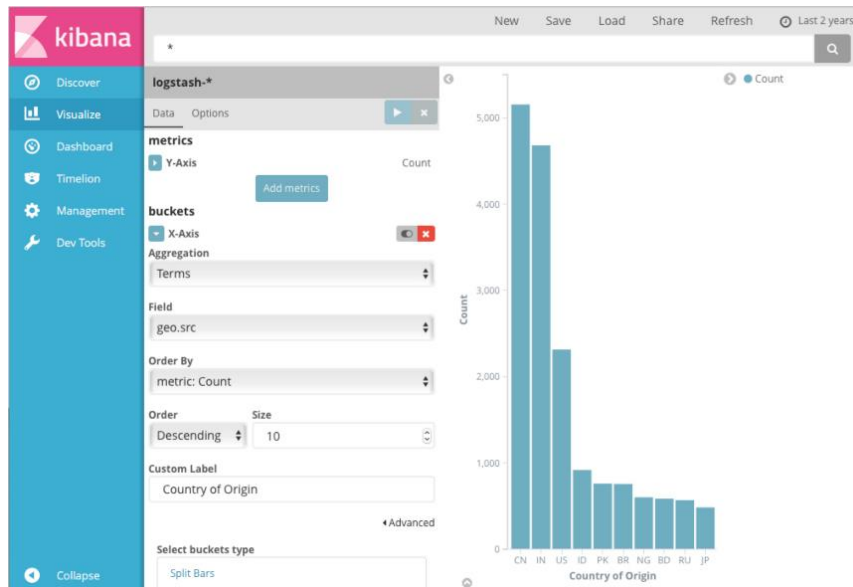


Figure 11 Kibana Visualize interface, simple bar chart

- Dashboard:** Once all kinds of visualizations are created, Dashboard takes over to display them all into a comprehensive data rich dashboard-like environment. Dashboards can easily be shared with a permalink. Every chart of the dashboard can be resized, dragged around or inspected further so as to view or download the specific data that are hidden behind it in csv or also check the specific Elasticsearch query behind and its response in JSON format. Of course, as you type queries on the query bar the whole dashboard is adjusted to your requirements. Plus, it allows you to enter specific filters independently of the queries or use the interactive input controls created on visualization step (dropdown list or range – shown on the figure below).

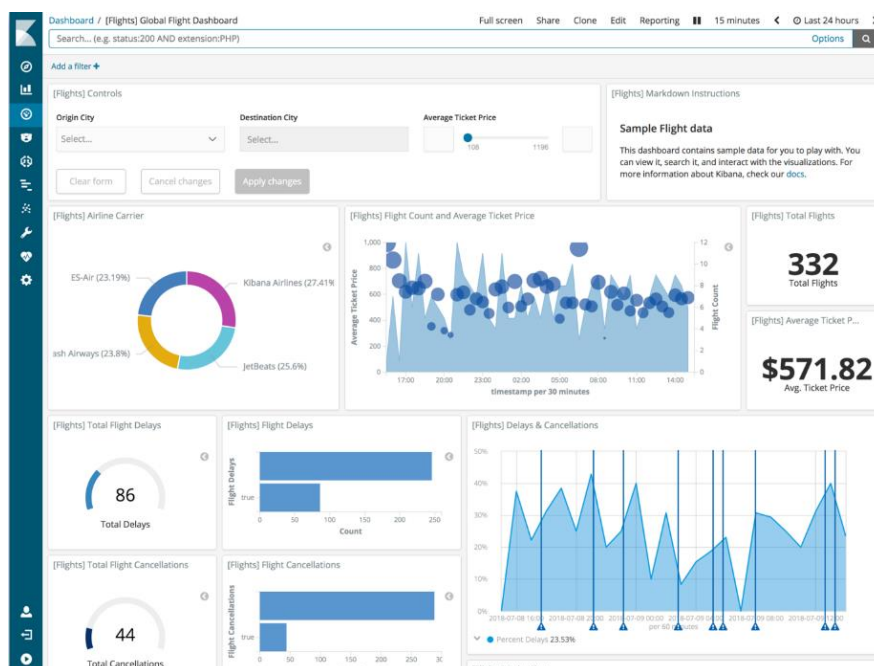


Figure 12 Kibana Dashboard interface

- **Timelion:** Timelion is a time series data visualizer interface. It's driven by a simple expression language you use to retrieve time series data, perform calculations to tease out the answers to complex questions, and visualize the results. Its elements can also be added to the Dashboard in order to construct one general picture of data.

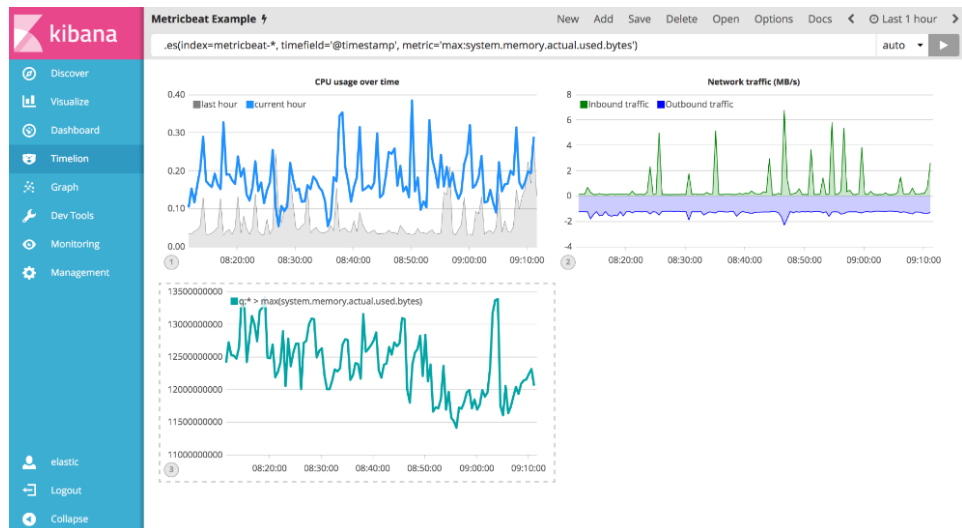


Figure 13 Kibana Timelion interface for time series visualization¹⁷

- **Canvas:** Another way of visualizing data is offered through Canvas. It is indeed a canvas which you can fill in with data but visualize it in a more creative way than the standard charts. Canvas combines data with colors, shapes, text, and your imagination to bring dynamic, data-rich data displays. Below there are two examples of such data displays for e-commerce and web traffic.

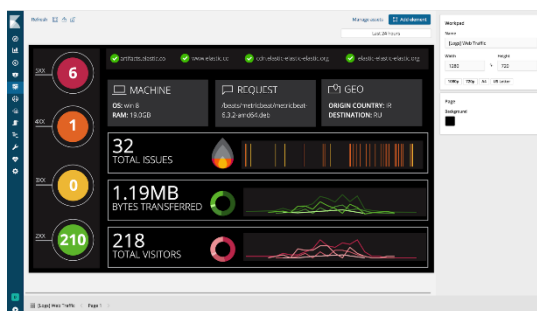


Figure 14 Kibana Canvas interface for web traffic

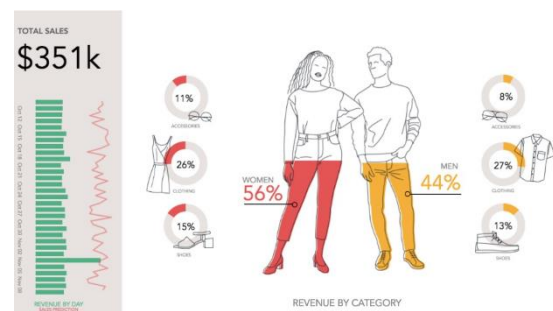


Figure 15 Kibana Canvas interface for e-commerce

- **Dev Tools:** Provides three developer tools to interact with data stored in Elasticsearch directly from Kibana interface. First is the Console, splitting screen into two: on the left side accepting curl-like commands to interact with the REST API of Elasticsearch and displaying results on the right side. Here is an example where we request all created indexes in Elasticsearch:

¹⁷ <https://www.elastic.co/guide/en/kibana/current/timelion-conditional.html>

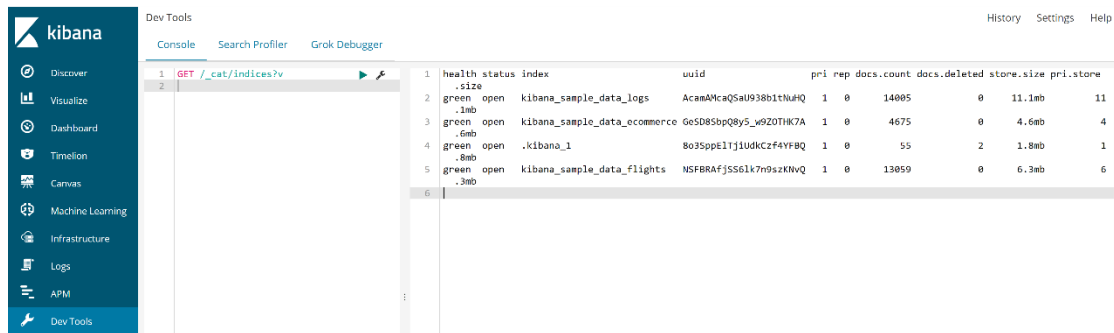


Figure 16 Kibana Dev Tools: Console

Also, within Dev Tools there is Search Profiler, which communicates with Profiler API used to inspect and analyze queries. The response is usually a large JSON file, which Kibana transforms into a visualization, allowing to investigate poorly performing queries. Last but not least, Grok debugger is the perfect tool to help you construct a proper pattern for your data. You can insert some sample data, try out different kind of patterns and immediately review the result.

- **Management:** This is the place where Kibana configuration takes place. Along with specifying a configuration and adding plugins, most importantly you can define your index patterns. These index patterns should match one or more indices stored on your Elasticsearch instance.

Also, as the time goes by, more and more features are getting added within Kibana. It also supports machine learning capabilities (identifying anomalous patterns in data, statistical rarity, unusual behaviors, prediction of future behavior), infrastructure (identifying infrastructure problems in real-time, explore metrics and logs of servers, containers or services), APM (automatically collects in-depth performance metrics and errors from inside your applications, visualizing application bottlenecks), Graph (discovery of data items relations, graph-based recommendations for e-commerce), Monitoring (health and performance data for the elastic products, alerts).

One last thing we want to focus upon regarding Kibana interface is the **query bar** of Kibana providing a very simple and intuitive way to search data, hiding beneath the powerful JSON-based Query DSL that we will explain further on section 3.4. Autocomplete and a simplified query syntax are available, making the querying experience a very user-friendly process.

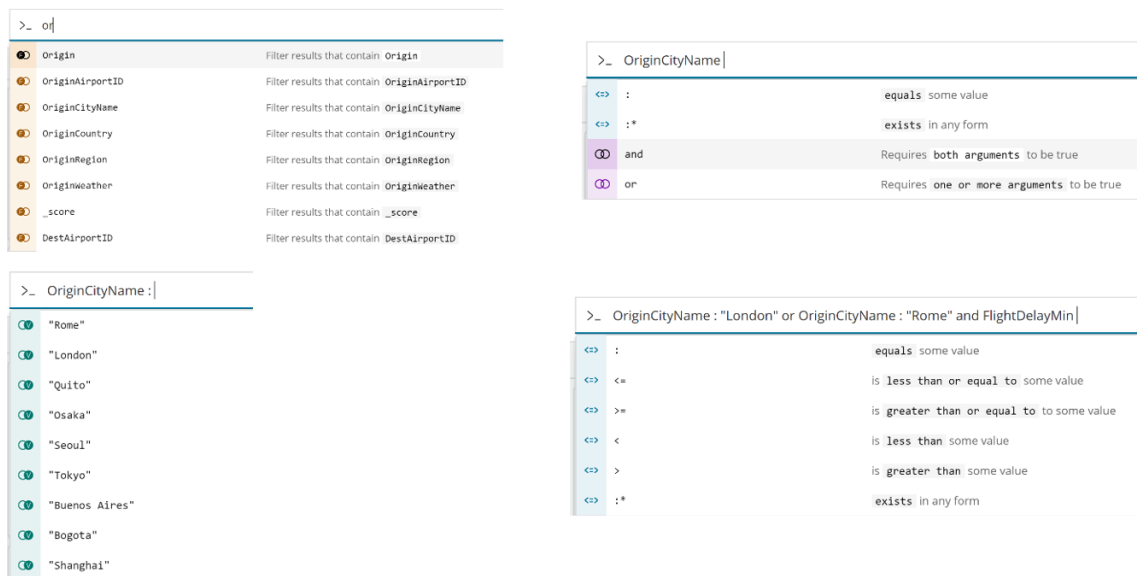


Figure 17 Constructing query using Auto-complete in Kibana query bar

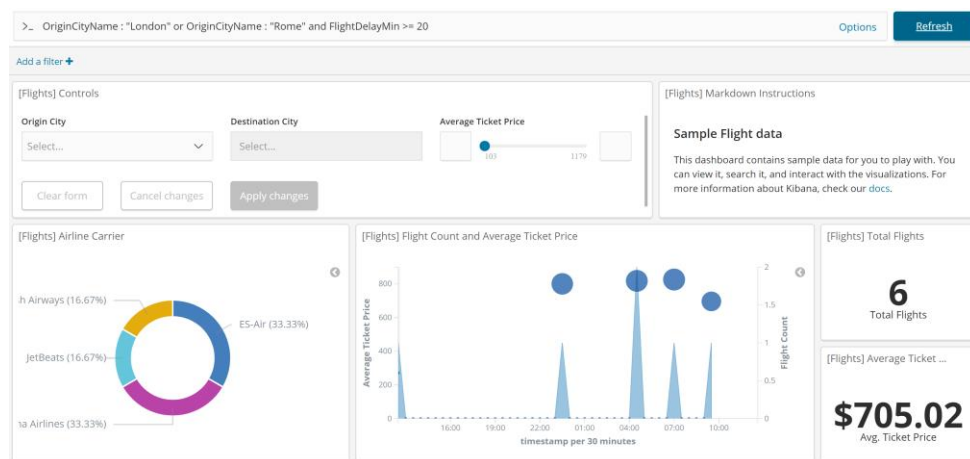


Figure 18 Adjusted visualizations upon result of the query on Kibana dashboard

Of course you are able to use the same query on Discover tab of Kibana, in order to directly get all the matching documents as illustrated below.

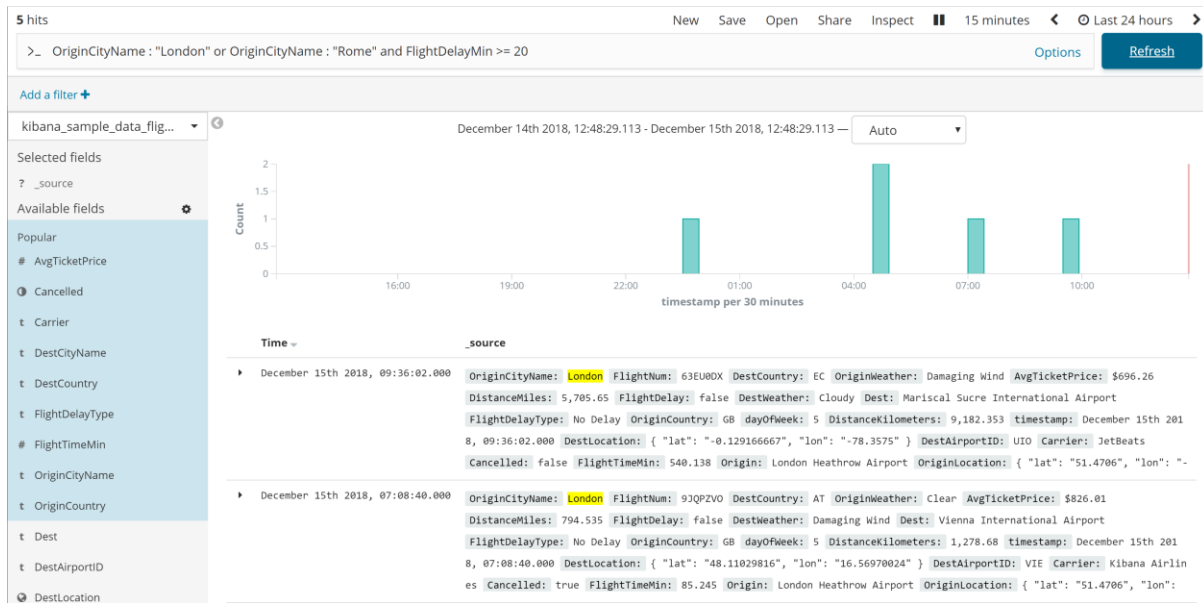


Figure 19 Adjusted documents list upon result of the query on Kibana discover

Overall, we believe that Kibana is a very powerful tool being capable of serving multiple use cases, for analyzing visualizing data in the most insightful way.

2.5 Industry use cases

The elastic products family can be used for a wide variety of applications. We hereby present some cases where elastic is a perfect fit and showcasing some real-world applications by big companies.

Elasticsearch can be a perfect fit for applications that require fast full text search on any kind of unstructured data, supporting also autocomplete suggestions. What is more, alerting based on specific event is another interesting aspect of Elasticsearch. For example, alert a user when the price of a specific product he has shown interest in falls below a specific amount on some company provider of his area.

Another typical case would be to collect and analyze log or transaction data, looking for trends, anomalies, aggregations, with any kind of data mining or machine learning techniques. Lastly, with the introduction of Kibana, visualizing a lot of data in a wide variety of ways has never been easier. Business intelligence is fully represented and heavily used within the elastic products supporting complex data analysis queries and providing insightful dashboards.

Worth mentioning is also the Elasticsearch capability on time series data like metrics and application events. With powerful Beats handling the delivery of data from source and Timelion user interface within Kibana handling the visualization, time series data has become one of the famous use cases of elastic products family.

Elasticsearch is already used by many companies and for a very wide variety of applications. Below there are a few sample use-cases of Elasticsearch¹⁸:

- **Wikipedia**: Full-text search to provide suggested text.
- **The Guardian**: Give editors current feedback about public opinion on published media through social and visitor data.
- **Stack Overflow**: Complete full-text search, geolocation queries and source related questions and/or answers.
- **GitHub**: Queries billions of lines of code with the search engine.
- **Netflix**: Monitoring and analysing customer service related operations and security related logs.
- **LinkedIn**: Support their load in real time and monitor performance and security.

As the uses of Elasticsearch continue to grow and change over time, it is expected that more and more use cases will be supported in the future.

3 PRACTICAL EXAMPLE

In this chapter, we present a practical example of using Elasticsearch with a specific dataset. Firstly, we show the steps needed to perform the installation of Elasticsearch. Then, we present the NYPD Motor Vehicle Collision¹⁹ data that we will use for our example. After, we present the steps needed to appropriately load the data into Elasticsearch with Beats and visualize them in Kibana, based on an example of Elasticsearch usage²⁰.

3.1 Installation Guide

In this section we will go through the installation of the four main products of the Elastic family: Elasticsearch, Kibana, Logstash and Filebeat. As you will see in the guide, installations are straightforward and you can get it all running within few minutes.

3.1.1 Installing Elasticsearch

Hereby the installation steps:

1. Download the appropriate (for your system) file from the download page: <https://www.elastic.co/downloads/elasticsearch>
2. Unzip folder and open a command line there.
3. Run `bin/elasticsearch` (or `bin\elasticsearch.bat` on Windows)
4. Run `curl http://localhost:9200/` (or `Invoke-RestMethod http://localhost:9200` with PowerShell on windows)

Elasticsearch is up and running on your localhost:9200. Point your browser there and you should get something like this: (note the tagline: “You Know, for Search”)

¹⁸ <https://www.quora.com/What-are-use-cases-of-Elasticsearch>

¹⁹ <https://data.cityofnewyork.us/Public-Safety/NYPD-Motor-Vehicle-Collisions/h9gi-nx95?>

²⁰ https://github.com/elastic/examples/tree/master/Exploring%20Public%20Datasets/nyc_traffic_accidents

```
1. {"name" : "ukLIVGf",
2.   "cluster_name" : "elasticsearch",
3.   "cluster_uuid" : "Vd155ZTJRbSlPC48ucvsVQ",
4.   "version" : {
5.     "number" : "6.5.2",
6.     "build_flavor" : "default",
7.     "build_type" : "zip",
8.     "build_hash" : "9434bed",
9.     "build_date" : "2018-11-29T23:58:20.891072Z",
10.    "build_snapshot" : false,
11.    "lucene_version" : "7.5.0",
12.    "minimum_wire_compatibility_version" : "5.6.0",
13.    "minimum_index_compatibility_version" : "5.0.0"  },
14.   "tagline" : "You Know, for Search" }
```

3.1.2 Installing Kibana

Hereby the installation steps:

1. Download the appropriate (for your system) file from the download page: <https://www.elastic.co/downloads/kibana>
2. Unzip folder and open config/kibana.yml in an editor. Set to your Elasticsearch instance:

```
1. # The URL of the Elasticsearch instance to use for all your queries.
2. elasticsearch.url: "http://localhost:9200"
```

3. Run `bin/kibana` (or `bin\kibana.bat` on Windows)
4. Point your browser at "`http://localhost:5601`"

Kibana is up and running on your localhost:5601. Point your browser there and you should see the Kibana interface.

3.1.3 Installing Logstash

Hereby the installation steps:

1. Download the appropriate (for your system) file from the download page: <https://www.elastic.co/downloads/logstash>
2. Prepare a proper logstash-simple.conf file:

```
1. input { stdin { } }
2.   output {
3.     elasticsearch { hosts => ["localhost:9200"] }
4.     stdout { codec => rubydebug }
5.   }
```

3. Run Logstash and specify the configuration file with the `-f` flag.

```
1. bin/logstash -f logstash-simple.conf
```

Logstash is up and running. As stated on the configuration file “input{ stdin{ } }” you can type a row of data on the command line and Logstash will process it and import to Elasticsearch. Of course, usually configuration file is much more sophisticated. Another example is the one below, where you can input data from a file.

```
1. input {
2.   file {
3.     path => "/tmp/access_log"
4.     start_position => "beginning"
5.   }
6. }
7.
8. filter {
9.   if [path] =~ "access" {
10.    mutate { replace => { "type" => "apache_access" } }
11.    grok { match => { "message" => "%{COMBINEDAPACHELOG}" } }
12.   }
13.   date { match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ] }
14.
15. }
16.
17. output {
18.   elasticsearch { hosts => ["localhost:9200"] }
19.   stdout { codec => rubydebug }
20. }
```

3.1.4 Installing Filebeat

Hereby the installation steps:

1. Download the appropriate (for your system) file from the download page:
<https://www.elastic.co/downloads/beats/filebeat>
2. Unzip folder and open Filebeat.yml in an editor. Set it so that it points at your data.

```
1. filebeat.prospectors:
2. - type: log
3.   paths:
4.     - C:/Users/sokpa/Desktop/use_case/nyc_collision/nyc_collision_data.csv
5.
6. output.elasticsearch:
7.   hosts: ["localhost:9200"]
8.   index: nyc_visionzero
9.   pipeline: nyc_collision
10.
11. setup.template.enabled: false
```

3. Start the demon by running `sudo ./Filebeat -e -c Filebeat.yml`

Your data will start importing into Elasticsearch and you can immediately visualize them in Kibana. As the import is processing, Kibana visualizations will be getting updated accordingly.

3.2 Importing the data

Once you have Elasticsearch, Kibana and Filebeat installed you are ready to go.

1. Run Elasticsearch and Kibana

```
<path_to_elasticsearch_root_dir>/bin/elasticsearch  
<path_to_kibana_root_dir>/bin/kibana
```

2. Make sure that both are up and running (change the ports accordingly if you have not used the defaults).
 - a. Point your browser at `localhost:9200` – should return the details with the tagline: “You Know, for Search”.
 - b. Point your browser at `localhost:5601` – should display Kibana UI.
3. Download input data

- a. Download the CSV version of the NYPD Motor Vehicle Collision dataset from the NYC Open Data Portal²¹. In this example, we are renaming the downloaded CSV file to `nyc_collision_data.csv`.

To perform this on linux you can simply run:

```
1. mkdir nyc_collision  
2. cd nyc_collision  
3. wget https://data.cityofnewyork.us/api/views/h9gi-nx95/rows.csv?accessType=DOWNLOAD -O nyc_collision_data.csv
```

4. Put the downloaded input data into one folder along with the 4 files attached to this project. You should now have a folder with:
 - a. `nyc_collision_data.csv` – raw input data file
 - b. `nyc_collision_filebeat.yml` – Filebeat config file for importing data
 - c. `nyc_collision_kibana.json` – Kibana config file to prebuild dashboard
 - d. `nyc_collision_pipeline.json` – pipeline config for processing CSV values
 - e. `nyc_collision_template.json` – template for the custom mapping of fields

Let us hereby explain each file and what its place is within our example.

nyc_collision_filebeat.yml

This file configures the run of Filebeat, which we need in order to import data into Elasticsearch. We need to specify the input, passing the input file path of `nyc_collision_data.csv` but also the output. In our case output is Elasticsearch, which is defined on hosts: [`localhost:9200`]. Remember that 9200 is the default port of Elasticsearch. However, note that it is possible to output our data from Filebeat into Logstash and let Logstash handle the route to Elasticsearch. Lastly, we provide a unique name for our index and pipeline.

²¹ <https://data.cityofnewyork.us/Public-Safety/NYPD-Motor-Vehicle-Collisions/h9gi-nx95?>

nyc_collision_pipeline

This is the ingest pipeline for processing csv lines. In other words, it is setting up a template within Elasticsearch, which will later be mapped to the import data in order to produce the defined fields along with their transformations. More specifically, defines the fields structure and types and perform any transformations that are needed (like trimming, converting types, remove unnecessary fields, combine or create fields based on the input). Let us now present the transformations that occur within that file:

- Firstly, the pattern of the whole row is defined under `grok>patterns`. In this pattern we use some custom patterns that are defined exactly after this, on “`pattern_definitions`”. As you can observe it is possible to use a custom pattern inside another custom pattern if all are defined.

```
"CUSTOM_DATE": "%{MONTHNUM}/{MONTHDAY}/{YEAR},%{CUSTOM_TIME}",
"CUSTOM_TIME": "%{HOUR:hour_of_day}:%{MINUTE}",
"LOCATION": "%{BASE10NUM},%{SPACE}%{BASE10NUM}"
```

- Then, it trims several fields and performs the appropriate type conversions following

```
1. {
2.   "convert": {
3.     "field": "number_of_persons_injured",
4.     "type": "integer",
5.     "ignore_failure": true
6.   }
7. },
```

- Plus, it includes some scripting, in order to calculate some fields values, again, following below structure:

```
1. {
2.   "script": {
3.     "lang": "painless",
4.     "inline":
5.       "if (ctx.number_of_persons_killed == '') {
6.         ctx.number_of_persons_killed = 0;
7.       }
8.       if (ctx.number_of_persons_injured == '') {
9.         ctx.number_of_persons_injured 0;
10.      }
11.
12.      ctx.number_persons_impacted = ctx.number_of_persons_killed +
13.      ctx.number_of_persons_injured;";
13.     "ignore_failure": true
14.   }
15. },
```

- Again, some different calculations occur in order to define an arraylist of items, based on the input data values.

```

1. {
2.   "script": {
3.     "lang": "painless",
4.     "inline": "HashSet factors = new HashSet();
5.               factors.add(ctx.contributing_factor_vehicle);
6.               factors.add(ctx.contributing_factor_vehicle_2);
7.               factors.add(ctx.contributing_factor_vehicle_3);
8.               factors.add(ctx.contributing_factor_vehicle_4);
9.               factors.add(ctx.contributing_factor_vehicle_5);
10.              factors.remove('Unspecified');
11.              factors.remove('');
12.              ctx.contributing_factor_vehicle = new ArrayList(facto
rs);",
13.     "ignore_failure": true
14.   }
15. },

```

- Lastly, it ignores some values that are no longer of use:

```

1. {
2.   "remove": {
3.     "field": "contributing_factor_vehicle_2",
4.     "ignore_failure": true
5.   }
6. },

```

nyc_collision_template.json

The index's settings and template, describing the fields of each document that will be stored. In Elasticsearch's terms this is called the Mapping, which is the process of defining how a document, and the fields it contains, are stored and indexed. In this specific case, the documents that will be stored, are as defined in the *nyc_collision_pipeline.yml* and discussed above.

nyc_collision_kibana.json

This file holds the exported kibana visualizations as defined in the original repository.

5. Now it is time to run all the configurations on the mapping fields schema we created above. Open a command line on the folder containing all our files. First, we install the pipeline instance into Elasticsearch using the below curl command:

```

1. curl -XPUT -H 'Content-Type: application/json' 'localhost:9200/_ingest/pipeline/nyc_collision' -d @nyc_collision_pipeline.json

```

Then, we install the index template into Elasticsearch executing below command:

```

1. curl -XPUT -H 'Content-Type: application/json' 'localhost:9200/_template/nyc_collision' -d @nyc_collision_template.json

```

2. By now, Elasticsearch is ready to receive our input. We need to feed it using Filebeat and the configuration file we created for this purpose. Move the config file (`nyc_collision_filebeat.yml`) into the Filebeat installation folder and then execute in the following command:

```
1. ./filebeat -e -c nyc_collision_filebeat.yml
```

Congratulations. You have successfully imported your data into Kibana.

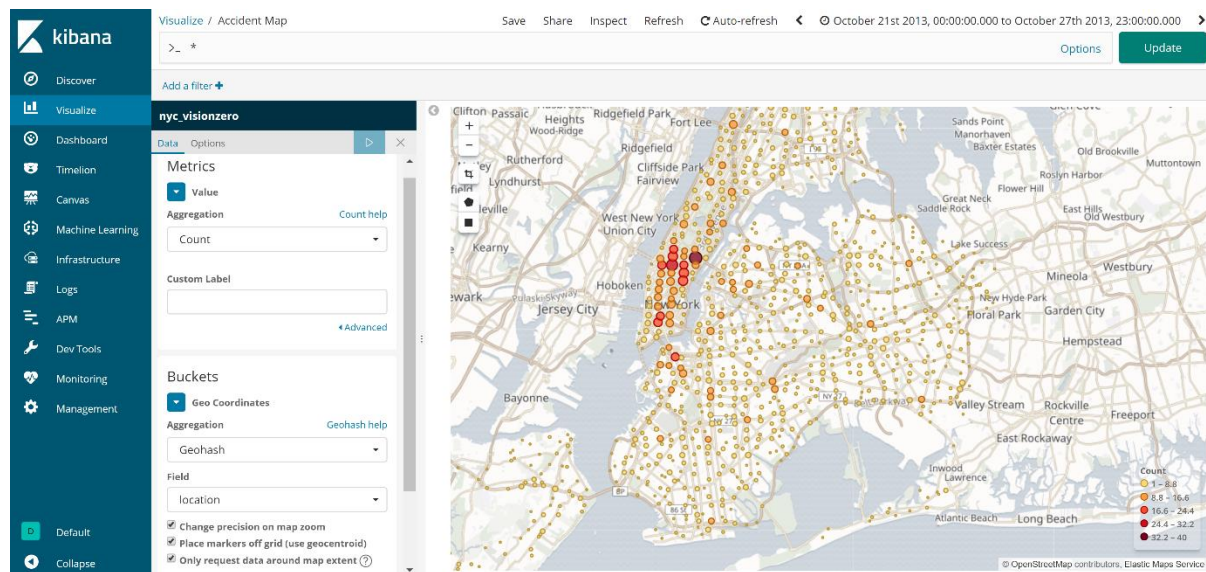
3.3 Building Visualizations in Kibana

After the data have been imported, we can now directly view them in Kibana. Just follow the below simple steps:

1. Point your browser to `localhost:5601` to access Kibana (adapt port if you have not used the default one).
2. On the menu items, go to Management tab > Index Patterns > Create index pattern
3. Specify `nyc_visionzero` as index pattern name and click on Next step.
4. Select `@Timestamp` as Time field and finish the wizard.
5. As we have provided the exported Kibana json file you can simply import it and directly view all visualizations. To do that simple go to Management > Select Saved Objects tab from the above menu bar > Import, and select `nyc_collision_kibana.json`
6. On the menu go to Dashboard, open NYC Motor Vehicles Collision dashboard and happy data hunting!

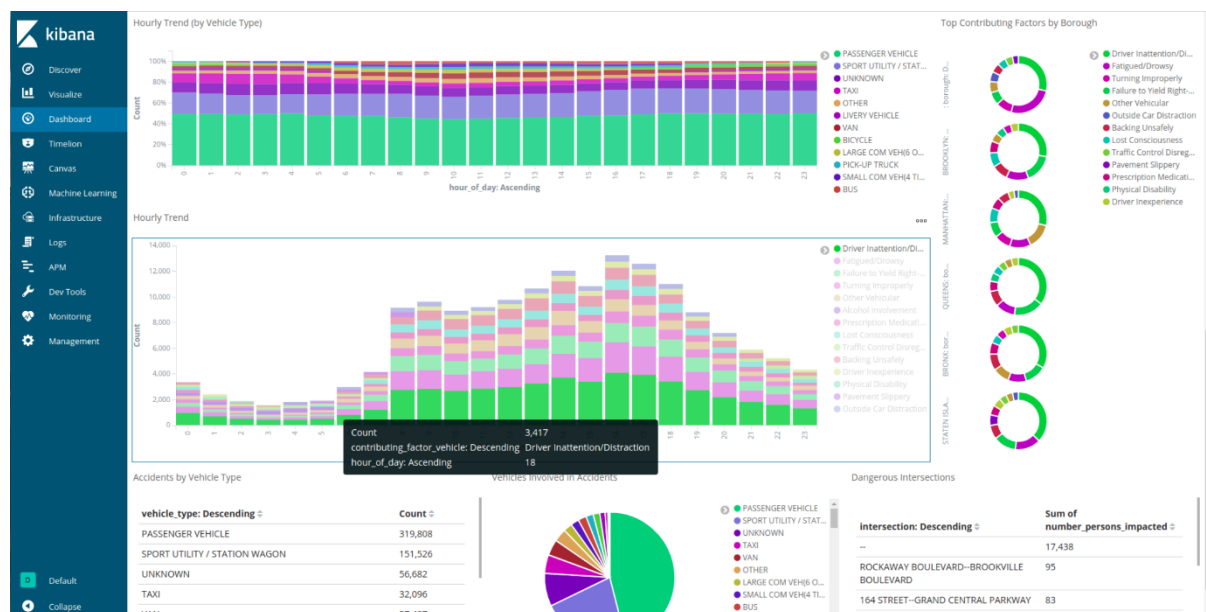
3.3.1 Accidents map

As an example, we hereby explain the creation of one visualization which is added to the dashboard. To build the accidents map where geolocation of each accident is plotted, we need to go on Visualize tab, click on add visualization (+) and select "Coordinate Map" type. Then we select our index: `nyx_visionzero`. On Metrics we let "Count" as we are interested in the number of accidents for a region. On buckets we select Geohash and we select location as the field to target and our accident maps is ready. It is worth noting that it is an interactive map, meaning that as we zoom out or zoom in, the data are adjusted accordingly.



3.3.2 Other Visualizations

Similarly, we can create other useful visualizations that can give us a good intuition over our data. Note that all visualizations in the dashboard adjust their data on the given query.



3.4 Querying Elasticsearch

It is time to see how some basic search engine queries look like in Elasticsearch. Queries are done via HTTP requests at the Elasticsearch server, which runs a CRUD HTTP API.

There are two main ways to query Elasticsearch:

1. URI Search, which we explore in 3.4.1
2. Request Body Search, which we explore in 3.4.2

3.4.1 URI Search

The easiest way to search an Elasticsearch cluster is through URI search. You can pass a simple query to Elasticsearch using the `q` query parameter. The following query will search the `nyc_visionzero` index for documents with a `borough` field equal to "MANHATTAN" (`q=borough:MANHATTAN`) and return 1 of them(`size=1`):

```
1. curl "http://localhost:9200/nyc_visionzero/_search?size=1;q=borough:MANHATTAN"
```

This returns the following results in JSON format:

```
1. {
2.   "took" : 16,
3.   "timed_out" : false,
4.   "_shards" : {
5.     "total" : 5,
6.     "successful" : 5,
7.     "skipped" : 0,
8.     "failed" : 0
9.   },
10.  "hits" : {
11.    "total" : 240736,
12.    "max_score" : 1.7550646,
13.    "hits" : [
14.      {
15.        "_index" : "nyc_visionzero",
16.        "_type" : "doc",
17.        "_id" : "5d_K1GcB7iJGd2DqSMju",
18.        "_score" : 1.7550646,
19.        "_source" : {
20.          "number_of_motorist_injured" : 0,
21.          "latitude" : "40.785183",
22.          "number_of_cyclist_killed" : 0,
23.          "on_street_name" : "",
24.          "source" : "/home/jp/Projects/adb_elasticsearch/nyc_collision_example/dat
a/nyc_collision_data.csv",
25.          "borough" : "MANHATTAN",
26.          "number_of_persons_killed" : 0,
27.          "zip_code" : "10024",
28.          "contributing_factor_vehicle" : [
29.            "Passing Too Closely"
30.          ],
31.          "number_persons_impacted" : 0,
32.          "intersection" : "--",
33.          "host" : {
34.            "name" : "jp-laptop"
35.          },
36.          "beat" : {
37.            "hostname" : "jp-laptop",
38.            "name" : "jp-laptop",
39.            "version" : "6.5.2"
40.          },
41.          "number_of_pedestrians_killed" : 0,
42.          "off_street_name" : "155 WEST 83 STREET",
43.          "hour_of_day" : 14,
44.          "longitude" : "-73.97512",
45.          "number_of_motorist_killed" : 0,
46.          "offset" : 5443859,
47.          "unique_key" : "3995295",
48.          "prospector" : {
49.            "type" : "log"
50.          },
```

```

51.         "vehicle_type" : [
52.             "Station Wagon/Sport Utility Vehicle"
53.         ],
54.         "message" : "10/06/2018,14:45,MANHATTAN,10024,40.785183,-
73.97512,\"(40.785183, -
73.97512)\\",,,155          WEST 83 STREET          ,0,0,0,0,0,0,0,0,0,0,Passing Too
Closely,Unspecified,,,,3995295,Station Wagon/Sport Utility Vehicle,,,,",
55.         "number_of_cyclist_injured" : 0,
56.         "input" : {
57.             "type" : "log"
58.         },
59.         "@timestamp" : "2018-10-06T14:45:00.000-05:00",
60.         "cross_street_name" : "",
61.         "number_of_pedestrians_injured" : 0,
62.         "number_of_persons_injured" : 0,
63.         "location" : "40.785183, -73.97512"
64.     }
65. }
66. ]
67. }
68. }

```

For the readability of the rest of the document, we will not present the results of the search queries from now on.

Note: Ranking Documents

You can notice in line 18 there is a score field returned. Scoring is a feature much needed in search engine databases to rank documents by their significance. This score is calculated against the documents in Elasticsearch based on the provided queries. Factors such as the length of a field, how often the specified term appears in the field, and (in the case of wildcard and fuzzy searches) how closely the term matches the specified value all influence the score. The calculated score is then used to order documents, usually from the highest score to lowest, and the highest scoring documents are then returned to the client. One can influence the scores of different queries in various ways, for example by using the boost parameter for specific word in the search query. This is especially useful if you want certain queries in a complex query to carry more weight than others and you are looking for the most significant documents.

By leveraging the Lucene syntax, we can build some impressive searches, such as a fuzzy search alternative of the previous search (note that now we are searching MAHNATTAN, which doesn't exist in the database, instead of MANHATTAN), but specify that a distance of 1 from the searched word is allowed (MAHNATTAN~1).

```
1. curl "http://localhost:9200/nyc_visionzero/_search?size=1;q=borough:MAHNATTAN~1"
```

Several options are available that allow you to customize the URI search by adding and/or clauses to constrain your searches, using regex expressions, wildcards or range queries. Moreover you can specify which analyzer to use (analyzer), whether the query should be fault-tolerant (lenient), and whether an explanation of the scoring should be provided (explain).

Although the URI search is a simple and efficient way to query your cluster, you'll quickly find that it doesn't support the full capacity of Elasticsearch. To profit from it, you must use Request Body Search. Using Request Body Search allows you to build a complex search request using various elements and query clauses that will match, filter, and order as well as manipulate documents based on multiple criteria.

3.4.2 Request Body Search

Request Body Search uses a JSON document that contains various elements to create a search. Not only can you specify search criteria, you can also specify the range and number of documents that you expect back, the fields that you want, and various other options. This is done using the Query DSL language.

Let's see a complex example, that will use some of the most used capabilities of Query DSL.

```
1. curl -XGET "http://localhost:9200/nyc_visionzero/_search" -H 'Content-Type: application/json' -d'
```

```
1. {
2.   "size": 500,
3.   "query": {
4.     "bool": {
5.       "must": [
6.         {
7.           "range": {
8.             "@timestamp": {
9.               "gte": "15/09/2018",
10.              "lte": "20/12/2018",
11.              "format": "dd/MM/yyyy"
12.            }
13.          }
14.        },
15.        {
16.          "fuzzy": {
17.            "borough": "MAHNATTAN"
18.          }
19.        }
20.      ],
21.      "should": [
22.        {
23.          "match": {
24.            "message": {
25.              "query": "PLACE"
26.            }
27.          }
28.        }
29.      ],
30.      "must_not": [
31.        {
32.          "match": {
33.            "message": {
34.              "query": "west avenue"
35.            }
36.          }
37.        }
38.      ]
39.    }
40.  }
41. }'
```

The above query returns the 500 most relevant accident documents that:

- Must have: (The clause must appear in matching documents and will contribute to the score)
 - A timestamp ranging from 15/09/2018 to 20/12/2018
 - A field borough that can be fuzzily matched to the word “MAHNATTAN”
- Should have (The clause should appear in matching documents and will contribute to the score. If must doesn't return anything, this can match documents, otherwise it just contributes to the scores)
 - Matching for the word “PLACE” in the field message (this field contains the csv row input as text)
- Must not have (The clause must not appear in matching documents)
 - Matching in the field message for the string “west avenue”

4 CONCLUSION

With the size of unstructured text data in the modern world, searching engine databases have become quite relevant. Smart data structures based on inverted indexes allow them to quickly perform full text analytics, document ranking based on relevance and a whole range of features needed to search text.

Elasticsearch is a powerful full-fledged search engine tool that offers a wide range of features that constantly get expanded within the Elastic Stack ecosystem. It is easy to get started, as it is very straightforward to add data with Logstash and create interactive visualizations of them with Kibana. With updated versions it gets even more features related to data analytics such as machine learning, time series analysis, rollup queries and geographical queries to name a few. Elastic products family is certainly a very promising engine that develops rapidly over the years, having already created a big enthusiastic open source community.