ElasticSearch

BY:

RUJUL PAREKH	18030242017
IMRAN QURAISHI	18030242021
OMKAR SATHAYE	18030242022
AKSHAY SAWANT	18030242023
SIDDHESH BHURKE	18030242036

CONTENTS

Page no.

1. Introduction…………………………………………………………………………..3
2. Architecture…………………………………………………………………………5
3. Implementation……………………………………………………………………..19
4. Applications…………………………………………………………………………25
5. Case Study……………………………………………………………………………27
6. Key benefits…………………………………………………………………………29
7. References…………………………………………………………………………..30

Introduction

Elastic Search is a cross platform scalable search solution which uses JSON over HTTP for user friendly. It was created by Shay Banon 9 years ago (Feb-8th 2010) as a successor to Compass-3 software. Elasticsearch BV was founded in 2012 to provide commercial services and products around Elasticsearch and related software. In June 2014, the company announced raising $70 million in a Series C funding round, just 18 months after forming the company. The round was led by New Enterprise Associates (NEA). Additional funders include Benchmark Capital and Index Ventures. This round brings total funding to $104M.

In March 2015, the company Elasticsearch changed their name to Elastic. In June 2018, Elastic filed for an initial public offering with an estimated valuation of between 1.5 and 3 billion dollars. On 5 October 2018, Elastic was listed on the New York Stock Exchange.

Elastic Search is distributed and can be used to search all kinds of documents. It provides scalable search, has near real-time search, and supports multitenancy. Customers looking for product information from the businesses with huge product and client base are facing the issues such as a long time in product information retrieval. This leads to poor user experience and in turn missing the potential customer.

Lag in search is attributed to the relational database used for the product design, where the data is scattered among multiple tables and retrieval of meaningful user information require fetching the data from them. The Relational Database works comparatively slow when it comes to huge data and fetching search results through queries from the database. Businesses nowadays looking for alternate ways where the data stored in such a way that the retrieval is quick. This can be achieved by adopting NOSQL rather than RDBMS for storing data. Elasticsearch is one such NOSQL distributed database. Elasticsearch relies on flexible data models to build and update visitors profiles to meet the demanding workload and low latency required for real-time engagement.

Relational database works comparatively slow when it comes to huge data and fetching search results through queries from the database. (There are ways to optimize this like indexing but then there are related limitations like we can't index every field. Row updates to heavily indexed tables would take time. People also scale their RDBMS vertically to improve performance.) This is a problem is overcome by Elasticsearch.

To perform fast querying Elastic Search uses 3 main components:

1. ElasticSearch : In elastic search the data is loaded and we can perform search operations on the data with ease and less lag. This component is executed first.

2. Logstack : Logstack is used to load the data in Elastic Search. So basically logstack is used to stack the data sequentially into elastic search so that it becomes easier for the elastic search engine to perform queries on the data and extract relevant information.

3. Kibana : Every tool requires a visualisation interface to make results interpreted easily. In elastic search we have Kibana which performs visualisations on the data. Kibana provides the user with an easy to understand interface and many graphical methods to apply on the data that will make data visualisation easy to interpret.

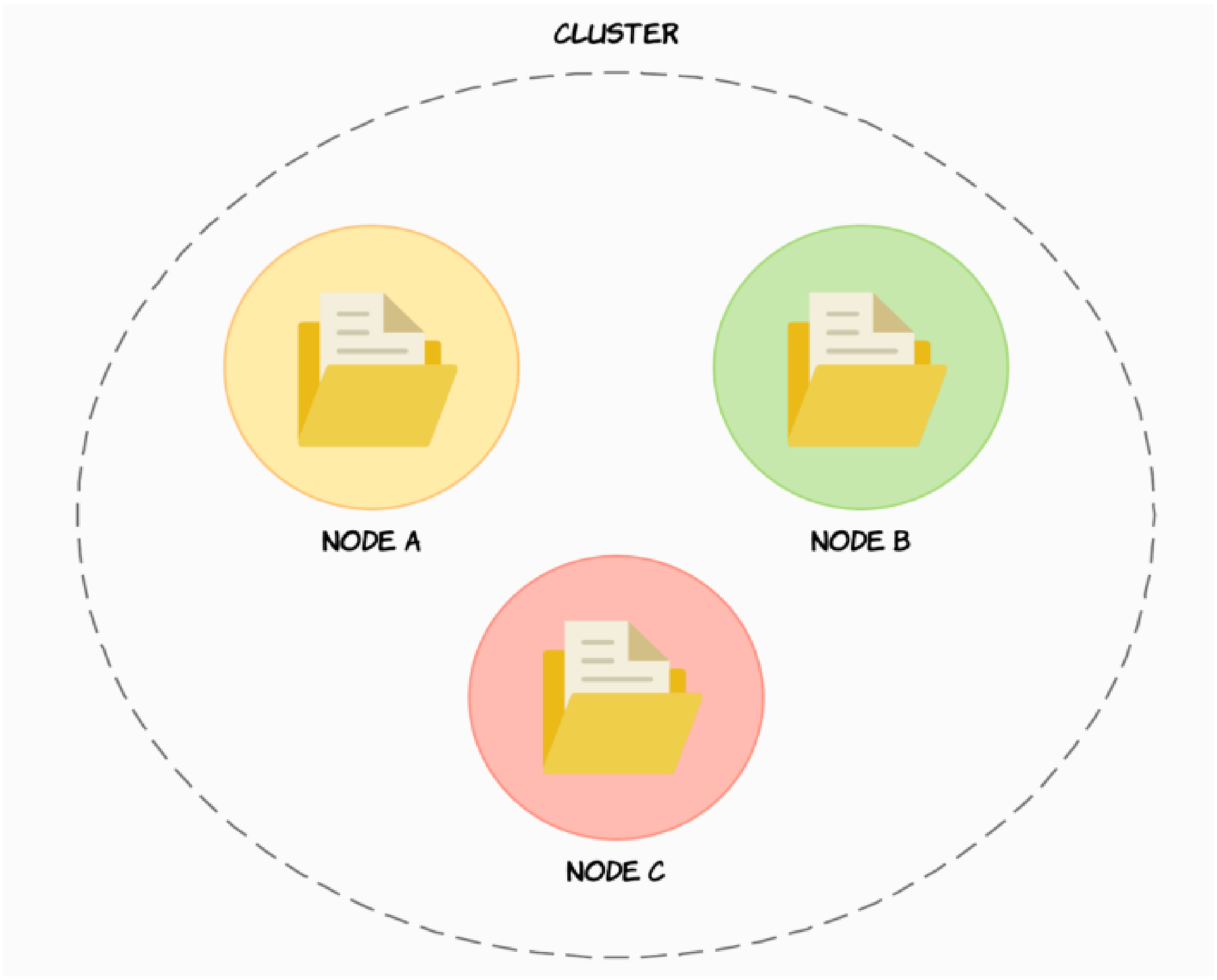
Key Features of Elastic Search:

- In elastic search indices can be divided into shards and each shard can have zero or more replicas. Each node hosts one or more shards, and acts as a coordinator to delegate operations to the correct shard(s). Rebalancing and routing are done automatically". Related data is often stored in the same index, which consists of one or more primary shards, and zero or more replica shards. Once an index has been created, the number of primary shards cannot be changed.
- Elasticsearch is developed alongside a data collection and log-parsing engine called Logstash, and an analytics and visualisation platform called Kibana. The three products are designed for use as an integrated solution, referred to as the "Elastic Stack" (formerly the "ELK stack").
- Elasticsearch uses Lucene and tries to make all its features available through the JSON and Java API. It supports faceting and percolating, which can be useful for notifying if new documents match for registered queries.
- Another feature is called "gateway" and handles the long-term persistence of the index for example, an index can be recovered from the gateway in the event of a server crash. Elasticsearch supports real-time GET requests, which makes it suitable as a NoSQL datastore, but it lacks distributed transactions.

Elasticsearch Architecture

1. Nodes & Clusters

To start things off, we will begin by talking about nodes and clusters, which are at the centre of the Elasticsearch architecture. A node is a server (either physical or virtual) that stores data and is part of what is called a cluster. A cluster is a collection of nodes, i.e. servers, and each node contains a part of the cluster's data, being the data that you add to the cluster. The collection of nodes therefore contains the entire data set for the cluster.



Each node participates in the indexing and searching capabilities of the cluster, meaning that a node will participate in a given search query by searching the data that it stores. For example, you might have some data on Node A and some other data on Node B, and both pieces of data match a given query. every node within the cluster can handle HTTP requests for clients that want to send a request to the cluster. This is done by using the HTTP REST API that the cluster exposes. A given node then receives this request and will be responsible for coordinating the rest of the work. Each node may also be assigned as being the so-called master node by default.

A master node is the node that is responsible for coordinating changes to the cluster, such as adding or removing nodes, creating or removing indices, etc. This master node updates the state of the cluster and it is the only node that may do this.

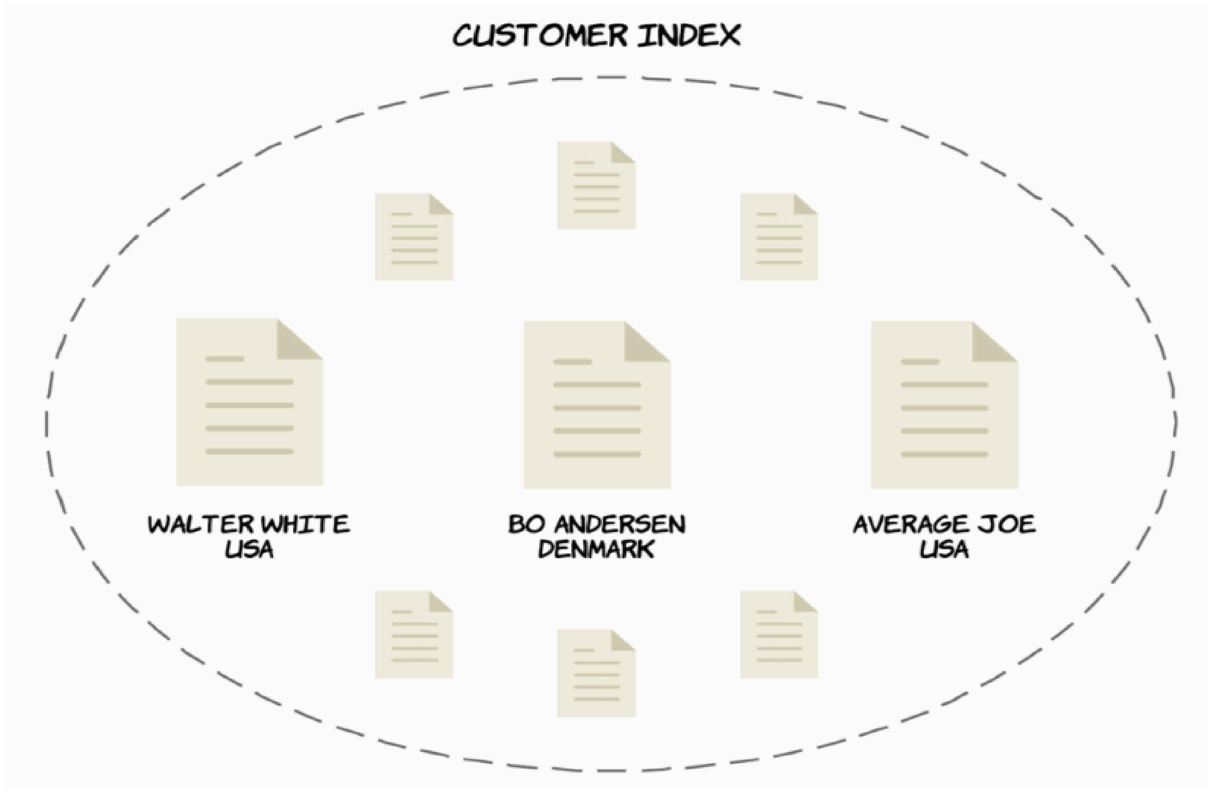
Both clusters and nodes are identified by unique names. For clusters, the default name is elasticsearch in all lowercase letters, and the default name for nodes is a Universally Unique Identifier, also referred to as a UUID. If you want or need to, you can change this default behavior. The names of nodes are important because that is how you can identify which physical or virtual machines correspond to which Elasticsearch nodes. By default, nodes join a cluster named elasticsearch, but you can configure nodes to join a specific cluster by specifying its name. However, the default behavior means that if you start up a number of nodes on your network, they will automatically join a cluster named elasticsearch. And, if no cluster already exists with that name, it will be formed.

You can have as many nodes running within a cluster that you want, and it is perfectly valid to have a cluster with only one node. The architecture of Elasticsearch is extremely scalable, particularly due to sharding (explained later), so scalability is not going to be an issue for you unless you are dealing with huge amounts of data. There are clusters out there with several terabytes of data, so chances are that this won't be a problem for you.

2. Indices & Documents:

Now that you know what clusters and nodes are, let's take a closer look at how data is organized and stored. Each data item that you store within your cluster is called a document, being a basic unit of information that can be indexed. Documents are JSON objects and would correspond to rows in a relational database. So, if you wanted to store a person, you could add an object with the name and country properties. You already know that data is stored across all of the nodes in the cluster, but how are the documents organized? Documents are stored within something called indices. An index is a collection of documents that have somewhat similar characteristics, i.e. are logically related. An example would be to have an index for product data, one for customer data, and one for orders.

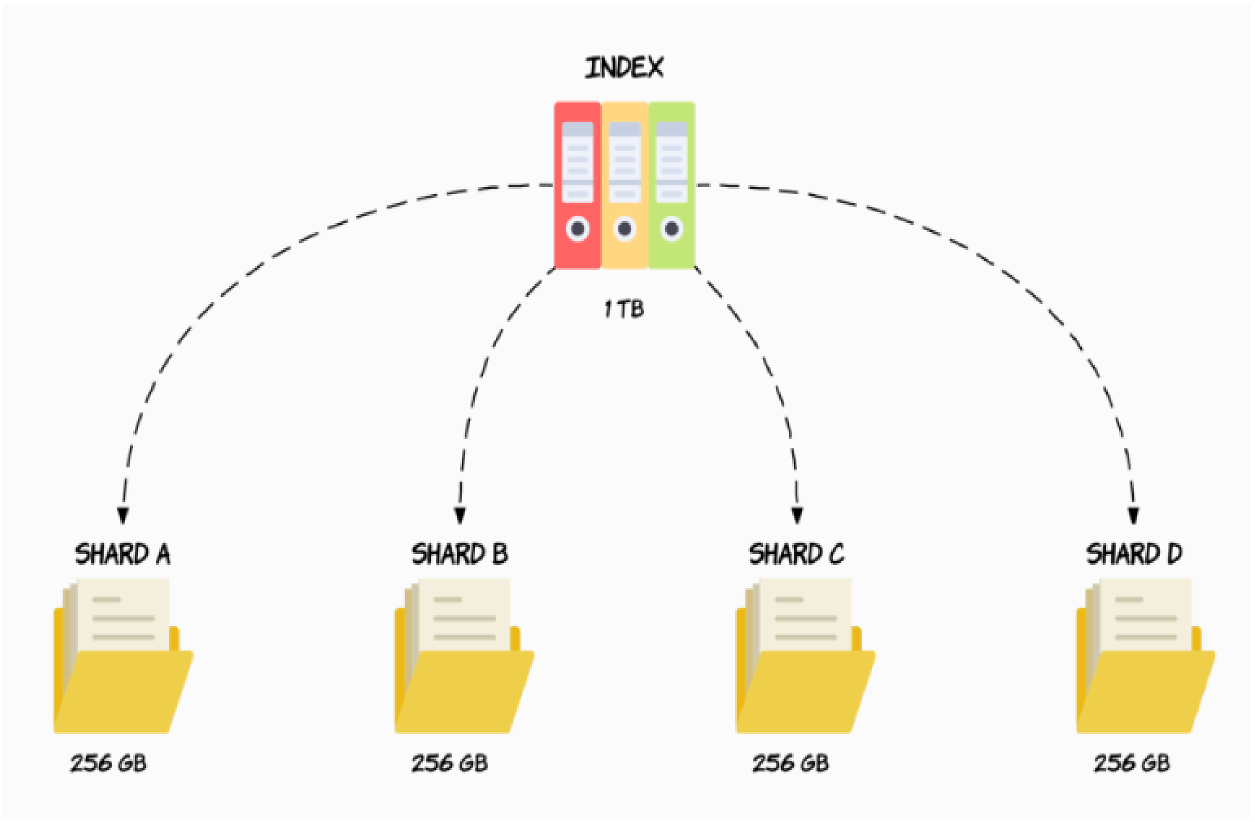
Documents have IDs assigned to them either automatically by Elasticsearch, or by you when adding them to an index. A document is uniquely identified by the index and its ID. You can add as many documents as you want to an index. As with clusters and nodes, indices are also identified by names, which must be in all lowercased letters. These names are then used when searching for documents, in which case you would specify the index to search through for matching documents. The same applies for adding, removing and updating documents.



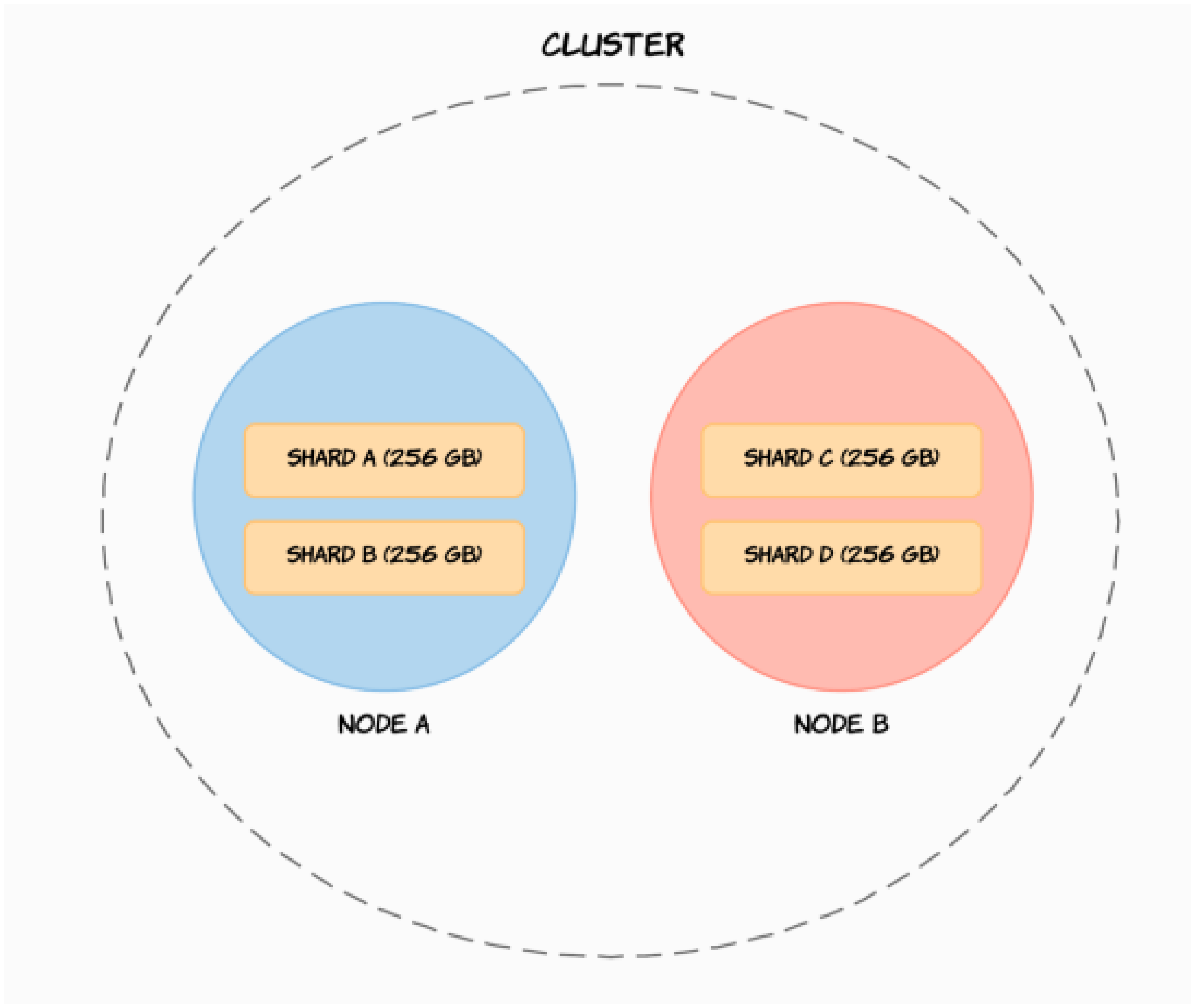
3. Sharding in Elasticsearch:

Elasticsearch is extremely scalable due to its distributed architecture. One of the reasons this is the case, is due to something called sharding.

Before getting into what sharding is, let's first talk about why it is needed in the first place. Suppose that you have an index containing lots of documents, totalling 1 terabyte of data. You have two nodes in your cluster, each with 512 gigabytes available for storing data. Clearly the entire index will not fit on either of the nodes, so splitting the index' data up somehow is necessary, or we would effectively be out of disk space. In scenarios like this where the size of an index exceeds the hardware limits of a single node, sharding comes to the rescue. Sharding solves this problem by dividing indices into smaller pieces named shards. So, a shard will contain a subset of an index' data and is in itself fully functional and independent, and you can kind of think of a shard as an "independent index." This is not entirely accurate, hence why I put that in quotation marks, but it's a decent way to think about it nevertheless. When an index is sharded, a given document within that index will only be stored within one of the shards.



The great thing about shards, is that they can be hosted on any node within the cluster. That being said, an index' shards will not necessarily be distributed across multiple physical or virtual machines, as this depends on the number of nodes in your cluster. So, in the case of the previous example, we could divide the 1 terabyte index into four shards, each containing 256 gigabytes of data, and these shards could then be distributed across the two nodes, meaning that the index as a whole now fits with the disk capacity that we have available.



There are two main reasons why sharding is important, with the first one being that it allows you to split and thereby scale volumes of data. So, if you have growing amounts of data, you will not face a bottleneck because you can always tweak the number of shards for a particular index. I will get back to how to specify the number of shards in just a moment. The other reason why sharding is important, is that operations can be distributed across multiple nodes and thereby parallelized. This results in increased performance, because multiple machines can potentially work on the same query. This is completely transparent to you as a user of Elasticsearch.

So how do you specify the number of shards an index has? You can optionally specify this at index creation time, but if you don't, a default number of 5 will be used. This is sufficient in most cases, since it allows for a good amount of growth in data before you need to worry about adding additional shards. How long it will take before you have to worry about that depends on the amount of data that is stored within a particular index and the hardware that you have available, i.e. the number of nodes and the amount of disk space. Of course, other indices and their amount of data comes into play as well, so how many shards you want depends on a couple of factors.

That being said, a default of 5 shards will get you a long way, and you won't have to deal with sharding yourself for quite a while unless you are already dealing with large volumes of data.

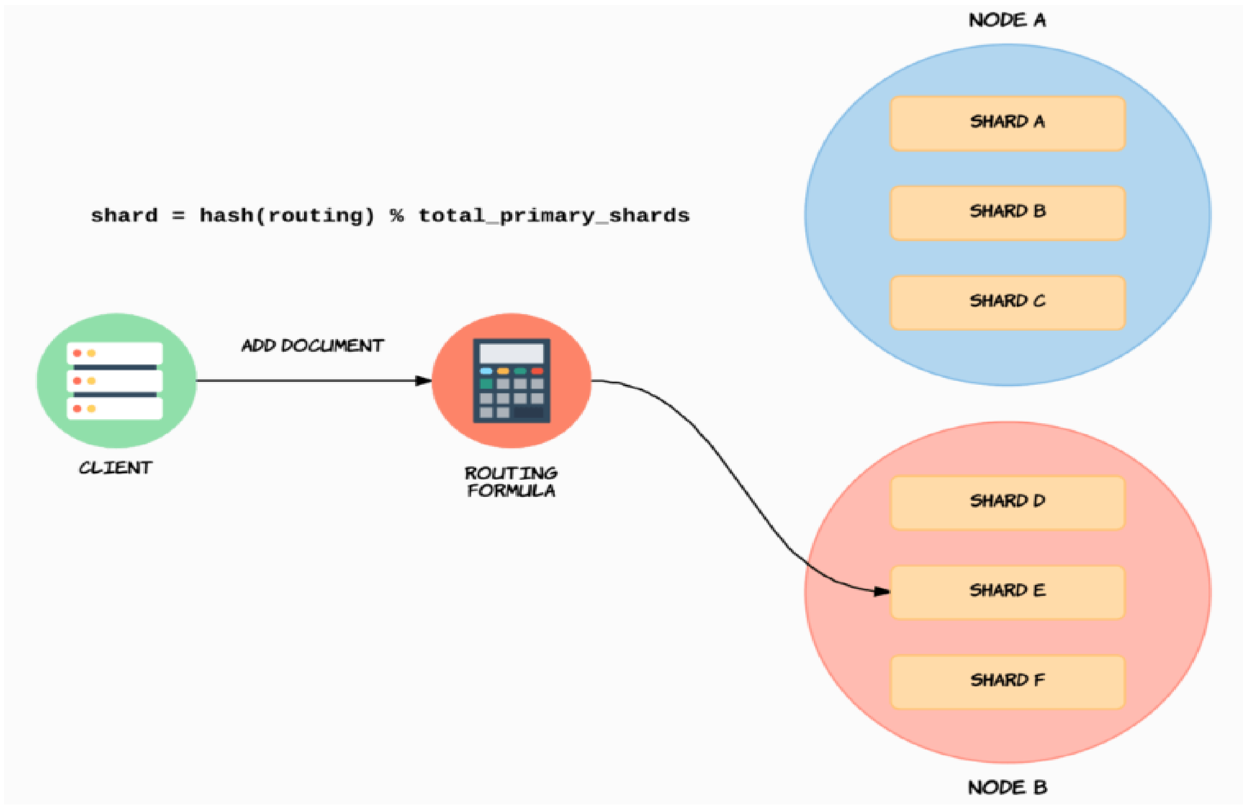
But what if you do need to change the number of shards for an index? If the index has already been created, you unfortunately cannot change the number of shards. What you would do instead, is to create a new index with the number of shards that you want and move your data over to the new index. Chances are that you will never have to do this if you are a developer, so you typically won't have to worry about it. Nevertheless, that is how you can change the number of shards for an index if you need to.

4. Distributing Documents across Shards (Routing):

We learned how data is stored on potentially more than one node in a cluster, and also how that is accomplished with sharding. But how does Elasticsearch know on which shard to store a new document, and how will it find it when retrieving it by ID? There needs to be a way of determining this, because surely it cannot be random. And also, documents should be distributed evenly between nodes by default, so that we won't have one shard containing way more documents than another. So, determining which shard a given document should be stored in or has been stored is, is called routing.

To make Elasticsearch as easy to use as possible, routing is handled automatically by default, and most users won't need to manually deal with it. The way it works by default, is that Elasticsearch uses a simple formula for determining the appropriate shard.

By default, the "routing" value will equal a given document's ID. This value is then passed through a hashing function, which generates a number that can be used for the division. The remainder of dividing the generated number with the number of primary shards in the index, will give the shard number. This is how Elasticsearch determines the location of specific documents. When executing search queries (i.e. not looking a specific document up by ID), the process is different, as the query is then broadcasted to all shards.

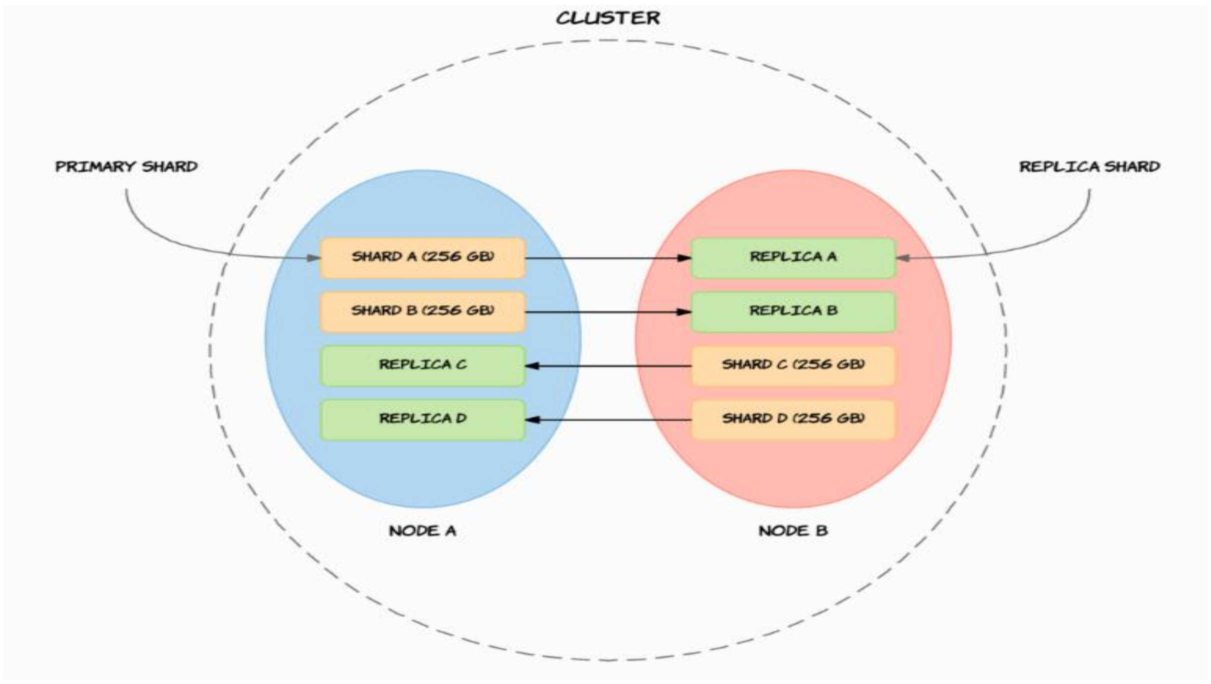


Remember how I mentioned that the number of shards for an index cannot be changed once an index has been created? Taking the routing formula into consideration, then we have the answer as to why this is the case. If we were to change the number of shards, then the result of running the routing formula would change for documents. Consider an example where a document has been stored on Shard A when we had five shards, because that is what the outcome of the routing formula was at the time. Suppose that we were able to change the number of shards, and that we changed it to seven. If we try to lookup the document by ID, the result of the routing formula might be different. Now the formula might route to Shard B, even though the document is actually stored on Shard A. This means that the document would never be found, and that would really cause some headaches. So that's why the number of shards cannot be changed once an index has been created, so you would have to create a new index and move the documents to it.

5. Replication in Elasticsearch:

Hardware can fail at any time, and software can be buggy at times. Let's face it, sometimes things just stop working. The more hardware capacity you add, the higher the risk that some hardware stops working, such as a hard drive breaking. Taking that into consideration, it's probably a good idea to have some kind of fault tolerance and failover mechanism in place. That's where replication comes into the picture.

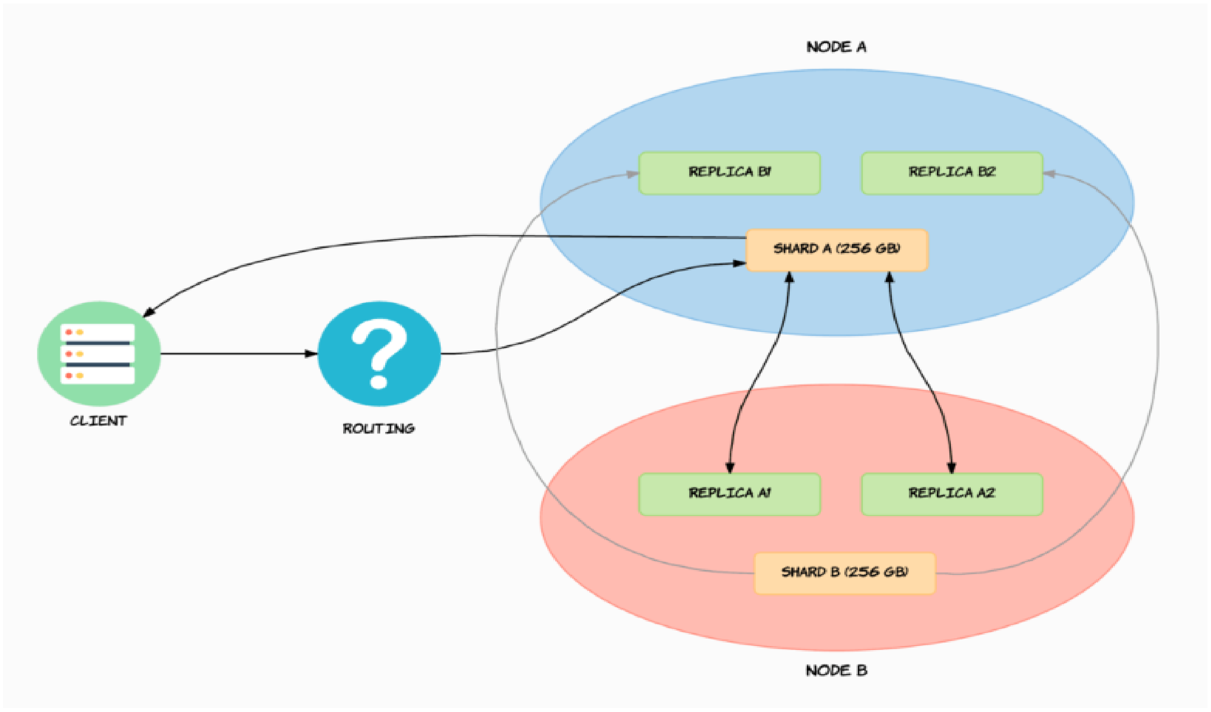
Elasticsearch natively supports replication of your shards, meaning that shards are copied. When a shard is replicated, it is referred to as either a replica shard, or just a replica if you are feeling lazy. The shards that have been replicated are referred to as primary shards. A primary shard and its replicas is referred to as a replication group.



The example that you see above, is with an index of one terabyte divided into four shards of each 256 gigabytes. The shards are now primary shards and each have a replica shard.

Replication serves two purposes, with the main one being to provide high availability in case nodes or shards fail. For replication to even be effective if something goes wrong, replica shards are never allocated to the same nodes as the primary shards, which you can also see on the above diagram. This means that even if an entire node fails, you will have at least one replica of any primary shards on that particular node. The other purpose of replication — or perhaps a side benefit — is increased performance for search queries. This is the case because searches can be executed on all replicas in parallel, meaning that replicas are actually part of the cluster's searching capabilities. Replicas are therefore not exclusively used for availability purposes, although that is often the primary motivation for using replication. As with shards, the number of replicas is defined when creating an index. The default number of replicas is one, being one for each shard. This means that by default, a cluster consisting of more than one node, will have 5 primary shards and 5 replicas, totalling 10 shards per index. This makes up a complete replica of your data, so either of the nodes can have a disk failure without you losing any data. The reason I said that the cluster should have more than one node, is that replicas are never stored on the same node as the primary shard that it is a replica of, as I mentioned before.

As with shards, the number of replicas is defined when creating an index. The default number of replicas is one, being one for each shard. This means that by default, a cluster consisting of more than one node, will have 5 primary shards and 5 replicas, totalling 10 shards per index. This makes up a complete replica of your data, so either of the nodes can have a disk failure without you losing any data. The reason I said that the cluster should have more than one node, is that replicas are never stored on the same node as the primary shard that it is a replica of, as I mentioned before.



going through a simple example based on the above diagram. We have a cluster with two nodes. We only have a single index consisting of two shards, each with two replicas. We have a client on the left-hand side, which would typically be a server communicating with the cluster. In this case, we want to delete a document from the index. At this point, Elasticsearch needs to find the correct replication group, and thereby also the primary shard. This is done with so-called routing, which is not something that we will get into right now, so you can consider that a black box. Just know that something happens there that finds the appropriate replication group and its primary shard – Shard A in this example. The operation is then routed to the primary shard where it is validated and then executed. Once the operation completes on the primary shard itself, the operation is sent to the replica shards within the replication group. In this case that means that the delete operation is sent to Replica A1 and Replica A2. When the operation successfully completes on both of these replicas, the primary shard — i.e. Shard A — acknowledges that the request was successful to the client.

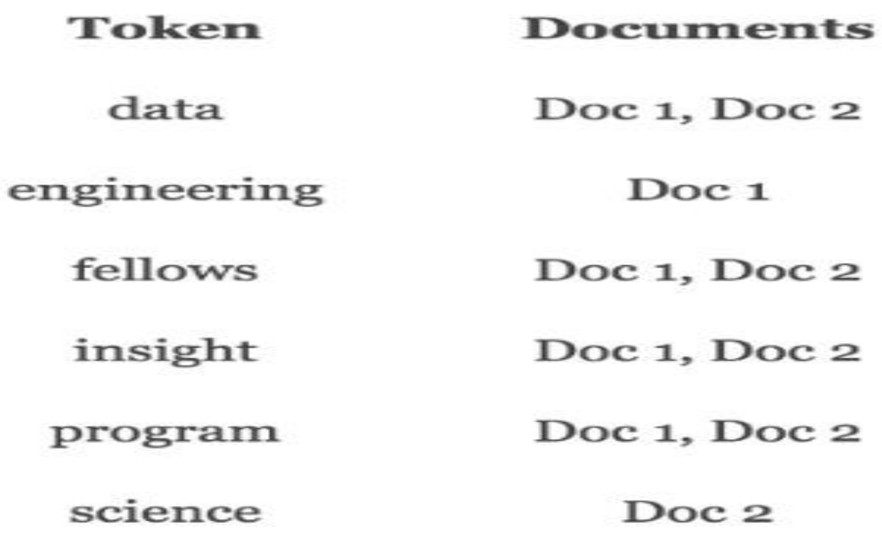
6. Storage Model:

Elasticsearch uses Apache Lucene, a full-text search library written in Java and developed by Doug Cutting (creator of Apache Hadoop), internally which uses a data structure called an inverted index designed to serve low latency search results. A document is the unit of data in Elasticsearch and an inverted index is created by tokenizing the terms in the document, creating a sorted list of all unique terms and associating a list of documents with where the word can be found.

It is very similar to an index at the back of a book which contains all the unique words in the book and a list of pages where we can find that word. When we say a document is indexed, we refer to the inverted index. Let's see how inverted index looks like for the following two documents:

Doc 1: Insight Data Engineering Fellows Program

Doc 2: Insight Data Science Fellows Program



Token	Documents
data	Doc 1, Doc 2
engineering	Doc 1
fellows	Doc 1, Doc 2
insight	Doc 1, Doc 2
program	Doc 1, Doc 2
science	Doc 2

If we want to find documents which contain the term "insight", we can scan the inverted index (where words are sorted), find the word "insight" and return the document IDs which contain this word, which in this case would be Doc 1 and Doc 2.

To improve searchability (e.g., serving same results for both lowercase and uppercase words), the documents are first analyzed and then indexed. Analyzing consists of two parts:

- Tokenizing sentences into individual words
- Normalizing words to a standard form

By default, Elasticsearch uses Standard Analyzer, which uses

- Standard tokenizer to split words on word boundaries
- Lowercase token filter to convert words to lowercase

There are many other analyzers available and you can read about them in the docs.

In order to serve relevant search results, every query made on the documents is also analyzed using the same analyzer used for indexing.

NOTE: The standard analyzer also uses stop token filter but it is disabled by default.

7. Anatomy of a Write:
   (C)reate

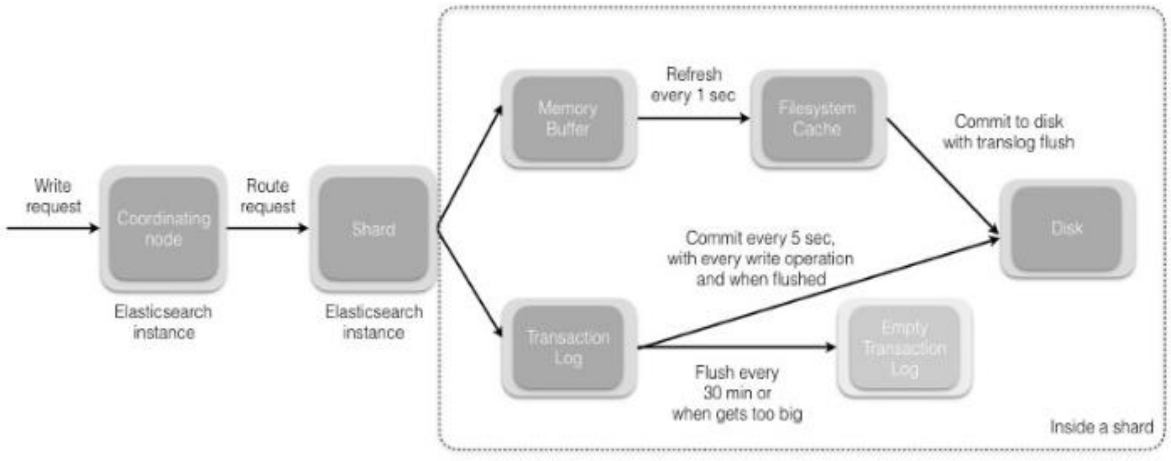
   When you send a request to the coordinating node to index a new document, the following set of operations take place:

   All the nodes in the Elasticsearch cluster contain metadata about which shard lives on which node. The coordinating node routes the document to the appropriate shard using the document ID (default). Elasticsearch hashes the document ID with murmur3 as the hash function and mods by the number of primary shards in the index to determine which shard the document should be indexed in.
   shard = hash(document_id) % (num_of_primary_shards)
   As the node receives the request from the coordinating node, the request is written to the translog (we'll cover translog in a follow up post) and the document is added to memory buffer. If the request is successful on the primary shard, the request is sent in parallel to the replica shards. The client receives acknowledgement that the request was successful only after the translog is fsync'ed on all primary and replica shards.

The memory buffer is refreshed at a regular interval (defaults to 1 sec) and the contents are written to a new segment in filesystem cache. This segment is not yet fsync'ed, however, the segment is open and the contents are available for search.

The translog is emptied and filesystem cache is fsync'ed every 30 minutes or when the translog gets too big. This process is called flush in Elasticsearch. During the flush process, the in-memory buffer is cleared and the contents are written to a new segment. A new commit point is created with the segments fsync'ed and flushed to disk. The old translog is deleted and a fresh one begins.

The figure below shows how the write request and data flows.



(U)pdate and (D)elete

Delete and Update operations are also write operations. However, documents in Elasticsearch are immutable and hence, cannot be deleted or modified to represent any changes. Then, how can a document be deleted/updated?

Every segment on disk has a .del file associated with it. When a delete request is sent, the document is not really deleted, but marked as deleted in the .del file. This document may still match a search query but is filtered out of the results. When segments are merged (we'll cover segment merging in a follow up post), the documents marked as deleted in the .del file are not included in the new merged segment.

Now, let's see how updates work. When a new document is created, Elasticsearch assigns a version number to that document. Every change to the document results in a new version number. When an update is performed, the old version is marked as deleted in the .del file and the new version is indexed in a new segment. The older version may still match a search query, however, it is filtered out from the results.

After the documents are indexed/updated, we would like to perform search requests. Let’s see how search requests are executed in Elasticsearch.

8. Anatomy of a (R)ead:

Read operations consist of two parts:

Query Phase

Fetch Phase

Let’s see how each phase works.

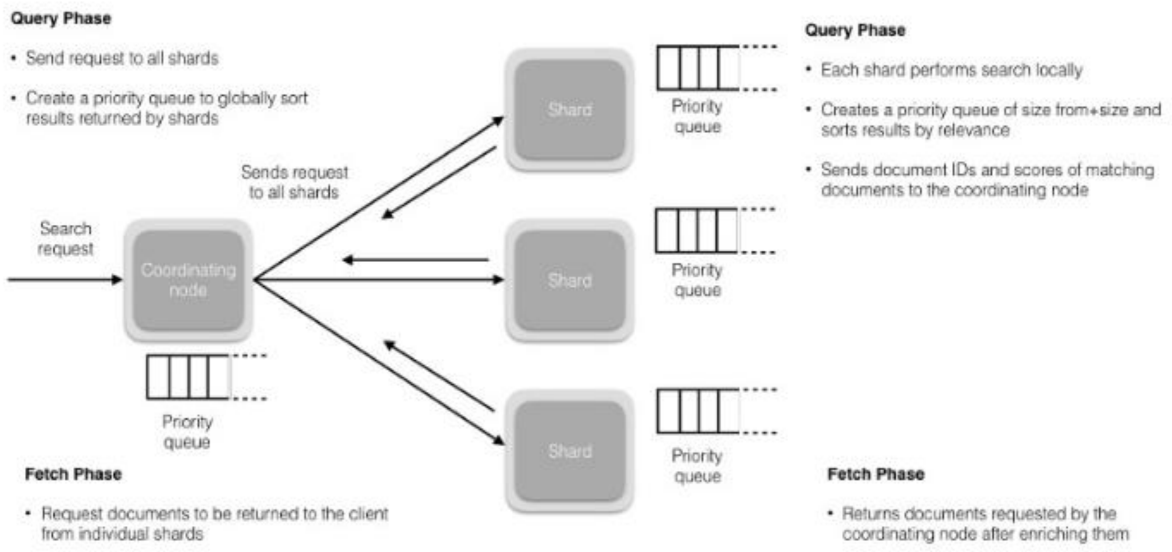
Query Phase

In this phase, the coordinating node routes the search request to all the shards (primary or replica) in the index. The shards perform search independently and create a priority queue of the results sorted by relevance score (we’ll cover relevance score later in the post). All the shards return the document IDs of the matched documents and relevant scores to the coordinating node. The coordinating node creates a new priority queue and sorts the results globally. There can be a lot of documents which match the results, however, by default, each shard sends the top 10 results to the coordinating node and the coordinating creates a priority queue sorting results from all the shards and returns the top 10 hits.

Fetch Phase

After the coordinating node sorts all the results to generate a globally sorted list of documents, it then requests the original documents from all the shards. All the shards enrich the documents and return them to the coordinating node.

The figure below shows how the read request and the data flows.


Query Phase
• Send request to all shards
• Create a priority queue to globally sort results returned by shards
Sends request to all shards
Search request
Coordinating node
Priority queue
Fetch Phase
• Request documents to be returned to the client from individual shards
Shard
Shard
Shard
Priority queue
Priority queue
Priority queue
Query Phase
• Each shard performs search locally
• Creates a priority queue of size from+size and sorts results by relevance
• Sends document IDs and scores of matching documents to the coordinating node
Fetch Phase
• Returns documents requested by the coordinating node after enriching them

Implementation

Step 1:

Install Elasticsearch, Kibana and logstash 6.5.4 from https://www.elastic.co/downloads.

Step 2:

Unzip the files into a folder.

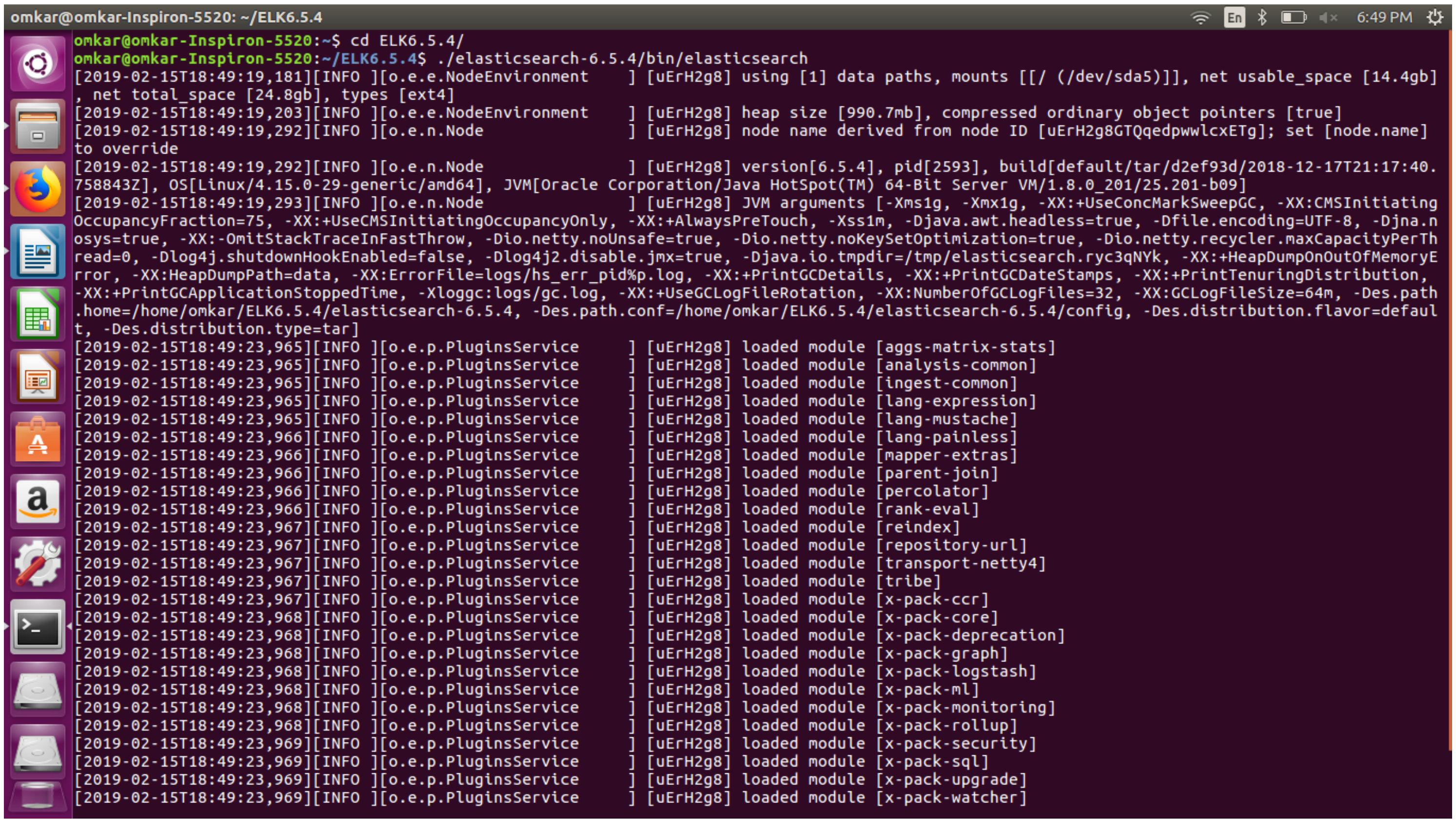
Step 3:

Download a dataset on which you want to perform operations. Here we have used BlackFriday dataset from Kaggle.

Step 4:

Start the elastic search engine.

Command: ./elasticsearch-6.5.4/bin/elasticsearch



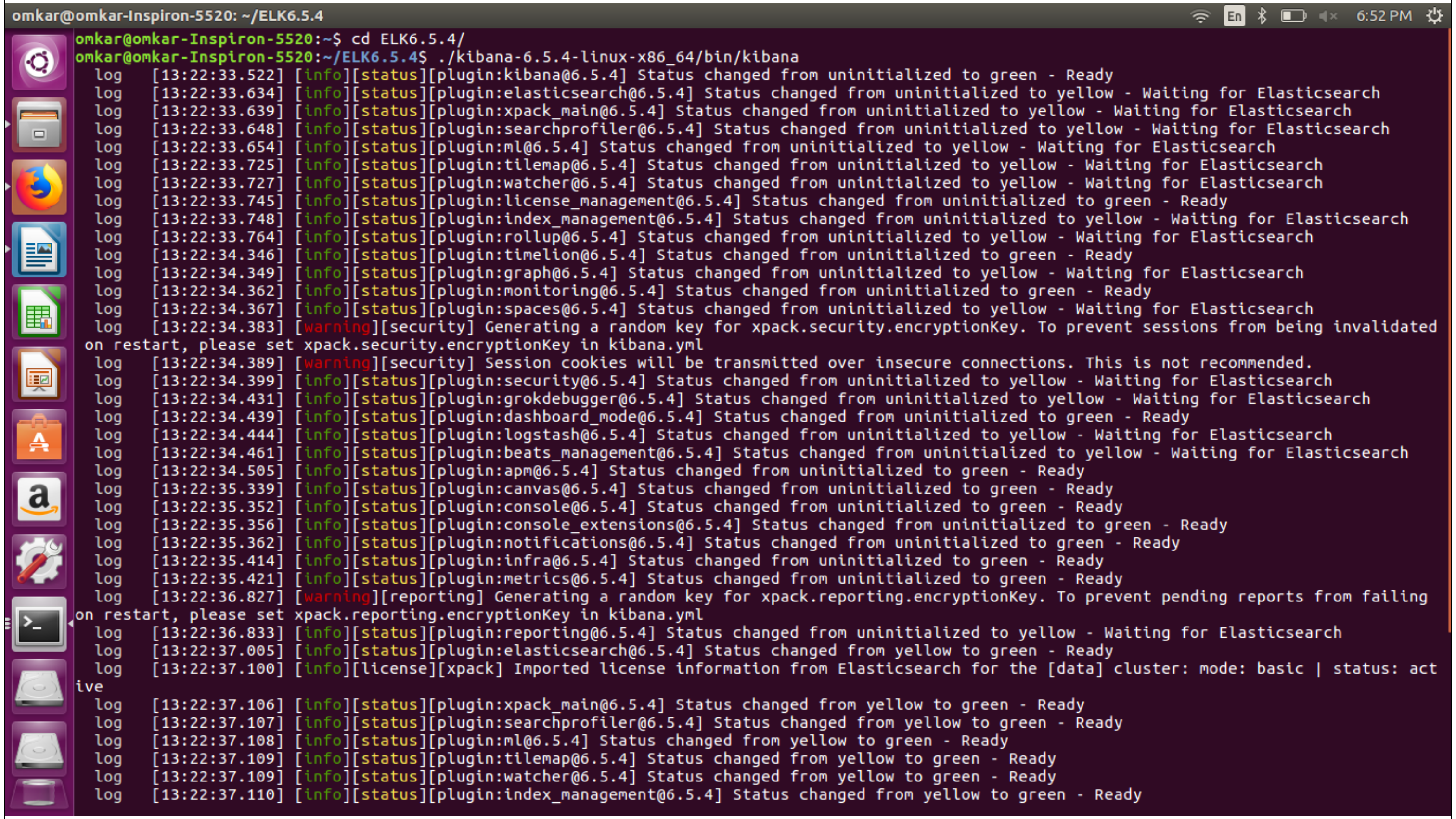
```
omkar@omkar-Inspiron-5520: ~/ELK6.5.4
omkar@omkar-Inspiron-5520:~$ cd ELK6.5.4/
omkar@omkar-Inspiron-5520:~/ELK6.5.4$ ./elasticsearch-6.5.4/bin/elasticsearch
[2019-02-15T18:49:19,181][INFO ][o.e.e.NodeEnvironment    ] [uErH2g8] using [1] data paths, mounts [[/ (/dev/sda5)]], net usable_space [14.4gb]
, net total_space [24.8gb], types [ext4]
[2019-02-15T18:49:19,203][INFO ][o.e.e.NodeEnvironment    ] [uErH2g8] heap size [990.7mb], compressed ordinary object pointers [true]
[2019-02-15T18:49:19,292][INFO ][o.e.n.Node               ] [uErH2g8] node name derived from node ID [uErH2g8GTQqedpwwlcxETg]; set [node.name]
to override
[2019-02-15T18:49:19,292][INFO ][o.e.n.Node               ] [uErH2g8] version[6.5.4], pid[2593], build[default/tar/d2ef93d/2018-12-17T21:17:40.
758843Z], OS[Linux/4.15.0-29-generic/amd64], JVM[Oracle Corporation/Java HotSpot(TM) 64-Bit Server VM/1.8.0_201/25.201-b09]
[2019-02-15T18:49:19,293][INFO ][o.e.n.Node               ] [uErH2g8] JVM arguments [-Xms1g, -Xmx1g, -XX:+UseConcMarkSweepGC, -XX:CMSInitiating
OccupancyFraction=75, -XX:+UseCMSInitiatingOccupancyOnly, -XX:+AlwaysPreTouch, -Xss1m, -Djava.awt.headless=true, -Dfile.encoding=UTF-8, -Djna.n
osys=true, -XX:-OmitStackTraceInFastThrow, -Dio.netty.noUnsafe=true, -Dio.netty.noKeySetOptimization=true, -Dio.netty.recycler.maxCapacityPerTh
read=0, -Dlog4j.shutdownHookEnabled=false, -Dlog4j2.disable.jmx=true, -Djava.io.tmpdir=/tmp/elasticsearch.ryc3qNYk, -XX:+HeapDumpOnOutOfMemoryE
rror, -XX:HeapDumpPath=data, -XX:ErrorFile=logs/hs_err_pid%p.log, -XX:+PrintGCDetails, -XX:+PrintGCDateStamps, -XX:+PrintTenuringDistribution,
-XX:+PrintGCApplicationStoppedTime, -Xloggc:logs/gc.log, -XX:+UseGCLogFileRotation, -XX:NumberOfGCLogFiles=32, -XX:GCLogFileSize=64m, -Des.path
.home=/home/omkar/ELK6.5.4/elasticsearch-6.5.4, -Des.path.conf=/home/omkar/ELK6.5.4/elasticsearch-6.5.4/config, -Des.distribution.flavor=defaul
t, -Des.distribution.type=tar]
[2019-02-15T18:49:23,965][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [aggs-matrix-stats]
[2019-02-15T18:49:23,965][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [analysis-common]
[2019-02-15T18:49:23,965][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [ingest-common]
[2019-02-15T18:49:23,965][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [lang-expression]
[2019-02-15T18:49:23,965][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [lang-mustache]
[2019-02-15T18:49:23,966][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [lang-painless]
[2019-02-15T18:49:23,966][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [mapper-extras]
[2019-02-15T18:49:23,966][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [parent-join]
[2019-02-15T18:49:23,966][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [percolator]
[2019-02-15T18:49:23,966][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [rank-eval]
[2019-02-15T18:49:23,967][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [reindex]
[2019-02-15T18:49:23,967][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [repository-url]
[2019-02-15T18:49:23,967][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [transport-netty4]
[2019-02-15T18:49:23,967][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [tribe]
[2019-02-15T18:49:23,967][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-ccr]
[2019-02-15T18:49:23,968][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-core]
[2019-02-15T18:49:23,968][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-deprecation]
[2019-02-15T18:49:23,968][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-graph]
[2019-02-15T18:49:23,968][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-logstash]
[2019-02-15T18:49:23,968][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-ml]
[2019-02-15T18:49:23,968][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-monitoring]
[2019-02-15T18:49:23,968][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-rollup]
[2019-02-15T18:49:23,969][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-security]
[2019-02-15T18:49:23,969][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-sql]
[2019-02-15T18:49:23,969][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-upgrade]
[2019-02-15T18:49:23,969][INFO ][o.e.p.PluginsService     ] [uErH2g8] loaded module [x-pack-watcher]
```

Fig: Starting Elastic Search Engine

Step 5:

Start Kibana

Command: ./kibana-6.5.4-linux-x86_64/bin/kibana

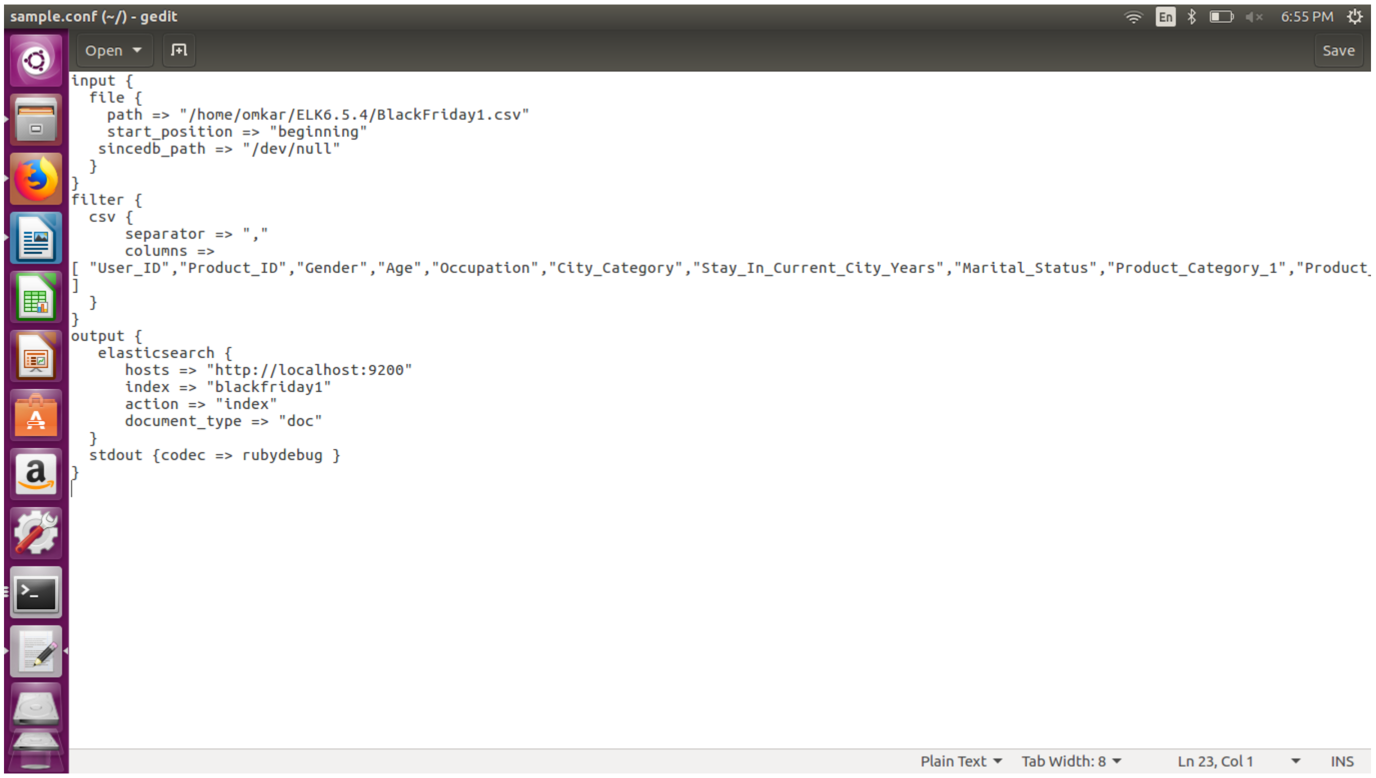


```
omkar@omkar-Inspiron-5520: ~/ELK6.5.4
omkar@omkar-Inspiron-5520:~$ cd ELK6.5.4/
omkar@omkar-Inspiron-5520:~/ELK6.5.4$ ./kibana-6.5.4-linux-x86_64/bin/kibana
  log   [13:22:33.522] [info][status][plugin:kibana@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:33.634] [info][status][plugin:elasticsearch@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:33.639] [info][status][plugin:xpack_main@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:33.648] [info][status][plugin:searchprofiler@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:33.654] [info][status][plugin:ml@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:33.725] [info][status][plugin:tilemap@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:33.727] [info][status][plugin:watcher@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:33.745] [info][status][plugin:license_management@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:33.748] [info][status][plugin:index_management@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:33.764] [info][status][plugin:rollup@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:34.346] [info][status][plugin:timelion@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:34.349] [info][status][plugin:graph@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:34.362] [info][status][plugin:monitoring@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:34.367] [info][status][plugin:spaces@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:34.383] [warning][security] Generating a random key for xpack.security.encryptionKey. To prevent sessions from being invalidated
 on restart, please set xpack.security.encryptionKey in kibana.yml
  log   [13:22:34.389] [warning][security] Session cookies will be transmitted over insecure connections. This is not recommended.
  log   [13:22:34.399] [info][status][plugin:security@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:34.431] [info][status][plugin:grokdebugger@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:34.439] [info][status][plugin:dashboard_mode@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:34.444] [info][status][plugin:logstash@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:34.461] [info][status][plugin:beats_management@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:34.505] [info][status][plugin:apm@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:35.339] [info][status][plugin:canvas@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:35.352] [info][status][plugin:console@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:35.356] [info][status][plugin:console_extensions@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:35.362] [info][status][plugin:notifications@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:35.414] [info][status][plugin:infra@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:35.421] [info][status][plugin:metrics@6.5.4] Status changed from uninitialized to green - Ready
  log   [13:22:36.827] [warning][reporting] Generating a random key for xpack.reporting.encryptionKey. To prevent pending reports from failing
on restart, please set xpack.reporting.encryptionKey in kibana.yml
  log   [13:22:36.833] [info][status][plugin:reporting@6.5.4] Status changed from uninitialized to yellow - Waiting for Elasticsearch
  log   [13:22:37.005] [info][status][plugin:elasticsearch@6.5.4] Status changed from yellow to green - Ready
  log   [13:22:37.100] [info][license][xpack] Imported license information from Elasticsearch for the [data] cluster: mode: basic | status: act
ive
  log   [13:22:37.106] [info][status][plugin:xpack_main@6.5.4] Status changed from yellow to green - Ready
  log   [13:22:37.107] [info][status][plugin:searchprofiler@6.5.4] Status changed from yellow to green - Ready
  log   [13:22:37.108] [info][status][plugin:ml@6.5.4] Status changed from yellow to green - Ready
  log   [13:22:37.109] [info][status][plugin:tilemap@6.5.4] Status changed from yellow to green - Ready
  log   [13:22:37.109] [info][status][plugin:watcher@6.5.4] Status changed from yellow to green - Ready
  log   [13:22:37.110] [info][status][plugin:index_management@6.5.4] Status changed from yellow to green - Ready
```

Fig: Starting Kibana

Step 6:

Writing a code to ingest Dataset into elastic search engine.



```
input {
  file {
    path => "/home/omkar/ELK6.5.4/BlackFriday1.csv"
    start_position => "beginning"
   sincedb_path => "/dev/null"
  }
}
filter {
  csv {
      separator => ","
      columns =>
[ "User_ID","Product_ID","Gender","Age","Occupation","City_Category","Stay_In_Current_City_Years","Marital_Status","Product_Category_1","Product_
]
  }
}
output {
   elasticsearch {
      hosts => "http://localhost:9200"
      index => "blackfriday1"
      action => "index"
      document_type => "doc"
  }
  stdout {codec => rubydebug }
}
```

Fig: Code

Code Explanation :

Code consists of three major parts:

- Input
- Filter
- Output

Input :

It consists the path of dataset. The position from where it should start.

Filter :

How CSV file will be seperated into different coloumns.

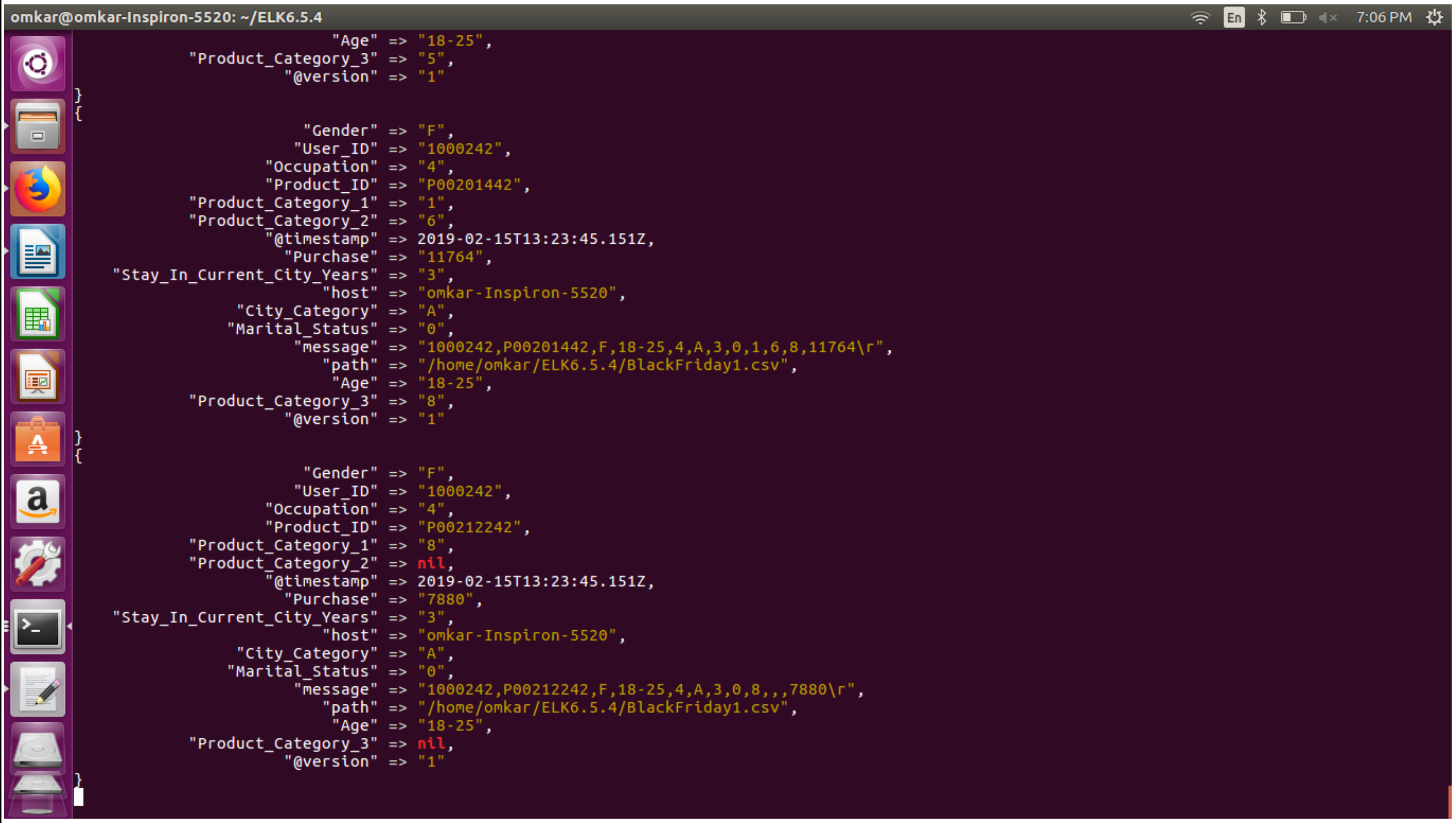
Output:

The most important part is index. The data will be uniqly identified by the index.

Step 7:

Starting Logstach and then ingesting the blackfriday dataset into search engine.

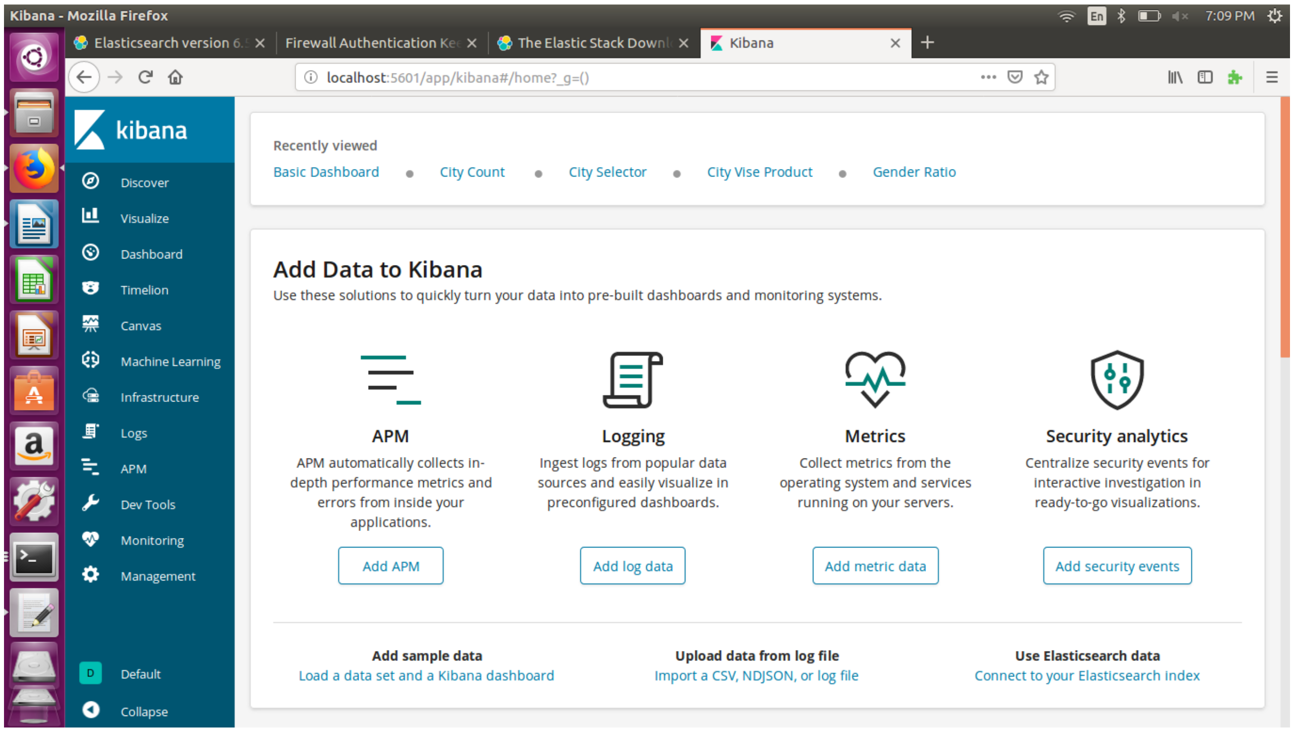
Command: ./logstash-6.5.4/bin/logstash -f /home/omkar/sample.conf



```
omkar@omkar-Inspiron-5520: ~/ELK6.5.4                                    7:06 PM
                     "Age" => "18-25",
        "Product_Category_3" => "5",
                  "@version" => "1"
}
{
                   "Gender" => "F",
                  "User_ID" => "1000242",
               "Occupation" => "4",
               "Product_ID" => "P00201442",
       "Product_Category_1" => "1",
       "Product_Category_2" => "6",
               "@timestamp" => 2019-02-15T13:23:45.151Z,
                 "Purchase" => "11764",
"Stay_In_Current_City_Years" => "3",
                     "host" => "omkar-Inspiron-5520",
            "City_Category" => "A",
           "Marital_Status" => "0",
                  "message" => "1000242,P00201442,F,18-25,4,A,3,0,1,6,8,11764\r",
                     "path" => "/home/omkar/ELK6.5.4/BlackFriday1.csv",
                      "Age" => "18-25",
       "Product_Category_3" => "8",
                 "@version" => "1"
}
{
                   "Gender" => "F",
                  "User_ID" => "1000242",
               "Occupation" => "4",
               "Product_ID" => "P00212242",
       "Product_Category_1" => "8",
       "Product_Category_2" => nil,
               "@timestamp" => 2019-02-15T13:23:45.151Z,
                 "Purchase" => "7880",
"Stay_In_Current_City_Years" => "3",
                     "host" => "omkar-Inspiron-5520",
            "City_Category" => "A",
           "Marital_Status" => "0",
                  "message" => "1000242,P00212242,F,18-25,4,A,3,0,8,,,7880\r",
                     "path" => "/home/omkar/ELK6.5.4/BlackFriday1.csv",
                      "Age" => "18-25",
       "Product_Category_3" => nil,
                 "@version" => "1"
}
```

Fig: Ingesting data into search engine.

Step 8:



Kibana - Mozilla Firefox

Elasticsearch version 6. | Firewall Authentication Ke | The Elastic Stack Downl | Kibana

localhost:5601/app/kibana#/home?_g=()

kibana

- Discover
- Visualize
- Dashboard
- Timelion
- Canvas
- Machine Learning
- Infrastructure
- Logs
- APM
- Dev Tools
- Monitoring
- Management
- Default
- Collapse

Recently viewed

Basic Dashboard • City Count • City Selector • City Vise Product • Gender Ratio

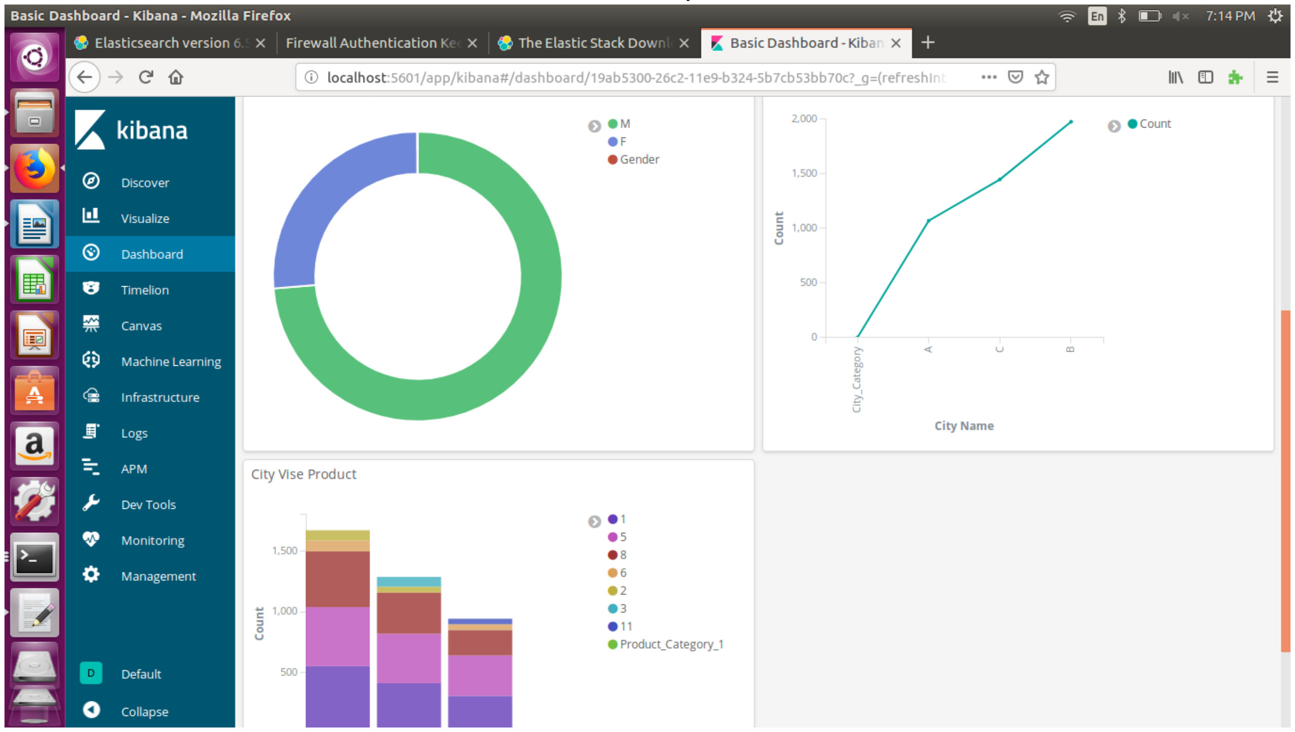
Add Data to Kibana

Use these solutions to quickly turn your data into pre-built dashboards and monitoring systems.

APM	Logging	Metrics	Security analytics
APM automatically collects in-depth performance metrics and errors from inside your applications.	Ingest logs from popular data sources and easily visualize in preconfigured dashboards.	Collect metrics from the operating system and services running on your servers.	Centralize security events for interactive investigation in ready-to-go visualizations.
Add APM	Add log data	Add metric data	Add security events

Add sample data	Upload data from log file	Use Elasticsearch data
Load a data set and a Kibana dashboard	Import a CSV, NDJSON, or log file	Connect to your Elasticsearch index

- Goto Management -> Index Pattern. Then select the index you want to perform operations on. Here I will select blackfriday1 index as I have mentioned that name in my
- code.
- Goto Visualize and select the visualizations you wish.



- Goto Dashboard and merge all these visulaizations.

Moreover, You can also interface Python with ElasticSearch. The enitire documentation along with example(code) can be found on https://elasticsearch-py.readthedocs.io/en/master/

Index API

The index API adds or updates a typed JSON document in a specific index, making it searchable. The following example inserts the JSON document into the "twitter" index, under a type called _doc with an id of 1:

```
PUT twitter/_doc/1
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
RESULT:
{
    "_shards" : {
        "total" : 2,
        "failed" : 0,
        "successful" : 2
    },
    "_index" : "twitter",
    "_type" : "_doc",
    "_id" : "1",
    "_version" : 1,
    "_seq_no" : 0,
    "_primary_term" : 1,
    "result" : "created"
}
```

SEARCH API

The search API allows you to execute a search query and get back search hits that match the query. The query can either be provided using a simple query string as a parameter, or using a request body.

For example, we can search on all documents within the twitter index:

```
GET /twitter/_search?q=user:kimchy
GET /kimchy,elasticsearch/_search?q=tag:wow
GET /_all/_search?q=tag:wow
```

Avg Aggregation

A single-value metrics aggregation that computes the average of numeric values that are extracted from the aggregated documents. These values can be extracted either from specific numeric fields in the documents, or be generated by a provided script.

Assuming the data consists of documents representing exams grades (between 0 and 100) of students we can average their scores with:

```
POST /exams/_search?size=0
{
    "aggs" : {
        "avg_grade" : { "avg" : { "field" : "grade" } }
    }
}
```

The above aggregation computes the average grade over all documents. The aggregation type is avg and the field setting defines the numeric field of the documents the average will be computed on. The above will return the following:

```
{
    ...
    "aggregations": {
        "avg_grade": {
            "value": 75.0
        }
    }
}
```

CASE STUDY

How is Maruti Techlabs using Elasticsearch for its client?

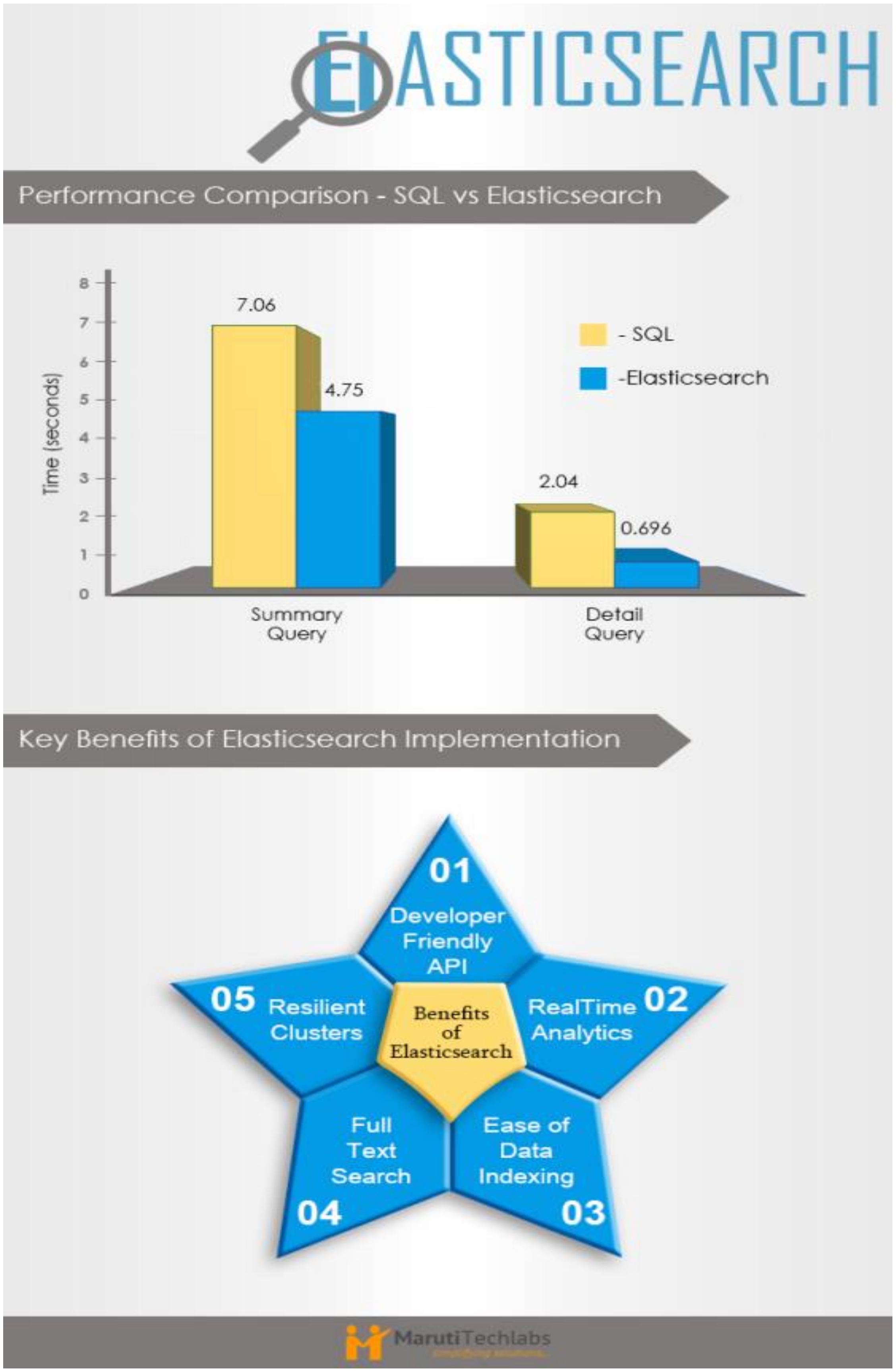
Maruti techlabs is using Elasticsearch for improving the user experience in searching data of used car parts for our client based in Austin, Texas.

A potential customer can find 'used parts' for his car on this portal. A huge amount of data (around 42 million data) affects the usability of the system performance and query response time. If a search requires data entities from a large data set, you could see a significant drag in query performance.

Standard tools like Relational Database Management Systems (RDBMS) are not suited for real-time big data analysis and dynamic conditions leading to time-outs. Thus, a complex search involves a mix of traditional databases from numerous vendors consisting of structured and unstructured data.

For this client, Maruti Techlabs chose Elasticsearch as the secondary data layer component. We have separate services for data import and result computation. So when data from vendors is maintained in SQL server it is simultaneously fed into Elasticsearch. Using Elasticsearch query response time was significantly reduced from 7.06 seconds to 4.75 seconds.

Scalability is another additional benefit of this new architecture. Leveraging Elasticsearch to build the data infrastructure has made it easier to linearly scale as new data nodes are added in the future.


ELASTICSEARCH
Performance Comparison - SQL vs Elasticsearch
Time (seconds)
0
1
2
3
4
5
6
7
8
7.06
4.75
2.04
0.696
- SQL
-Elasticsearch
Summary Query
Detail Query
Key Benefits of Elasticsearch Implementation
01 Developer Friendly API
02 RealTime Analytics
03 Ease of Data Indexing
04 Full Text Search
05 Resilient Clusters
Benefits of Elasticsearch
MarutiTechlabs

Key benefits of Elasticsearch implementation:

1. Developer-Friendly API: Any actions can be performed using RESTful api so its developer friendly.

2. Real-Time Analytics: Any updated results of customer events, such as page views, website navigation, shopping cart use, or any other kind of online or digital activity is immediately available for search and analytics.

3. Ease of Data Indexing: Data indexing is a way of sorting a number of records on multiple fields. Elasticsearch is schema-free and document-oriented.

4. Full-Text Search: In a full-text search, a search engine examines all of the words in every stored document as it tries to match search criteria.

5. Resilient Clusters: Elasticsearch clusters are resilient — they will detect new or failed nodes. A cluster may contain multiple indices that can be queried independently or as a group.

References

https://www.elastic.co/

https://www.elastic.co/products/elasticsearch

https://en.wikipedia.org/wiki/Elasticsearch

https://www.tutorialspoint.com/elasticsearch

https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-index_.html

https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-avg-aggregation.html

https://www.elastic.co/guide/en/elasticsearch/reference/current/search-search.html