Working with ElasticSearch 7.0

TriveraTech
TECHNOLOGY TRAINING

Table of Contents

1. Introducing Elastic Stack

TriveraTech
TECHNOLOGY TRAINING

Introducing Elastic Stack

In this lesson, we will cover the following topics:

- What is Elasticsearch, and why use it?
- A brief history of Elasticsearch and Apache Lucene
- Elastic Stack components
- Use cases of Elastic Stack

What is Elasticsearch, and why use it?

Let's look at the key benefits of using Elasticsearch as your data store:

- Schemaless, document-oriented
- Searching
- Analytics
- Rich client library support and the REST API
- Easy to operate and easy to scale
- Near real-time
- Lightning-fast
- Fault-tolerant

Schemaless and document-oriented

- JSON documents naturally support this type of data. For example, take a look at the following document:

```
{
 "name": "John Smith",
 "address": "121 John Street, NY, 10010",
 "age": 40
}
```

Schemaless and document-oriented

- This document may represent a customer's record. Here the record has the name, address, and age fields of the customer.
- Another record may look like the following:

```
{
 "name": "John Doe",
 "age": 38,
 "email": "john.doe@company.org"
}
```

Searching capability

- The core strength of Elasticsearch lies in its text-processing capabilities.
- Elasticsearch is great at searching, especially full-text searches, Let's understand what a full-text search is:
- Full-text search means searching through all the terms of all the documents available in the database.
- This requires the entire contents of all documents to be parsed and stored beforehand. When you hear full-text search, think of Google Search.

Analytics

- Apart from searching, the second most important functional strength of Elasticsearch is analytics.

- Yes, what was originally known as just a full-text search engine is now used as an analytics engine in a variety of use cases.

- Many organizations are running analytics solutions powered by Elasticsearch in production.

Rich client library support and the REST API

- Elasticsearch has very rich client library support to make it accessible to many programming languages.

- There are client libraries available for Java, C#, Python, JavaScript, PHP, Perl, Ruby, and many more.

- Apart from the official client libraries, there are community-driven libraries for 20 plus programming languages.

Easy to operate and easy to scale

- Elasticsearch can run on a single node and easily scale out to hundreds of nodes.

- It is very easy to start a single node instance of Elasticsearch; it works out of the box without any configuration changes and scales to hundreds of nodes.

Near real-time capable

- Typically, data is available for queries within a second after being indexed (saved).

- Not all big data storage systems are real-time capable.

- Elasticsearch allows you to index thousands to hundreds of thousands of documents per second and makes them available for searching almost immediately.

Lightning–fast

- Elasticsearch uses Apache Lucene as its underlying technology.By default, Elasticsearch indexes all the fields of your documents.

- This is extremely invaluable as you can query or search by any field in your records.

- You will never be in a situation in which you think, If only I had chosen to create an index on this field.

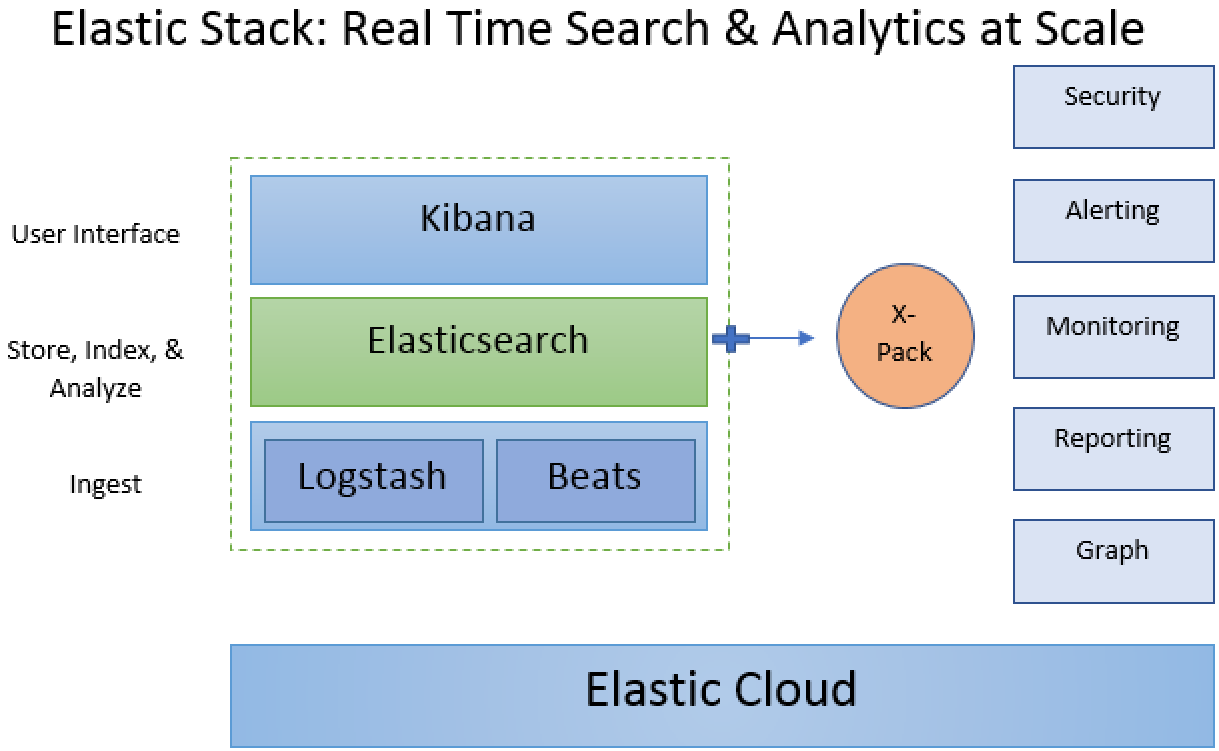
Fault-tolerant

- Elasticsearch clusters can keep running even when there are hardware failures such as node failure and network failure.

- In the case of node failure, it replicates all the data on the failed node to another node in the cluster.

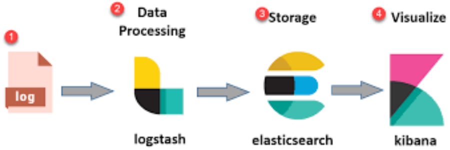
- In the case of network failure, Elasticsearch seamlessly elects master replicas to keep the cluster running.

Exploring the components of the Elastic Stack



Elasticsearch

- Elasticsearch is at the heart of the Elastic Stack.
- It stores all your data and provides search and analytic capabilities in a scalable way.
- We have already looked at the strengths of Elasticsearch and why you would want to use it.
- Elasticsearch can be used without using any other components to power your application in terms of search and analytics.



Logstash

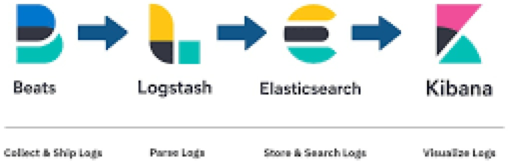
- Logstash helps centralize event data such as logs,metrics, or any other data in any format.

- It can perform a number of transformations before sending it to a stash of your choice.

- It is a key component of the Elastic Stack, used to centralize the collection and transformation processes in your data pipeline.

Beats

- Beats is a platform of open-source lightweight data shippers. Its role is complementary to Logstash.
- Logstash is a server-side component, whereas Beats has a role on the client side.
- Beats consists of a core library, libbeat, which provides an API for shipping data from the source, configuring the input options, and implementing logging.



Kibana

- Kibana is the visualization tool for the Elastic Stack, and can help you gain powerful insights about your data in Elasticsearch.
- It is often called a window into the Elastic Stack.
- It offers many visualizations includinghistograms,maps, line charts, time series, and more.
- You can build visualizations with just a few clicks and interactively explore data.

X-Pack

- X-Pack adds essential features to make the Elastic Stack production-ready.
- It adds security, monitoring, alerting, reporting, graph, and machine learning capabilities to the Elastic Stack.

Elastic Cloud

- Elastic Cloud is the cloud-based, hosted, and managed setup of the Elastic Stack components.
- The service is provided by Elastic (https://www.elastic.co/), which is behind the development of Elasticsearch and other Elastic Stack components.
- All Elastic Stack components are open source except X-Pack (and Elastic Cloud).

Use cases of Elastic Stack

The following list of example use cases is by no means exhaustive, but highlights some of the most common ones:

- Log and security analytics
- Product search
- Metrics analytics
- Web searches and website searches

Log and security analytics

Application support teams face a great challenge in administering and managing large numbers of applications deployed across tens or hundreds of servers. The application infrastructure could have the following components:

- Web servers
- Application servers
- Database servers
- Message brokers

Log and security analytics

- Typically, enterprise applications have all, or most, of the types of servers described earlier, and there are multiple instances of each server.
- In the event of an error or production issue, the support team has to log in to individual servers and look at the errors.
- It is quite inefficient to log in to individual servers and look at the raw log files.
- The Elastic Stack provides a complete toolset to collect, centralize, analyze, visualize, alert, and report errors as they occur.

Product search

- A product search involves searching for the most relevant product from thousands or tens of thousands of products and presenting the most relevant products at the top of the list before other, less relevant, products.

- You can directly relate this problem to e-commerce websites, which sell huge numbers of products sold by many vendors or resellers.

Metrics analytics

- Elastic Stack has excellent analytics capabilities, thanks to the rich Aggregations API in Elasticsearch.
- This makes it a perfect tool for analyzing data with lots of metrics.
- Metric data consists of numeric values as opposed to unstructured text such as documents and web pages.
- Some examples are data generated by sensors, Internet of Things (IoT) devices, metrics generated by mobile devices, servers, virtual machines, network routers, switches, and so on. The list is endless.

Web search and website search

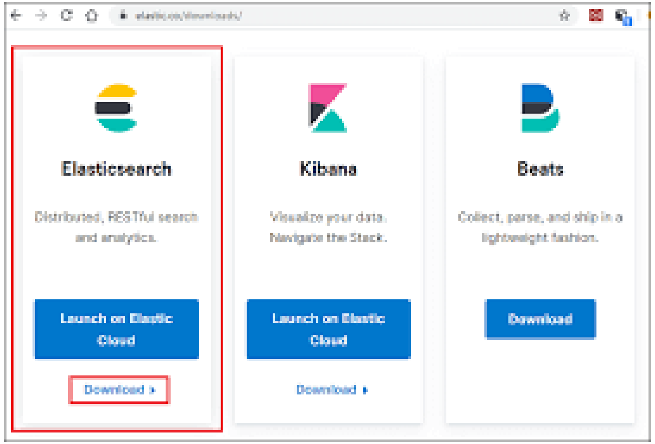
- Elasticsearch can serve as a search engine for your website and perform a Google-like search across the entire content of your site.

- GitHub, Wikipedia, and many other platforms power their searches using Elasticsearch.

- Elasticsearch can be leveraged to build content aggregation platforms.

Downloading and installing

- Now that we have enough motivation and reasons to learn about Elasticsearch and the Elastic Stack, let's start by downloading and installing the key components.
- Firstly, we will download and install Elasticsearch and Kibana.
- We will install the other components as we need them on the course of our journey.



Elasticsearch

Distributed, RESTful search and analytics.

Kibana

Visualize your data. Navigate the Stack.

Beats

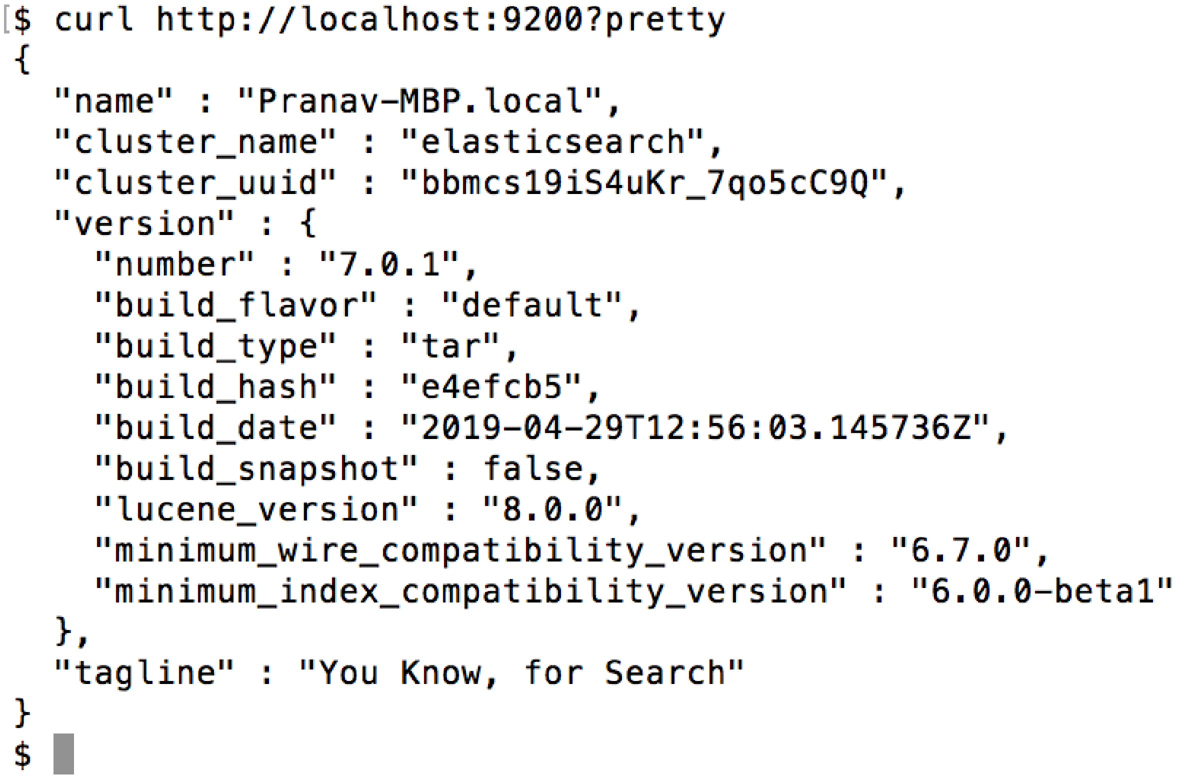
Collect, parse, and ship in a lightweight fashion.

Installing Elasticsearch

We will use the ZIP format as it is the least intrusive and the easiest for development purposes:

- Go to https://www.elastic.co/downloads/elasticsearch and download the ZIP distribution. You can also download an older version if you are looking for an exact version.
- Extract the file and change your directory to the top-level extracted folder. Run bin/elasticsearch or bin/elasticsearch.bat.
- Run curl http://localhost:9200 or open the URL in your favorite browser.

You should see an output like this



```
[$ curl http://localhost:9200?pretty
{
  "name" : "Pranav-MBP.local",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "bbmcs19iS4uKr_7qo5cC9Q",
  "version" : {
    "number" : "7.0.1",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "e4efcb5",
    "build_date" : "2019-04-29T12:56:03.145736Z",
    "build_snapshot" : false,
    "lucene_version" : "8.0.0",
    "minimum_wire_compatibility_version" : "6.7.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
$ 
```

Installing Kibana

Kibana is also available in a variety of packaging formats such as ZIP, TAR.GZ, RMP, and DEB for 32-bit and 64-bit architecture machines:

- Go to https://www.elastic.co/downloads/kibana and download the ZIP or TAR.GZ distribution for the platform that you are on.
- Extract the file and change your directory to the top-level extracted folder. Run bin/kibana or bin/kibana.bat.
- Open http://localhost:5601 in your favorite browser.

Summary

- In this lesson, we started by understanding the motivations of various search and analytics technologies other than relational databases and NoSQL stores.
- We looked at the strengths of Elasticsearch, which is at the heart of the Elastic Stack.
- We then looked at the rest of the components of the Elastic Stack and how they fit into the ecosystem.

Complete Lab 1

2. Getting Started with Elasticsearch

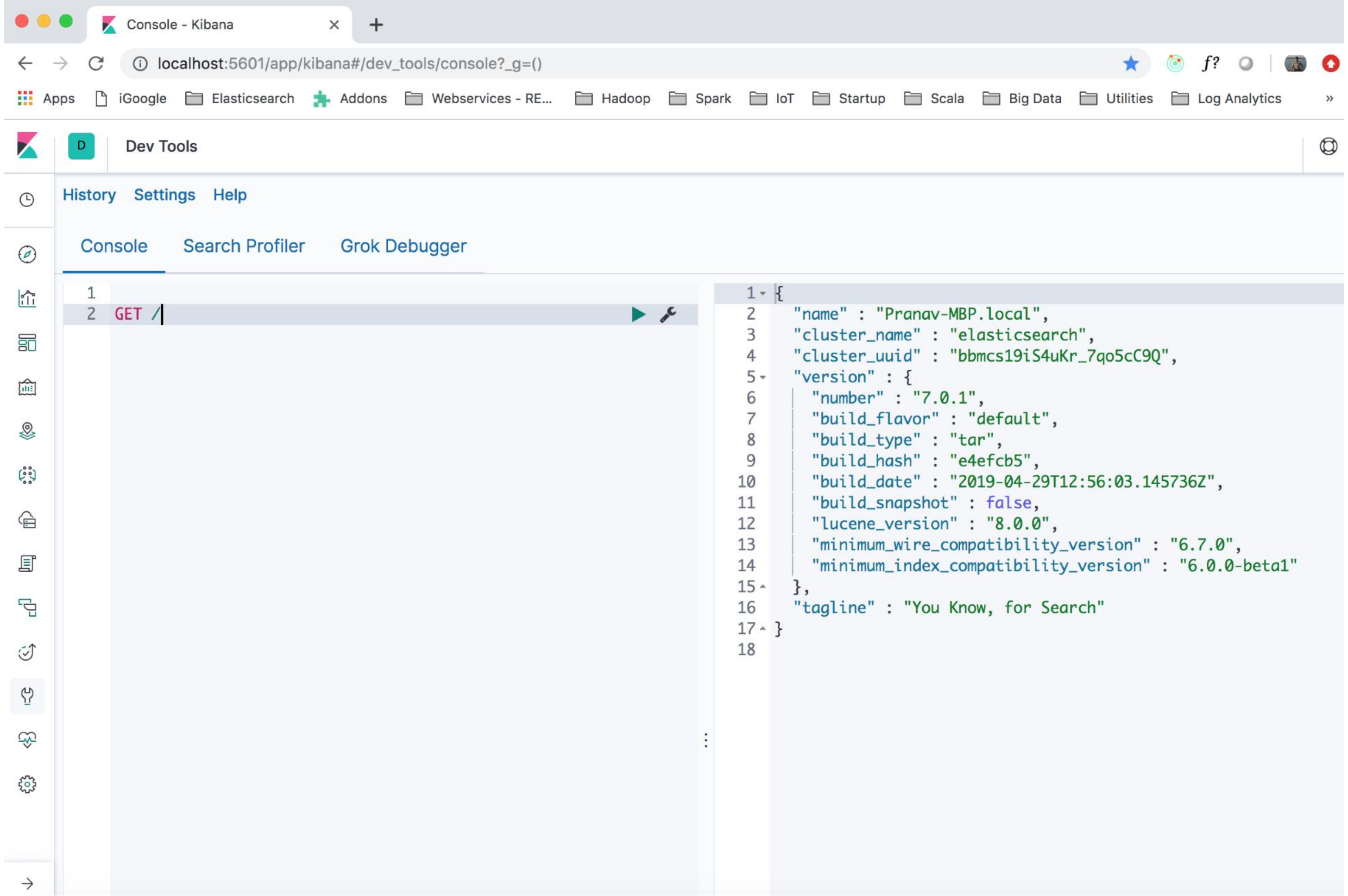
Getting Started with Elasticsearch

We will cover the following topics in this lesson:

- Using the Kibana Console UI
- Core concepts of Elasticsearch
- CRUD operations
- Creating indexes and taking control of mapping
- REST API overview

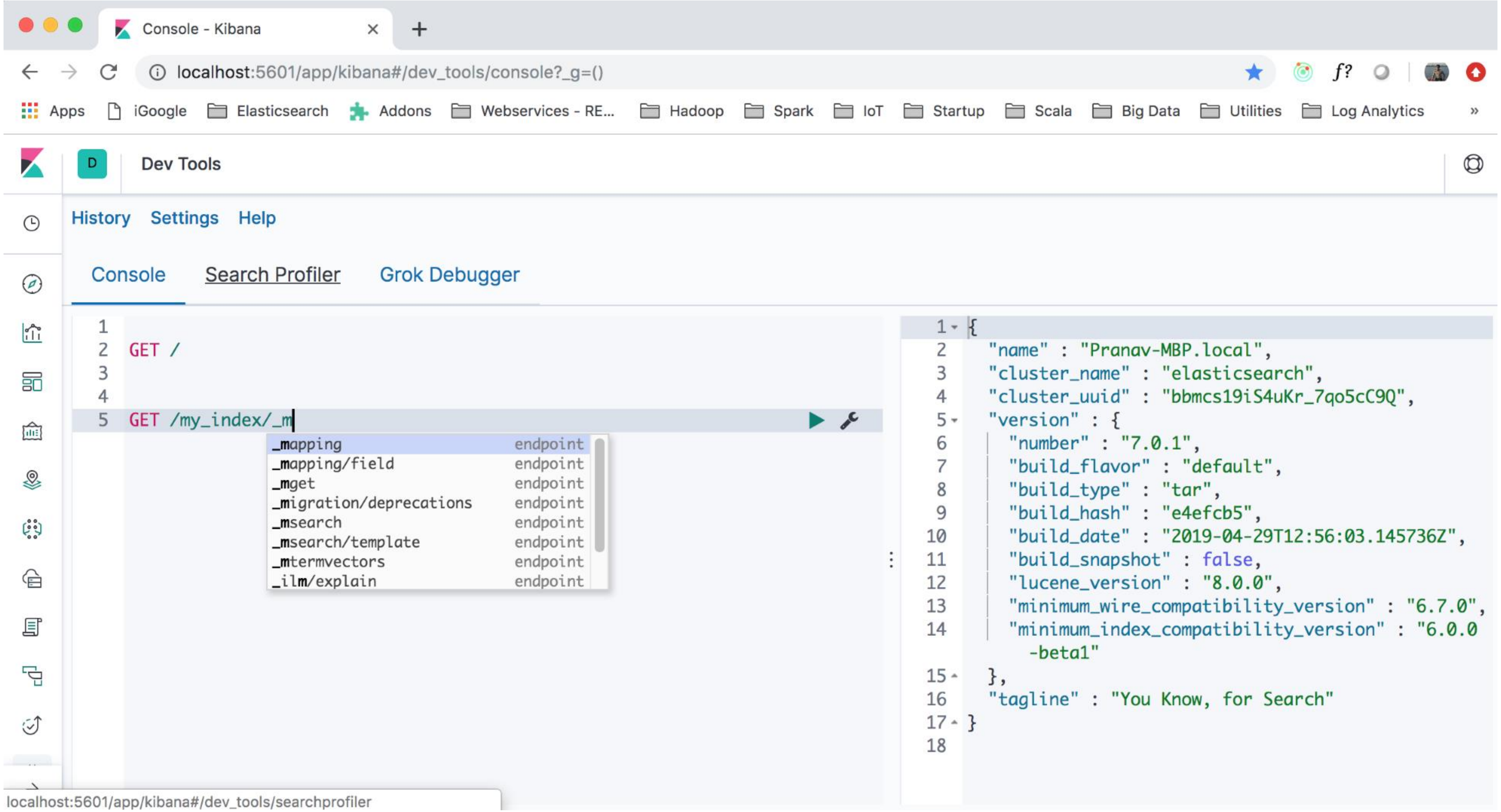
Using the Kibana Console UI

- Before we start writing our first queries to interact with Elasticsearch, we should familiarize ourselves with a very important tool: Kibana Console.
- This is important because Elasticsearch has a very rich REST API, allowing you to do all sorts of operations with Elasticsearch.
- Kibana Console has an editor that is very capable and aware of the REST API.
- It allows for auto completion, and for the formatting of queries as you write them.



```
GET /
```

```
{
  "name" : "Pranav-MBP.local",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "bbmcs19iS4uKr_7qo5cC9Q",
  "version" : {
    "number" : "7.0.1",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "e4efcb5",
    "build_date" : "2019-04-29T12:56:03.145736Z",
    "build_snapshot" : false,
    "lucene_version" : "8.0.0",
    "minimum_wire_compatibility_version" : "6.7.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```



```

GET /


GET /my_index/_m
```

```
_mapping                    endpoint
_mapping/field              endpoint
_mget                       endpoint
_migration/deprecations     endpoint
_msearch                    endpoint
_msearch/template           endpoint
_mtermvectors               endpoint
_ilm/explain                endpoint
```

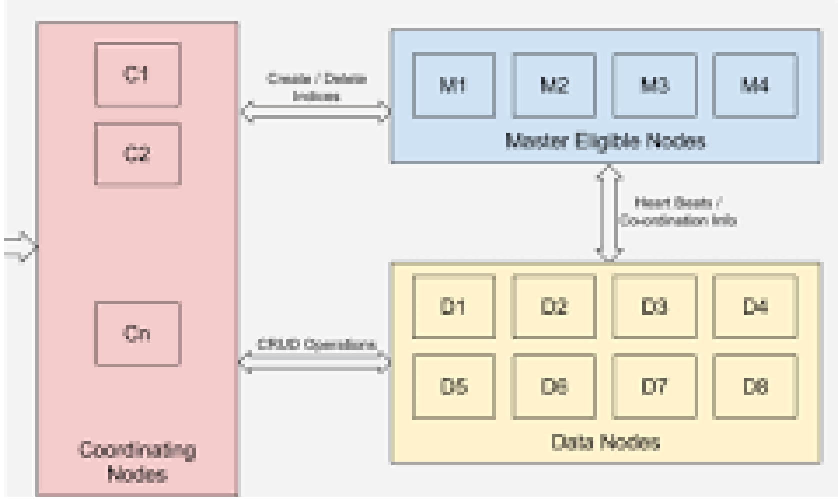
```
{
  "name" : "Pranav-MBP.local",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "bbmcs19iS4uKr_7qo5cC9Q",
  "version" : {
    "number" : "7.0.1",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "e4efcb5",
    "build_date" : "2019-04-29T12:56:03.145736Z",
    "build_snapshot" : false,
    "lucene_version" : "8.0.0",
    "minimum_wire_compatibility_version" : "6.7.0",
    "minimum_index_compatibility_version" : "6.0.0
      -beta1"
  },
  "tagline" : "You Know, for Search"
}

```

Core concepts of Elasticsearch

We will look at the following core abstractions of Elasticsearch:

- Indexes
- Types
- Documents
- Clusters
- Nodes
- Shards and replicas
- Mappings and types
- Inverted indexes



Core concepts of Elasticsearch

Let's start learning about these with an example:

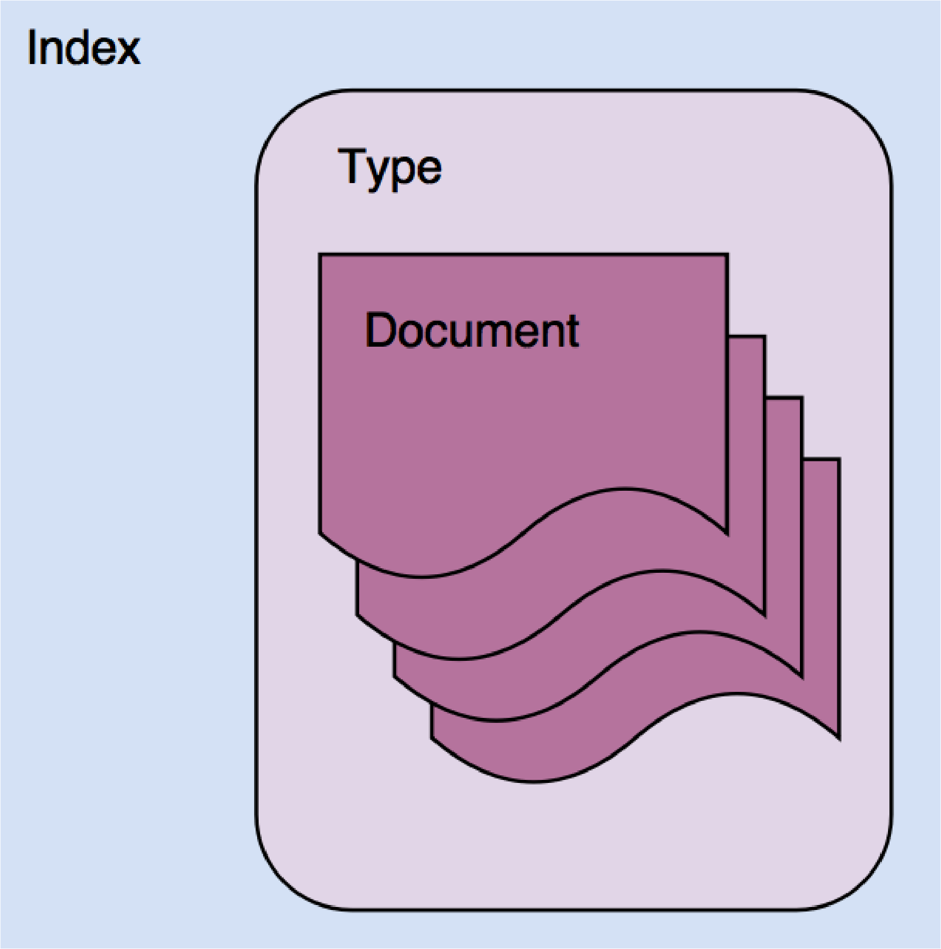
```
PUT /catalog/_doc/1
{
   "sku": "SP000001",
   "title": "Elasticsearch for Hadoop",
   "description": "Elasticsearch for Hadoop",
   "author": "Vishal Shukla",
   "ISBN": "1785288997",
   "price": 26.99
}
```

Core concepts of Elasticsearch

- All of the examples that are written for the Kibana Console UI can be very easily converted into curl commands that can be executed from the command line.
- The following is the curl version of the previous Kibana Console UI command:

```
curl -XPUT http://localhost:9200/catalog/_doc/1 -d '{ "sku": "SP000001", "title": "Elasticsearch for Hadoop", "description": "Elasticsearch for Hadoop", "author": "Vishal Shukla", "ISBN": "1785288997", "price": 26.99}'
```

Indexes



Types

- In our example of a product catalog, the document that was indexed was of the product type.
- Each document stored in the product type represents one product.
- Since the same index cannot have other types, such as customers, orders, and order line items, and more, types help in logically grouping or organizing the same kind of documents within an index.

Types

- The following code is for the index for customers:

```
PUT /customers/_doc/1
{
 "firstName": "John",
 "lastName": "Smith",
 "contact": {
 "mobile": "212-xxx-yyyy"
 },
 ...
}
```

Types

- The following code is for the index for products:

```
PUT /products/_doc/1
{
 "title": "Apple iPhone Xs (Gold, 4GB RAM, 64GB
Storage, 12 MP Dual Camera, 458 PPI Display)",
 "price": 999.99,
 ...
}
```

Documents

- Documents contain multiple fields. Each field in the JSON document is of a particular type.
- In the product catalog example that we saw earlier, these fields were sku, title, description, and price.
- Each field and its value can be seen as a key-value pair in the document, where key is the field name and value is the field value.
- The field name is similar to a column name in a relational database.

Documents

These fields are as follows:

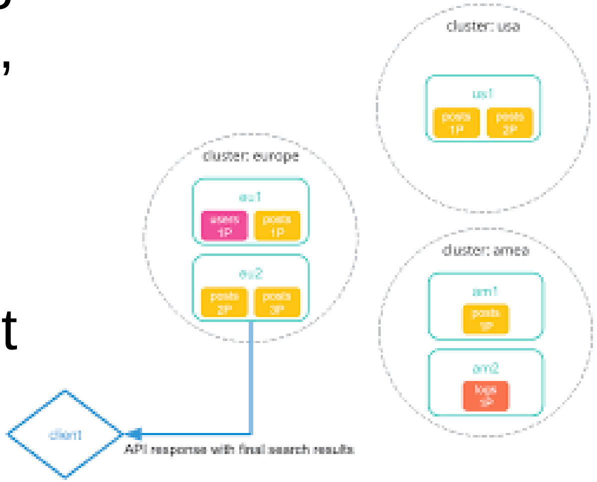
- _id: This is the unique identifier of the document within the type, just like a primary key in a database table. It can be autogenerated or specified by the user.
- _type: This field contains the type of the document.
- _index: This field contains the index name of the document.

Nodes

- An Elasticsearch node is a single server of Elasticsearch, which may be part of a larger cluster of nodes.
- It participates in indexing, searching, and performing other operations that are supported by Elasticsearch.
- Every Elasticsearch node is assigned a unique ID and name when it is started.
- A node can also be assigned a static name via the node.name parameter in the Elasticsearch configuration file, config/elasticsearch.yml.

Clusters

- A cluster hosts one or more indexes and is responsible for providing operations such as searching, indexing, and aggregations.
- A cluster is formed by one or more nodes.
- Every Elasticsearch node is always part of a cluster, even if it is just a single node cluster.



Shards and replicas

- First, let's understand what a shard is, An index contains documents of one or more types.
- Shards help in distributing an index over the cluster.
- Shards help in dividing the documents of a single index over multiple nodes.
- There is a limit to the amount of data that can be stored on a single node, and that limit is dictated by the storage, memory, and processing capacities of that node.

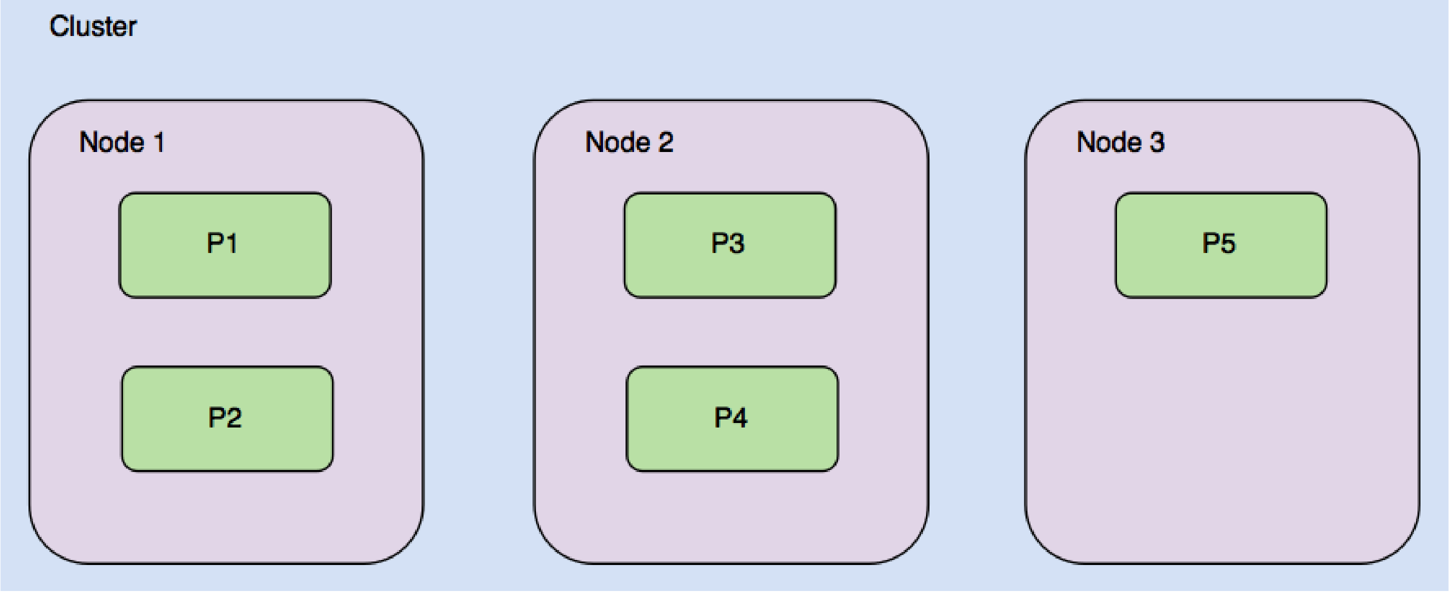
Documents

The process of dividing the data among shards is called sharding. Sharding is inherent in Elasticsearch and is a way of scaling and parallelizing, as follows:

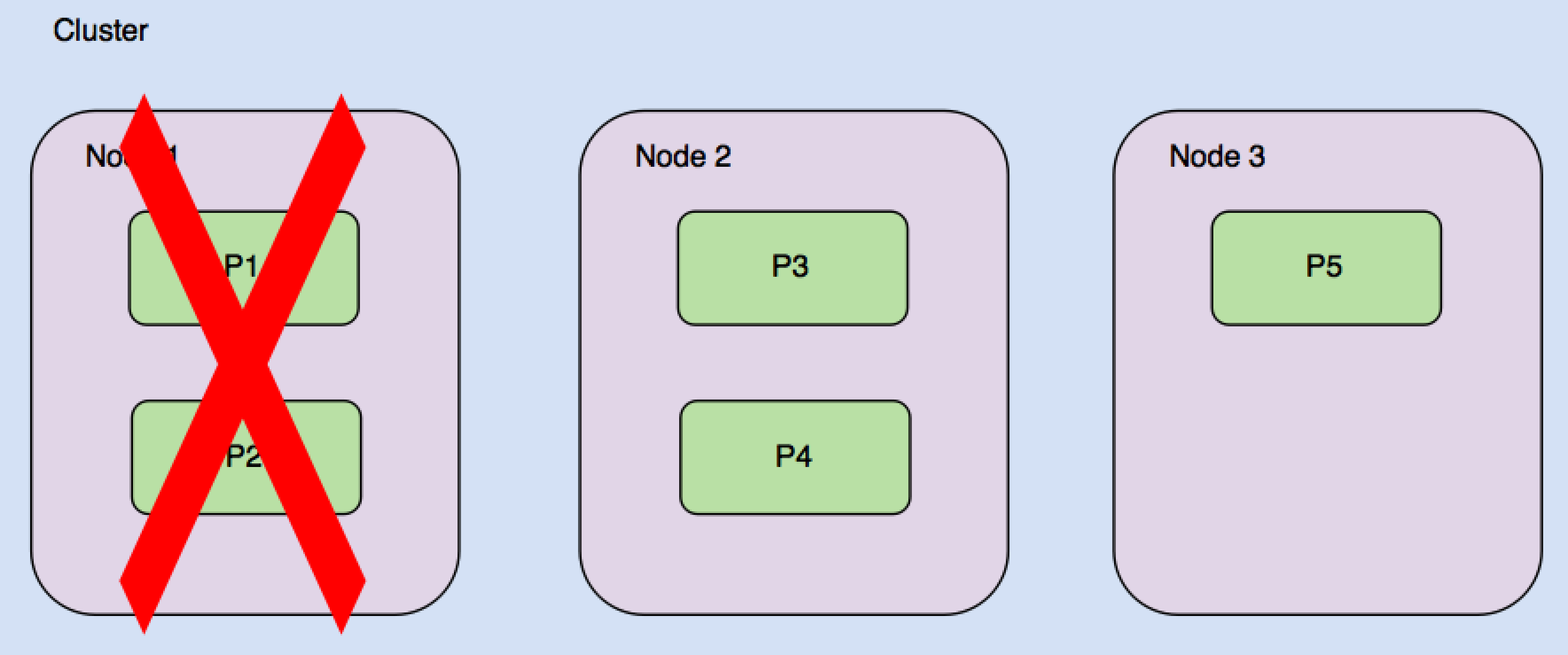
- It helps in utilizing storage across different nodes of the cluster
- It helps in utilizing the processing power of different nodes of the cluster

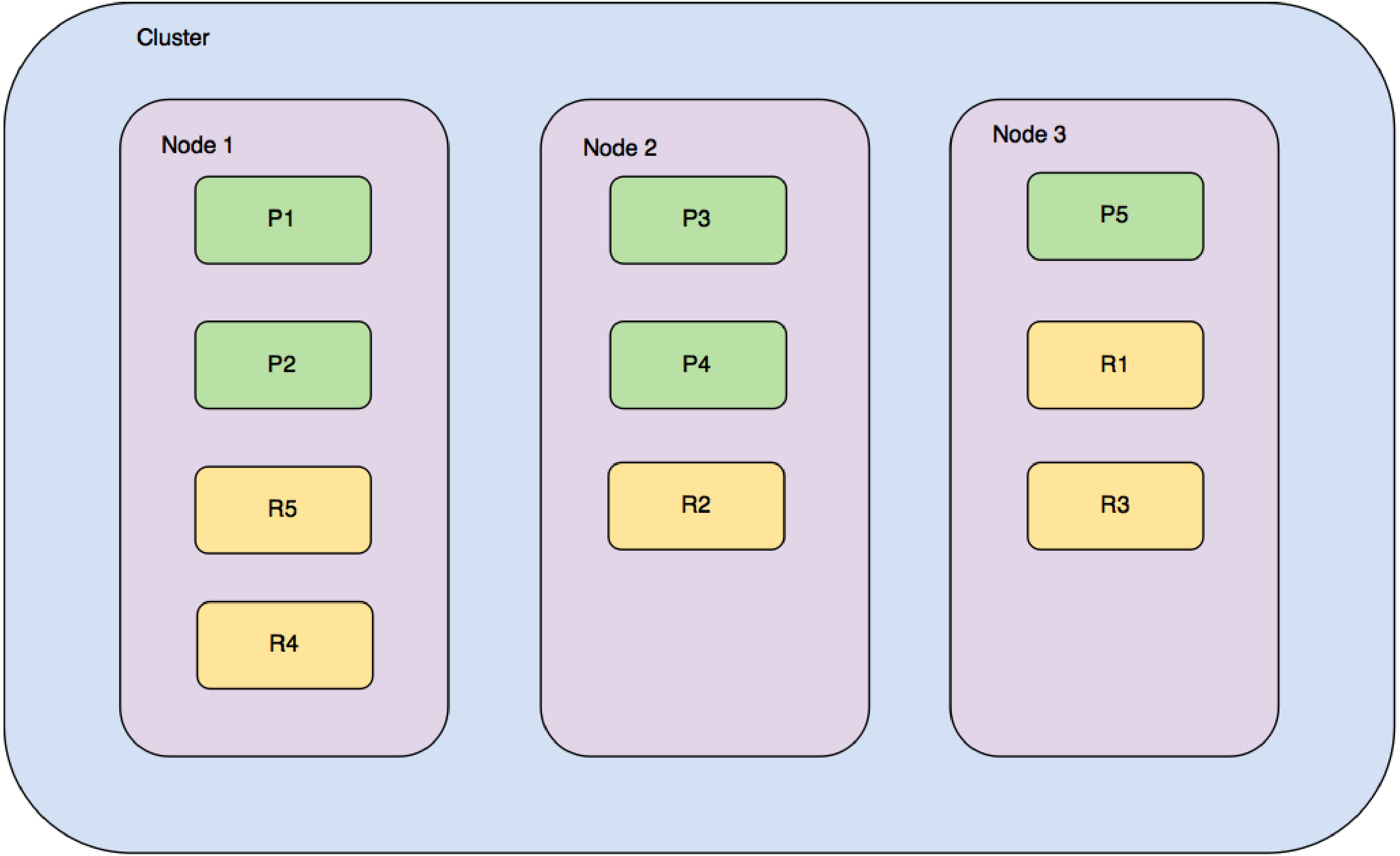
Documents

- The following diagram illustrates how five shards of one index may be distributed on a three-node cluster:



- Now, imagine that one of the nodes (Node 1) goes down.
- With Node 1, we also lose the share of data, which was stored in shards P1 and P2:




Cluster
Node 1
P1
P2
R5
R4
Node 2
P3
P4
R2
Node 3
P5
R1
R3

Mappings and datatypes

- Elasticsearch is schemaless, meaning that you can store documents with any number of fields and types of fields.
- In a real-world scenario, data is never completely schemaless or unstructured.
- There are always some sets of fields that are common across all documents in a type.
- In fact, types within the indexes should be created based on common fields.

Datatypes

- Elasticsearch supports a wide variety of datatypes for different scenarios where you want to store text data, numbers, booleans, binary objects, arrays, objects, nested types, geo-points, geo-shapes, and many other specialized datatypes, such as IPv4 and IPv6 addresses.

- In a document, each field has a datatype associated with it.

Mappings

```
PUT /catalog/_doc/2
{
    "sku": "SP000002",
    "title": "Google Pixel Phone 32GB - 5 inch display",
    "description": "Google Pixel Phone 32GB - 5 inch display
(Factory Unlocked US Version)",
    "price": 400.00,
    "resolution": "1440 x 2560 pixels",
    "os": "Android 7.1"
}
```

Mappings

Remember, unlike relational databases, we didn't have to define the fields that would be part of each document. In fact, we didn't even have to create an index with the name catalog. When the first document about the product type was indexed in the index catalog, the following tasks were performed by Elasticsearch:

- Creating an index with the name catalog
- Defining the mappings for the type of documents that will be stored in the index's default type – _doc

Creating an index with the name catalog

- The first step involves creating an index, because the index doesn't exist already.
- The index is created using the default number of shards.
- We will look at a concept called index templates – you can create templates for any new indexes.
- Sometimes, an index needs to be created on the fly, just like in this case, where the insertion of the first document triggers the creation of a new index.

Defining the mappings for the type of product

- The second step involves defining the mappings for the type of product.
- This step is executed because the type catalog did not exist before the first document was indexed.
- Remember the analogy of type with a relational database table.
- The table needs to exist before any row can be inserted.

Defining the mappings for the type of product

- To see the mappings of the product type in the catalog index, execute the following command in the Kibana Console UI:

```
GET /catalog/_mapping
```

Defining the mappings for the type of product

- The response should look like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/2_1.txt

Defining the mappings for the type of product

- Each text datatype field is mapped as follows:

```
"field_name": {
"type": "text",
"fields": {
"keyword": {
"type": "keyword",
"ignore_above": 256
}
 }
}
```

Inverted indexes

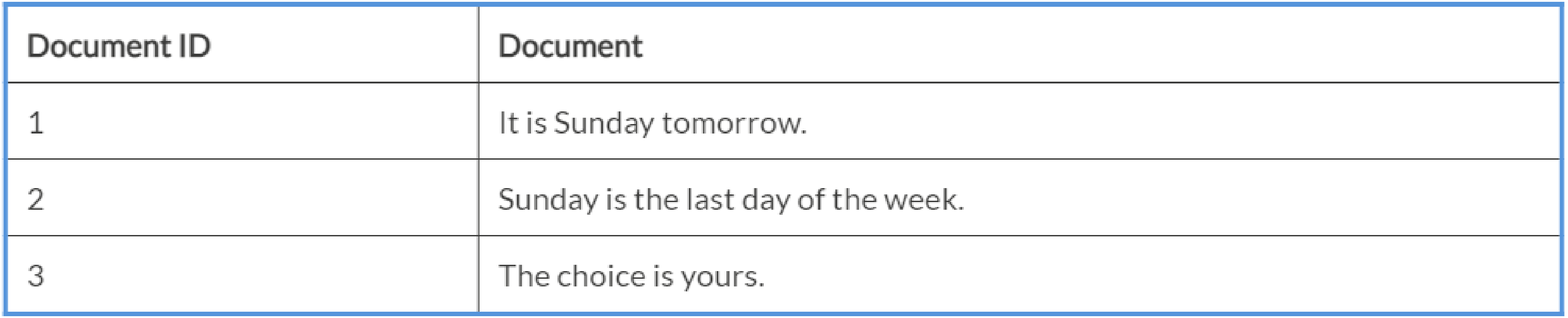
- An inverted index is the core data structure of Elasticsearch and any other system supporting full-text search.

- An inverted index is similar to the index that you see at the end of any course.

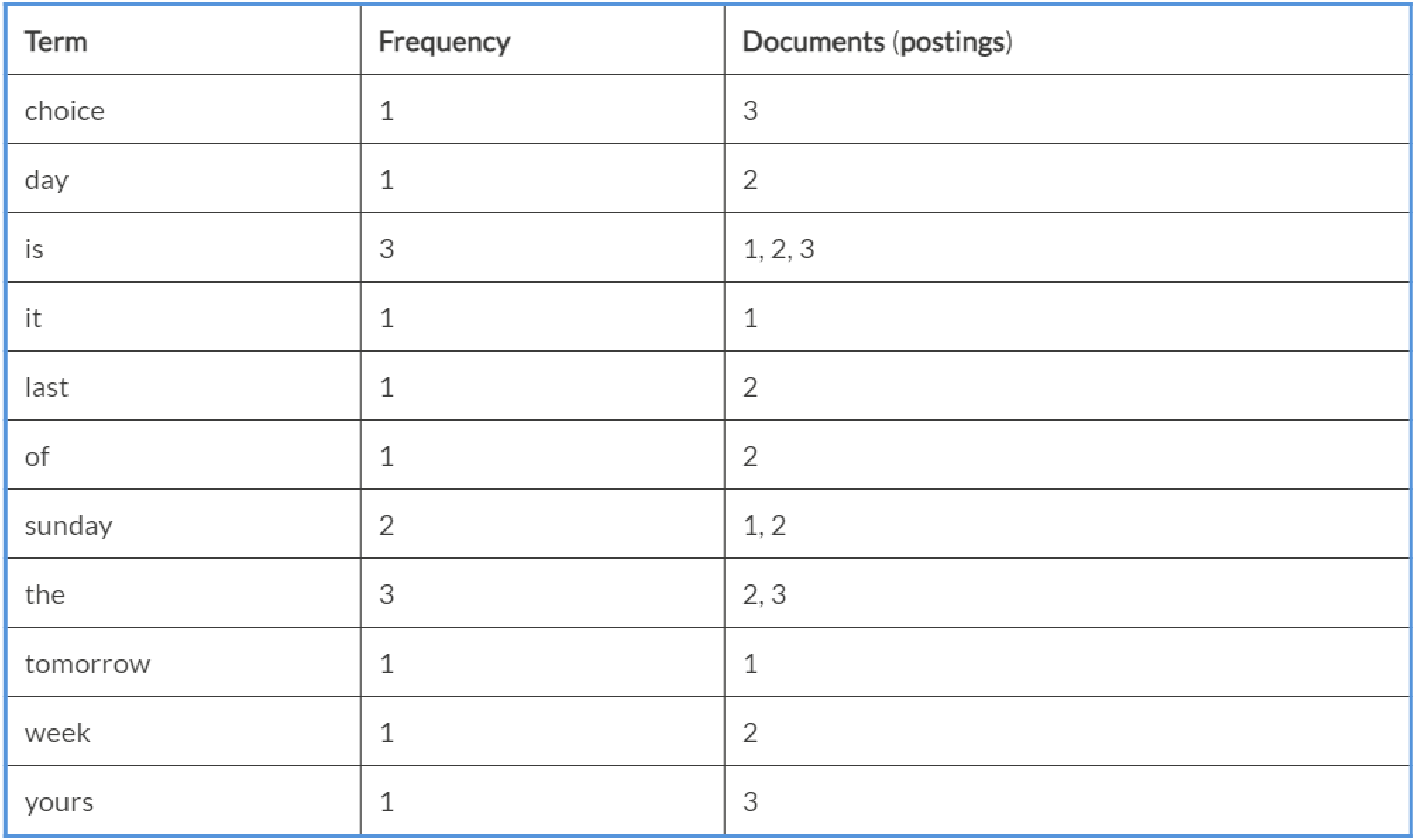
- It maps the terms that appear in the documents to the documents.

Inverted indexes

- For example, you may build an inverted index from the following strings:



Document ID	Document
1	It is Sunday tomorrow.
2	Sunday is the last day of the week.
3	The choice is yours.



Term	Frequency	Documents (postings)
choice	1	3
day	1	2
is	3	1, 2, 3
it	1	1
last	1	2
of	1	2
sunday	2	1, 2
the	3	2, 3
tomorrow	1	1
week	1	2
yours	1	3

CRUD operations

To understand how to perform CRUD operations, we will cover the following APIs. These APIs fall under the category of document APIs, which deal with documents:

- Index API
- Get API
- Update API
- Delete API

Index API

- In Elasticsearch terminology, adding (or creating) a document to a type within an index of Elasticsearch is called an indexing operation.

- There are two ways we can index a document:

1. Indexing a document by providing an ID
2. Indexing a document without providing an ID

Indexing a document by providing an ID

- The format of this request is PUT /<index>/<type>/<id>, with the JSON document as the body of the request:

```
PUT /catalog/_doc/1
{
    "sku": "SP000001",
    "title": "Elasticsearch for Hadoop",
    "description": "Elasticsearch for Hadoop",
    "author": "Vishal Shukla",
    "ISBN": "1785288997",
    "price": 26.99
}
```

Indexing a document without providing an ID

- The format of this request is POST /<index>/<type>, with the JSON document as the body of the request:

```
POST /catalog/_doc
{
    "sku": "SP000003",
    "title": "Mastering Elasticsearch",
    "description": "Mastering Elasticsearch",
    "author": "Bharvi Dixit",
    "price": 54.99
}
```

- The ID, in this case, will be generated by Elasticsearch. It is a hash string, as highlighted in the response:

```
{
  "_index" : "catalog",
  "_type" : "_doc",
  "_id" : "1ZFMpmoBa_wgE5i2FfWV",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 4,
  "_primary_term" : 1
}
```

Get API

- The get API is useful for retrieving a document when you already know the ID of the document.

- It is essentially a get by primary key operation, as follows:

```
GET /catalog/_doc/1ZFMpmoBa_wgE5i2FfWV
```

- The format of this request is GET /<index>/<type>/<id>.
- The response would be as expected:

```
{
  "_index" : "catalog",
  "_type" : "_doc",
  "_id" : "1ZFMpmoBa_wgE5i2FfWV",
  "_version" : 1,
  "_seq_no" : 4,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "sku" : "SP000003",
    "title" : "Mastering Elasticsearch",
    "description" : "Mastering Elasticsearch",
"author": "Bharvi Dixit",
    "price": 54.99
}
}
```

Update API

- The update API is useful for updating the existing document by ID.
- The format of an update request is POST <index>/<type>/<id>/_update, with a JSON request as the body:

```
POST /catalog/_update/1
{
"doc": {
"price": "28.99"
}
}
```

- The response of the update request is as follows:

```
{
"_index": "catalog",
"_type": "_doc",
"_id": "1",
"_version": 2,
"result": "updated",
"_shards": {
"total": 2,
"successful": 1,
"failed": 0
}
}
```

- The following example uses doc_as_upsert to merge into the document with an ID of 3 or insert a new document if it doesn't exist:

```
POST /catalog/_update/3
{
"doc": {
    "author": "Albert Paro",
    "title": "Elasticsearch 5.0 Cookbook",
    "description": "Elasticsearch 5.0 Cookbook Third Edition",
"price": "54.99"
},
"doc_as_upsert": true
}
```

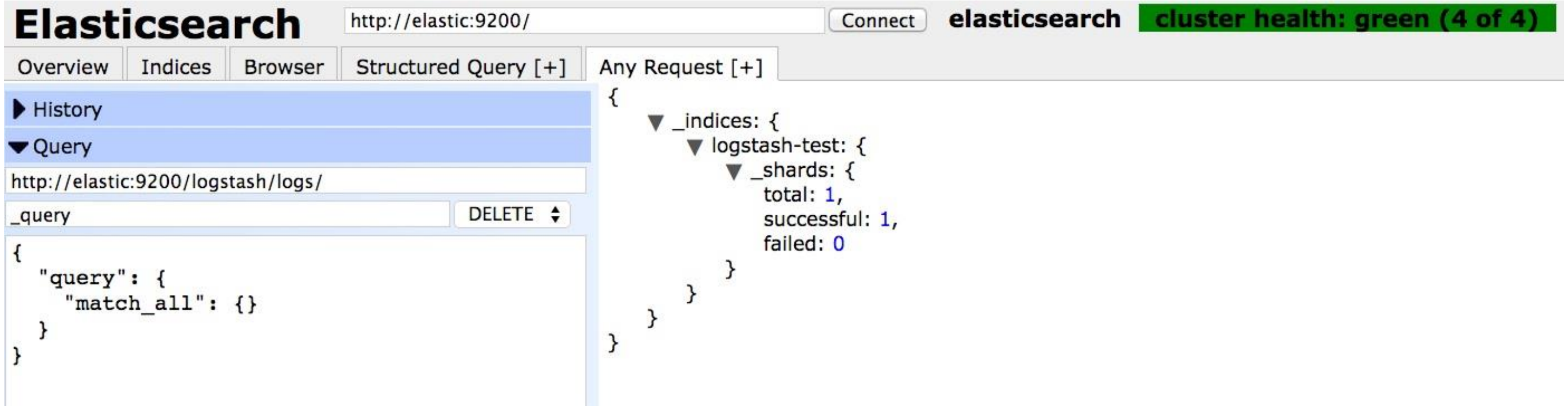
- We can update the value of a field based on the existing value of that field or another field in the document.
- The following update uses an inline script to increase the price by two for a specific product:

```
POST /catalog/_update/1ZFMpmoBa_wgE5i2FfWV
{
  "script": {
    "source": "ctx._source.price += params.increment",
    "lang": "painless",
    "params": {
      "increment": 2
    }
  }
}
```

Delete API

- The delete API lets you delete a document by ID:

DELETE /catalog/_doc/1ZFMpmoBa_wgE5i2FfWV



http://elastic:9200/

http://elastic:9200/logstash/logs/

_query

DELETE

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

```
{
  ▼ _indices: {
    ▼ logstash-test: {
      ▼ _shards: {
          total: 1,
          successful: 1,
          failed: 0
        }
      }
    }
}
```

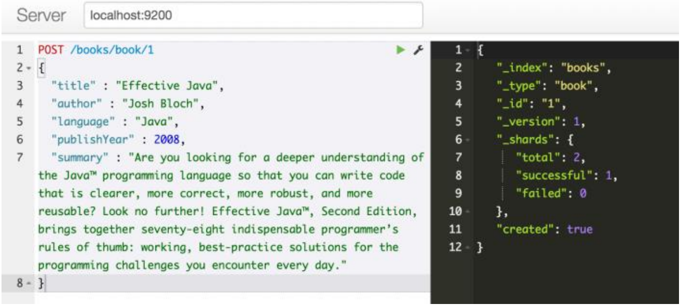
- The response of the delete operation is as follows:

```
{
  "_index" : "catalog",
  "_type" : "_doc",
  "_id" : "1ZFMpmoBa_wgE5i2FfWV",
  "_version" : 4,
  "result" : "deleted",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 9,
  "_primary_term" : 1
}
```

Creating indexes and taking control of mapping

Usually, you wouldn't want to let things happen automatically, as you would want to control how indexes are created and also how mapping is created. We will see how you can take control of this process in this section, and we will look at the following:

- Creating an index
- Creating a mapping
- Updating a mapping



localhost:9200

```
POST /books/book/1
{
  "title" : "Effective Java",
  "author" : "Josh Bloch",
  "language" : "Java",
  "publishYear" : 2008,
  "summary" : "Are you looking for a deeper understanding of the Java™ programming language so that you can write code that is clearer, more correct, more robust, and more reusable? Look no further! Effective Java™, Second Edition, brings together seventy-eight indispensable programmer's rules of thumb: working, best-practice solutions for the programming challenges you encounter every day."
}
```

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

Creating an index

- You can create an index and specify the number of shards and replicas to create:

```
PUT /catalog
{
  "settings": {
    "index": {
      "number_of_shards": 5,
      "number_of_replicas": 2
    }
  }
}
```

Creating an index

- It is possible to specify a mapping for a type at the time of index creation.
- The following command will create an index called catalog, with five shards and two replicas.
- Additionally, it also defines a type called my_type with two fields, one of the text type and another of the keyword type:

https://github.com/fenago/elasticsearch/blob/master/snippets/2_2.txt

Creating type mapping in an existing index

- A type can be added within an index after the index is created using the following code.
- The mappings for the type can be specified as follows:

```
PUT /catalog/_mapping
{
  "properties": {
    "name": {
      "type": "text"
    }
  }
}
```

Creating type mapping in an existing index

- Let's add a couple of documents after creating the new type:

```
POST /catalog/_doc
{
"name": "books"
}
POST /catalog/_doc
{
"name": "phones"
}
```

Creating type mapping in an existing index

- Elasticsearch will assign a type automatically based on the value that you insert for the new field.
- It only takes into consideration the first value that it sees to guess the type of that field:

```
POST /catalog/_doc
{
 "name": "music",
 "description": "On-demand streaming music"
}
```

Creating type mapping in an existing index

- When the new document is indexed with fields, the field is assigned a datatype based on its value in the initial document.

- Let's look at the mapping after this document is indexed:

https://github.com/fenago/elasticsearch/blob/master/snippets/2_3.txt

Updating a mapping

- Let's add a code field, which is of the keyword type, but with no analysis:

```
PUT /catalog/_mapping
{
  "properties": {
    "code": {
      "type": "keyword"
    }
  }
}
```

Updating a mapping

- This mapping is merged into the existing mappings of the _doc type.
- The mapping looks like the following after it is merged:

https://github.com/fenago/elasticsearch/blob/master/snippets/2_4.txt

Updating a mapping

- Any subsequent documents that are indexed with the code field are assigned the right datatype:

```
POST /catalog/_doc
{
"name": "sports",
"code": "C004",
"description": "Sports equipment"
}
```

REST API overview

The APIs that deal with Elasticsearch are categorized into the following types of APIs:

- Document APIs
- Search APIs
- Aggregation APIs
- Indexes APIs
- Cluster APIs
- cat APIs

Common API conventions

All Elasticsearch REST APIs share some common features. They can be used across almost all APIs. In this section, we will cover the following features:

- Formatting the JSON response
- Dealing with multiple indexes

Formatting the JSON response

- By default, the response of all the requests is not formatted.

- It returns an unformatted JSON string in a single line:

```
curl -XGET http://localhost:9200/catalog/_doc/1
```

Formatting the JSON response

- The following response is not formatted:

```
{"_index":"catalog","_type":"product","_id":"1","_version":3,"found":true,"_source":{
 "sku": "SP000001",
 "title": "Elasticsearch for Hadoop",
 "description": "Elasticsearch for Hadoop",
 "author": "Vishal Shukla",
 "ISBN": "1785288997",
 "price": 26.99
}}
```

- Passing pretty=true formats the response:

```
curl -XGET http://localhost:9200/catalog/_doc/1?pretty=true
{
 "_index" : "catalog",
 "_type" : "product",
 "_id" : "1",
 "_version" : 3,
 "found" : true,
 "_source" : {
 "sku" : "SP000001",
 "title" : "Elasticsearch for Hadoop",
 "description" : "Elasticsearch for Hadoop",
 "author" : "Vishal Shukla",
 "ISBN" : "1785288997",
 "price" : 26.99
 }
}
```

Dealing with multiple indexes

We cover the following scenarios when dealing with multiple indexes within a cluster:

- Searching all documents in all indexes
- Searching all documents in one index
- Searching all documents of one type in an index
- Searching all documents in multiple indexes
- Searching all documents of a particular type in all indexes

Dealing with multiple indexes

- This will return all the documents from all the indexes of the cluster.

- The response looks like the following, and it is truncated to remove the unnecessary repetition of documents:

https://github.com/fenago/elasticsearch/blob/master/snippets/2_5.txt

Searching all documents in one index

- The following code will search for all documents, but only within the catalog index:

```
GET /catalog/_search
```

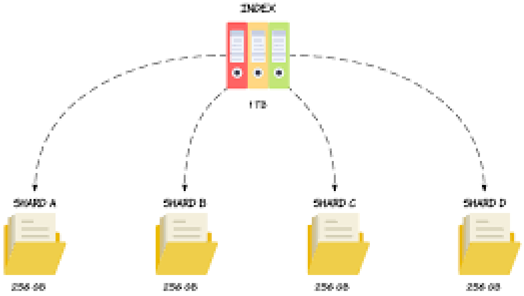
- You can also be more specific and include the type in addition to the index name, like so:

```
GET /catalog/_doc/_search
```

Searching all documents in multiple indexes

- The following will search for all the documents within the catalog index and an index named my_index:

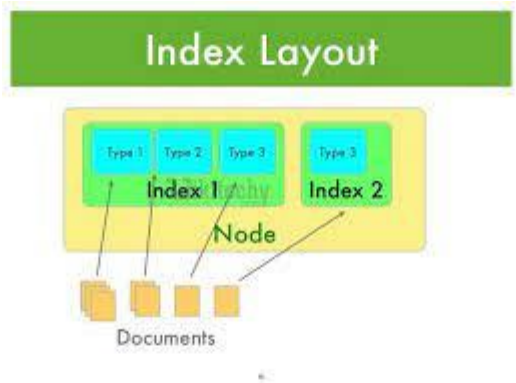
```
GET /catalog,my_index/_search
```



Searching all the documents of a particular type in all indexes

- The following will search all the indexes in the cluster, but only documents of the product type will be searched:

```
GET /_all/_doc/_search
```



Summary

- In this lesson, we learned about the essential Kibana Console UI and curl commands that we can use to interact with Elasticsearch with the REST API.
- Then, we looked at the core concepts of Elasticsearch.
- We performed customary CRUD operations, which are required as support for any data store.
- We took a closer look at how to create indexes, and how to create and manage mappings.

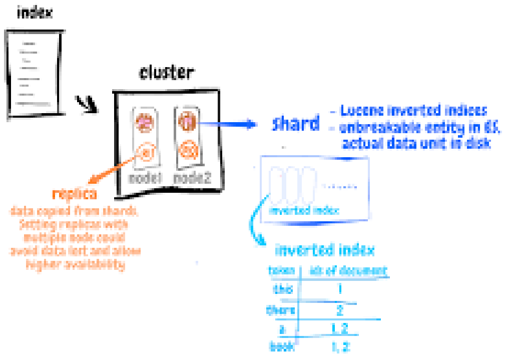
Complete Lab 2

3. Searching - What is Relevant

Searching - What is Relevant

We will cover the following topics in this lesson:

- The basics of text analysis
- Searching from structured data
- Writing compound queries
- Searching from full-text
- Modeling relationships


index
cluster
node1
node2
shard
- Lucene inverted indices
- unbreakable entity in ES, actual data unit in disk
replica
data copied from shards. Setting replicas with multiple node could avoid data lost and allow higher availability
inverted index
inverted index
token
ids of document
this
1
there
2
a
1, 2
book
1, 2

The basics of text analysis

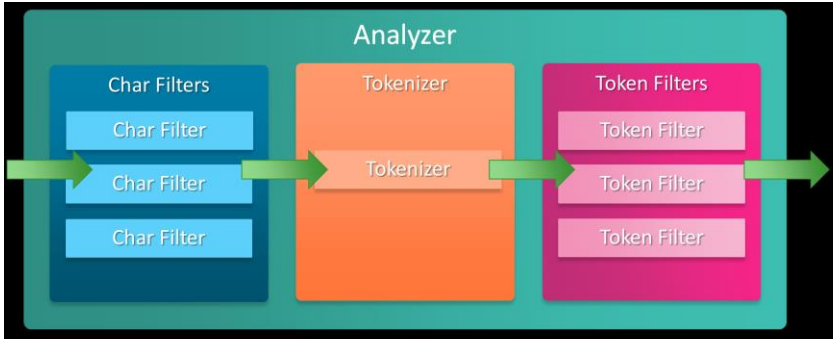
In the following sections, we will cover the following topics:

- Understanding Elasticsearch analyzers
- Using built-in analyzers
- Implementing autocomplete with a custom analyzer

Understanding Elasticsearch analyzers

The analyzer performs this process of breaking up input character streams into terms. This happens twice:

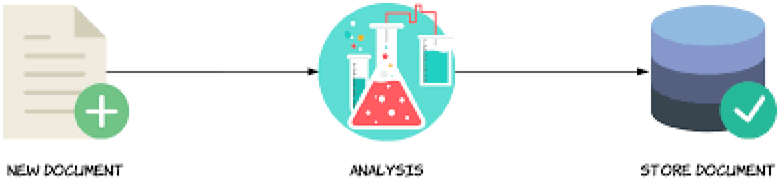
- At the time of indexing
- At the time of searching



Understanding Elasticsearch analyzers

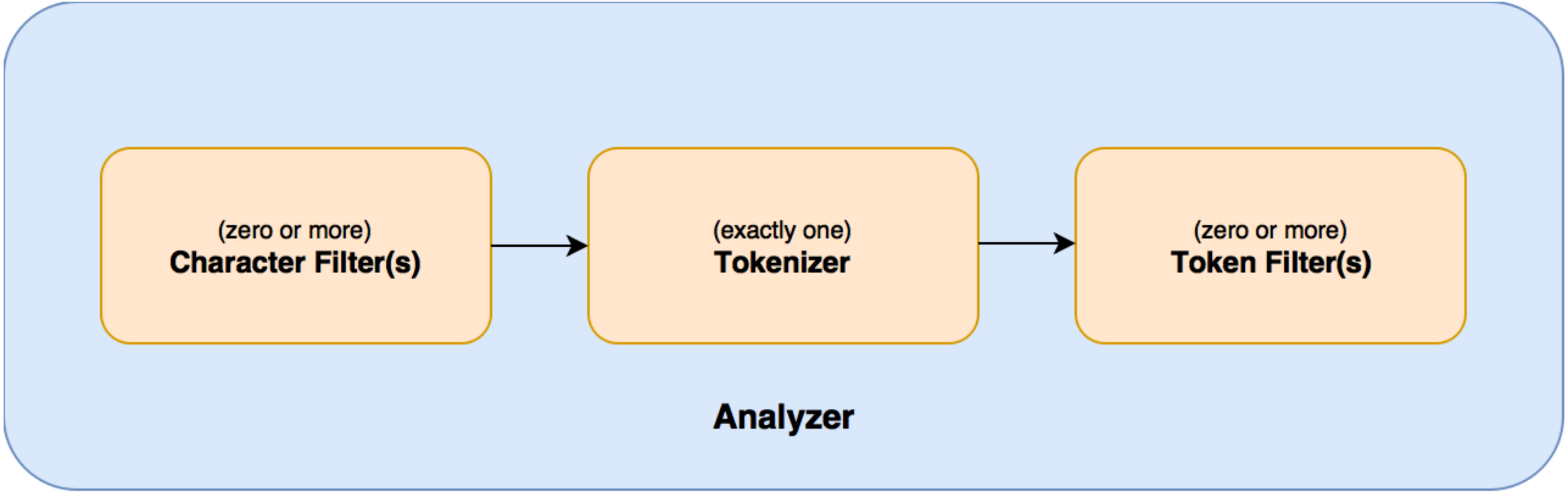
Elasticsearch uses analyzers to analyze text data. An analyzer has the following components:

- Character filters: Zero or more
- Tokenizer: Exactly one
- Token filters: Zero or more



Understanding Elasticsearch analyzers

- The following diagram depicts the components of an analyzer:



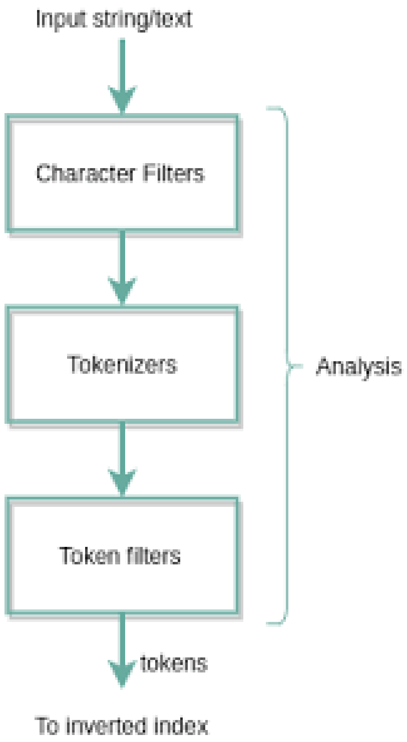
Character filters

- When composing an analyzer, we can configure zero or more-character filters.
- A character filter works on a stream of characters from the input field; each character filter can add, remove, or change the characters in the input field.

Character filters

For example, you may want to transform emoticons into some text that represents those emoticons:

- :) should be translated to _smile_
- :(should be translated to _sad_
- :D should be translated to _laugh_



Character filters

```
"char_filter": {
    "my_char_filter": {
     "type": "mapping",
     "mappings": [
      ":) => _smile_",
      ":( => _sad_",
      ":D => _laugh_"
     ]
    }
   }
```

Tokenizer

- An analyzer has exactly one tokenizer, The responsibility of a tokenizer is to receive a stream of characters and generate a stream of tokens &These tokens are used to build an inverted index.

- A token is roughly equivalent to a word & In addition to breaking down characters into words or tokens, it also produces, in its output, the start and end offset of each token in the input stream.

Standard tokenizer

- Loosely speaking, the standard tokenizer breaks down a stream of characters by separating them with whitespace characters and punctuation.
- The following example shows how the standard tokenizer breaks a character stream into tokens:

```
POST _analyze
{
  "tokenizer": "standard",
  "text": "Tokenizer breaks characters into tokens!"
}
```

Standard tokenizer

- The preceding command produces the following output; notice the start_offset, end_offset, and positions in the output:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_1.txt

Token filters

- There can be zero or more token filters in an analyzer.
- Every token filter can add, remove, or change tokens in the input token stream that it receives.
- Since it is possible to have multiple token filters in an analyzer, the output of each token filter is sent to the next one until all token filters are considered.
- Elasticsearch comes with several token filters, and they can be used to compose your own custom analyzers.

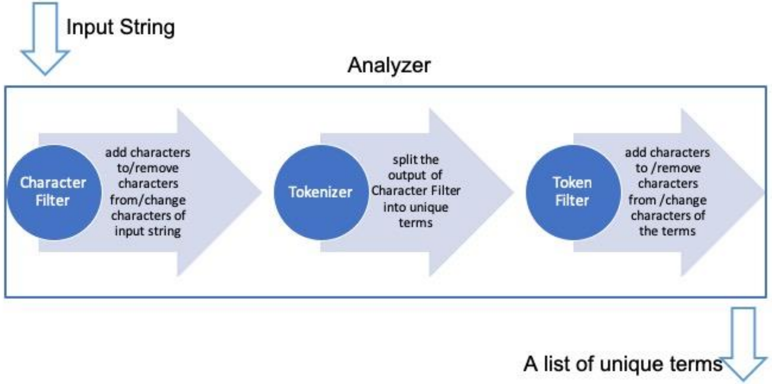
Using built-in analyzers

- Elasticsearch comes with several built-in analyzers that can be used directly.
- Almost all these analyzers work without any need for additional configuration, but they provide the flexibility of configuring some parameters.

Standard analyzer

Standard Analyzer is suitable for many languages and situations. It can also be customized for the underlying language or situation. Standard analyzer comprises of the following components:

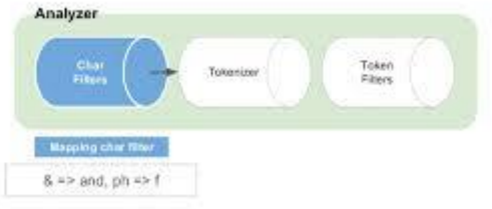
- Tokenizer
- Token filters


Input String
Analyzer
Character Filter
add characters to/remove characters from/change characters of input string
Tokenizer
split the output of Character Filter into unique terms
Token Filter
add characters to /remove characters from /change characters of the terms
A list of unique terms

Standard analyzer

- Let's see how Standard analyzer works by default with an example:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_2.txt



Standard analyzer

- Let's check how Elasticsearch will do the analysis for the my_text field whenever any document is indexed in this index.
- We can do this test using the _analyze API, as we saw earlier:

```
POST index_standard_analyzer/_analyze
{
  "field": "my_text",
  "text": "The Standard Analyzer works this way."
}
```

Standard analyzer

- The output of previous command shows the following tokens:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_3.txt

Standard analyzer

- Even though the Standard Analyzer has a stop token filter, none of the tokens are filtered out. Therefore the _analyze output has all words as tokens.

- Let's create another index that uses English language stop words:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_4.txt

Standard analyzer

- When you try the _analyze API on the new index, you will see that it removes the stopwords, such as the and this:

```
POST
index_standard_analyzer_english_stopwords/_analyze
{
  "field": "my_text",
  "text": "The Standard Analyzer works this way."
}
```

Standard analyzer

- Previous code returns a response like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_5.txt

Implementing autocomplete with a custom analyzer

- In certain situations, you may want to create your own custom analyzer by composing character filters, tokenizers, and token filters of your choice.
- Please remember that most requirements can be fulfilled by one of the built-in analyzers with some configuration.
- Let's create an analyzer that can help when implementing autocomplete functionality.

Implementing autocomplete with a custom analyzer

- If we were to use Standard Analyzer at indexing time, the following terms would be generated for the field with the Learning Elastic Stack 7 value:

```
GET /_analyze
{
  "text": "Learning Elastic Stack 7",
  "analyzer": "standard"
}
```

Implementing autocomplete with a custom analyzer

- For example, if the user has typed elas, it should still recommend Learning Elastic Stack 7 as a product.

- Let's compose an analyzer that can generate terms such as el, ela, elas, elast, elasti, elastic, le, lea, and so on:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_6.txt

Implementing autocomplete with a custom analyzer

- Given that the following two products are indexed, and the user has typed Ela so far, the search should return both products:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_7.txt

Implementing autocomplete with a custom analyzer

- We are going to use product catalog data taken from the popular e-commerce site www.amazon.com.
- The data is downloadable from http://dbs.uni-leipzig.de/file/Amazon-GoogleProducts.zip.
- Before we start with the queries, let's create the required index and import some data:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_8.txt

Implementing autocomplete with a custom analyzer

- After you have imported the data, verify that it is imported

with the following query:

```
GET /amazon_products/_search
{
  "query": {
    "match_all": {}
  }
}
```

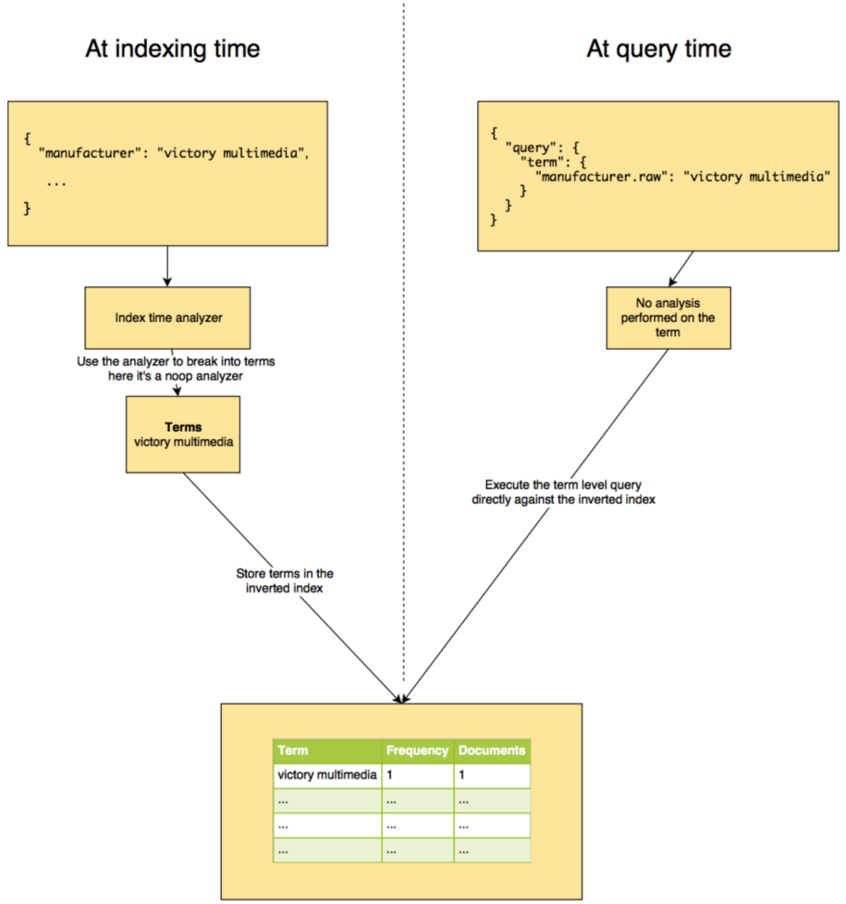
Searching from structured data

- In certain situations, we may want to find out whether a given document should be included or not; that is, a simple binary answer.

- On the other hand, there are other types of queries that are relevance-based.

- Such relevance-based queries also return a score against each document to say how well that document fits the query.

Term level query flow



At indexing time

```
{
  "manufacturer": "victory multimedia",
    ...
}
```

Index time analyzer

Use the analyzer to break into terms
here it's a noop analyzer

Terms
victory multimedia

Store terms in the
inverted index

At query time

```
{
  "query": {
    "term": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

No analysis
performed on the
term

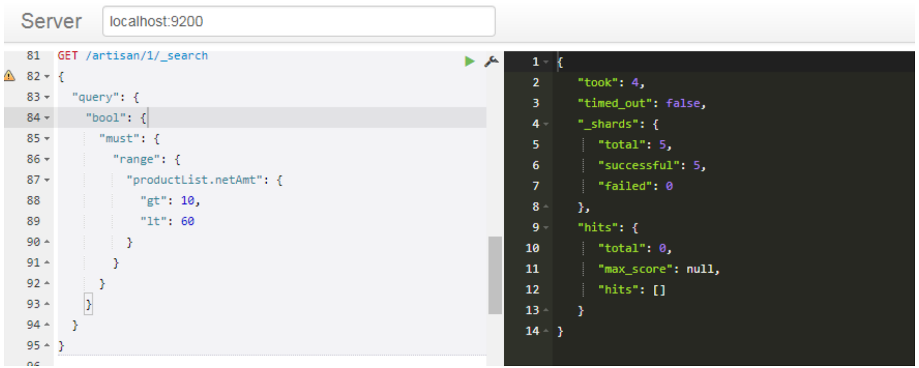
Execute the term level query
directly against the inverted index

Term	Frequency	Documents
victory multimedia	1	1
...	...	...
...	...	...
...	...	...

Range query

Range queries can be applied to fields with datatypes that have natural ordering. We will look at how to apply range queries in the following ways:

- On numeric types
- With score boosting
- On dates



Server localhost:9200

```
GET /artisan/1/_search
{
  "query": {
    "bool": {
      "must": {
        "range": {
          "productList.netAmt": {
            "gt": 10,
            "lt": 60
          }
        }
      }
    }
  }
}
```

```
{
  "took": 4,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 0,
    "max_score": null,
    "hits": []
  }
}
```

Let's look at the most typical range query on a numeric field.

Range query on numeric types

- Suppose we are storing products with their prices in an Elasticsearch index and we want to get all products within a range.
- The following is the query to get products in the range of $10 to $20:

```
GET /amazon_products/_search
{
  "query": {
    "range": {
      "price": {
        "gte": 10,
        "lte": 20
      }
    }
  }
}
```

Range query

- The response of previous query looks like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_9.txt

Range query with score boosting

- The range query allows you to provide a boost parameter to enhance its score relative to other query/queries that it is combined with:

```
GET /amazon_products/_search
{
  "from": 0,
  "size": 10,
  "query": {
    "range": {
      "price": {
        "gte": 10,
        "lte": 20,
"boost": 2.2
      }
    }
  }
}
```

Range query on dates

- A range query can also be applied to date fields since dates are also inherently ordered. You can specify the date format while querying a date range:

```
GET /orders/_search
{"query":{"range":{"orderDate":{"gte":"01/09/2017","lte":"30/09/2017","format":"dd/MM/yyyy"}}}}
```

Range query on dates

- Elasticsearch allows us to use dates with or without the time in its queries.
- It also supports the use of special terms, including nowto denote the current time.
- For example, the following query queries data from the last 7 days up until now, that is, data from exactly 24 x 7 hours ago till now with a precision of milliseconds:

```
GET /orders/_search
{"query":{"range":{"orderDate":{"gte":"now-7d","lte":"now"}}}}
```

Exists query

- Sometimes it is useful to obtain only records that have non-null and non-empty values in a certain field.
- For example, getting all products that have description fields defined:

```
GET /amazon_products/_search
{
  "query": {
    "exists": {
      "field": "description"
    }
  }
}
```

Term query

- When we defined the manufacturer field, we stored it as both text and keyword fields.
- When doing an exact match, we have to use the field with the keyword type:

```
GET /amazon_products/_search
{
  "query": {
    "term": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

- The response looks like the following (only the partial response is included):

```
{
  ...
  "hits": {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
"max_score": 5.965414,
    "hits": [
      {
        "_index": "amazon_products",
        "_type": "products",
        "_id": "AV5rBfPNNI_2eZGciIHC",
"_score": 5.965414,
   ...
```

Term query

- It needs to be wrapped inside a constant_score filter:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "manufacturer.raw": "victory multimedia"
        }
      }
    }
  }
}
```

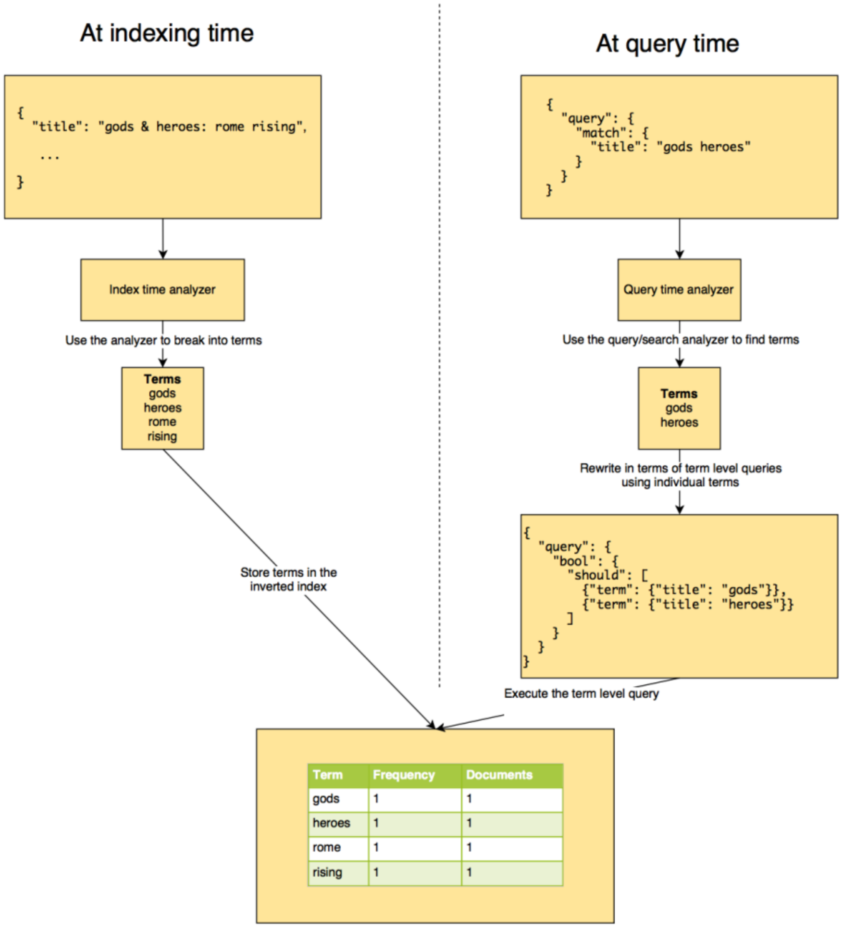
Searching from the full text

- Full-text queries can work on unstructured text fields, These queries are aware of the analysis process.

- Full-text queries apply the analyzer on the search terms before performing the actual search operation.

- That determines the right analyzer to be applied by first checking whether a field-level search_analyzer is defined, and then by checking whether a field-level analyzer is defined.

High level query flow



At indexing time

```
{
  "title": "gods & heroes: rome rising",
   ...
}
```

Index time analyzer

Use the analyzer to break into terms

Terms
gods
heroes
rome
rising

Store terms in the inverted index

At query time

```
{
  "query": {
    "match": {
      "title": "gods heroes"
    }
  }
}
```

Query time analyzer

Use the query/search analyzer to find terms

Terms
gods
heroes

Rewrite in terms of term level queries using individual terms

```
{
  "query": {
    "bool": {
      "should": [
        {"term": {"title": "gods"}},
        {"term": {"title": "heroes"}}
      ]
    }
  }
}
```

Execute the term level query

Term	Frequency	Documents
gods	1	1
heroes	1	1
rome	1	1
rising	1	1

Match query

- A match query is the default query for most full-text search requirements.
- It is one of the high-level queries that is aware of the analyzer used for the underlying field.
- Let's get an understanding of what this means under the hood.

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

Match query

- In fact, in this particular case, the match query gets converted into a term query, such as the following:

```
GET /amazon_products/_search
{
  "query": {
    "term": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

Match query

- Let's see what happens if you execute a match query against a text field, which is a real use case for a full-text query:

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer": "victory multimedia"
    }
  }
}
```

Operator

- By default, if the search term specified results in multiple terms after applying the analyzer, we need a way to combine the results from individual terms.

- As we saw in the preceding example, the default behavior of the match query is to combine the results using the or operator, that is, one of the terms has to be present in the document's field.

Operator

- It can be changed to use the and operator using the following query:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_10.txt

Minimum should match

- Instead of applying the and operator, we can keep the or operator and specify at least how many terms should match in each document for it to be included in the result.

- This allows for finer-grained control:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_11.txt

Fuzziness

- With the fuzziness parameter, we can turn thematch query into a fuzzy query.

- This fuzziness is based on the Levenshtein edit distance, to turn one term into another by making several edits to the original text.

- Edits can be insertions, deletions, substitutions, or the transposition of characters in the original term.

Fuzziness

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer": {
        "query": "victor multimedia",
        "fuzziness": 1
      }
    }
  }
}
```

- If we wanted to still allow more room for errors to be correctable, the fuzziness should be increased to 2.
- For example, a fuzziness of 2 will even match victer. Victory is two edits away from victer:

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer": {
        "query": "victer multimedia",
        "fuzziness": 2
      }
    }
  }
}
```

Match phrase query

- When you want to match a sequence of words, as opposed to separate terms in a document, the match_phrase query can be useful.

- For example, the following text is present as part of the description for one of the products:

real video saltware aquarium on your desktop!

- The match query can include all those documents that have any of the terms, even when they are out of order within the document:

```
GET /amazon_products/_search
{
  "query": {
    "match_phrase": {
      "description": {
        "query": "real video saltware aquarium"
      }
    }
  }
}
```

Match phrase query

- The response will look like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_12.txt

- For example, a slop value of 1 would allow one missing word in the search text but would still match the document:

```
GET /amazon_products/_search
{
  "query": {
    "match_phrase": {
      "description": {
        "query": "real video aquarium",
        "slop": 1
      }
    }
  }
}
```

Multi match query

The multi_match query can be used with different options. We will look at the following options:

- Querying multiple fields with defaults
- Boosting one or more fields
- With types of multi_match queries
- Let's look at each option, one by one.

Querying multiple fields with defaults

- We want to provide a product search functionality in our web application.

- When the end user searches for some terms, we want to query both the title and description fields.

- This can be done using the multi_match query.

Querying multiple fields with defaults

- The following query will find all of the documents that have the terms monitor or aquarium in the title or the description fields:

```
GET /amazon_products/_search
{
  "query": {
    "multi_match": {
      "query": "monitor aquarium",
      "fields": ["title", "description"]
    }
  }
}
```

Boosting one or more fields

- In an e-commerce type of web application, the user intends to search for an item, and they might search for some keywords.

- What if we want the title field to be more important than the description? If one or more of the search terms appears in the title, it is definitely a more relevant product than the ones that have those values only in the description.

Boosting one or more fields

- Let's make the title field three times more important than the description field.
- This can be done by using the following syntax:

```
GET /amazon_products/_search
{
  "query": {
    "multi_match": {
      "query": "monitor aquarium",
      "fields": ["title^3", "description"]
    }
  }
}
```

Writing compound queries

- This class of queries can be used to combine one or more queries to come up with a more complex query.
- Some compound queries convert scoring queries into non-scoring queries and combine multiple scoring and non-scoring queries.
- We will look at the following compound queries:

1. Constant score query
2. Bool query

Constant score query

- Elasticsearch supports querying both structured data and full text.
- While full-text queries need scoring mechanisms to find the best matching documents, structured searches don't need scoring.
- The constant score query allows us to convert a scoring query that normally runs in a query context to a non-scoring filter context.

- For example, a term query is normally run in a query context.
- This means that when Elasticsearch executes a term query, it not only filters documents but also scores all of them:

```
GET /amazon_products/_search
{
  "query": {
    "term": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

- The response contains the score for every document. Please see the following partial response:

```
{
  ...,
  "hits": {
    "total": 3,
    "max_score": 5.966147,
    "hits": [
      {
        "_index": "amazon_products",
        "_type": "products",
        "_id": "AV5rBfasNI_2eZGciIbg",
        "_score": 5.966147,
        "_source": {
          "price": "19.95",
   ...
}
```

- The original query can be converted to run in a filter context using the following constant_score query:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "manufacturer.raw": "victory multimedia"
        }
      }
    }
  }
}
```

- It assigns a neutral score of 1 to each document by default.
- Please note the partial response in the following code:

```
{
  ...,
  "hits": {
    "total": 3,
"max_score": 1,
    "hits": [
      {
        "_index": "amazon_products",
        "_type": "products",
        "_id": "AV5rBfasNI_2eZGcilbg",
"_score": 1,
        "_source": {
          "price": "19.95",
          "description": ...
      }
   ...
}
```

- It is possible to specify a boost parameter, which will assign that score instead of the neutral score of 1:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "manufacturer.raw": "victory multimedia"

        }
      },
"boost": 1.2
    }
  }
}
```

Bool query

- The bool query in Elasticsearch is your Swiss Army knife.
- It can help you write many types of complex queries.
- If you are come from an SQL background, you already know how to filter based on multiple AND and OR conditions in the WHERE clause.
- The bool query allows you to combine multiple scoring and non-scoring queries.

Bool query

- A bool query has the following sections:

```
GET /amazon_products/_search
{
  "query": {
    "bool": {
      "must": [...],        scoring queries executed in query context
      "should": [...],      scoring queries executed in query context
      "filter": {},         non-scoring queries executed in filter context
      "must_not": [...]     non-scoring queries executed in filter context
    }
  }
}
```

Combining OR conditions

- To find all of the products in the price range 10 to 13, OR manufactured by valuesoft:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_13.txt

Combining AND and OR conditions

- Find all products in the price range 10 to 13, AND manufactured by valuesoft or pinnacle:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_14.txt

Adding NOT conditions

- It is possible to add NOT conditions, that is, specifically filtering out certain clauses using the must_not clause in the bool filter.
- For example, find all of the products in the price range 10 to 20, but they must not be manufactured by encore.
- The following query will do just that:

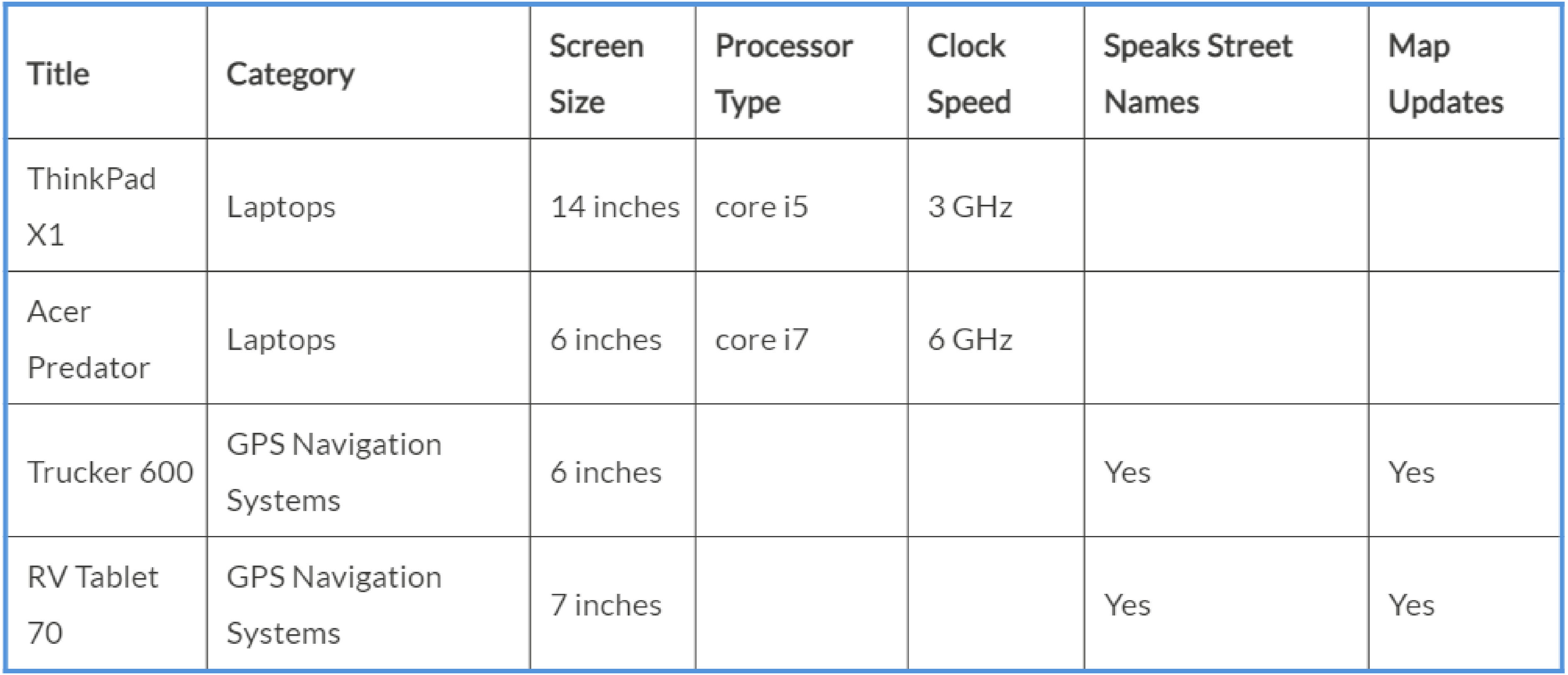
https://github.com/fenago/elasticsearch/blob/master/snippets/3_15.txt

- The bool query with the must_not element is useful for negate any query.
- To negate or apply a NOT filter to the query, it should be wrapped inside the bool with must_not, as follows:

```
GET /amazon_products/_search
{
  "query": {
    "bool": {
      "must_not": {
.... original query to be negated ...
      }
    }
  }
}
```

Modeling relationships

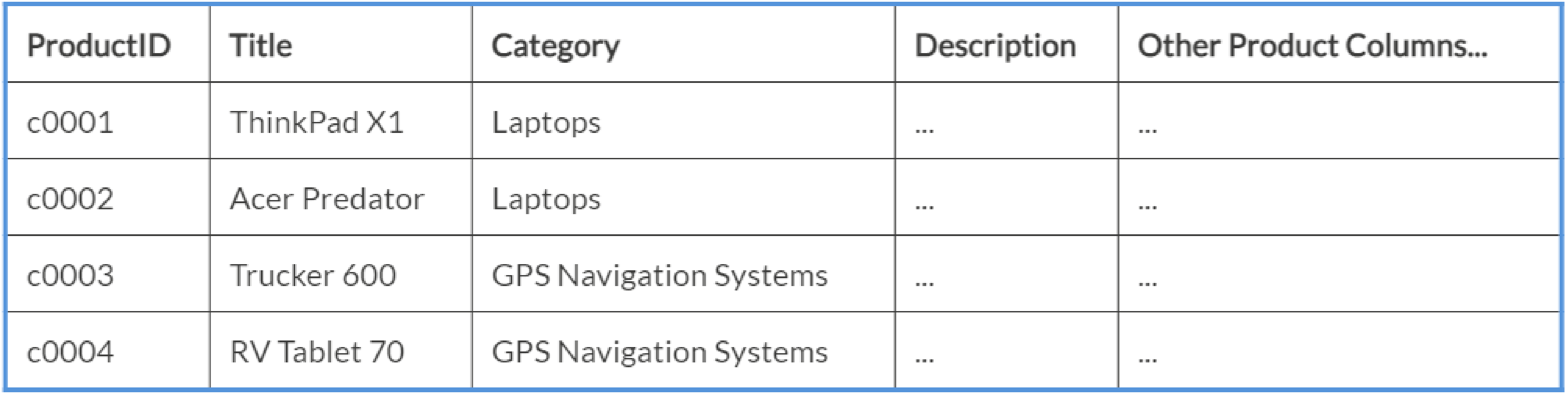
- At the same time, products in the Automobile GPS Systems category may have features such as screen size, whether GPS can speak street names, or whether it has free lifetime map updates available.
- Because we may have tens of thousands of products in hundreds of product categories, we may have tens of thousands of features.
- One solution might be to create one field for each feature.



Title	Category	Screen Size	Processor Type	Clock Speed	Speaks Street Names	Map Updates
ThinkPad X1	Laptops	14 inches	core i5	3 GHz		
Acer Predator	Laptops	6 inches	core i7	6 GHz		
Trucker 600	GPS Navigation Systems	6 inches			Yes	Yes
RV Tablet 70	GPS Navigation Systems	7 inches			Yes	Yes

Modeling relationships

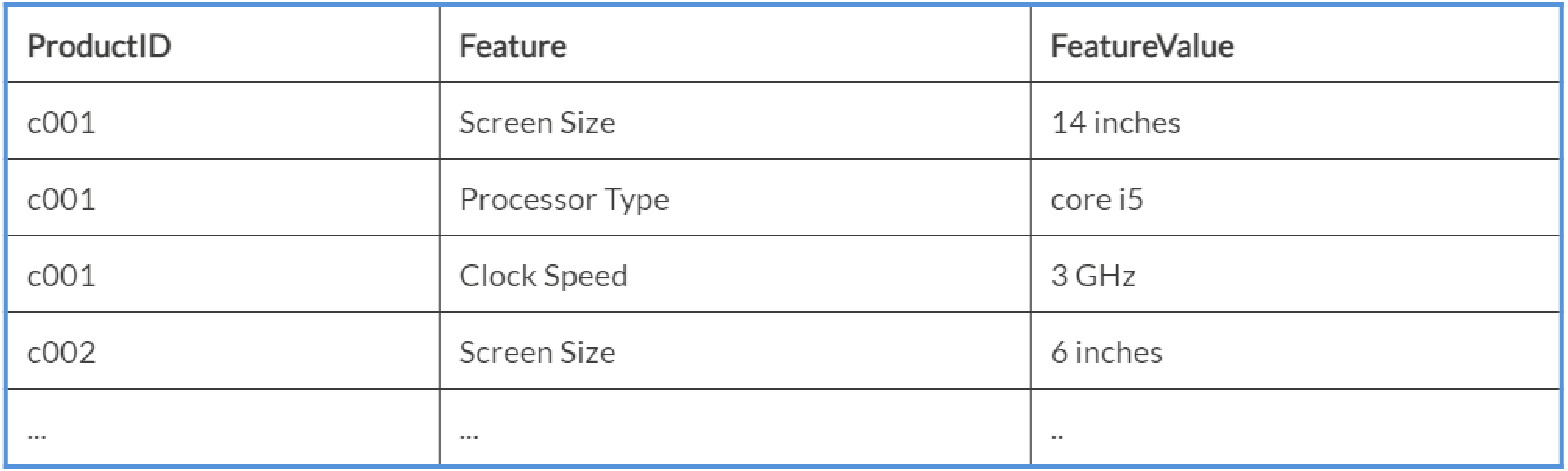
- The product table would be modeled as follows:



ProductID	Title	Category	Description	Other Product Columns...
c0001	ThinkPad X1	Laptops	...	...
c0002	Acer Predator	Laptops	...	...
c0003	Trucker 600	GPS Navigation Systems	...	...
c0004	RV Tablet 70	GPS Navigation Systems	...	...

Modeling relationships

- The features would be modeled as a separate table, where the ProductID and Feature may be a composite primary key:



ProductID	Feature	FeatureValue
c001	Screen Size	14 inches
c001	Processor Type	core i5
c001	Clock Speed	3 GHz
c002	Screen Size	6 inches
...	...	..

- The join datatype mapping that establishes the relationship is defined as follows:

```
PUT /amazon_products_with_features
{
 ...
 "mappings": {
  "doc": {
   "properties": {
    ...
"product_or_feature": {
     "type": "join",
     "relations": {
      "product": "feature"
     }
    },
    ...
   }
  }
 }
}
```

Modeling relationships

- When indexing product records, we use the following syntax:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_16.txt

- The value of product_or_feature, which is set to product suggests that this document is referring to a product.
- A feature record is indexed as follows:

```
PUT amazon_products_with_features/doc/c0001_screen_size?routing=c0001
{
  "product_or_feature": {
    "name": "feature",
    "parent": "c0001"
  },
  "feature_key": "screen_size",
  "feature_value": "14 inches",
  "feature": "Screen Size"
}
```

has_child query

- If you want to get products based on some condition on the features, you can use has_child queries.
- The outline of a has_child query is as follows:

```
GET <index>/_search
{
  "query": {
    "has_child": {
      "type": <the child type against which to run the following query>,
      "query": {
        <any elasticsearch query like term, bool, query_string to be run against the child
type>
      }
    }
  }
}
```

has_child query

- Let's learn about this through an example.

- We want to get all of the products where processor_series is Core i7 from the example dataset that we loaded:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_17.txt

has_child query

- The result of this has_child query is that we get back all the products that satisfy the query mentioned under the has_child element executed against all the features.

- The response should look like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_18.txt

has_parent query

- We want to get all the features of a specific product that has the product id = c0003.

- We can use a has_parent query as follows:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_19.txt

has_parent query

- The result is all the features of that product:

https://github.com/fenago/elasticsearch/blob/master/snippets/3_20.txt

parent_id query

- You guessed correctly; it is the parent_id query, where we obtain all children using the parent's ID:

```
GET /amazon_products_with_features/_search
{
  "query": {
    "parent_id": {
      "type": "feature",
      "id": "c0001"
    }
  }
}
```

Summary

- In this lesson, we took a deep dive into the search capabilities of Elasticsearch.
- We looked at the role of analyzers and the anatomy of an analyzer, saw how to use some of the built-in analyzers that come with Elasticsearch, and saw how to create custom analyzers.
- Along with a solid background on analyzers, we learned about two main types of queries—term-level queries and full-text queries.

Complete Lab 3

4. Analytics with Elasticsearch

Analytics with Elasticsearch

In this lesson, we will look at how Elasticsearch can serve as our analytics engine. We will cover the following topics:

- The basics of aggregations
- Preparing data for analysis
- Metric aggregations
- Bucket aggregations
- Pipeline aggregations

The basics of aggregations

- While learning about searching, we used the following API:

```
POST /<index_name>/_search
{
  "query":
  {
    ... type of query ...
  }
}
```

The basics of aggregations

- The aggregations, or aggs, element allows us to aggregate data.
- All aggregation requests take the following form:

```
POST /<index_name>/_search
{
  "aggs": {
    ... type of aggregation ...
          },
  "query": {  ... type of query ... },              //optional query part
  "size": 0                                     //size typically set to 0
}
```

Bucket aggregations

- Bucket aggregations segment the data in question (defined by the query context) into various buckets that are identified by the buckets key.

- Bucket aggregation evaluates each document in the context by deciding which bucket it falls into.

Bucket aggregations

- For people who are coming from an SQL background, a query that has GROUP BY, such as the following query,does the following withbucket aggregations:

```
SELECT column1, count(*) FROM table1 GROUP BY
column1;
```

Metric aggregations

- Metric aggregations work on numerical fields.
- They compute the aggregate value of a numerical field in the given context.
- For example, let's suppose that we have a table containing the results of a student's examination.
- Each record contains marks obtained by the student.

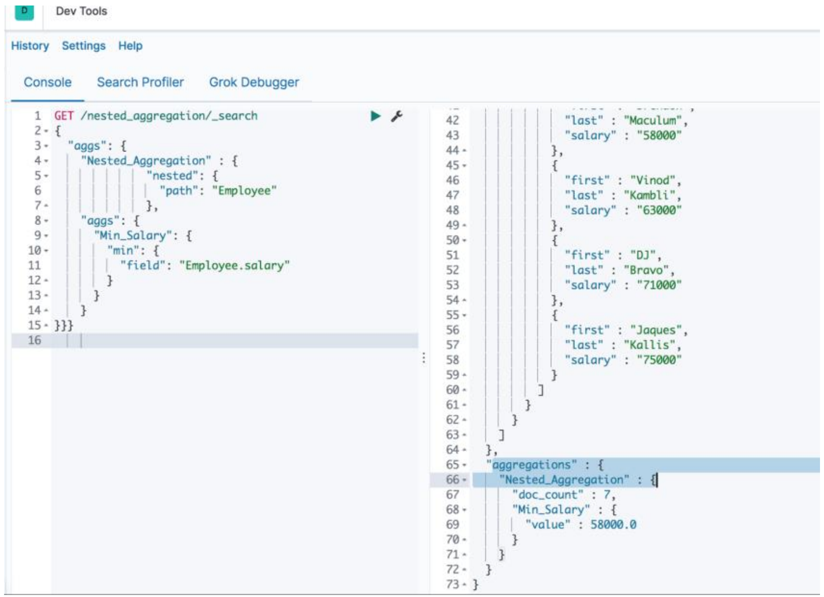
Metric aggregations

- In SQL terms, the following query gives a rough analogy of what a metric aggregation may do:

```
SELECT avg(score) FROM results;
```

Matrix aggregations

- Matrix aggregations were introduced with Elasticsearch version 5.0.
- Matrix aggregations work on multiple fields and compute matrices across all the documents within the query context.



```
GET /nested_aggregation/_search
{
  "aggs": {
    "Nested_Aggregation" : {
              "nested": {
                "path": "Employee"
              },
    "aggs": {
      "Min_Salary": {
        "min": {
          "field": "Employee.salary"
        }
      }
    }
}}}
```

```
              "last" : "Maculum",
              "salary" : "58000"
            },
            {
              "first" : "Vinod",
              "last" : "Kambli",
              "salary" : "63000"
            },
            {
              "first" : "DJ",
              "last" : "Bravo",
              "salary" : "71000"
            },
            {
              "first" : "Jaques",
              "last" : "Kallis",
              "salary" : "75000"
            }
          ]
        }
      }
    ]
  },
  "aggregations" : {
    "Nested_Aggregation" : {
      "doc_count" : 7,
      "Min_Salary" : {
        "value" : 58000.0
      }
    }
  }
}
```

Pipeline aggregations

- Pipeline aggregations are higher order aggregations that can aggregate the output of other aggregations.

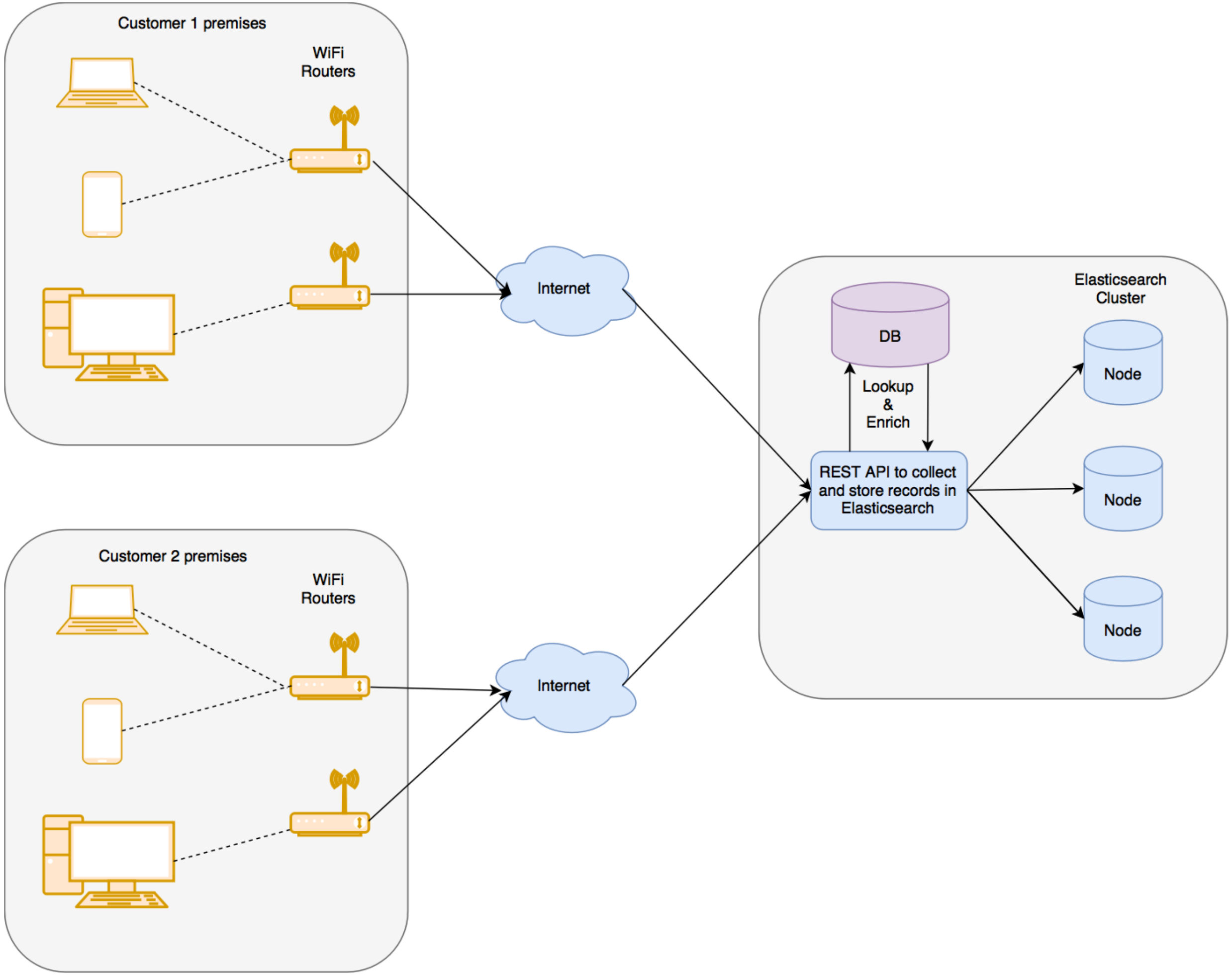
- These are useful for computing something, such as derivatives & We will look at some pipeline aggregations later in this lesson.

- This was an overview about the different types of aggregations supported by Elasticsearch at a high level.

Preparing data for analysis

We will cover the following topics while we prepare and load the data into the local Elasticsearch instance:

- Understanding the structure of the data
- Loading the data using Logstash


Customer 1 premises
WiFi Routers
Internet
Customer 2 premises
WiFi Routers
Internet
Elasticsearch Cluster
DB
Lookup & Enrich
REST API to collect and store records in Elasticsearch
Node
Node
Node

Understanding the structure of the data

- Finally, the enriched records are stored in Elasticsearch in a flat data structure.

- One record looks as follows:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_1.txt

Loading the data using Logstash

- To import the data, please follow the instructions in this course's accompanying source code repository on GitHub, at https://github.com/fenago/elasticsearch. This can be found in the v7.0 branch.
- Please clone or download the repository from GitHub & The instructions for importing data are at the following path within the project.
- Once you have cloned the repository, check out the v7.0 branch.

Loading the data using Logstash

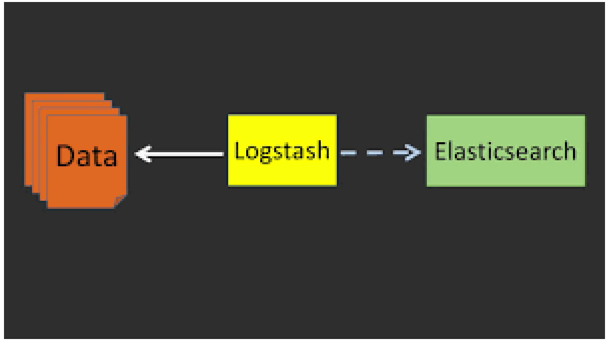
- Once you have imported the data, verify that your data has been imported with the following query:

```
GET /bigginsight/_search
{
  "query": {
    "match_all": {}
   },
  "size": 1
}
```

Loading the data using Logstash

- You should see a response similar to the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_2.txt



Metric aggregations

In this section, we will go over the following metric aggregations:

- Sum, average, min, and max aggregations
- Stats and extended stats aggregations
- Cardinality aggregations

Sum, average, min, and max aggregations

- Finding the sum of a field, the minimum value for a field, the maximum value for a field, or an average, are very common operations.

- For people who are familiar with SQL, the query to find the sum is as follows:

```
SELECT sum(downloadTotal) FROM usageReport;
```

Sum aggregation

- Here is how to write a simple sum aggregation:

```
GET bigginsight/_search?track_total_hits=true
{
 "aggregations": {                    1
    "download_sum": {                    2
      "sum": {                      3
        "field": "downloadTotal"      4
      }
    }
  },
  "size": 0                         5
}
```

- The response should look like the following:

```
{
  "took": 92,
  ...
  "hits": {
    "total" : {
      "value" : 242836,          1
      "relation" : "eq"
    },             1
    "max_score": 0,
    "hits": []
  },
  "aggregations": {              2
    "download_sum": {              3
      "value": 2197438700        4
    }
  }
}
```

- The average aggregation finds an average across all the documents in the querying context:

```
GET bigginsight/_search
{
 "aggregations": {
   "download_average": {                 1
     "avg": {                      2
       "field": "downloadTotal"
     }
   }
  },
  "size": 0
}
```

- The min aggregation is how we will find the minimum value of the downloadTotal field in the entire index/type:

```
GET bigginsight/_search
{
 "aggregations": {
    "download_min": {
      "min": {
        "field": "downloadTotal"
      }
    }
  },
  "size": 0
}
```

Max aggregation

- Here's how we will find the maximum value of the downloadTotal field in the entire index/type:

```
GET bigginsight/_search
{
 "aggregations": {
    "download_max": {
      "max": {
        "field": "downloadTotal"
      }
    }
  },
  "size": 0
}
```

Stats and extended stats aggregations

- These aggregations compute some common statistics in a single request, without having to issue multiple requests.

- This saves resources on the Elasticsearch side, as well, because the statistics are computed in a single pass, rather than being requested multiple times.

- The client code also becomes simpler if you are interested in more than one of these statistics.

- Stats aggregation computes the sum, average, min, max, and count of documents in a single pass:

```
GET bigginsight/_search
{
 "aggregations": {
    "download_stats": {
      "stats": {
        "field": "downloadTotal"
      }
    }
  },
  "size": 0
}
```

- The response should look like the following:

```
{
  "took": 4,
  …,
  "hits": {
    "total" : {
      "value" : 10000,
      "relation" : "gte"
    },
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "download_stats": {
      "count": 242835,
      "min": 0,
      "max": 241213,
      "avg": 9049.102065188297,
      "sum": 2197438700
    }
  }
}
```

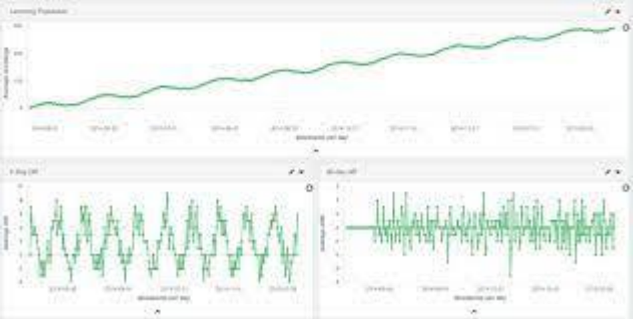
- The extended stats aggregation returns a few more statistics in addition to the ones returned by the stats aggregation:

```
GET bigginsight/_search
{
 "aggregations": {
    "download_estats": {
      "extended_stats": {
        "field": "downloadTotal"
      }
    }
  },
  "size": 0
}
```

Extended stats aggregation

- The response looks like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_3.txt



Cardinality aggregation

- Finding the count of unique elements can be done with the cardinality aggregation. It is similar to finding the result of a query such as the following:

```
select count(*) from (select distinct username from usageReport) u;
```

- Let's look at how we can find out the count of unique users for which we have network traffic data:

```
GET bigginsight/_search
{
 "aggregations": {
    "unique_visitors": {
      "cardinality": {
        "field": "username"
      }
    }
  },
  "size": 0
}
```

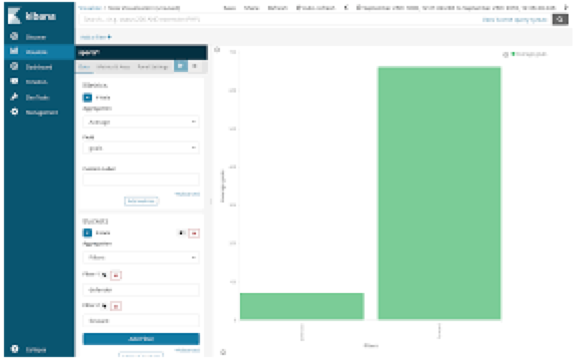
- The cardinality aggregation response is just like the other metric aggregations:

```
{
  "took": 110,
  …,
  "hits": {
    "total" : {
      "value" : 10000,
      "relation" : "gte"
    },
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "unique_visitors": {
      "value": 79
    }
  }
}
```

Bucket aggregations

In this section, we will cover the following topics, keeping the network traffic data example at the center:

- Bucketing on string data
- Bucketing on numerical data
- Aggregating filtered data
- Nesting aggregations
- Bucketing on custom conditions
- Bucketing on date/time data
- Bucketing on geospatial data



Bucketing on string data

Some examples of scenarios in which you may want to segment the data by a string typed field are as follows:

- Segmenting the network traffic data per department
- Segmenting the network traffic data per user
- Segmenting the network traffic data per application, or per category

Terms aggregation

- Which are the top categories, that is, categories that are surfed the most by users?
- We are interested in the most surfed categories – not in terms of the bandwidth used, but just in terms of counts (record counts).
- In a relational database, we could write a query like the following:

```
SELECT category, count(*) FROM usageReport GROUP BY category ORDER BY count(*) DESC;
```

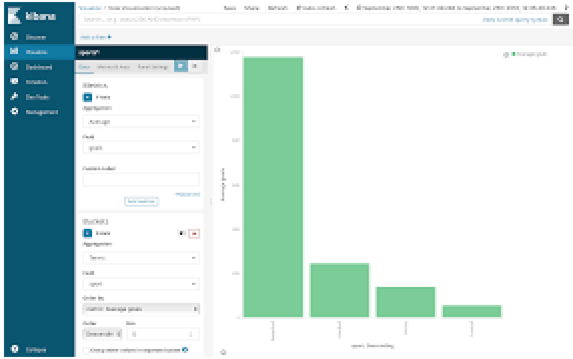
- The Elasticsearch aggregation query, which would do a similar job, can be written as follows:

```
GET /bigginsight/_search
{
  "aggs": {                          1
    "byCategory": {                      2
      "terms": {                     3
        "field": "category"              4
      }
    }
  },
  "size": 0                          5
}
```

Terms aggregation

- The response looks like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_4.txt



Terms aggregation

- Next, we want to find out the top applications in terms of the maximum number of records for each application:

```
GET /bigginsight/_search?size=0
{
  "aggs": {
    "byApplication": {
      "terms": {
        "field": "application"
      }
    }
  }
}
```

- Returns a response like the following:

```
{
  ...,
  "aggregations": {
    "byApplication": {
      "doc_count_error_upper_bound": 6339,
      "sum_other_doc_count": 129191,
      "buckets": [
        {
          "key": "Skype",
          "doc_count": 26115
        },
        ...
}
```

- To get the top n buckets instead of the default 10, we can use the size parameter inside the terms aggregation:

```
GET /bigginsight/_search?size=0
{
 "aggs": {
  "byApplication": {
   "terms": {
    "field": "application",
    "size": 15
   }
  }
 }
}
```

Bucketing on numerical data

- Another common scenario is when we want to segment or slice the data into various buckets, based on a numerical field.
- For example, we may want to slice the product data by different price ranges, such as up to $10, $10 to $50, $50 to $100, and so on.
- You may want to segment the data by age group, employee count, and so on.

Histogram aggregation

- We have some records of network traffic usage data.
- The usage field tells us about the number of bytes that are used for uploading or downloading data.
- Let's try to divide or slice all the data based on the usage:

```
POST /bigginsight/_search?size=0
{
 "aggs": {
 "by_usage": {
 "histogram": {
 "field": "usage",
 "interval": 1000
 }
 }
 }
}
```

- The response should look like the following (truncated for brevity):

```
{
  ...,
  "aggregations": {
    "by_usage": {
      "buckets": [
        {
          "key": 0.0,
          "doc_count": 30060
        },
        {
          "key": 1000.0,
          "doc_count": 42880
        },
        {
          "key": 2000.0,
          "doc_count": 42041
        },
...
}
```

Range aggregation

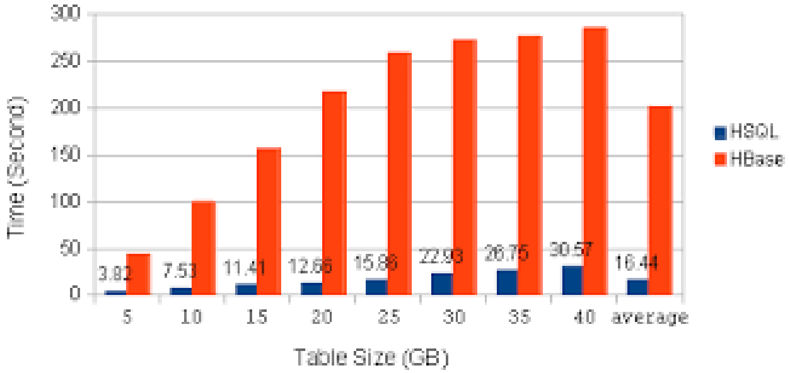
- The to value is exclusive, and is not included in the current bucket's range:

```
POST /bigginsight/_search?size=0
{
  "aggs": {
    "by_usage": {
      "range": {
        "field": "usage",
        "ranges": [
          { "to": 1024 },
          { "from": 1024, "to": 102400 },
          { "from": 102400 }
        ]
      }
    }
  }
}
```

Range aggregation

- The response looks like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_5.txt



- It is possible to specify custom key labels for the range buckets, as follows:

```
POST /bigginsight/_search?size=0
{
 "aggs": {
  "by_usage": {
   "range": {
    "field": "usage",
    "ranges": [
     { "key": "Upto 1 kb", "to": 1024 },
     { "key": "1 kb to 100 kb","from": 1024, "to": 102400 },
     { "key": "100 kb and more", "from": 102400 }
    ]
   }
  }
 }
}
```

Aggregations on filtered data

- In our quest to learn about different bucket aggregations, let's take a very short detour to understand how to apply aggregations on filtered data.
- So far, we have been applying all of our aggregations on all the data of the given index/type.
- In the real world, you will almost always need to apply some filters before applying aggregations (either metric or bucket aggregations).

- Now, what we want to do is find the top category for a specific customer, not for all of the customers:

```
GET /bigginsight/_search?size=0&track_total_hits=true
{
  "query": {
    "term": {
      "customer": "Linkedin"
    }
  },
  "aggs": {
    "byCategory": {
      "terms": {
        "field": "category"
      }
    }
  }
}
```

Aggregations on filtered data

- Let's look at the response of this query to understand this better:

```
{
 "took": 18,
 ...,
 "hits": {
"total" : {
   "value" : 76607,
   "relation" : "eq"
  },
  "max_score": 0,
  "hits": []
 },
 ...
}
```

```
GET /bigginsight/_search?size=0
{
  "query": {
    "bool": {
      "must": [
        {"term": {"customer": "Linkedin"}},
        {"range": {"time": {"gte": 1506277800000, "lte": 1506294200000}}}
      ]
    }
  },
  "aggs": {
    "byCategory": {
      "terms": {
        "field": "category"
      }
    }
  }
}
```

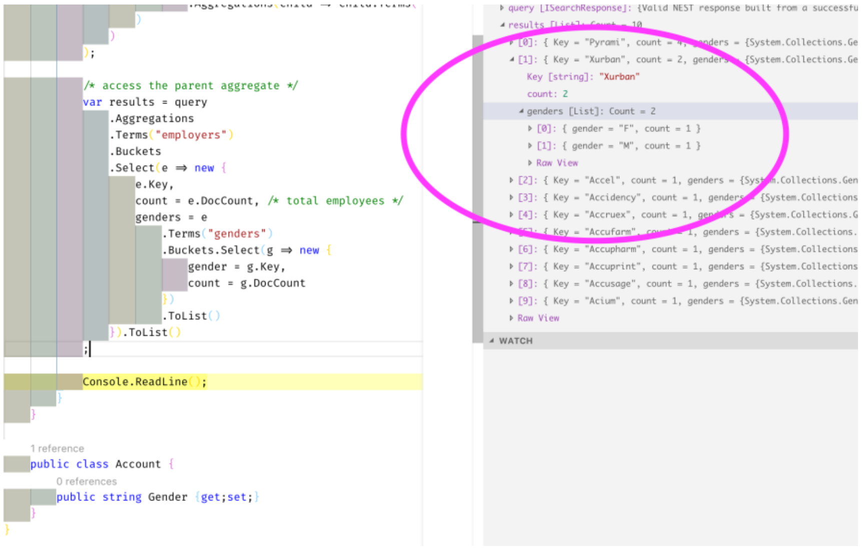
Nesting aggregations

- Bucket aggregations split the context into one or more buckets.
- We can restrict the context of the aggregation by specifying the query element, as we saw in the previous section.
- When a metric aggregation is nested inside a bucket aggregation, the metric aggregation is computed within each bucket.

Nesting aggregations

- The following query does exactly this.
- Please refer to the annotated numbers, which correspond to the three main objectives of the the following query:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_6.txt



```
                .Aggregations(child => child.Terms(
            )
        )
    );

    /* access the parent aggregate */
    var results = query
        .Aggregations
        .Terms("employers")
        .Buckets
        .Select(e => new {
            e.Key,
            count = e.DocCount, /* total employees */
            genders = e
                .Terms("genders")
                .Buckets.Select(g => new {
                    gender = g.Key,
                    count = g.DocCount
                })
                .ToList()
        }).ToList()
    ;

    Console.ReadLine();
  }
}

1 reference
public class Account {
    0 references
    public string Gender {get;set;}
}
}
```

```
▸ query [ISearchResponse]: {Valid NEST response built from a successfu
◂ results [List]: Count = 10
  ▸ [0]: { Key = "Pyrami", count = 4, genders = {System.Collections.Ge
  ◂ [1]: { Key = "Xurban", count = 2, genders = {System.Collections.Ge
      Key [string]: "Xurban"
      count: 2
    ◂ genders [List]: Count = 2
      ▸ [0]: { gender = "F", count = 1 }
      ▸ [1]: { gender = "M", count = 1 }
      ▸ Raw View
  ▸ [2]: { Key = "Accel", count = 1, genders = {System.Collections.Gen
  ▸ [3]: { Key = "Accidency", count = 1, genders = {System.Collections
  ▸ [4]: { Key = "Accruex", count = 1, genders = {System.Collections.G
  ▸ [5]: { Key = "Accufarm", count = 1, genders = {System.Collections.
  ▸ [6]: { Key = "Accupharm", count = 1, genders = {System.Collections
  ▸ [7]: { Key = "Accuprint", count = 1, genders = {System.Collections
  ▸ [8]: { Key = "Accusage", count = 1, genders = {System.Collections.
  ▸ [9]: { Key = "Acium", count = 1, genders = {System.Collections.Gen
  ▸ Raw View
◂ WATCH
```

Nesting aggregations

- The response looks like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_7.txt

- Please see the following partial query, which has been modified to sort the buckets in descending order of the total_usage metric:

```
GET /bigginsight/usageReport/_search
{
  ...,
  "aggs": {
    "by_users": {
      "terms": {
        "field": "username",
"order": { "total_usage": "desc"}
      },
      "aggs": {
        ...
...
}
```

Nesting aggregations

- Who are the top two users in each department, given the total bandwidth consumed by each user?

- The following query will help us get that answer:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_8.txt

Bucketing on custom conditions

- Sometimes, what we want is more control over how the buckets are created.
- The aggregations that we have looked at so far have dealt with a single type of field.
- If the given field that we want to slice data from is of the string type, we generally use the terms aggregation.
- If the field is of the numerical type, we have a few choices, including histogram, range aggregation, and others, to slice the data into different segments.

Filter aggregation

- We want to create a bucket of all records that have category = Chat:

```
POST /bigginsight/_search?size=0
{
 "aggs": {
   "chat": {
     "filter": {
       "term": {
         "category": "Chat"
       }
     }
   }
 }
}
```

- The response should look like the following:

```
{
  "took": 4,
  …,
  "hits": {
    "total" : {
      "value" : 10000,
      "relation" : "gte"
    },
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "chat": {
      "doc_count": 52277
    }
  }
}
```

Filters aggregation

- With filters aggregation, you can create multiple buckets, each with its own specified filter that will cause the documents satisfying that filter to fall into the related bucket. Let's look at an example.

- Suppose that we want to create multiple buckets to understand how much of the network traffic was caused by the Chat category.

- It allows us to write arbitrary filters to create buckets:

```
GET bigginsight/_search?size=0
{
  "aggs": {
    "messages": {
      "filters": {
        "filters": {
          "chat": { "match": { "category": "Chat" }},
          "skype": { "match": { "application": "Skype" }},
          "other_than_skype": {
            "bool": {
              "must": {"match": {"category": "Chat"}},
              "must_not": {"match": {"application": "Skype"}}
            }
          }
        }
      }
    }
  }
}
```

Bucketing on date/time data

In the context of the network traffic example that we are going through, the following questions can be answered through time series analysis of the data:

- How are the bandwidth requirements changing for my organization over a period of time?
- Which are the top applications, over a period of time, in terms of bandwidth usage?

Date Histogram aggregation

Using Date Histogram aggregation, we will see how we can create buckets on a date field. In the process, we will go through the following stages:

- Creating buckets across time periods
- Using a different time zone
- Computing other metrics within sliced time intervals
- Focusing on a specific day and changing intervals

Creating buckets across time periods

- The following query will create buckets on different values of time, grouped by one-day intervals:

```
GET /bigginsight/_search?size=0                    1
{
  "aggs": {
    "counts_over_time": {
      "date_histogram": {                          2
        "field": "time",
        "interval": "1d"                           3
      }
    }
  }
}
```

Creating buckets across time periods

- The response looks like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_9.txt

Using a different time zone

```
GET /bigginsight/_search?size=0
{
  "aggs": {
    "counts_over_time": {
      "date_histogram": {
        "field": "time",
        "interval": "1d",
        "time_zone": "+05:30"
      }
    }
  }
}
```

Using a different time zone

- The response now looks like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_10.txt

Computing other metrics within sliced time intervals

- We have just sliced the data across time by using the Date Histogram to create the buckets on the time field.
- This gave us the document counts in each bucket. Next, we will try to answer the following question:
- What is the day-wise total bandwidth usage for a given customer?

```
GET /bigginsight/_search?size=0
{
  "query": { "term": {"customer": "Linkedin"} },
  "aggs": {
    "counts_over_time": {
      "date_histogram": {
        "field": "time",
        "interval": "1d",
        "time_zone": "+05:30"
      },
      "aggs": {
        "total_bandwidth": {
          "sum": { "field": "usage" }
        }
      }
    }
  }
}
```

- The following is the shortened response to the query:

```
{
  ..,
  "aggregations": {
    "counts_over_time": {
      "buckets": [
        {
          "key_as_string": "2017-09-23T00:00:00.000+05:30",
          "key": 1506105000000,
          "doc_count": 18892,
          "total_bandwidth": {
            "value": 265574303
          }
        },
        ...
      ]
    }
  }
}
```

Focusing on a specific day and changing intervals

- What we are doing is also called drilling down in the data, Often, the result of the previous query is displayed as a line chart, with time on the x axis and data used on the y axis.
- If we want to zoom in on a specific day from that line chart, the following query can be useful:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_11.txt

Focusing on a specific day and changing intervals

- The shortened response would look like the following:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_12.txt

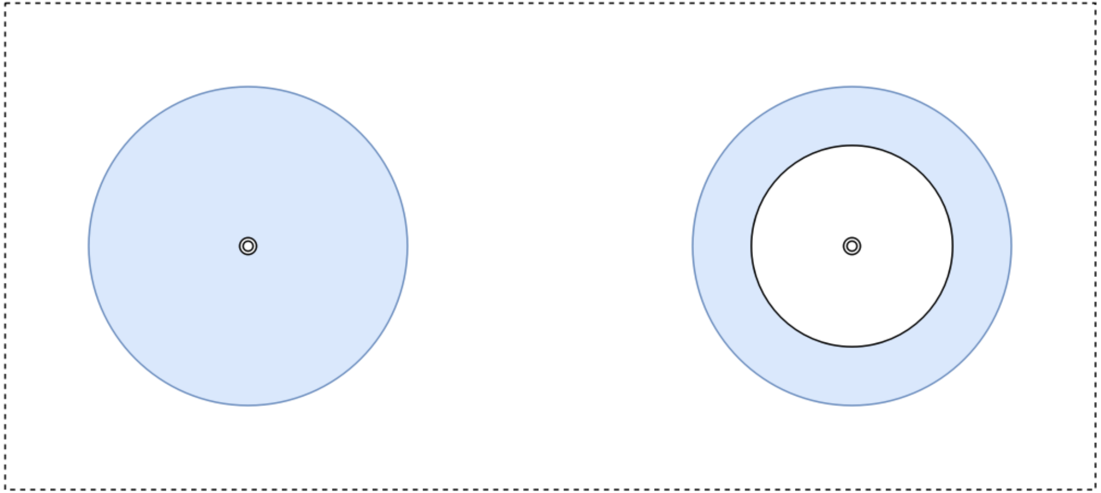
Bucketing on geospatial data

- Another powerful feature of bucket aggregation is the ability to do geospatial analysis on the data.

- If your data contains fields of the geo-point datatype, where the coordinates are captured, you can perform some interesting analysis, which can be rendered on a map to give you better insight into the data.

Geodistance aggregation

- Geodistance aggregation helps in creating buckets of distances from a given geo-point.
- This can be better illustrated using a diagram:



- The shaded area is from the center up to the given radius, forming a circle:

```
GET bigginsight/_search?size=0
{
  "aggs": {
    "within_radius": {
      "geo_distance": {
        "field": "location",
        "origin": {"lat": 23.102869,"lon": 72.595692},
        "ranges": [{"to": 5}]
      }
    }
  }
}
```

- let's look at what happens if you specify both from and to in the geodistance aggregation.
- This will correspond to the right circle in the preceding diagram:

```
GET bigginsight/_search?size=0
{
  "aggs": {
    "within_radius": {
      "geo_distance": {
        "field": "location",
        "origin": {"lat": 23.102869,"lon": 72.595692},
        "ranges": [{"from": 5, "to": 10}]
      }
    }
  }
}
```

GeoHash grid aggregation

- Lower values of precision represent larger geographical areas, while higher values represent smaller, more precise geographical areas:

```
GET bigginsight/_search?size=0
{
  "aggs": {
    "geo_hash": {
      "geohash_grid": {
        "field": "location",
        "precision": 7
      }
    }
  }
}
```

GeoHash grid aggregation

- The data that we have in our network traffic example is spread over a very small geographical area, so we have used a precision of 7.

- The supported values for precision are from 1 to 12. Let's look at the response to this request:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_13.txt

GeoHash grid aggregation

- When you try a precision value of 9, you will see the following response:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_14.txt

Pipeline aggregations

Pipeline aggregations are a relatively new feature, and they are still experimental. At a high level, there are two types of pipeline aggregation:

- Parent pipeline aggregations have the pipeline aggregation nested inside other aggregations
- Sibling pipeline aggregations have the pipeline aggregation as the sibling of the original aggregation from which pipelining is done

Calculating the cumulative sum of usage over time

- Using cumulative sum aggregation, we can also compute the cumulative bandwidth usage at the end of every hour of the day.

- Let's look at the query and try to understand it:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_15.txt

Calculating the cumulative sum of usage over time

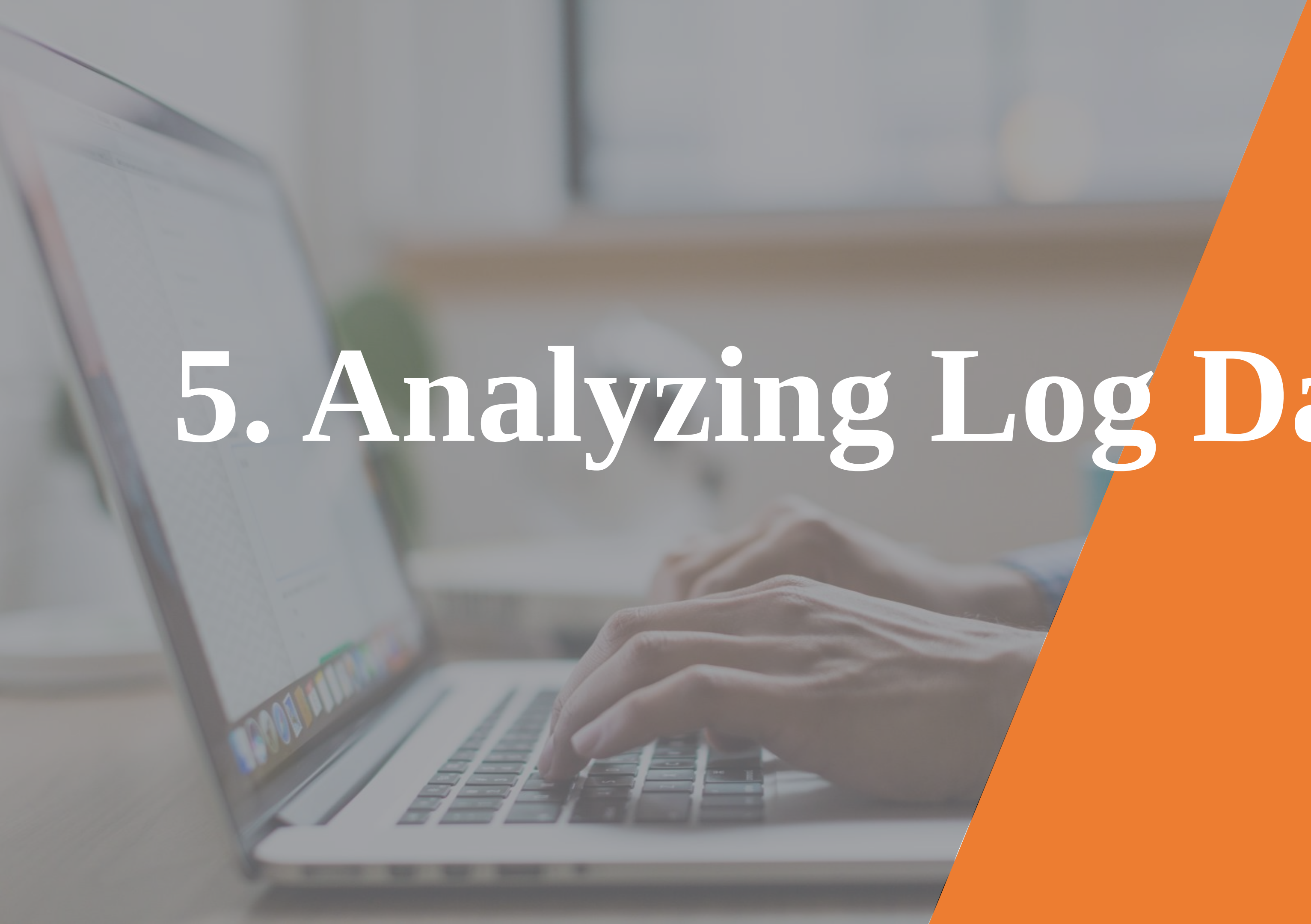
- The response should look as follows.
- It has been truncated for brevity:

https://github.com/fenago/elasticsearch/blob/master/snippets/4_16.txt

Summary

- In this lesson, you learned how to use Elasticsearch to build powerful analytics applications.
- We covered how to slice and dice the data to get powerful insight.
- We started with metric aggregation and dealt with numerical datatypes.
- We then covered bucket aggregation in order to find out how to slice the data into buckets or segments, in order to drill down into specific segments.

Complete Lab 4

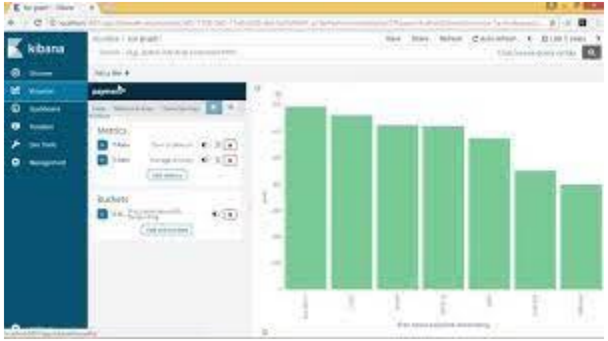


5. Analyzing Log Data

Analyzing Log Data

In this lesson, we will be exploring Logstash, another key component of the Elastic Stack that is mainly used as an ETL (Extract, Transform, and Load) engine. We will also be exploring the following topics:

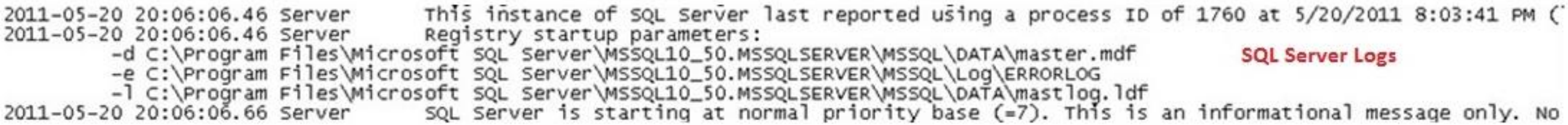
- Log Analysis challenges
- Using Logstash
- The Logstash architecture
- Overview of Logstash plugins
- Ingest node



Log analysis challenges

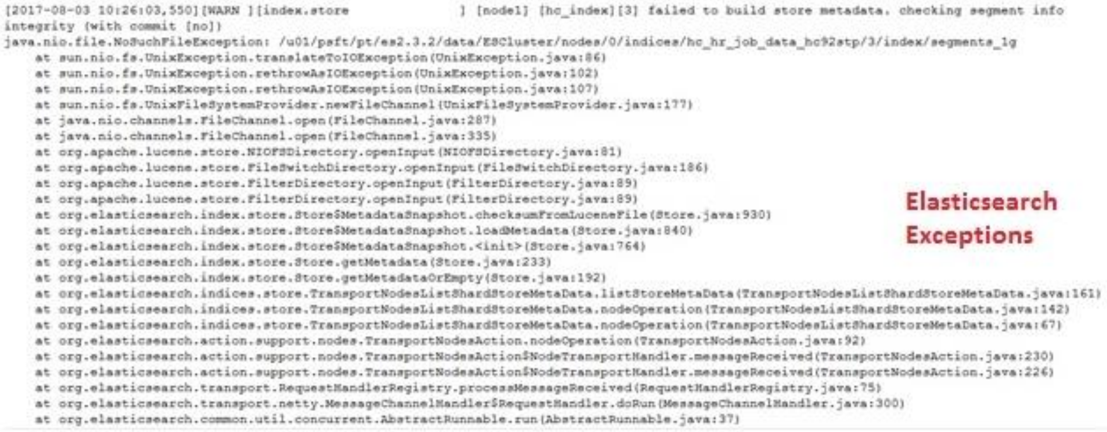
- Logs are defined as records of incidents or observations.
- Logs are generated by a wide variety of resources, such as systems, applications, devices, humans, and so on.
- A log is typically made of two things; that is, a timestamp (the time the event was generated) and data (the information related to the event):

Log = Timestamp + Data



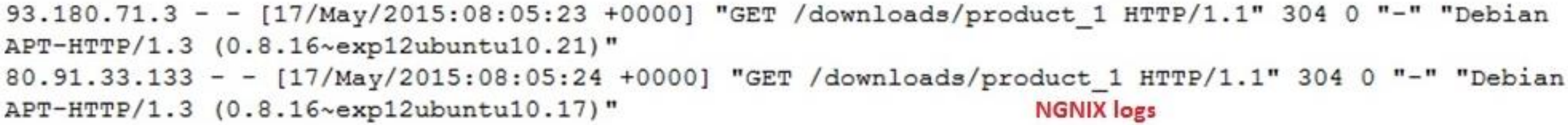
```
2011-05-20 20:06:06.46 Server      This instance of SQL Server last reported using a process ID of 1760 at 5/20/2011 8:03:41 PM (
2011-05-20 20:06:06.46 Server      Registry startup parameters:
         -d C:\Program Files\Microsoft SQL Server\MSSQL10_50.MSSQLSERVER\MSSQL\DATA\master.mdf
         -e C:\Program Files\Microsoft SQL Server\MSSQL10_50.MSSQLSERVER\MSSQL\Log\ERRORLOG
         -l C:\Program Files\Microsoft SQL Server\MSSQL10_50.MSSQLSERVER\MSSQL\DATA\mastlog.ldf
2011-05-20 20:06:06.66 Server      SQL Server is starting at normal priority base (=7). This is an informational message only. No
```

SQL Server Logs



```
[2017-08-03 10:26:03,550][WARN ][index.store              ] [node1] [hc_index][3] failed to build store metadata. checking segment info
integrity (with commit [no])
java.nio.file.NoSuchFileException: /u01/psft/pt/es2.3.2/data/ESCluster/nodes/0/indices/hc_hr_job_data_hc92stp/3/index/segments_1g
    at sun.nio.fs.UnixException.translateToIOException(UnixException.java:86)
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:102)
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:107)
    at sun.nio.fs.UnixFileSystemProvider.newFileChannel(UnixFileSystemProvider.java:177)
    at java.nio.channels.FileChannel.open(FileChannel.java:287)
    at java.nio.channels.FileChannel.open(FileChannel.java:335)
    at org.apache.lucene.store.NIOFSDirectory.openInput(NIOFSDirectory.java:81)
    at org.apache.lucene.store.FileSwitchDirectory.openInput(FileSwitchDirectory.java:186)
    at org.apache.lucene.store.FilterDirectory.openInput(FilterDirectory.java:89)
    at org.apache.lucene.store.FilterDirectory.openInput(FilterDirectory.java:89)
    at org.elasticsearch.index.store.Store$MetadataSnapshot.checksumFromLuceneFile(Store.java:930)
    at org.elasticsearch.index.store.Store$MetadataSnapshot.loadMetadata(Store.java:840)
    at org.elasticsearch.index.store.Store$MetadataSnapshot.<init>(Store.java:764)
    at org.elasticsearch.index.store.Store.getMetadata(Store.java:233)
    at org.elasticsearch.index.store.Store.getMetadataOrEmpty(Store.java:192)
    at org.elasticsearch.indices.store.TransportNodesListShardStoreMetaData.listStoreMetaData(TransportNodesListShardStoreMetaData.java:161)
    at org.elasticsearch.indices.store.TransportNodesListShardStoreMetaData.nodeOperation(TransportNodesListShardStoreMetaData.java:142)
    at org.elasticsearch.indices.store.TransportNodesListShardStoreMetaData.nodeOperation(TransportNodesListShardStoreMetaData.java:67)
    at org.elasticsearch.action.support.nodes.TransportNodesAction.nodeOperation(TransportNodesAction.java:92)
    at org.elasticsearch.action.support.nodes.TransportNodesAction$NodeTransportHandler.messageReceived(TransportNodesAction.java:230)
    at org.elasticsearch.action.support.nodes.TransportNodesAction$NodeTransportHandler.messageReceived(TransportNodesAction.java:226)
    at org.elasticsearch.transport.RequestHandlerRegistry.processMessageReceived(RequestHandlerRegistry.java:75)
    at org.elasticsearch.transport.netty.MessageChannelHandler$RequestHandler.doRun(MessageChannelHandler.java:300)
    at org.elasticsearch.common.util.concurrent.AbstractRunnable.run(AbstractRunnable.java:37)
```

Elasticsearch
Exceptions



```
93.180.71.3 - - [17/May/2015:08:05:23 +0000] "GET /downloads/product_1 HTTP/1.1" 304 0 "-" "Debian
APT-HTTP/1.3 (0.8.16~exp12ubuntu10.21)"
80.91.33.133 - - [17/May/2015:08:05:24 +0000] "GET /downloads/product_1 HTTP/1.1" 304 0 "-" "Debian
APT-HTTP/1.3 (0.8.16~exp12ubuntu10.17)"
```

NGINX logs

Log analysis challenges

- Correlating events occur across multiple systems at the same time.
- Some example time formats that can be seen in the logs are as follows:

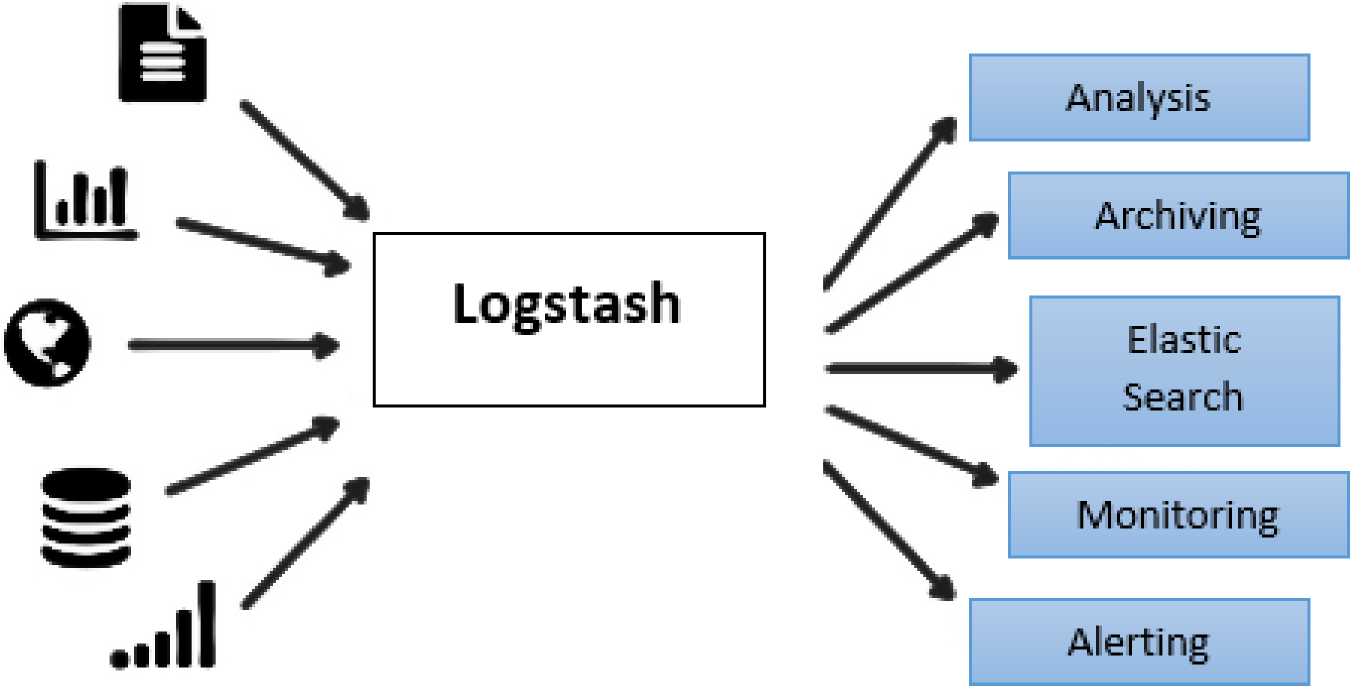
```
Nov 14 22:20:10
[10/Oct/2000:13:55:36 -0700]
172720538
053005 05:45:21
1508832211657
```

Using Logstash

- Logstash is a popular open source data collection engine with real-time pipelining capabilities.

- Logstash allows us to easily build a pipeline that can help in collecting data from a wide variety of input sources, and parse, enrich, unify, and store it in a wide variety of destinations.

Using Logstash



Installation and configuration

Prerequisites

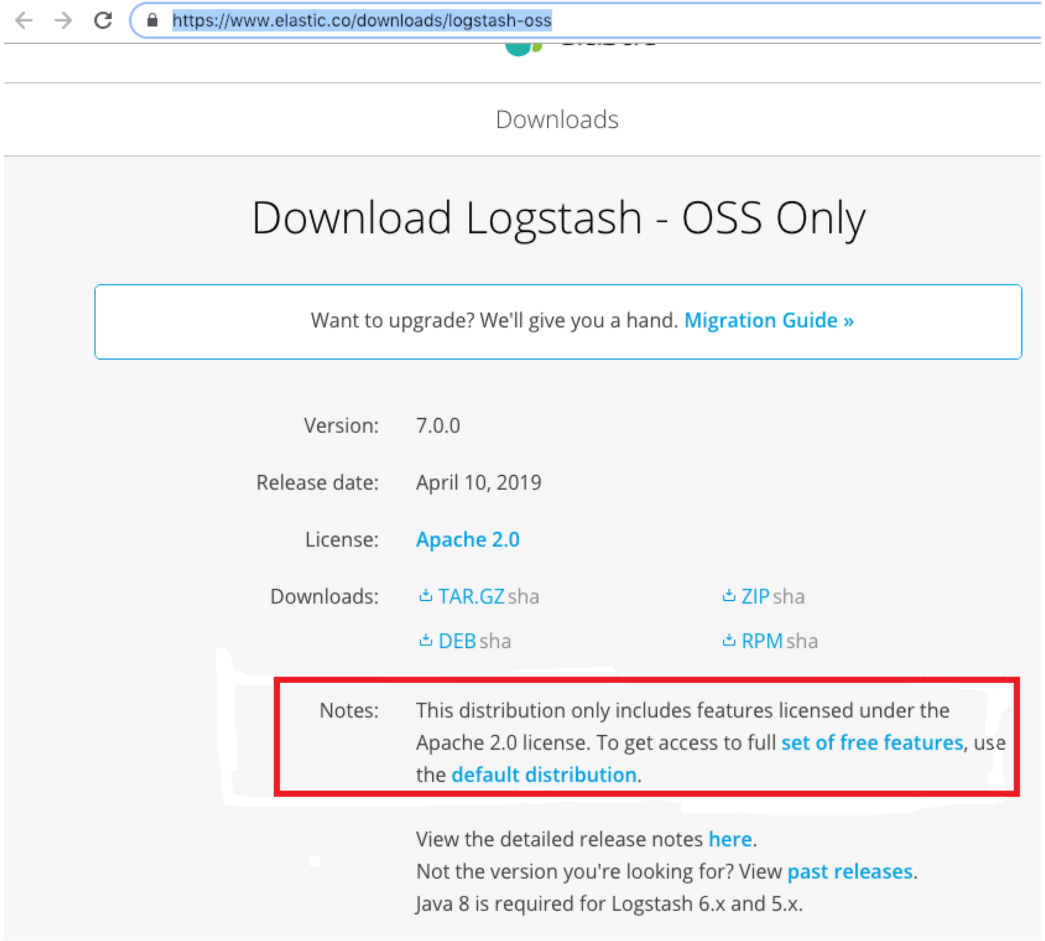
- Java runtime is required to run Logstash. Logstash requires Java 8.
- Make sure that JAVA_HOME is set as an environment variable, and to check your Java version, run the following command:

```
java -version
```

- You should see the following output:

```
java version "1.8.0_65"Java(TM) SE
RuntimeEnvironment(build 1.8.0_65-b17)JavaHotSpot(TM)64-
BitServer VM (build 25.65-b01, mixed mode)
```

Downloading and installing Logstash



https://www.elastic.co/downloads/logstash-oss

Downloads

Download Logstash - OSS Only

Want to upgrade? We'll give you a hand. Migration Guide »

Version:	7.0.0
Release date:	April 10, 2019
License:	Apache 2.0
Downloads:	TAR.GZ sha ZIP sha DEB sha RPM sha
Notes:	This distribution only includes features licensed under the Apache 2.0 license. To get access to full set of free features, use the default distribution.

View the detailed release notes here.
Not the version you're looking for? View past releases.
Java 8 is required for Logstash 6.x and 5.x.

Installing on Windows

- Rename the downloaded file logstash-7.0.0.zip. Unzip the downloaded file.
- Once unzipped, navigate to the newly created folder, as shown in the following code snippet:

```
E:\>cd logstash-oss-7.0.0
```

Installing on Linux

- Unzip the tar.gz package and navigate to the newly created folder, as follows:

```
$>tar -xzf logstash-oss-7.0.0.tar.gz
$>cd logstash-7.0.0/
```

Running Logstash

- Let's ensure that Logstash works fine after installation by running the following command with a simple configuration (the logstash pipeline) as a parameter:

```
E:\logstash-7.0.0\bin>logstash -e "input { stdin { } } output
{ stdout {} }"
```

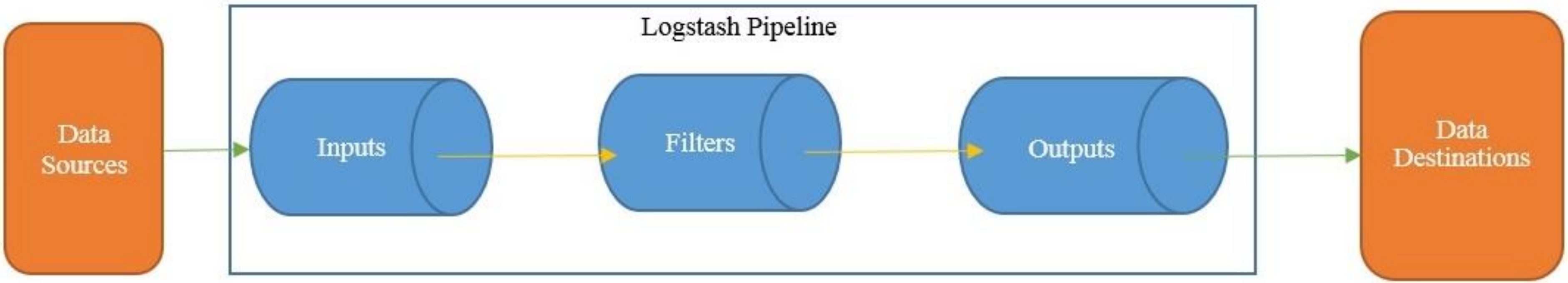
Running Logstash

- You should get the following logs:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_1.txt

The Logstash architecture

- The Logstash event processing pipeline has three stages, that is, Inputs, Filters, and Outputs.
- A Logstash pipeline has two required elements, that is, input and output, and one option element known as filters:



The Logstash architecture

- The Logstash pipeline is stored in a configuration file that ends with a .conf extension.
- The three sections of the configuration file are as follows:

```
input
{
}
filter
{
}
output
{
}
```

The Logstash architecture

- Let's use the same configuration that we used in the previous section, with some little modifications, and store it in a file:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_2.txt

The Logstash architecture

- Let's run Logstash using this new pipeline/configuration that's stored in the simple.conf file, as follows:

```
E:\logstash-7.0.0\bin>logstash -f ../conf/simple.conf
```

The Logstash architecture

- Once Logstash has started, enter any input, say, LOGSTASH IS AWESOME, and you should see the response, as follows:

```
{
   "@version" => "1",
       "host" => "SHMN-IN",
  "@timestamp" => 2017-11-03T11:42:56.221Z,
    "message" => "LOGSTASH IS AWESOME\r"
}
```

Overview of Logstash plugins

- By default, as part of the Logstash distribution, many common plugins are available out of the box.
- You can verify the list of plugins that are part of the current installation by executing the following command:

```
E:\logstash-7.0.0\bin>logstash-plugin list
```

Overview of Logstash plugins

- Using the --group flag, followed by either input, filter, output, or codec, you can find the list of installed input, filters, output, codecs, and plugins, respectively.
- For example:

```
E:\logstash-7.0.0\bin>logstash-plugin list --group filter
```

- You can list all the plugins containing a name fragment by passing the name fragment to logstash-plugin list.
- For example:

```
E:\logstash-7.0.0\bin>logstash-plugin list 'pager'
```

Installing or updating plugins

- If the required plugin is not bundled by default, you can install it using the bin\logstash-plugin install command.
- For example, to install the logstash-output-email plugin, execute the following command:

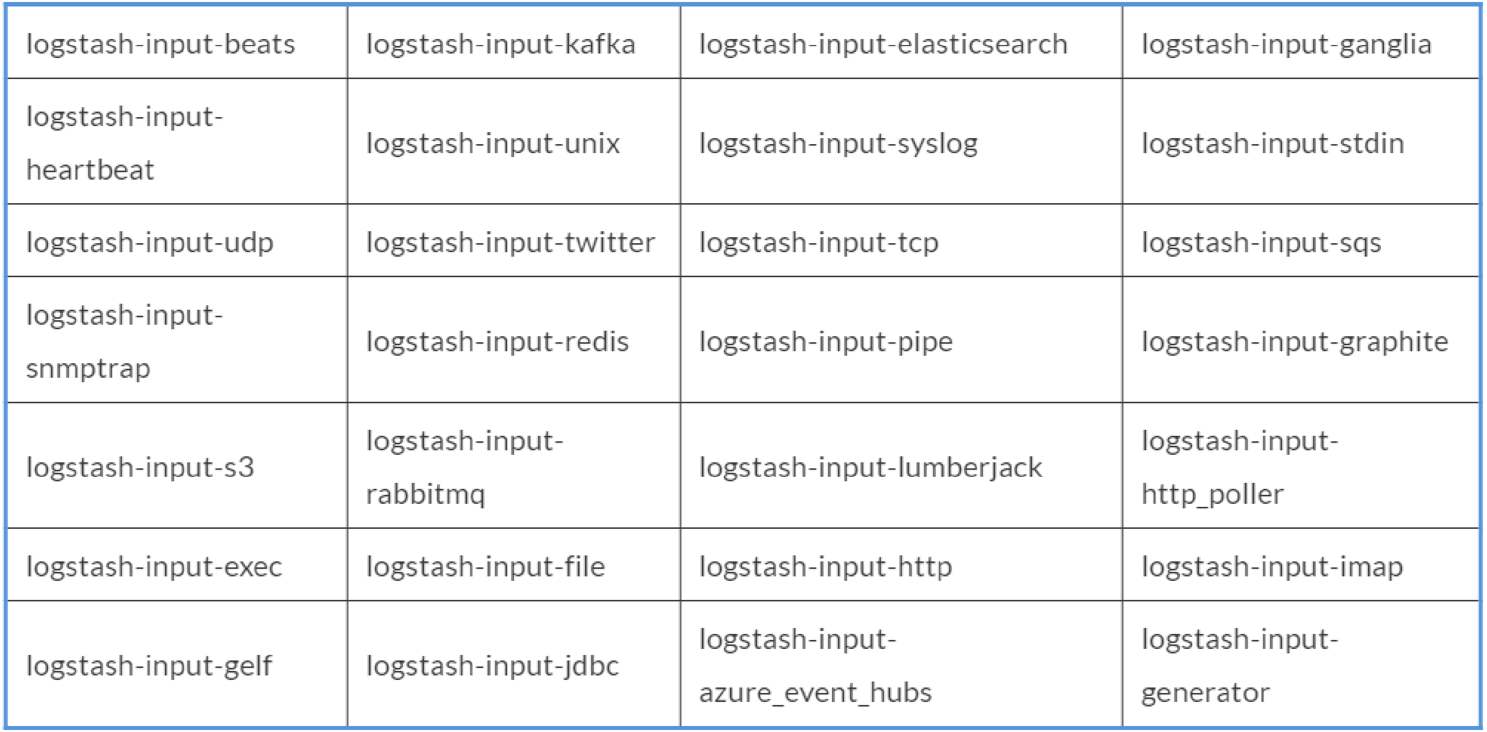
```
E:\logstash-7.0.0\bin>logstash-plugin install logstash-output-email
```

Installing or updating plugins

- By using the bin\logstash-plugin update command and passing the plugin name as a parameter to the command, you can get the latest version of the plugin:

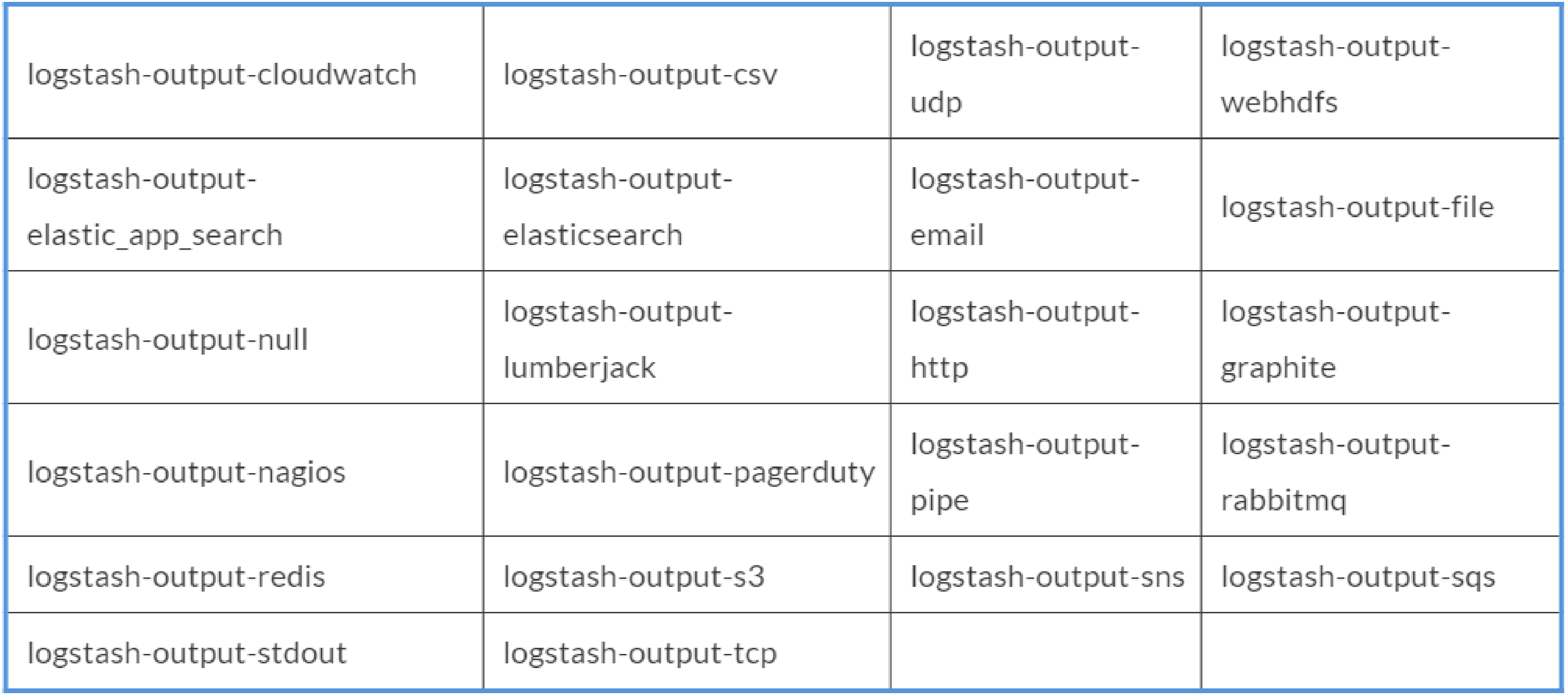
```
E:\logstash-oss-7.0.0\bin>logstash-plugin update
logstash-output-s3
```

Input plugins



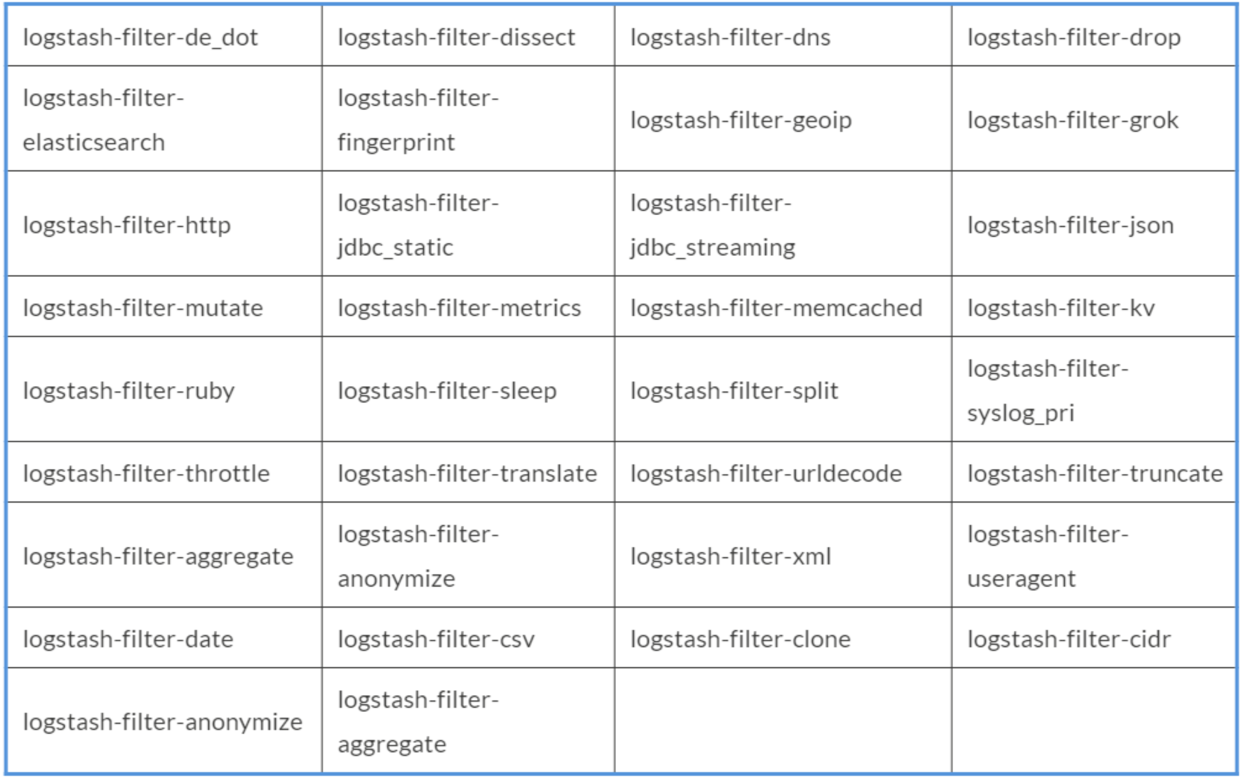
logstash-input-beats	logstash-input-kafka	logstash-input-elasticsearch	logstash-input-ganglia
logstash-input-heartbeat	logstash-input-unix	logstash-input-syslog	logstash-input-stdin
logstash-input-udp	logstash-input-twitter	logstash-input-tcp	logstash-input-sqs
logstash-input-snmptrap	logstash-input-redis	logstash-input-pipe	logstash-input-graphite
logstash-input-s3	logstash-input-rabbitmq	logstash-input-lumberjack	logstash-input-http_poller
logstash-input-exec	logstash-input-file	logstash-input-http	logstash-input-imap
logstash-input-gelf	logstash-input-jdbc	logstash-input-azure_event_hubs	logstash-input-generator

Output plugins



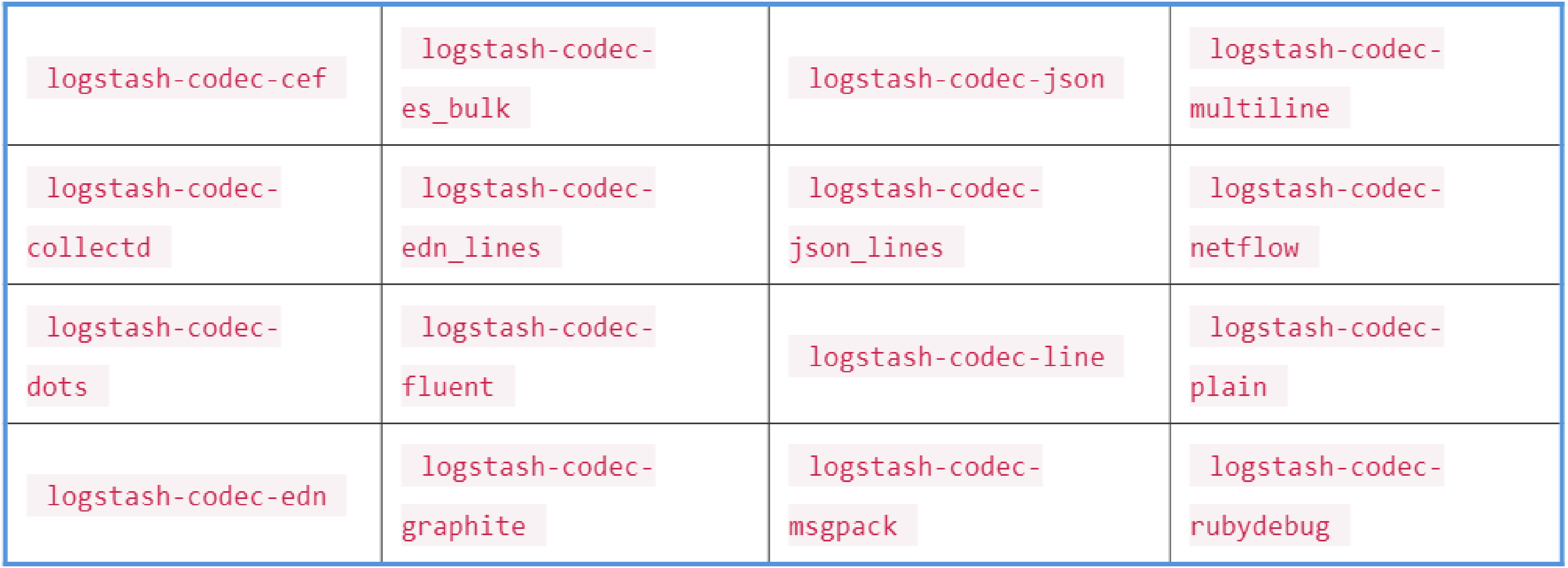
logstash-output-cloudwatch	logstash-output-csv	logstash-output-udp	logstash-output-webhdfs
logstash-output-elastic_app_search	logstash-output-elasticsearch	logstash-output-email	logstash-output-file
logstash-output-null	logstash-output-lumberjack	logstash-output-http	logstash-output-graphite
logstash-output-nagios	logstash-output-pagerduty	logstash-output-pipe	logstash-output-rabbitmq
logstash-output-redis	logstash-output-s3	logstash-output-sns	logstash-output-sqs
logstash-output-stdout	logstash-output-tcp		

Filter plugins



logstash-filter-de_dot	logstash-filter-dissect	logstash-filter-dns	logstash-filter-drop
logstash-filter-elasticsearch	logstash-filter-fingerprint	logstash-filter-geoip	logstash-filter-grok
logstash-filter-http	logstash-filter-jdbc_static	logstash-filter-jdbc_streaming	logstash-filter-json
logstash-filter-mutate	logstash-filter-metrics	logstash-filter-memcached	logstash-filter-kv
logstash-filter-ruby	logstash-filter-sleep	logstash-filter-split	logstash-filter-syslog_pri
logstash-filter-throttle	logstash-filter-translate	logstash-filter-urldecode	logstash-filter-truncate
logstash-filter-aggregate	logstash-filter-anonymize	logstash-filter-xml	logstash-filter-useragent
logstash-filter-date	logstash-filter-csv	logstash-filter-clone	logstash-filter-cidr
logstash-filter-anonymize	logstash-filter-aggregate		

Codec plugins



`logstash-codec-cef`	`logstash-codec-es_bulk`	`logstash-codec-json`	`logstash-codec-multiline`
`logstash-codec-collectd`	`logstash-codec-edn_lines`	`logstash-codec-json_lines`	`logstash-codec-netflow`
`logstash-codec-dots`	`logstash-codec-fluent`	`logstash-codec-line`	`logstash-codec-plain`
`logstash-codec-edn`	`logstash-codec-graphite`	`logstash-codec-msgpack`	`logstash-codec-rubydebug`

Exploring input plugins

File

- The file plugin is used to stream events from file(s) line by line. It works in a similar fashion to the tail -0f Linux\Unix command.
- For each file, it keeps track of any changes in the file, and the last location from where the file was read, only sends the data since it was last read.
- It also automatically detects file rotation, This plugin also provides the option to read the file from the beginning of the file.

- Let's take some example configurations to understand this plugin better:

```
#sample configuration 1
#simple1.conf
input
{ file{
    path => "/usr/local/logfiles/*"
 }
}
 output
{
 stdout {
 codec => rubydebug
 }
}
```

Exploring input plugins

- The preceding configuration specifies the streaming of all the new entries (that is, tailing the files) to the files found under the /usr/local/logfiles/ location:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_3.txt

Beats

- The Beats input plugin allows Logstash to receive events from the Elastic Beats framework.
- Beats are a collection of lightweight daemons that collect operational data from your servers and ship to configured outputs such as Logstash, Elasticsearch, Redis, and so on.
- There are several Beats available, including Metricbeat, Filebeat, Winlogbeat, and so on.

Beats

- By using the beats input plugin, we can make Logstash listen on desired ports for incoming Beats connections:

```
#beats.conf
input {
  beats {
 port => 1234
  }
}
output {
  elasticsearch {
  }
}
```

Beats

- port is the only required setting for this plugin, The preceding configuration makes Logstash listen for incoming Beats connections and index into Elasticsearch.
- When you start Logstash with the preceding configuration, you may notice Logstash starting an input listener on port 1234 in the logs, as follows:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_4.txt

Beats

- You can start multiple listeners to listen for incoming Beats connections as follows:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_5.txt

JDBC

- This plugin is used to import data from a database to Logstash.
- Each row in the results set would become an event, and each column would get converted into fields in the event.
- Using this plugin, you can import all the data at once by running a query, or you can periodically schedule the import using cron syntax (using the schedule parameter/setting).

JDBC

- Let's look at an example:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_6.txt

IMAP

- This plugin is used to read emails from an IMAP server.
- This plugin can be used to read emails and, depending on the email context, the subject of the email, or specific senders, it can be conditionally processed in Logstash and used to raise JIRA tickets, pagerduty events, and so on.
- The required configurations are host, password, and user.

IMAP

- User and password are where you need to specify the user credentials to authenticate/connect to the IMAP server:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_7.txt

Elasticsearch

- This plugin is used for transferring events from Logstash to Elasticsearch.
- This plugin is the recommended approach for pushing events/log data from Logstash to Elasticsearch.
- Once the data is in Elasticsearch, it can be easily visualized using Kibana.
- This plugin requires no mandatory parameters and it automatically tries to connect to Elasticsearch, which is hosted on localhost:9200.

Elasticsearch

- The simple configuration of this plugin would be as follows:

```
#elasticsearch1.conf
input {
  stdin{
  }
 }
output {
 elasticsearch {
  }
}
```

Elasticsearch

- Often, Elasticsearch will be hosted on a different server that's usually secure, and we might want to store the incoming data in specific indexes.

- Let's look at an example of this:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_8.txt

CSV

- This plugin is used for storing output in the CSV format.
- The required parameters for this plugin are the path parameter, which is used to specify the location of the output file, and fields, which specifies the field names from the event that should be written to the CSV file.
- If a field does not exist on the event, an empty string will be written.

CSV

- Let's look at an example. In the following configuration, Elasticsearch is queried against the apachelogs index for all documents matching statuscode:200, and the message, @timestamp, and host fields are written to a .csv file:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_9.txt

Kafka

- This plugin is used to write events to a Kafka topic.
- It uses the Kafka Producer API to write messages to a topic on the broker.
- The only required configuration is the topic_id.

Kafka

- Let's look at a basic Kafka configuration:

```
#kafka.conf
input {
  stdin{
  }
 }
output {
kafka {
            bootstrap_servers => "localhost:9092"
            topic_id => 'logstash'
    }
}
```

PagerDuty

- This output plugin will send notifications based on preconfigured services and escalation policies.

- The only required parameter for this plugin is the service_key to specify the Service API Key.

PagerDuty

- Let's look at a simple example with basic pagerduty configuration.

- In the following configuration, Elasticsearch is queried against the ngnixlogs index for all documents matching statuscode:404, and pagerduty events are raised for each document returned by Elasticsearch:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_10.txt

JSON

- This codec is useful if the data consists of .json documents, and is used to encode (if used in output plugins) or decode (if used in input plugins) the data in the .json format.
- If the data being sent is a JSON array at its root, multiple events will be created (that is, one per element).

JSON

- The simple usage of a JSON codec plugin is as follows:

```
input{
    stdin{
    codec => "json"
    }
}
```

Rubydebug

- This codec will output your Logstash event data using the Ruby Awesome Print library.
- The simple usage of this codec plugin is as follows:

```
output{
    stdout{
    codec => "rubydebug"
    }
}
```

Multiline

- The sample usage of this codec plugin is shown in the following snippet:

```
input {
  file {
    path =>"/var/log/access.log"
    codec => multiline {pattern =>"^\s "
      negate =>false
      what =>"previous"}}}
```

Ingest node

- Prior to Elasticsearch 5.0, if we wanted to preprocess documents before indexing them to Elasticsearch, then the only way was to make use of Logstash or preprocess them programmatically/manually and then index them to Elasticsearch.

- Elasticsearch lacked the ability to preprocess/transform the documents, and it just indexed the documents as they were.

Ingest node

- If an Elasticsearch node is implemented with the default configuration, by default, it would be master, data, and ingest enabled (that is, it would act as a master node, data node, and ingest node).
- To disable ingest on a node, configure the following setting in the elasticsearch.yml file:

```
node.ingest: false
```

Ingest node

- To preprocess a document before indexing, we must define the pipeline (which contains sequences of steps known as processors for transforming an incoming document).
- To use a pipeline, we simply specify the pipeline parameter on an index or bulk request to tell the ingest node which pipeline to use:

```
POST my_index/my_type?pipeline=my_pipeline_id
{"key":"value"}
```

Defining a pipeline

- A pipeline defines a series of processors.
- Each processor transforms the document in some way.
- Each processor is executed in the order in which it is defined in the pipeline.
- A pipeline consists of two main fields: a description and a list of processors.

Defining a pipeline

- The typical structure of a pipeline is as follows:

```
{"description":"...","processors":[...]}
```

Put pipeline API

- Let's look at an example. As we can see in the following code, we have defined a new pipeline named firstpipeline, which converts the value present in the message field into upper case:

```
curl -X PUT http://localhost:9200/_ingest/pipeline/firstpipeline -H
'content-type: application/json'
  -d '{
  "description" : "uppercase the incoming value in the message field",
  "processors" : [
    {
      "uppercase" : {
        "field": "message"
      }
    }
  ]
}'
```

Defining a pipeline

- Let's look at an example for this. As we can see in the following code, we have created a new pipeline called secondpipeline that converts the uppercase value present in the message field and renames the message field to data.
- It creates a new field named label with the testlabel value:

https://github.com/fenago/elasticsearch/blob/master/snippets/5_11.txt

Defining a pipeline

- Let's make use of the second pipeline to index a sample document:

```
curl -X PUT
'http://localhost:9200/myindex/mytpe/1?pipeline=secondpipeline' -H 'content-type: application/json' -d '{
  "message":"elk is awesome"
}'
```

- Let's retrieve the same document and validate the transformation:

```
curl -X GET   http://localhost:9200/myindex/mytpe/1   -H 'content-type: application/json'
```

Response:

```
{
    "_index": "myindex",
    "_type": "mytpe",
    "_id": "1",
    "_version": 1,
    "found": true,
    "_source": {
"label": "testlabel",
        "data": "ELK IS AWESOME"
    }
}
```

Get pipeline API

- This API is used to retrieve the definition of an existing pipeline.
- Using this API, you can find the details of a single pipeline definition or find the definitions of all the pipelines.
- The command to find the definition of all the pipelines is as follows:

```
curl -X GET http://localhost:9200/_ingest/pipeline -H
'content-type: application/json'
```

Get pipeline API

- To find the definition of an existing pipeline, pass the pipeline ID to the pipeline API.

- The following is an example of finding the definition of the pipeline named secondpipeline:

```
curl -X GET
http://localhost:9200/_ingest/pipeline/secondpipeline  -H
'content-type: application/json'
```

Delete pipeline API

- The delete pipeline API deletes pipelines by ID or wildcard match.

- The following is an example of deleting the pipeline named firstpipeline:

```
curl -X DELETE
http://localhost:9200/_ingest/pipeline/firstpipeline  -H
'content-type: application/json'
```

Simulate pipeline API

- This pipeline can be used to simulate the execution of a pipeline against the set of documents provided in the body of the request.

- You can either specify an existing pipeline to execute against the provided documents or supply a pipeline definition in the body of the request.

Simulate pipeline API

- The following is an example of simulating an existing pipeline:

```
curl -X POST http://localhost:9200/_ingest/pipeline/firstpipeline/_simulate -H 'content-type: application/json' -d '{
  "docs" : [
    { "_source": {"message":"first document"} },
    { "_source": {"message":"second document"} }
  ]
}'
```

- The following is an example of a simulated request, with the pipeline definition in the body of the request itself:

```
curl -X POST http://localhost:9200/_ingest/pipeline/_simulate -H 'content-type: application/json' -d '{
  "pipeline" : {
    "processors":[
      {
 "join": {
          "field": "message",
          "separator": "-"
        }
      }]
  },
  "docs" : [
    { "_source": {"message":["first","document"]} }
  ]
}'
```

Summary

- In this lesson, we laid out the foundations of Logstash. We walked you through the steps to install and configure Logstash to set up basic data pipelines, and studied Logstash's architecture.
- We also learned about the ingest node that was introduced in Elastic 5.x, which can be used instead of a dedicated Logstash setup.

Complete Lab 5

6. Building Data Pipelines with Logstash

TriveraTech
TECHNOLOGY TRAINING

Building Data Pipelines with Logstash

In this lesson, we will be covering the following topics:

- Parsing and enriching logs using Logstash
- The Elastic Beats platform
- Installing and configuring Filebeats for shipping logs

Parsing and enriching logs using Logstash

- The analysis of structured data is easier and helps us find meaningful/deeper analysis, rather than trying to perform analysis on unstructured data.

- Most analysis tools depend on structured data, Kibana, which we will be making use of for analysis and visualization, can be used effectively if the data in Elasticsearch is right

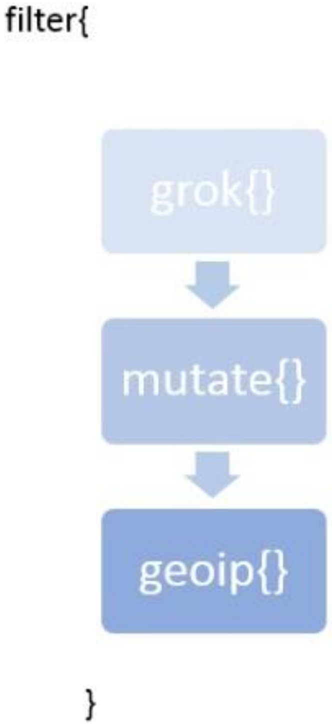
Parsing and enriching logs using Logstash

- Log data is typically made up of two parts, as follows:

logdata = timestamp + data

Filter plugins

- A filter plugin is used to perform transformations on data.
- It allows us to combine one or more plugins, and the order of the plugins defines the order in which the data is transformed.
- A sample filter section in a Logstash pipeline would look as follows:



CSV filter

- This filter is useful for parsing .csv files.
- This plugin takes an event containing CSV data, parses it, and stores it as individual fields.
- Let's take some sample data and use a CSV filter to parse data out of it.
- Store the following data in a file named users.csv:

```
FName,LName,Age,Salary,EmailId,Gender
John,Thomas,25,50000,John.Thomas,m
Raj, Kumar,30,5000,Raj.Kumar,f
Rita,Tony,27,60000,Rita.Tony,m
```

- In doing so, you can let the plugin know that it needs to detect column names automatically, as follows:

```
#csv_file.conf
input {
   file{
      path => "D:\es\logs\users.csv"
      start_position => "beginning"
      }
}
filter {
    csv{
    autodetect_column_names => true
}
}
output {
  stdout {
    codec => rubydebug
        }
}
```

Mutate filter

- You can perform general mutations on fields using this filter.
- The fields in the event can be renamed, converted, stripped, and modified.
- Let's enhance the csv_file.conf file we created in the previous section with the mutate filter and understand its usage.
- The following code block shows the use of the mutate filter:

https://github.com/fenago/elasticsearch/blob/master/snippets/6_1.txt

Grok filter

- This is a powerful and often used plugin for parsing the unstructured data into structured data, thus making the data easily queryable/filterable.
- In simple terms, Grok is a way of matching a line against a pattern (which is based on a regular expression) and mapping specific parts of the line to dedicated fields.
- The general syntax of a grok pattern is as follows:

```
%{PATTERN:FIELDNAME}
```

Grok filter

- By default, groked fields are strings.
- To cast either to float or int values, you can use the following format:

%{PATTERN:FIELDNAME:type}

Grok filter

- Patterns consist of a label and a regex. For example:

USERNAME [a-zA-Z0-9._-]+

- Patterns can contain other patterns, too; for example:

HTTPDATE %{MONTHDAY}/%{MONTH}/%{YEAR}:%{TIME} %{INT}

Grok filter

- If a pattern is not available, then you can use a regular expression by using the following format:

```
(?<field_name>regex)
```

Grok filter

- For example, regex (?<phone>\d\d\d-\d\d\d-\d\d\d\d) would match telephone numbers, such as 123-123-1234, and place the parsed value into the phone field.

- Let's look at some examples to understand grok better:

https://github.com/fenago/elasticsearch/blob/master/snippets/6_2.txt

Grok filter

- If the input line is of the "2017-10-11T21:50:10.000+00:00 tmi_19 001 this is a random message" format, then the output would be as follows:

https://github.com/fenago/elasticsearch/blob/master/snippets/6_3.txt

Date filter

- This plugin is used for parsing the dates from the fields, This plugin is very handy and useful when working with time series events.

- By default, Logstash adds a @timestamp field for each event, representing the time it processed the event.

- But the user might be interested in the actual timestamp of the generated event rather than the processed timestamp.

Date filter

- We can use the plugin like so:

```
filter {
    date {
    match => [ "timestamp", "dd/MMM/YYYY:HH:mm:ss Z"
]
         }
}
```

Date filter

- The user can keep the event time processed by Logstash, too:

```
filter {
    date {
    match => [ "timestamp", "dd/MMM/YYYY:HH:mm:ss Z"
]
target => "event_timestamp"
          }
}
```

Date filter

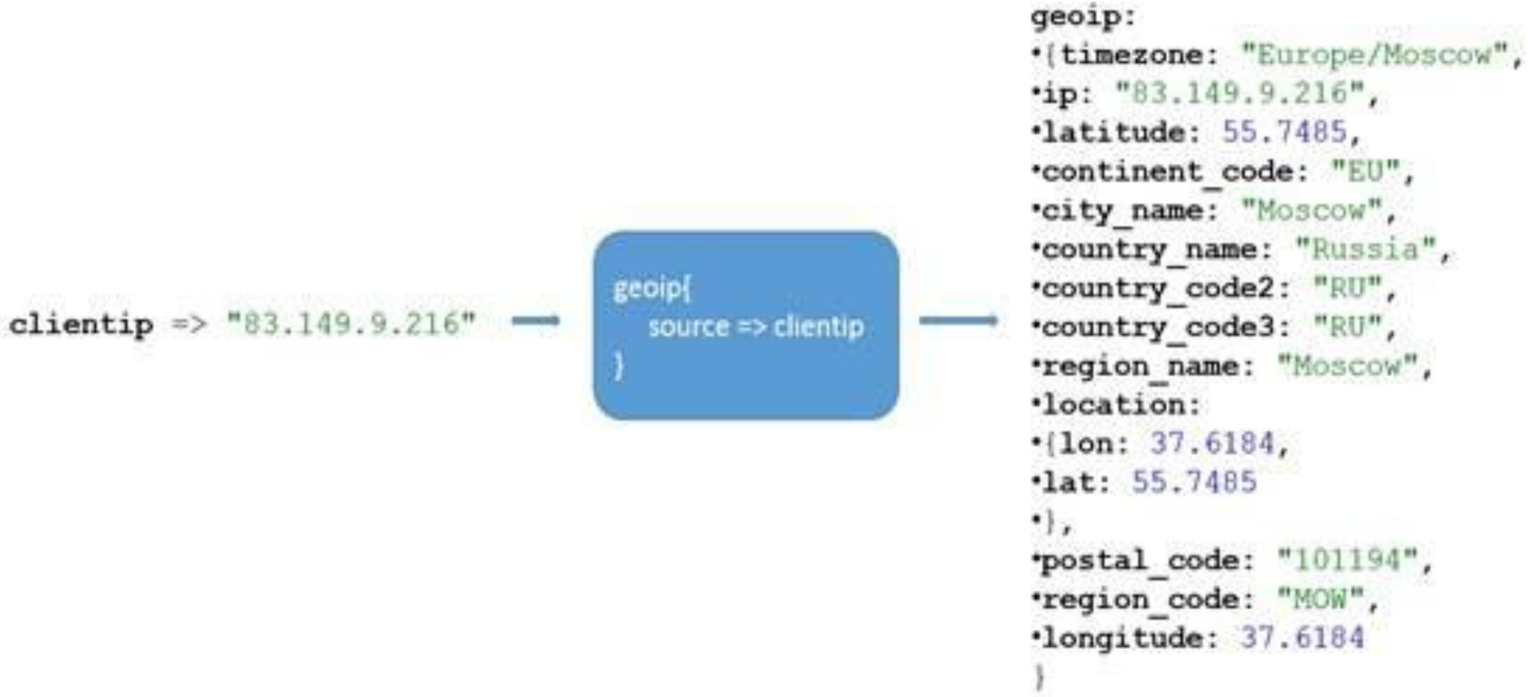
- If the time field has multiple possible time formats, then those can be specified as an array of values to the match parameter:

```
match => [ "eventdate", "dd/MMM/YYYY:HH:mm:ss Z",
"MMM dd yyyy HH:mm:ss","MMM d yyyy HH:mm:ss",
"ISO8601" ]
```

Geoip filter

- This plugin is used to enrich the log information.
- Given the IP address, it adds the geographical location of the IP address.
- It finds the geographical information by performing a lookup against the GeoLite2 City database for valid IP addresses and populates fields with results.
- The GeoLite2 City database is a product of the Maxmind organization and is available under the CCA-ShareAlike 4.0 license.

Geoip filter

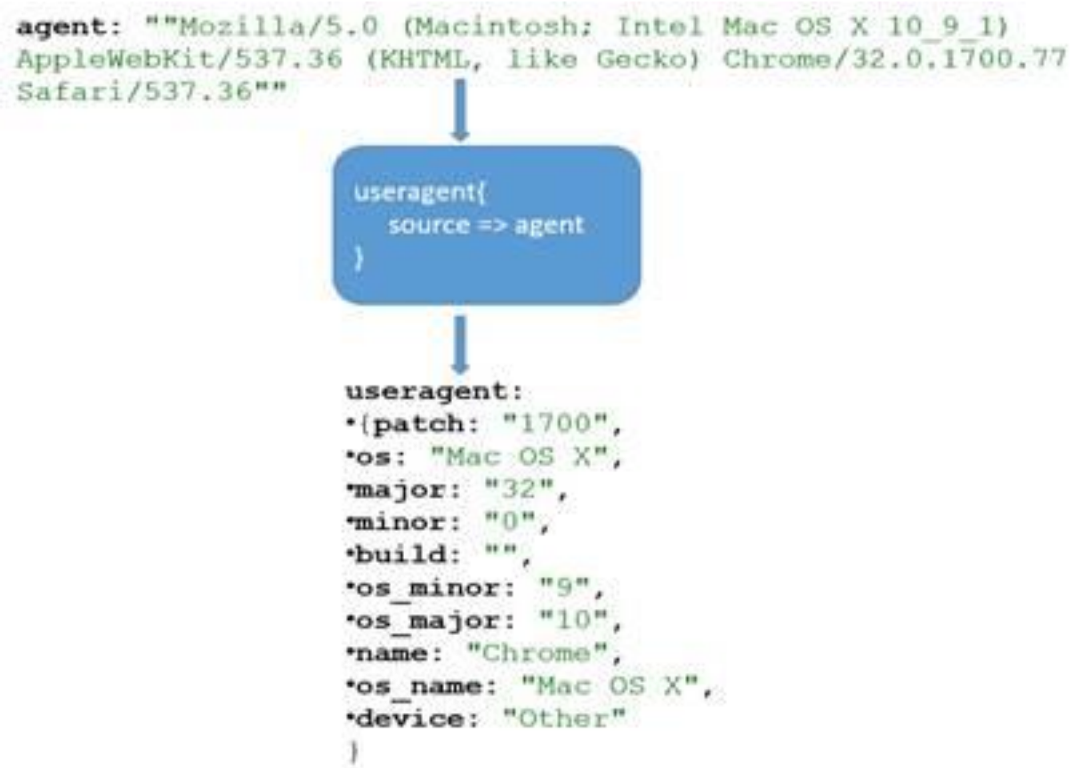


```
clientip => "83.149.9.216"
```

```
geoip{
    source => clientip
}
```

```
geoip:
•{timezone: "Europe/Moscow",
•ip: "83.149.9.216",
•latitude: 55.7485,
•continent_code: "EU",
•city_name: "Moscow",
•country_name: "Russia",
•country_code2: "RU",
•country_code3: "RU",
•region_name: "Moscow",
•location:
•{lon: 37.6184,
•lat: 55.7485
•},
•postal_code: "101194",
•region_code: "MOW",
•longitude: 37.6184
}
```

Useragent filter



```
agent: ""Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/32.0.1700.77
Safari/537.36""
```

```
useragent{
    source => agent
}
```

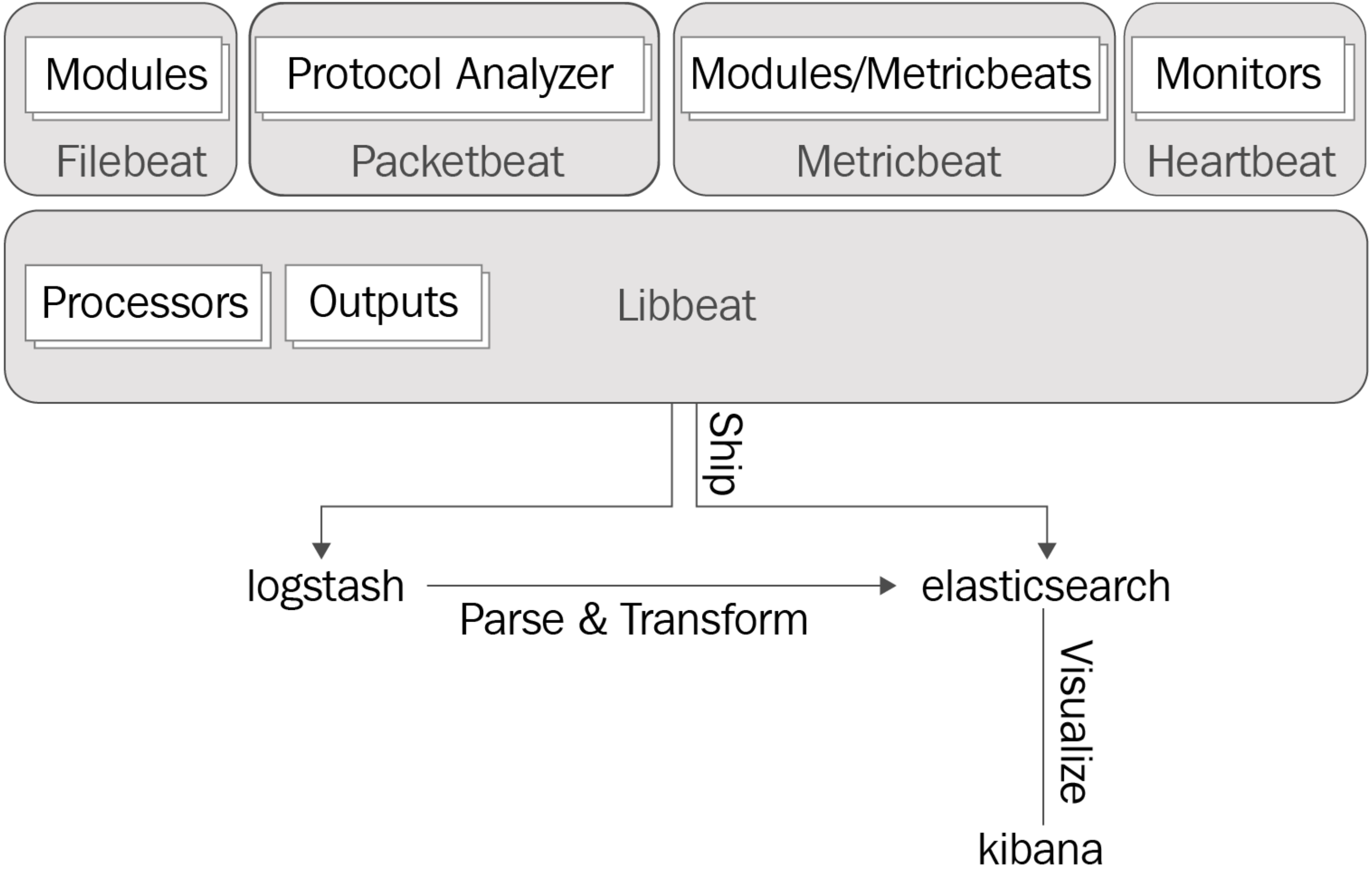
```
useragent:
•{patch: "1700",
•os: "Mac OS X",
•major: "32",
•minor: "0",
•build: "",
•os_minor: "9",
•os_major: "10",
•name: "Chrome",
•os_name: "Mac OS X",
•device: "Other"
}
```

Introducing Beats

- Beats are lightweight data shippers that are installed as agents on edge servers to ship operational data to Elasticsearch.

- Just like Elasticsearch, Logstash, Kibana, and Beats are open source products too.

- Depending on the use case, Beats can be configured to ship the data to Logstash to transform events prior to pushing the events to Elasticsearch.

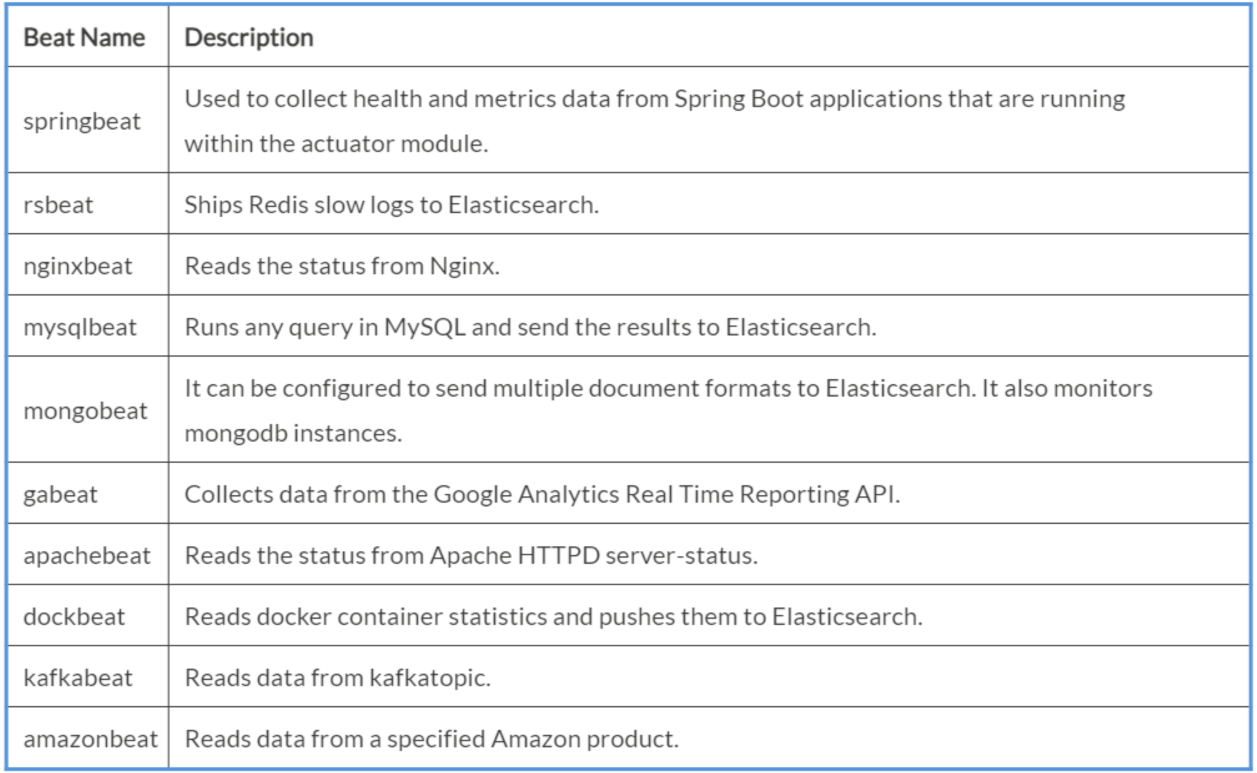

Modules
Filebeat
Protocol Analyzer
Packetbeat
Modules/Metricbeats
Metricbeat
Monitors
Heartbeat
Processors
Outputs
Libbeat
Ship
logstash
Parse & Transform
elasticsearch
Visualize
kibana

Beats by Elastic.co

Let's take a look at some used Beats commonly used by Elastic.co.

- Filebeat
- Metricbeat
- Packetbeat
- Heartbeat
- Winlogbeat
- Auditbeat
- Journalbeat
- Functionbeat

Community Beats



Beat Name	Description
springbeat	Used to collect health and metrics data from Spring Boot applications that are running within the actuator module.
rsbeat	Ships Redis slow logs to Elasticsearch.
nginxbeat	Reads the status from Nginx.
mysqlbeat	Runs any query in MySQL and send the results to Elasticsearch.
mongobeat	It can be configured to send multiple document formats to Elasticsearch. It also monitors mongodb instances.
gabeat	Collects data from the Google Analytics Real Time Reporting API.
apachebeat	Reads the status from Apache HTTPD server-status.
dockbeat	Reads docker container statistics and pushes them to Elasticsearch.
kafkabeat	Reads data from kafkatopic.
amazonbeat	Reads data from a specified Amazon product.

Logstash versus Beats

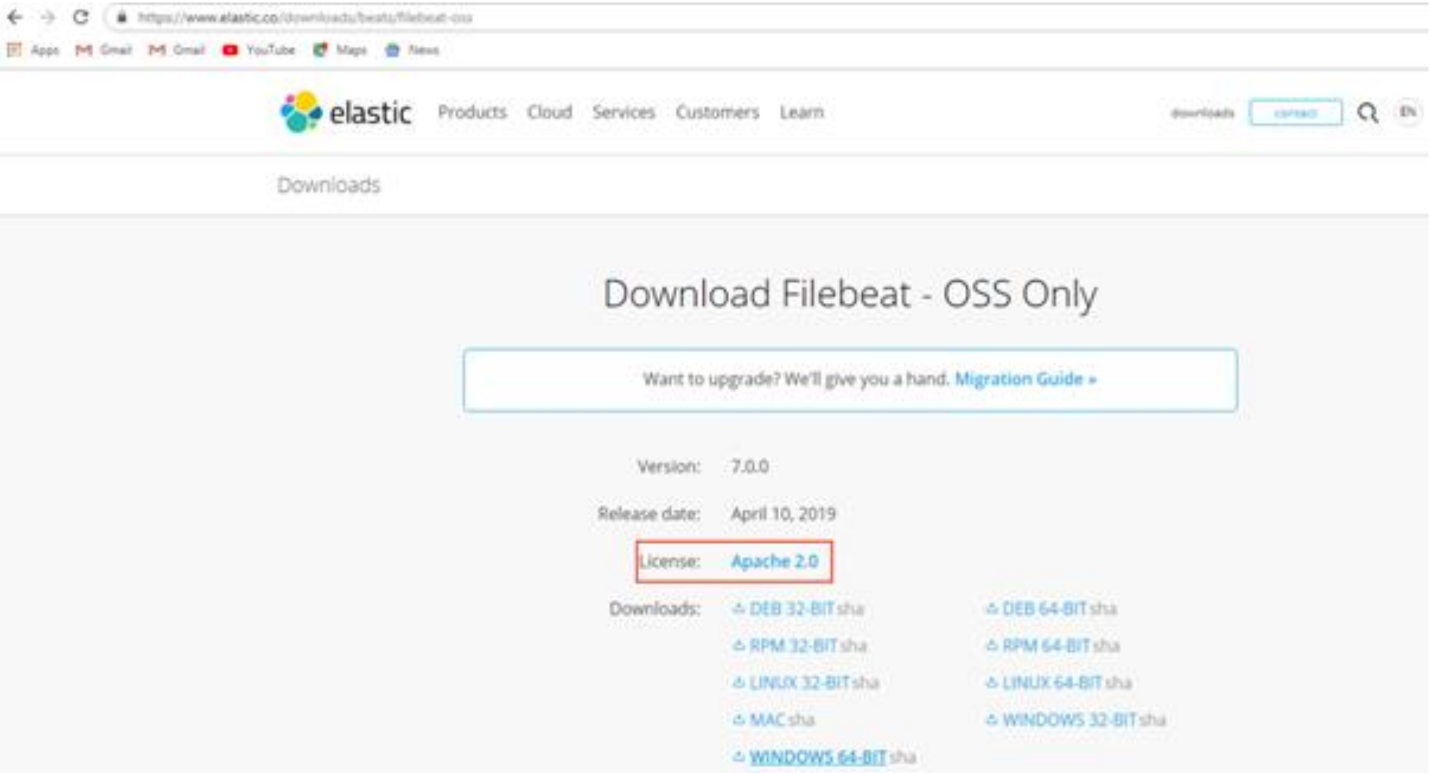
- After reading through the Logstash and Beats introduction, you might be confused as to whether Beats is a replacement for Logstash, the difference between them, or when to use one over the other.

- Beats are lightweight agents and consume fewer resources, and hence are installed on the edge servers where the operational data needs to be collected and shipped.

Filebeat

- Filebeat is an open source, lightweight log shipping agent that is installed as an agent to ship logs from local files.
- It monitors log directories, tails the files, and sends them to Elasticsearch, Logstash, Redis, or Kafka.
- It allows us to send logs from different systems to a centralized server.
- The logs can then be parsed or processed from here.

Downloading and installing Filebeat



Download Filebeat - OSS Only

Want to upgrade? We'll give you a hand. Migration Guide »

Version:	7.0.0
Release date:	April 10, 2019
License:	Apache 2.0

Downloads:

- DEB 32-BIT sha
- RPM 32-BIT sha
- LINUX 32-BIT sha
- MAC sha
- WINDOWS 64-BIT sha
- DEB 64-BIT sha
- RPM 64-BIT sha
- LINUX 64-BIT sha
- WINDOWS 32-BIT sha

Installing on Windows

- Unzip the downloaded file and navigate to the extracted location, as follows:

```
E:> cd E:\filebeat-7.0.0-windows-x86_64
```

Installing on Windows

- Run the following commands from the PowerShell prompt to install Filebeat as a Windows service:

```
PS >cd E:\filebeat-7.0.0-windows-x86_64
 PS E:\filebeat-7.0.0-windows-x86_64>.\install-service-filebeat.ps1
```

- In the event script execution is disabled on your system, you will have to set the execution policy for the current session:

```
PowerShell.exe -ExecutionPolicy UnRestricted -File .\install-service-filebeat.ps1
```

Installing on Linux

- Unzip the tar.gz package and navigate to the newly created folder, as follows:

```
$> tar -xzf filebeat-7.0.0-linux-x86_64.tar.gz
$> cd filebeat
```

- To install using dep or rpm, execute the appropriate commands in the Terminal:
- deb:

```
curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-7.0.0-amd64.deb
sudo dpkg -i filebeat-7.0.0-amd64.deb
```

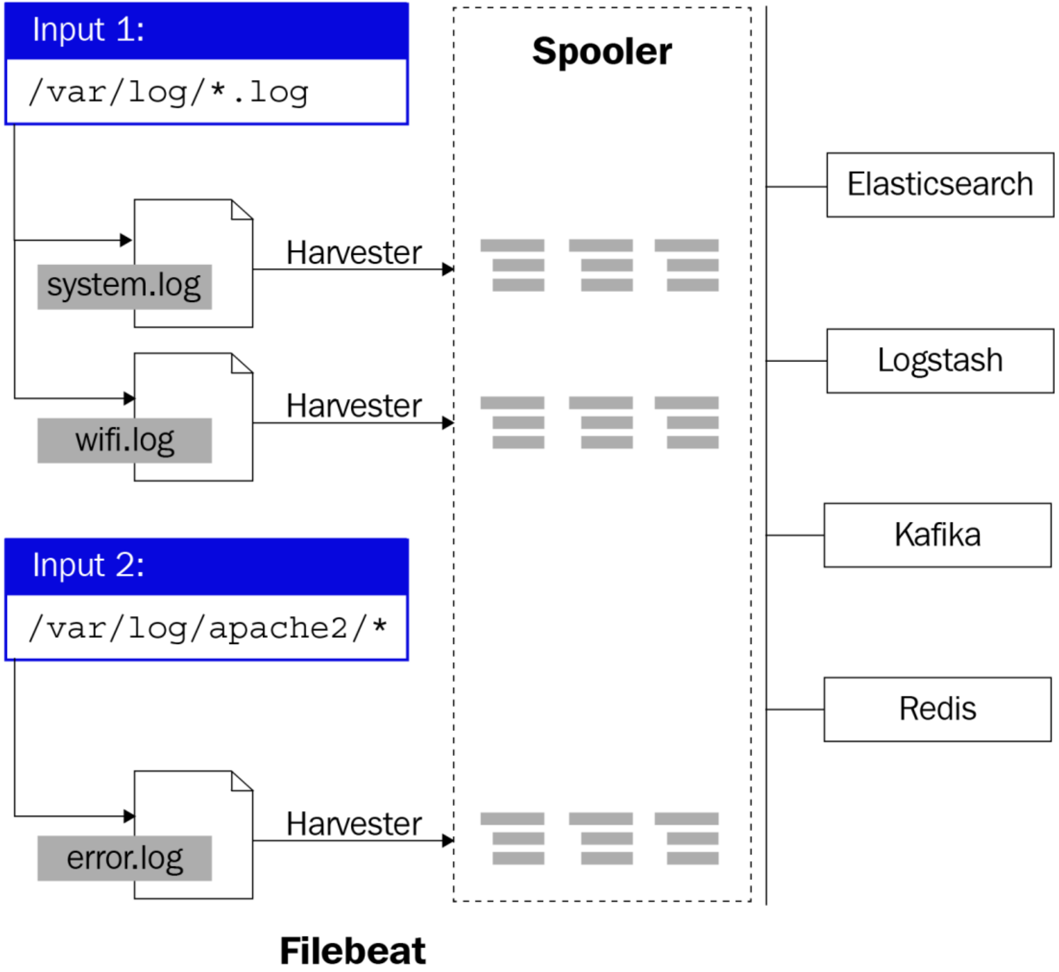
Installing on Linux

- rpm:

```
curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-7.0.0-x86_64.rpm
sudo rpm -vi filebeat-7.0.0-x86_64.rpm
```

Architecture

- Filebeat is made up of key components called inputs, harvesters, and spoolers.
- These components work in unison in order to tail files and allow you to send event data to the specified output.
- The input is responsible for identifying the list of files to read logs from.
- The input is configured with one or many file paths, from which it identifies the files to read logs from; it starts a harvester for each file.


Input 1:
/var/log/*.log
system.log
Harvester
wifi.log
Harvester
Input 2:
/var/log/apache2/*
error.log
Harvester
Spooler
Elasticsearch
Logstash
Kafika
Redis
Filebeat

Configuring Filebeat

The filebeat.yml file contains the following important sections:

- Filebeat inputs
- Filebeat modules
- Elasticsearch template settings
- Filebeat general/global options
- Kibana dashboard settings
- Output configuration
- Processors configuration
- Logging configuration

Configuring Filebeat

- let's see what a simple configuration would look like.
- As we can see in the following configuration, when Filebeat is started, it looks for files ending with the .log extension in the E:\fenago\logs\ path.
- It ships the log entries of each file to Elasticsearch, which is configured as the output, and is hosted at localhost:9200:

https://github.com/fenago/elasticsearch/blob/master/snippets/6_4.txt

Configuring Filebeat

- Place some log files in E:\fenago\logs\. To get Filebeat to ship the logs, execute the following command:

Windows:

```
E:\>filebeat-7.0.0-windows-x86_64>filebeat.exe
```

Linux:

```
[locationOfFilebeat]$ ./filebeat
```

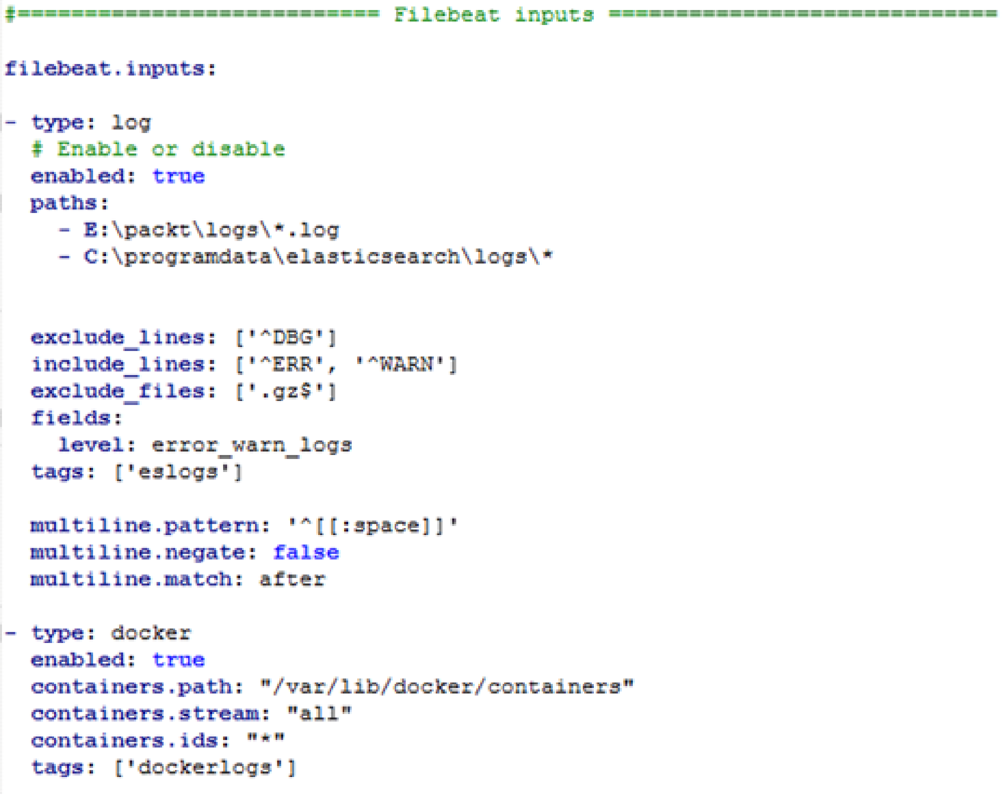
Configuring Filebeat

- To validate whether the logs were shipped to Elasticsearch, execute the following command:

https://github.com/fenago/elasticsearch/blob/master/snippets/6_5.txt

Filebeat inputs

- A sample configuration is as follows:



```
#============================= Filebeat inputs ==============================

filebeat.inputs:

- type: log
  # Enable or disable
  enabled: true
  paths:
    - E:\packt\logs\*.log
    - C:\programdata\elasticsearch\logs\*


  exclude_lines: ['^DBG']
  include_lines: ['^ERR', '^WARN']
  exclude_files: ['.gz$']
  fields:
    level: error_warn_logs
  tags: ['eslogs']

  multiline.pattern: '^[[:space]]'
  multiline.negate: false
  multiline.match: after

- type: docker
  enabled: true
  containers.path: "/var/lib/docker/containers"
  containers.stream: "all"
  containers.ids: "*"
  tags: ['dockerlogs']
```

Filebeat general/global options

- **registry_file:** It is used to specify the location of the registry file, which is used to maintain information about files, such as the last offset read and whether the read lines are acknowledged by the configured outputs or not.
- The default location of the registry is ${path.data}/registry:

```
filebeat.registry_file: /etc/filebeat/registry
```

Filebeat general/global options

- **shutdown_timeout:** This setting specifies how long Filebeat waits on shutdown for the publisher to finish.
- This ensures that if there is a sudden shutdown while filebeat is in the middle of sending events, it won't wait for the output to acknowledge all events before it shuts down.
- Hence, the filebeat waits for a certain time before it actually shuts down:

```
filebeat.shutdown_timeout: 10s
```

Filebeat general/global options

- **registry_flush:** This setting specifies the time interval when registry entries are to be flushed to the disk:

```
filebeat.registry_flush: 5s
```

- **name:** The name of the shipper that publishes the network data. By default, hostname is used for this field:

```
name: "dc1-host1"
```

Filebeat general/global options

- **tags:** The list of tags that will be included in the tags field of every event Filebeat ships.
- Tags make it easy to group servers by different logical properties and aids filtering of events in Kibana and Logstash:

```
tags: ["staging", "web-tier","dc1"]
```

- **max_procs:** The maximum number of CPUs that can be executed simultaneously.
- The default is the number of logical CPUs available in the system:

```
max_procs: 2
```

Output configuration

- **Elasticsearch:** It is used to send the events directly to Elasticsearch.

- A sample Elasticsearch output configuration is as follows:

```
output.elasticsearch:
  enabled: true
  hosts: ["localhost:9200"]
```

Output configuration

- If Elasticsearch is secure, then the credentials can be passed using the username and password settings:

```
output.elasticsearch:
  enabled: true
  hosts: ["localhost:9200"]
  username: "elasticuser"
  password: "password"
```

Output configuration

- To ship an event to the Elasticsearch ingest node pipeline so that it can be preprocessed before it is stored in Elasticsearch, the pipeline information can be provided using the pipleline setting:

```
output.elasticsearch:
  enabled: true
  hosts: ["localhost:9200"]
  pipeline: "apache_log_pipeline"
```

Output configuration

- **logstash:** This is used to send events to Logstash.
- A sample Logstash output configuration is as follows:

```
output.logstash:
  enabled: true
  hosts: ["localhost:5044"]
```

Output configuration

- To enable load balancing of events across the Logstash hosts, use the loadbalance flag, set to true:

```
output.logstash:
hosts: ["localhost:5045", "localhost:5046"]
  loadbalance: true
```

Output configuration

- **console:** This is used to send the events to stdout. The events are written in JSON format. It is useful during debugging or testing.
- A sample console configuration is as follows:

```
output.console:
  enabled: true
pretty: true
```

Logging

- A sample configuration is as follows:

```
logging.level: debug
logging.to_files: true
logging.files:
 path: C:\logs\filebeat
 name: metricbeat.log
 keepfiles: 10
```

Filebeat modules

A module is made up of one or more filesets. A fileset is made up of the following:

- Filebeat input configurations that contain the default paths needed to look out for logs. It also provides configuration for combining multiline events when needed.
- An Elasticsearch Ingest pipeline definition to parse and enrich logs.
- Elasticsearch templates, which define the field definitions so that appropriate mappings are set to the fields of the events.
- Sample Kibana dashboards, which can be used for visualizing logs.

Filebeat modules

- Since each module comes with the default configuration, make the appropriate changes in the module configuration file.

- The basic configuration for the redis module is as follows:

https://github.com/fenago/elasticsearch/blob/master/snippets/6_6.txt

Filebeat modules

- To enable modules, execute the modules enable command, passing one or more module names:

Windows:

```
E:\filebeat-7.0.0-windows-x86_64>filebeat.exe modules enable redis mysql
```

Linux:

```
[locationOfFileBeat]$./filebeat modules enable redis mysql
```

Filebeat modules

- To disable modules, execute the modules disable command, passing one or more module names to it. For example:

Windows:

```
E:\filebeat-7.0.0-windows-x86_64>filebeat.exe modules disable redis mysql
```

Linux:

```
[locationOfFileBeat]$./filebeat modules disable redis mysql
```

Filebeat modules

- Once the module is enabled, to load the recommended index template for writing to Elasticsearch, and to deploy sample dashboards for visualizing data in Kibana, execute thesetup command, as follows:

Windows:

```
E:\filebeat-7.0.0-windows-x86_64>filebeat.exe -e setup
```

Linux:

```
[locationOfFileBeat]$./filebeat -e setup
```

Filebeat modules

- Rather than enabling the modules by passing them as command-line parameters, you can enable the modules in the filebeat.yml configuration file itself, and start Filebeat as usual:

```
filebeat.modules:
-module: nginx
-module: mysql
```

Filebeat modules

- For the configuration file, use the following code:

```
filebeat.modules:
-module: nginx
  access:
    var.paths: ["C:\ngnix\access.log*"]
```

Filebeat modules

- For the command line, use the following code:

Windows:

```
E:\filebeat-7.0.0-windows-x86_64>filebeat.exe-e-modules=nginx-M"nginx.access.var.paths=[C:\ngnix\access.log*]"
```

Linux:

```
[locationOfFileBeat]$./filebeat-e-modules=nginx-M"nginx.access.var.paths=[\var\ngnix\access.log*]"
```

Summary

- In this lesson, we covered the powerful filter section of Logstash, which can be used for parsing and enriching log events.
- We have also covered some commonly used filter plugins.
- Then, we covered the Beats framework and took an overview of various Beats, including Filebeat, Heartbeat, Packetbeat, and so on, covering Filebeat in detail.

Complete Lab 6