RX-M
Software Consulting

Introduction to Elasticsearch

A comprehensive introduction to the

Elasticsearch. A distributed, open source search and analytics engine

Overview

Day One

1. Overview
2. Architecture and Components
3. Data Importation
4. Search
5. Analyzers

Day Two

5. Mappings
6. More Search
7. Suggesters
8. Aggregations
9. Bringing it All Together

Prerequisites:

- Basic proficiency at the command line
- Basic proficiency with a flavor of linux
- A desire for data

Equivalent experience:

- Interest in Search or Elasticsearch specificlly

Administrative Info

- Course: Introduction to Elasticsearch
- Length: 2 Days
- Format: Lecture/Labs/Discussion
- Schedule: 9:30AM – 5:30PM
  15 minute break, AM & PM
  1 hour lunch @ noon
  Lab time at the end of each AM and PM session
- Location: Fire exits, Restrooms, Security, other matters
- Attendees: Name/Role/Experience/Goals for the Course

Administrative Info

- Location: Fire exits, Restrooms, Security, other matters
- Attendees: This intensive two day hands on course is designed to help working technology professionals master the essential aspects and operations of Elasticsearch. The course covers all of the key concepts and administrative tasks necessary to deploy and use a production Elasticsearch system. Attendees will learn how to search, aggregate and analyze large volumes of real-time data using Elasticsearch's highly available schema-less platform. The distributed horizontally scaled architecture of Elasticsearch is covered along with best practice deployment and operations patterns. Students will gain familiarity with the Elasticsearch query DSL and various APIs including Document, Search and Indices. Attendees will leave with a clear understanding of Elasticsearch and how to use it to extract high value data insights over large scale streaming datasets in real-time.

Lecture and Lab

- Our Goals in this class are two fold:
  1. Familiarize you with the general concepts and ecosystem surrounding Elasticsearch
  2. Give you practical experience and/or working with the Elasticsearch (JSON, JSON, JSON, JSON)

Day 1 Agenda

- Elasticsearch Overview
- Elasticsearch Architecture and Clustering
- Data Importation
- Search
- Analyzers

Who am I?

- Michael Forrester
- Itamazonian@gmail.com
- https://www.linkedin.com/in/performingpro
- System/DevOps/Operations/SR Engineer, Architect, Manager of Geeks, Director of Operations/Infrastructure/DevOps.
- Startups, medium, and large companies.

Who are you?

- What do you do?
- What is your interest in Elasticsearch?
- What is your experience with Elasticsearch?
- What is important about ES to you?

Module 1: Overview of Elasticsearch

Objectives

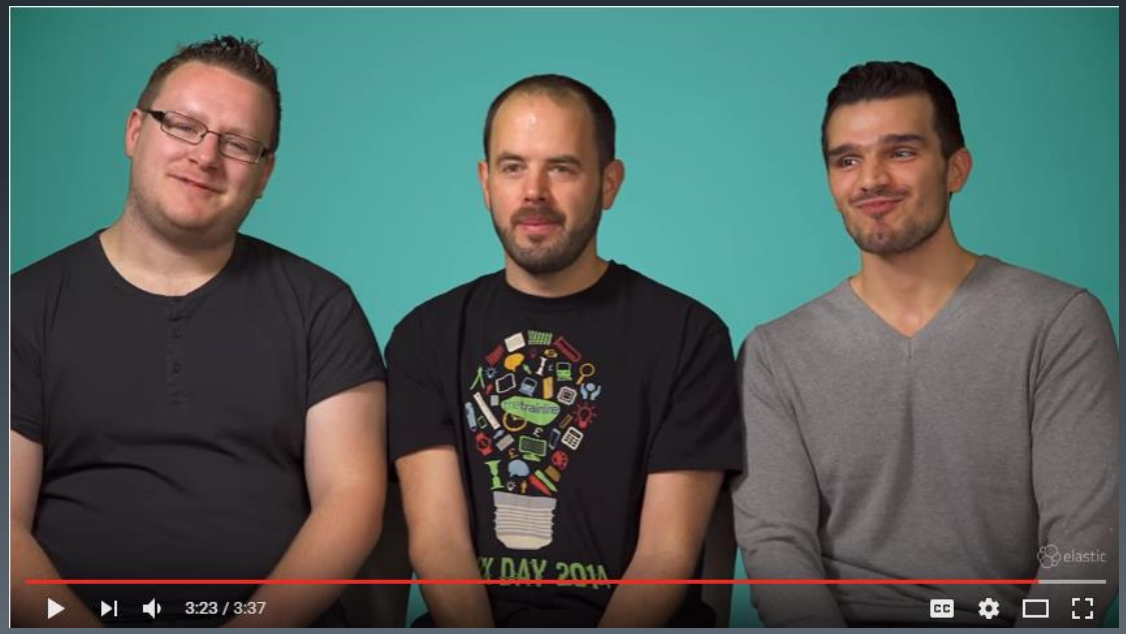
- Intention and Purpose of Elasticsearch
- A Brief History of Elasticsearch
- Installing and Starting Elasticsearch

What is Elasticsearch?

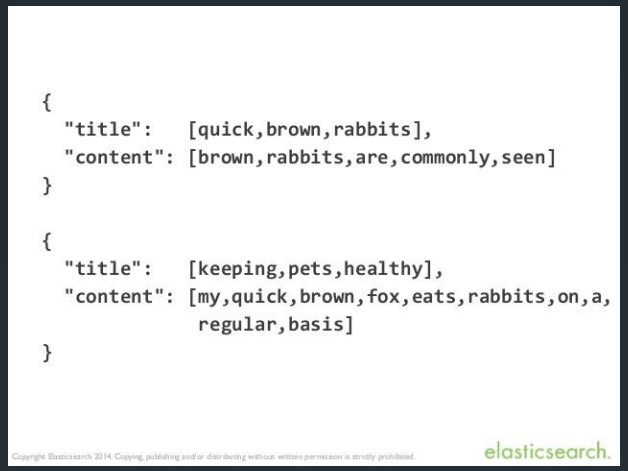
Elasticsearch is an open-source, broadly-distributable, readily-scalable, enterprise-grade search engine. Accessible through an extensive and elaborate API, Elasticsearch can power extremely fast searches that support your data discovery applications.

Elastic{ON} 2015 conference attendees were asked

"What is Elasticsearch?“



What about Lucene?



- **Apache Lucene** is a free and open-source information retrieval software library,
- It is supported by the Apache Software Foundation and is released under the Apache Software License.
- Doug Cutting originally wrote Lucene in Java in 1999.[2]
- It became its own top-level Apache project in February 2005.
- Lucene formerly included a number of sub-projects, such as Lucene.NET, Mahout, Tika and Nutch. Lucene.NET, Mahout, Nutch, and Tika are now independent top-level projects.
- In March 2010, the Apache Solr search server joined as a Lucene sub project, merging the developer communities.
- Version 4.0 was released on October 12, 2012.[3]
- The latest version of Lucene is 6.1.0 which was released on June 17, 2016.[3]

A Short Timeline for Elasticsearch

- 2004 - Shay Banon created Compass in 2004.[2] While thinking about the third version of Compass he realized that it would be necessary to rewrite big parts of Compass to "create a scalable search solution".[2] So he created "a solution built from the ground up to be distributed" and used a common interface, JSON over HTTP, suitable for programming languages other than Java as well.[2]
- 2010 - Shay Banon released the first version of Elasticsearch in February 2010.[3]
- 2012 - Elasticsearch BV was founded in 2012 to provide commercial services and products around Elasticsearch and related software.[4]
- 2014 - In June 2014, the company announced raising $70 million in a Series C funding round, just 18 months after forming the company. This round brings total funding to $104M.[5]
- 2015 - In October 2015, Elastic.co released a major release, 2.0, of Elasticsearch which incorporated major changes.

elastic
User Group

A History of Elasticsearch Releases

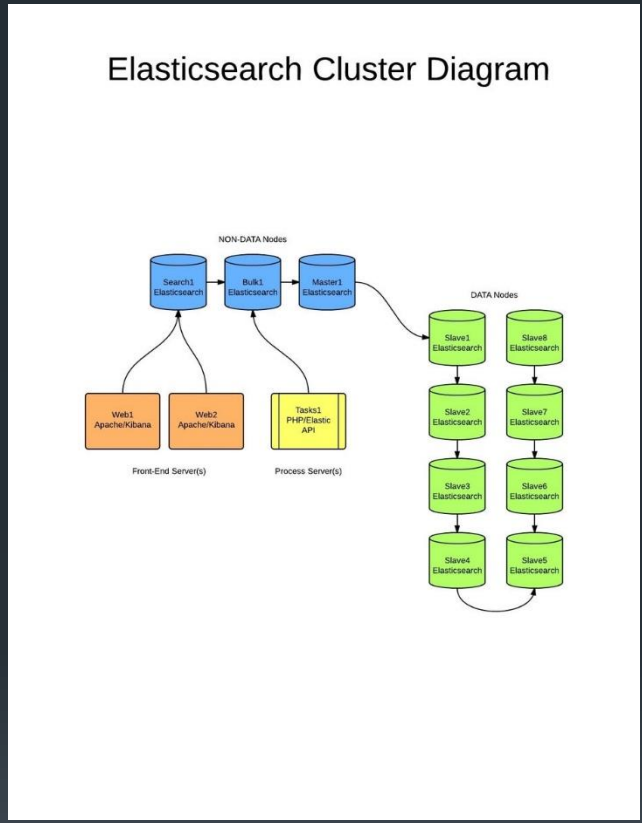
Version	Original release date	Latest version	Release date	Maintenance Status[6]
0.4	2010-02-08	0.4.0	2010-02-08	No longer supported
0.5	2010-03-05[7]	0.5.1	2010-03-09	No longer supported
0.6	2010-04-09[8]	0.6.0	2010-04-09	No longer supported
0.7	2010-05-14[9]	0.7.1	2010-05-17[10]	No longer supported
0.8	2010-05-27[11]	0.8.0	2010-05-27	No longer supported
0.9	2010-07-26[12]	0.9.0	2010-07-26	No longer supported
0.10	2010-08-27[13]	0.10.0	2010-08-27	No longer supported
0.11	2010-09-29[14]	0.11.0	2010-09-29	No longer supported
0.12	2010-10-18[15]	0.12.1	2010-10-27	No longer supported
0.13	2010-11-18[16]	0.13.1	2010-12-03	No longer supported
0.14	2010-12-27[17]	0.14.4	2011-01-31	No longer supported
0.15	2011-02-18[18]	0.15.2	2011-03-07	No longer supported
0.16	2011-04-23[19]	0.16.5	2011-07-26	No longer supported
0.17	2011-07-19[20]	0.17.10	2011-11-16	No longer supported
0.18	2011-10-26[21]	0.18.7	2012-01-10[22]	No longer supported
0.19	2012-03-01[23]	0.19.12	2012-12-04[24]	No longer supported
0.20	2012-12-07[25]	0.20.6	2013-03-25[26]	No longer supported
0.90	2013-04-29[27]	0.90.13	2014-03-25[28]	No longer supported
1.0	2014-02-12[29]	1.0.3	2014-04-16[30]	No longer supported
1.1	2014-03-25[28]	1.1.2	2014-05-22[31]	No longer supported
1.2	2014-05-22[31]	1.2.4	2014-08-13[32]	No longer supported
1.3	2014-07-23[33]	1.3.9	2015-02-19[34]	No longer supported
1.4	2014-11-05[35]	1.4.5	2015-04-27[36]	Still supported
1.5	2015-03-23[37]	1.5.2	2015-04-27[36]	Still supported
1.6	2015-06-09[38]	1.6.2	2015-07-29[39]	Still supported
1.7	2015-07-16[40]	1.7.5	2016-02-02[41]	Still supported
2.0	2015-10-28[42]	2.0.2	2015-12-17[43]	Still supported
2.1	2015-11-24[44]	2.1.2	2016-02-02[41]	Still supported
2.2	2016-02-02[41]	2.2.2	2016-03-30[45]	Still supported
2.3	2016-03-30[45]	2.3.4	2016-05-18[46]	Latest
5.0	2016-04-05[47]	5.0.0 Alpha 4	2016-06-30[48]	Latest preview version

Legend: Old version | Older version, still supported | **Latest version** | Latest preview version

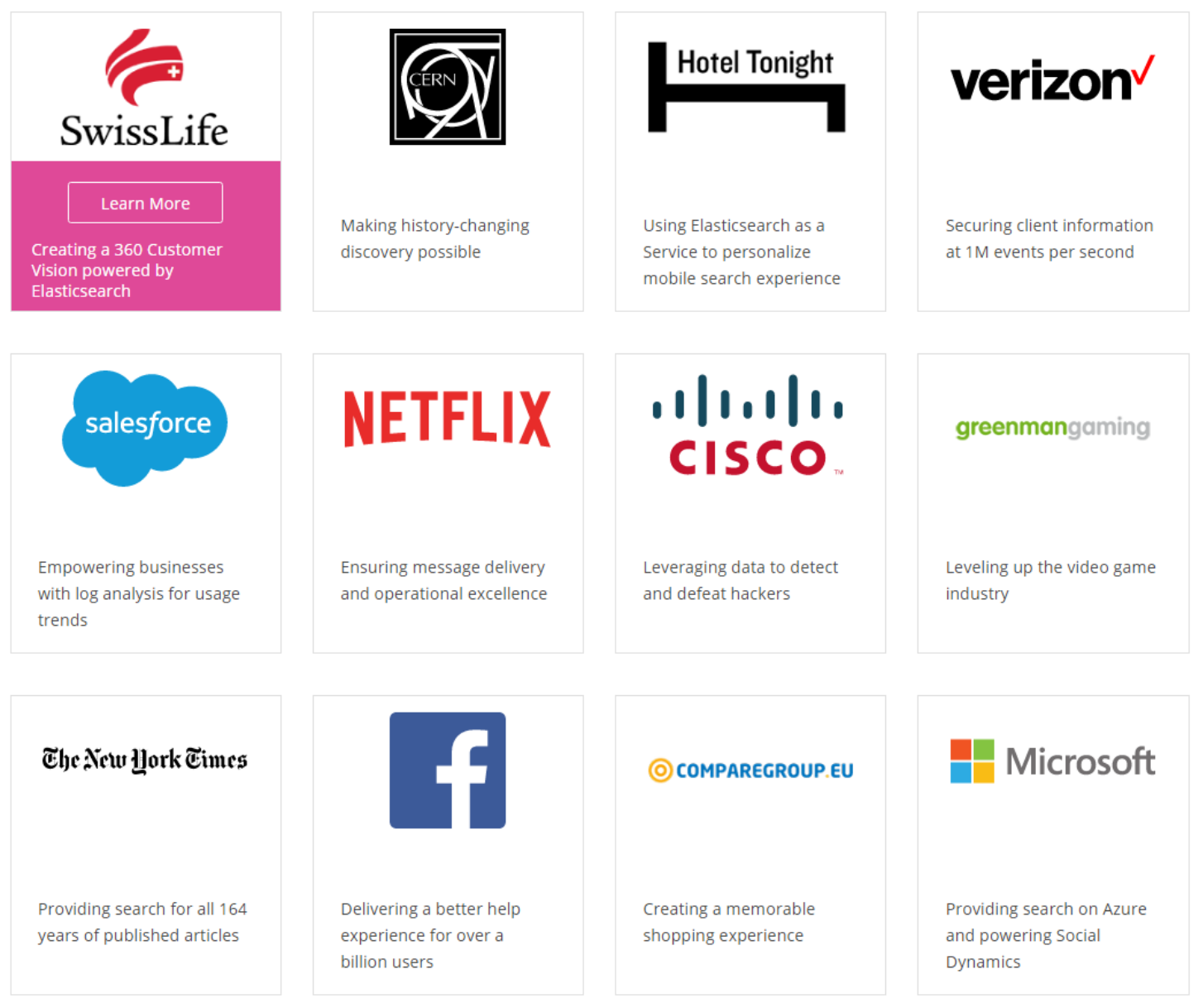
- 1st release in Feb 2010
- 1.0 release in April 2014
- 2.0 release in December 2015

Key Features of Elasticsearch

- Elasticsearch is a distributed RESTful search engine built for the cloud. Features include:
- Distributed and Highly Available Search Engine.
  - Each index is fully sharded with a configurable number of shards.
  - Each shard can have one or more replicas.
  - Read / Search operations performed on any of the replica shards.
- Multi Tenant with Multi Types.
  - Support for more than one index.
  - Support for more than one type per index.
  - Index level configuration (number of shards, index storage, …).
- Various set of APIs
  - HTTP RESTful API
  - Native Java API.
  - All APIs perform automatic node operation rerouting.
- Document oriented
  - No need for upfront schema definition.
  - Schema can be defined per type for customization of the indexing process.
- Reliable, Asynchronous Write Behind for long term persistency.
- (Near) Real Time Search.
- Built on top of Lucene
  - Each shard is a fully functional Lucene index
  - All the power of Lucene easily exposed through simple configuration / plugins.
- Per operation consistency
  - Single document level operations are atomic, consistent, isolated and durable.
- Open Source under the Apache License, version 2 (“ALv2”)



Who uses Elasticsearch?



Generic Installation and Setup

- Installation and Setup are part of the the 1st Lab that will be covered in class.
- Generically the installation steps are the same:
  - Designate your OS and server of choice
  - Install Version 7 or 8 of the Java JRE. OpenJDK is acceptable for 7 though Oracle is preferred for Production installations.
  - Obtain and install Elasticsearch 2.3 via source/compilation, brew, or apt-get/yum
  - Configure via elasticsearch.yml
  - Load Data
  - Execute Queries
- Setup any configuration changes in the elasticsearch.yml or logging.yml
- Setting automated startup or start manually with bin/elasticsearch (*nix) or elasticsearch.bat (windows)
- Explore your cluster
- Load Data
- Search for meaningful data

```
#!/usr/bin/env bash

IP="127.0.0.1"
HOST=`es2`
sed -i "/$IP/ s/.*/$IP\tlocalhost\t$HOST/g" /etc/hosts

sudo apt-get update
sudo apt-get upgrade

# install curl
sudo apt-get -y install curl

# install openjdk-7
sudo apt-get purge openjdk*
sudo apt-get -y install openjdk-8-jdk

# install ES
wget -qO - https://packages.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
echo "deb http://packages.elastic.co/elasticsearch/2.x/debian stable main" | sudo tee -a /etc/apt/sources.list.d/e
lasticsearch-2.x.list
sudo apt-get update && sudo apt-get install elasticsearch
sudo update-rc.d elasticsearch defaults 95 10

# either of the next two lines is needed to be able to access "localhost:9200" from the host os
sudo echo "network.bind_host: 0" >> /etc/elasticsearch/elasticsearch.yml
sudo echo "network.host: 0.0.0.0" >> /etc/elasticsearch/elasticsearch.yml

# enable cors (to be able to use Sense)
sudo echo "http.cors.enabled: true" >> /etc/elasticsearch/elasticsearch.yml
sudo echo "http.cors.allow-origin: /https?:\/\/localhost(:[0-9]+)?/" >> /etc/elasticsearch/elasticsearch.yml

# enable dynamic scripting
sudo echo "script.inline: on" >> /etc/elasticsearch/elasticsearch.yml
sudo echo "script.indexed: on" >> /etc/elasticsearch/elasticsearch.yml

sudo /etc/init.d/elasticsearch start

sudo /usr/share/elasticsearch/bin/plugin install -b --verbose license
sudo /usr/share/elasticsearch/bin/plugin install -b --verbose marvel-agent

wget -qO - https://packages.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
echo "deb http://packages.elastic.co/kibana/4.5/debian stable main" | sudo tee -a /etc/apt/sources.list
```

```
sudo apt-get update && sudo apt-get install kibana
sudo echo "elasticsearch.url: "http://localhost:9200"" >> /opt/kibana/config/kibana.yml
sudo update-rc.d kibana defaults 95 10
sudo /bin/systemctl daemon-reload
sudo /bin/systemctl enable kibana.service
sudo /etc/init.d/kibana restart
sudo /opt/kibana/bin/kibana plugin --install elastic/sense
sudo /opt/kibana/bin/kibana plugin --install elasticsearch/marvel/latest
sudo chown kibana:root /opt/kibana/optimize/.babelcache.json
sudo /etc/init.d/elasticsearch restart
sudo /etc/init.d/kibana restart
```

Supported Operating Systems

	CentOS/RHEL 6.x/7.x	Oracle Enterprise Linux 6/7 with RHEL Kernel only	Ubuntu 12.04/14.04	SLES 11 SP4**/12	OpenSuSE 13.2	Windows Server 2012/R2	Windows Server 2008	Debian 7	Solaris/SmartOS	Amazon Linux*
ES 1.4.x	✔	✔	✔	✔	✔	✔	✖	✔	✖	✖
ES 1.5.x	✔	✔	✔	✔	✔	✔	✖	✔	✖	✖
ES 1.6.x	✔	✔	✔	✔	✔	✔	✖	✔	✖	✖
ES 1.7.x	✔	✔	✔	✔	✔	✔	✖	✔	✖	✖
ES 2.0.x	✔	✔	✔	✔	✔	✔	✖	✔	✖	✔
ES 2.1.x	✔	✔	✔	✔	✔	✔	✖	✔	✖	✔
ES 2.2.x	✔	✔	✔	✔	✔	✔	✖	✔	✖	✔
ES 2.3.x	✔	✔	✔	✔	✔	✔	✖	✔	✖	✔

- Note that Windows 2012 only is supported and Amazon Linux requires 2.0.

Supported Java Virtual Machines

Product and JVM

	Oracle JRE 1.6	Oracle JVM 1.7u55+	Oracle JVM 1.8u20+	IcedTea OpenJDK 1.6	IcedTea OpenJDK 1.7.0.55+	IBM JVM 6	IBM JVM 7	Azul Zing
ES 1.4.x	✖	✔	✔	✖	✔	✖	✖	✖
ES 1.5.x	✖	✔	✔	✖	✔	✖	✖	✖
ES 1.6.x	✖	✔	✔	✖	✔	✖	✖	✖
ES 1.7.x	✖	✔	✔	✖	✔	✖	✖	✖
ES 2.0.x	✖	✔	✔	✖	✔	✖	✖	✖
ES 2.1.x	✖	✔	✔	✖	✔	✖	✖	✖
ES 2.2.x	✖	✔	✔	✖	✔	✖	✖	✖
ES 2.3.x	✖	✔	✔	✖	✔	✖	✖	✖

Any JVM can be used really, but use these if you are running a production, qa environment.

Sources to Obtain Elasticsearch

- Elastic.co source download (All that run a JVM)
- Elastic.co Yum repository (RHEL and clones)
- Elastic.co Apt repository (Ubuntu)
- Brew Install (OS X)
- Official Docker Repository

Once we have Java set up, we can then download and run Elasticsearch. The binaries are available from `www.elastic.co/downloads` along with all the releases that have been made in the past. For each release, you have a choice among a `zip` or `tar` archive, or a `DEB` or `RPM` package. For simplicity, let's use the tar file.

Let's download the Elasticsearch 2.3.4 tar as follows (Windows users should download the zip package):

```
curl -L -O https://download.elastic.co/elasticsearch/release/org/elasticsearch/dist
```

Then extract it as follows (Windows users should unzip the zip package):

```
tar -xvf elasticsearch-2.3.4.tar.gz
```

It will then create a bunch of files and folders in your current directory. We then go into the bin directory as follows:

```
cd elasticsearch-2.3.4/bin
```

And now we are ready to start our node and single cluster (Windows users should run the elasticsearch.bat file):

```
./elasticsearch
```

How to use this image

You can run the default `elasticsearch` command simply:

```
$ docker run -d elasticsearch
```

You can also pass in additional flags to `elasticsearch`:

```
$ docker run -d elasticsearch elasticsearch -Des.node.name="TestNode"
```

This image comes with a default set of configuration files for `elasticsearch`, but if you want to provide your own set of configuration files, you can do so via a volume mounted at `/usr/share/elasticsearch/config`:

```
$ docker run -d -v "$PWD/config":/usr/share/elasticsearch/config elasticsearch
```

This image is configured with a volume at `/usr/share/elasticsearch/data` to hold the persisted index data. Use that path if you would like to keep the data in a mounted volume:

```
$ docker run -d -v "$PWD/esdata":/usr/share/elasticsearch/data elasticsearch
```

This image includes `EXPOSE 9200 9300` (default `http.port`), so standard container linking will make it automatically available to the linked containers.

Where are things installed – deb and rpm

Default Paths

Below are the default paths that elasticsearch will use, if not explicitly changed.

deb and rpm

Type	Description	Location Debian/Ubuntu	Location RHEL/CentOS
home	Home of elasticsearch installation.	`/usr/share/elasticsearch`	`/usr/share/elasticsearch`
bin	Binary scripts including `elasticsearch` to start a node.	`/usr/share/elasticsearch/bin`	`/usr/share/elasticsearch/bin`
conf	Configuration files `elasticsearch.yml` and `logging.yml`.	`/etc/elasticsearch`	`/etc/elasticsearch`
conf	Environment variables including heap size, file descriptors.	`/etc/default/elasticsearch`	`/etc/sysconfig/elasticsearch`

Where are things installed – deb and rpm

logs	Log files location	`/var/log/elasticsearch`	`/var/log/elasticsearch`
plugins	Plugin files location. Each plugin will be contained in a subdirectory.	`/usr/share/elasticsearch/plugins`	`/usr/share/elasticsearch/plugins`
repo	Shared file system repository locations.	Not configured	Not configured
script	Location of script files.	`/etc/elasticsearch/scripts`	`/etc/elasticsearch/scripts`

Where are things installed - zip

zip and tar.gz

edit

Type	Description	Location
home	Home of elasticsearch installation	`{extract.path}`
bin	Binary scripts including `elasticsearch` to start a node	`{extract.path}/bin`
conf	Configuration files `elasticsearch.yml` and `logging.yml`	`{extract.path}/config`
data	The location of the data files of each index / shard allocated on the node	`{extract.path}/data`
logs	Log files location	`{extract.path}/logs`
plugins	Plugin files location. Each plugin will be contained in a subdirectory	`{extract.path}/plugins`
repo	Shared file system repository locations.	Not configured
script	Location of script files.	`{extract.path}/config/scripts`

Configuration Files


```
##################### ElasticSearch Configuration Example #####################

# This file contains an overview of various configuration settings,
# targeted at operations staff. Application developers should
# consult the guide at <http://elasticsearch.org/guide>.
#
# The installation procedure is covered at
# <http://elasticsearch.org/guide/en/elasticsearch/reference/current/setup.html>.
#
# ElasticSearch comes with reasonable defaults for most settings,
# so you can try it out without bothering with configuration.
#
# Most of the time, these defaults are just fine for running a production
# cluster. If you're fine-tuning your cluster, or wondering about the
# effect of certain configuration option, please _do ask_ on the
# mailing list or IRC channel [http://elasticsearch.org/community].

# Any element in the configuration can be replaced with environment variables
# by placing them in ${...} notation. For example:
#
# node.rack: ${RACK_ENV_VAR}

# For information on supported formats and syntax for the config file, see
# <http://elasticsearch.org/guide/en/elasticsearch/reference/current/setup-configuration.html>


################################### Cluster ###################################

# Cluster name identifies your cluster for auto-discovery. If you're running
# multiple clusters on the same network, make sure you're using unique names.
#
cluster.name: od-fts1


#################################### Node #####################################

# Node names are generated dynamically on startup, so you're relieved
# from configuring them manually. You can tie this node to a specific name:
#
node.name: "od-fts1a"

# Every node can be configured to allow or deny being eligible as the master,
# and to allow or deny to store the data.
#
# Allow this node to be eligible as a master node (enabled by default):
#
node.master: true
```

- Found in the “config” directory of your installation (i.e. /etc/elasticsearch, config, etc)
- Elasticsearch.yml
  - Can define node name, cluster name, master nodes, etc
- Logging.yml
  - Logs with Log4j (java) and controls logging level, detail, etc.
- Values declared inside these files are permanent versus transient.
- Once configuration is set; start up your cluster for management even if it is one node

Exploring your Cluster

- The REST API
- Now that we have our node (and cluster) up and running, the next step is to understand how to communicate with it. Fortunately, Elasticsearch provides a very comprehensive and powerful REST API that you can use to interact with your cluster. Among the few things that can be done with the API are as follows:
- Check your cluster, node, and index health, status, and statistics
- Administer your cluster, node, and index data and metadata
- Perform CRUD (Create, Read, Update, and Delete) and search operations against your indexes
- Execute advanced search operations such as paging, sorting, filtering, scripting, aggregations, and many others

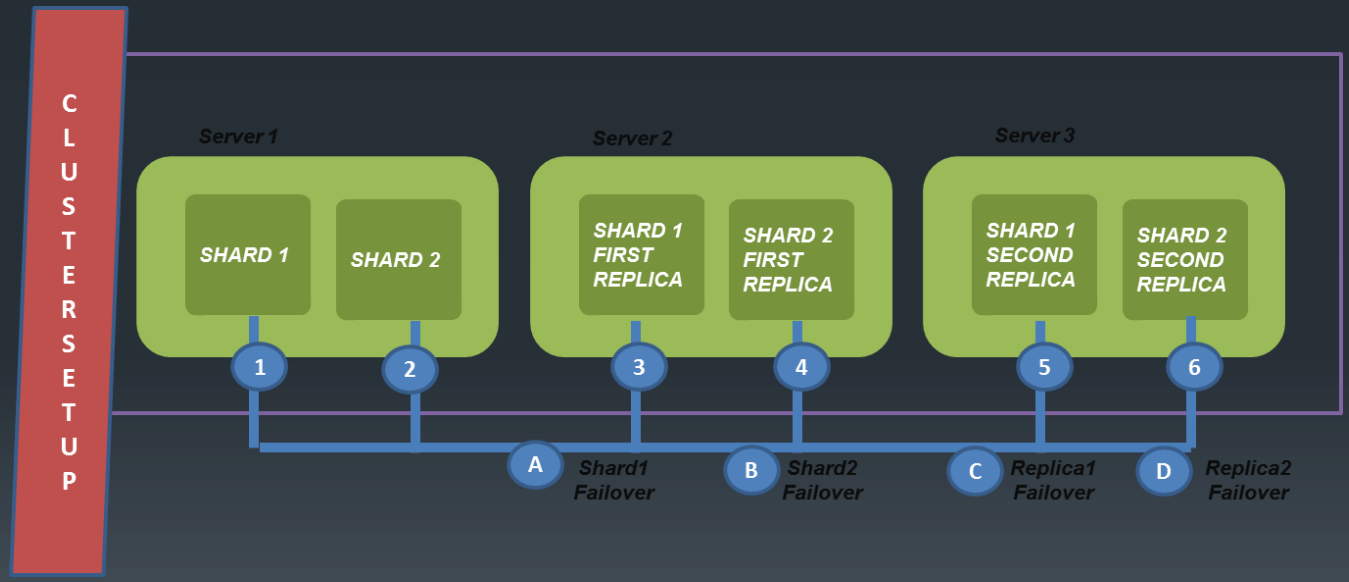
```
curl 'localhost:9200/_cat/health?v'
```

```
epoch      timestamp cluster       status node.total node.data shards pri relo init
1394735289 14:28:09  elasticsearch green           1         1      0   0    0    0
```

Summary

- Elasticsearch is an open-source, broadly-distributable, readily-scalable, enterprise-grade search engine
- Managed through an HTTP RESTful API
- Schema less with a data driven schema
- Written in Java and backed by Apache Lucene
- Used by many major tech players in the market
- Near Real Time Search
- Runs supported on Windows and Linux in production, but on anything that runs in a JVM.

Module 2: Architecture and Components

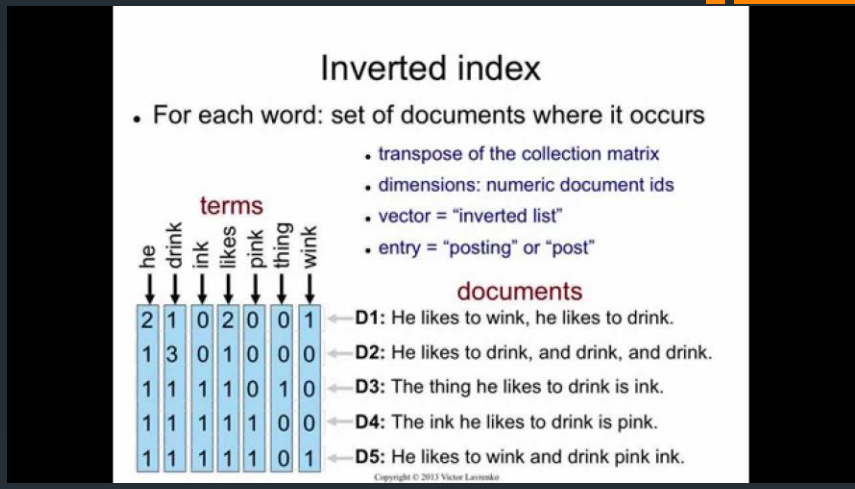


Objectives

- Key Concepts of Elasticsearch
- Key Components of Elasticsearch Architecture
- Clustering
- Indexes

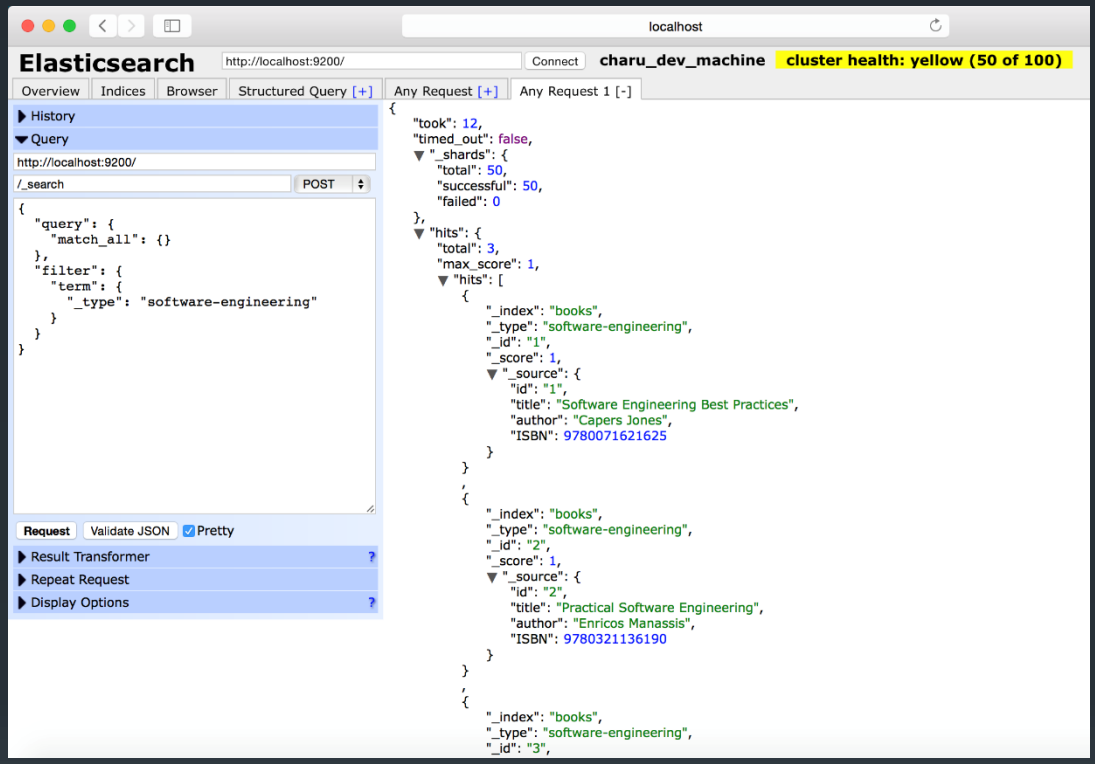
Index and Indices

- Index - An index is the logical place where Elasticsearch stores the data. Each index can be spread onto multiple Elasticsearch nodes and is divided into one or more smaller pieces called shards that are physically placed on the hard drives. If you are coming from the relational database world, you can think of an index like a table. However, the index structure is prepared for fast and efficient full text searching and, in particular, does not store original values. That structure is called an inverted index (https://en.wikipedia.org/wiki/Inverted_index).
- If you know MongoDB, you can think of the Elasticsearch index as a collection in MongoDB. If you are familiar with CouchDB, you can think about an index as you would about the CouchDB database. Elasticsearch can hold many indices located on one machine or spread them over multiple servers. Every index is built of one or more shards, and each shard can have many replicas.



Documents

- **DOCUMENT -** The main entity stored in Elasticsearch is a document. A document can have multiple fields, each having its own type and treated differently. Using the analogy to relational databases, a document is a row of data in a database table. All the documents with a field called title need to have the same data type for it, for example, string.
- Documents consist of fields, and each field may occur several times in a single document (such a field is called **multivalued**). Each field has a type (text, number, date, and so on). The field types can also be complex—a field can contain other subdocuments or arrays. The field type is important to Elasticsearch because type determines how various operations such as analysis or sorting are performed.
- Unlike the relational databases, documents don't need to have a fixed structure—every document may have a different set of fields, and in addition to this, fields don't have to be known during application development. Of course, one can force a document structure with the use of schema. From the client's point of view, a document is a JSON object (see more about the JSON format at https://en.wikipedia.org/wiki/JSON). Each document is stored in one index and has its own unique identifier, which can be generated automatically by Elasticsearch, and document type. The thing to remember is that the document identifier needs to be unique inside an index and should be for a given type. This means that, in a single index, two documents can have the same unique identifier if they are not of the same type.



Document Types

```
{
   "data": {
      "mappings": {
        "_type": {
          "type": "string",
          "index": "not_analyzed"
        },
        "name": {
          "type": "string"
        }
        "address": {
          "type": "string"
        }
        "timestamp": {
          "type": "long"
        }
        "message": {
          "type": "string"
        }
      }
   }
}
```

- **DOCUMENT TYPE -** In Elasticsearch, one index can store many objects serving different purposes. For example, a blog application can store articles and comments. The document type lets us easily differentiate between the objects in a single index. Every document can have a different structure, but in real-world deployments, dividing documents into types significantly helps in data manipulation. Of course, one needs to keep the limitations in mind. That is, different document types can't set different types for the same property. For example, a field called title must have the same type across all document types in a given index.

Analysis

```
index :
    analysis :
        analyzer :
            standard :
                type : standard
                stopwords : [stop1, stop2]
            myAnalyzer1 :
                type : standard
                stopwords : [stop1, stop2, stop3]
                max_token_length : 500
            # configure a custom analyzer which is
            # exactly like the default standard analyzer
            myAnalyzer2 :
                tokenizer : standard
                filter : [standard, lowercase, stop]
        tokenizer :
            myTokenizer1 :
                type : standard
                max_token_length : 900
            myTokenizer2 :
                type : keyword
                buffer_size : 512
        filter :
            myTokenFilter1 :
                type : stop
                stopwords : [stop1, stop2, stop3, stop4]
            myTokenFilter2 :
                type : length
                min : 0
                max : 2000
```

- Analysis is the process of converting full text to terms. Depending on which analyzer is used, these phrases: FOO BAR, Foo-Bar, foo,bar will probably all result in the terms foo and bar. These terms are what is actually stored in the index. A full text query (not a term query) for FoO:bAR will also be analyzed to the terms foo,bar and will thus match the terms stored in the index. It is this process of analysis (both at index time and at search time) that allows elasticsearch to perform full text queries. Also see text and term.

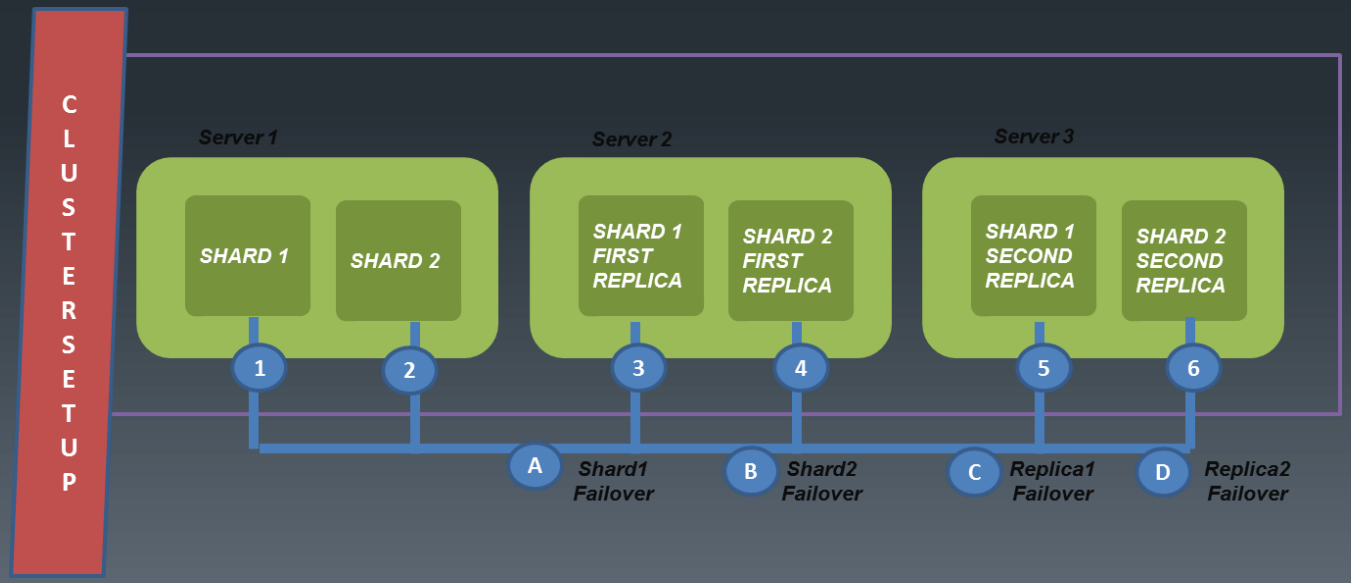
Mappings

```
PUT my_index (1)
{
  "mappings": {
    "user": { (2)
      "_all":       { "enabled": false  }, (3)
      "properties": { (4)
        "title":    { "type": "string"  }, (5)
        "name":     { "type": "string"  }, (6)
        "age":      { "type": "integer" }  (7)
      }
    },
    "blogpost": { (8)
      "properties": { (9)
        "title":    { "type": "string"  }, (10)
        "body":     { "type": "string"  }, (11)
        "user_id":  {
          "type":   "string", (12)
          "index":  "not_analyzed"
        },
        "created":  {
          "type":   "date", (13)
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
  }
}
```

- Mapping - Every field of the document must be properly analyzed depending on its type. For example, a different analysis chain is required for the numeric fields (numbers shouldn't be sorted alphabetically) and for the text fetched from web pages (for example, the first step would require you to omit the HTML tags as it is useless information). To be able to properly analyze at indexing and querying time, Elasticsearch stores the information about the fields of the documents in so-called mappings. Every document type has its own mapping, even if we don't explicitly define it.

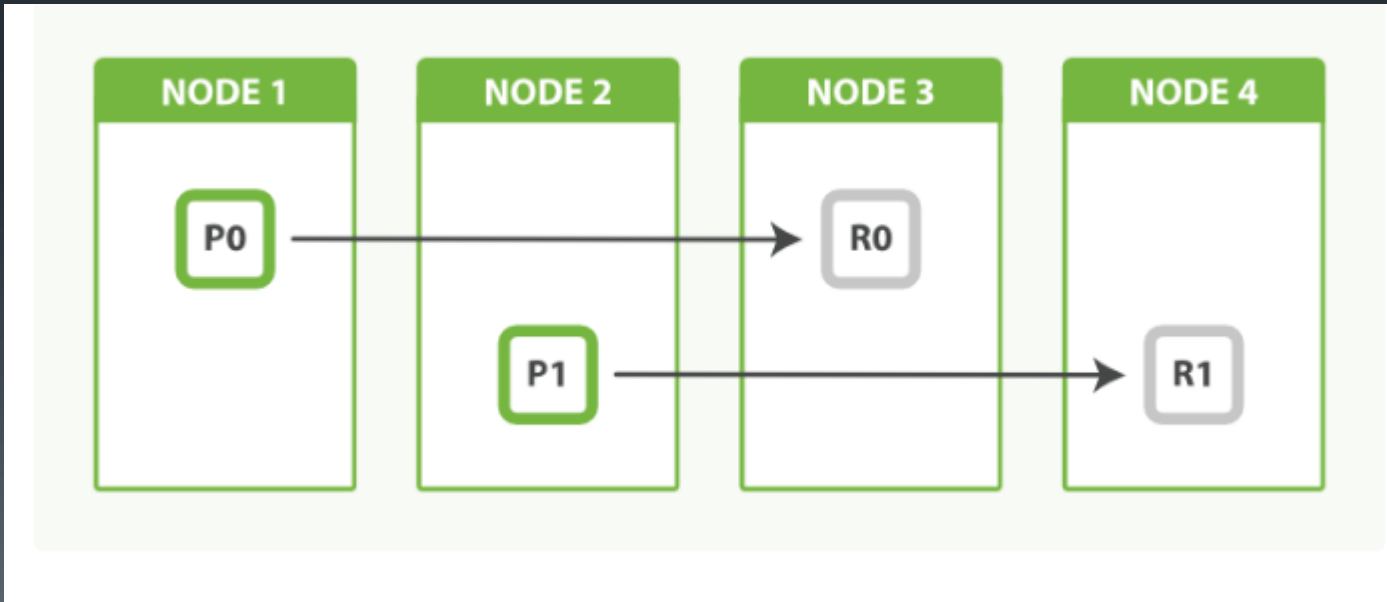
Nodes and Clusters

- Nodes and clusters- Elasticsearch can work as a standalone, single-search server. Nevertheless, to be able to process large sets of data and to achieve fault tolerance and high availability, Elasticsearch can be run on many cooperating servers. Collectively, these servers connected together are called a cluster and each server forming a cluster is called a node.



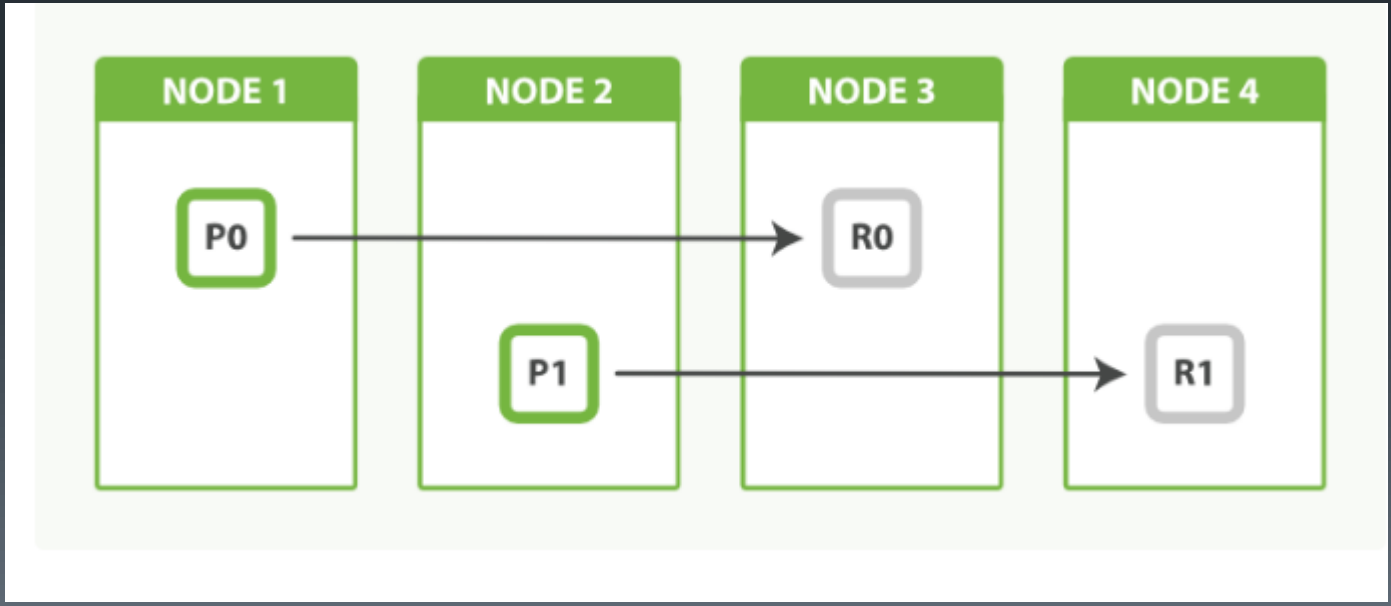
Shards

- **SHARDS -** When we have a large number of documents, we may come to a point where a single node may not be enough—for example, because of RAM limitations, hard disk capacity, insufficient processing power, and an inability to respond to client requests fast enough. In such cases, an index (and the data in it) can be divided into smaller parts called **shards** (where each shard is a separate Apache Lucene index). Each shard can be placed on a different server, and thus your data can be spread among the cluster nodes. When you query an index that is built from multiple shards, Elasticsearch sends the query to each relevant shard and merges the result in such a way that your application doesn't know about the shards. In addition to this, having multiple shards can speed up indexing, because documents end up in different shards and thus the indexing operation is parallelized.



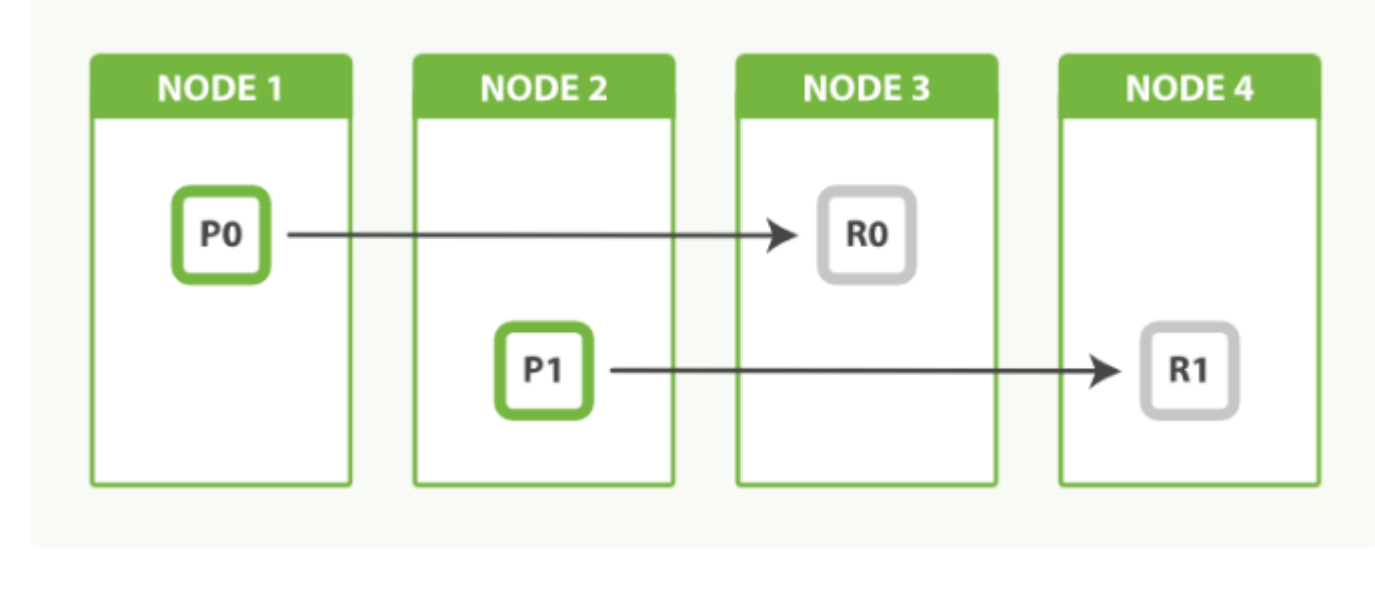
Primary Shards

- Each document is stored in a single primary shard. When you index a document, it is indexed first on the primary shard, then on all replicas of the primary shard. By default, an index has 5 primary shards. You can specify fewer or more primary shards to scale the number of documents that your index can handle. You cannot change the number of primary shards in an index, once the index is created. See also routing



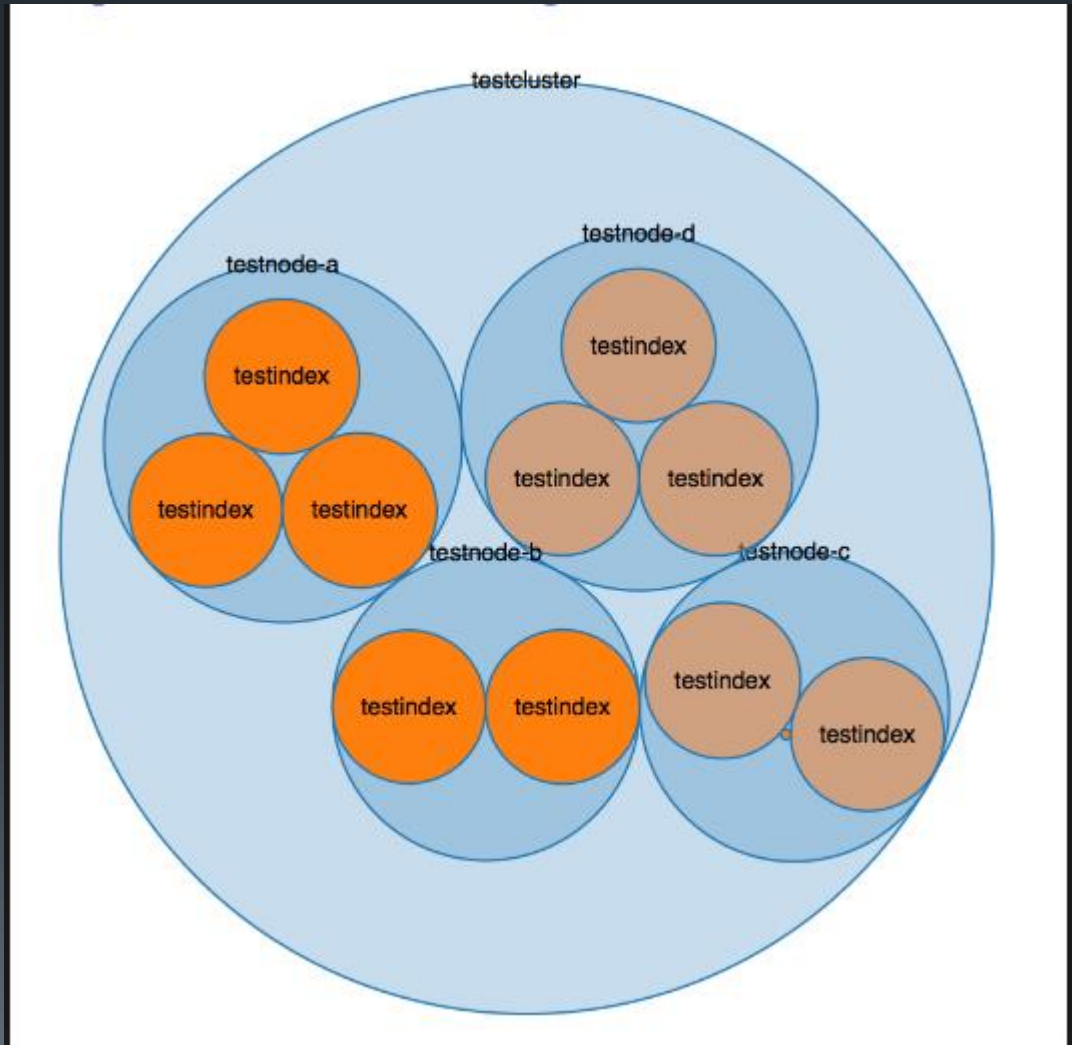
Replica Shards

- Each primary shard can have zero or more replicas. A replica is a copy of the primary shard, and has two purposes:
- increase failover: a replica shard can be promoted to a primary shard if the primary fails
- increase performance: get and search requests can be handled by primary or replica shards. By default, each primary shard has one replica, but the number of replicas can be changed dynamically on an existing index. A replica shard will never be started on the same node as its primary shard.

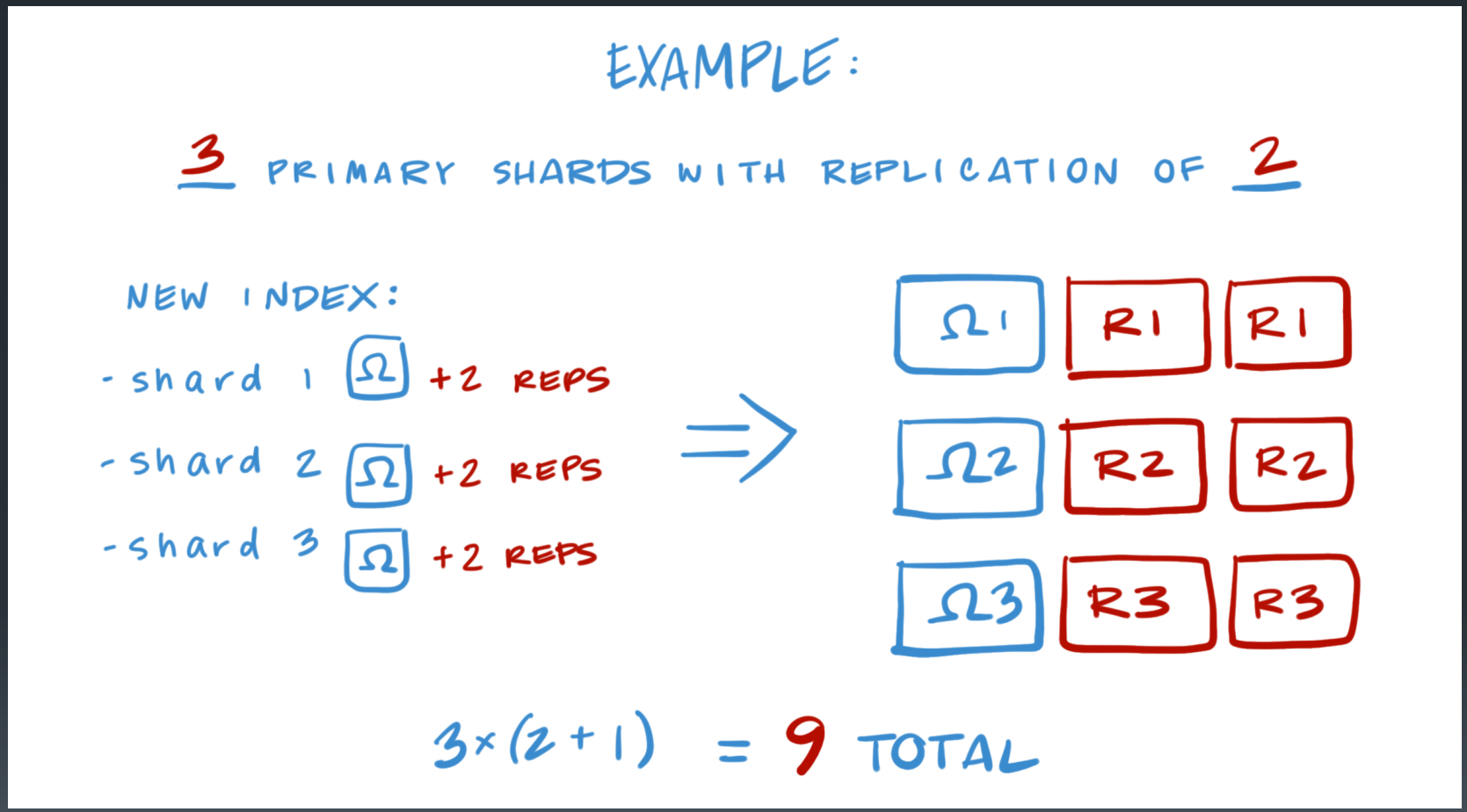


Clustering – Overview

A cluster consists of one or more nodes which share the same cluster name. The cluster acts as a team with the purpose of managing, accepting, and providing ES data. Each cluster is comprised of separate servers (virtual or physical) and has a single master node which is chosen automatically by the cluster and which can be replaced if the current master node fails.



Shard Allocation



- Note again that every shard contains a set of data from an index.
- Read Replicas are set copies of primary shards that are read-only.
- Read Replicas can be promoted to primary if a primary fails.

Node Roles

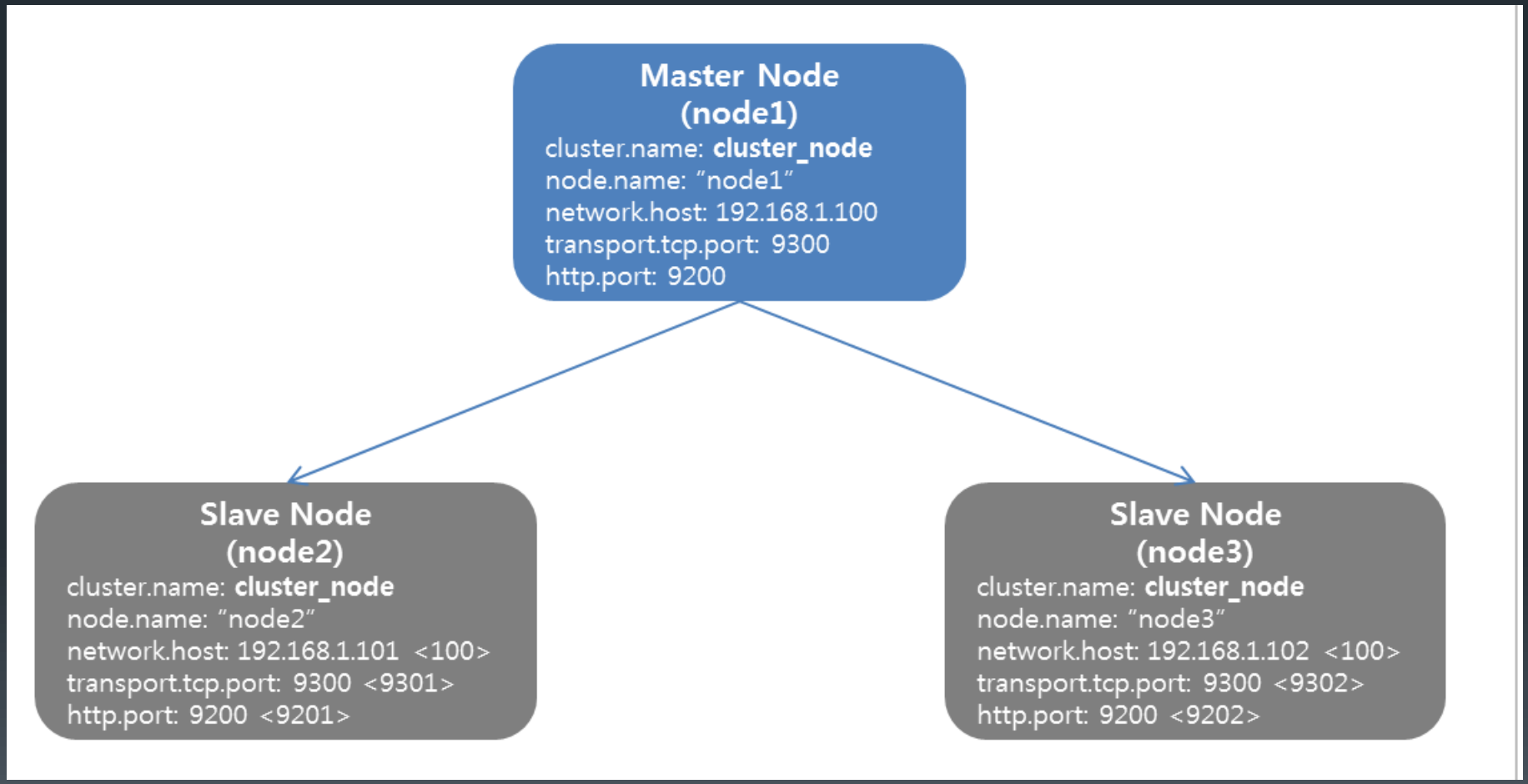
- **Node roles -** Elasticsearch nodes can be configured to work in one of the following roles:
  - **Master**: The node responsible for maintaining the global cluster state, changing it depending on the needs, and handling the addition and removal of nodes. There can only be a single master node active in a single cluster.
  - **Data**: The node responsible for holding the data and executing data related operations (indexation and searching) on the shards that are present locally for the node.
  - **Client**: The node responsible for handling requests. For the indexing requests, the client nodeforwards the request to the appropriate primary shard and, for the search requests, it sends it to all the relevant shards and aggregates the results.

```
curl -XGET 'localhost:9200/_cat/master?v'
```

The response returned by Elasticsearch for my local two-node cluster looks as follows:

```
id                     host      ip        node
Cfj3tzqpSNi5SZx4g8osAg 127.0.0.1 127.0.0.1 Skin
```

Master Nodes



- One node must be elected master in a cluster.
- You can have more than one master and you must have a quorum for operations to resume/begin.

Master Election

- **MASTER ELECTION CONFIGURATION -** Imagine that you have a cluster that is built of 10 nodes. Everything is working fine until one day when your network fails and 3 of your nodes are disconnected from the cluster, but they still see each other. Because of the Zen discovery and master election process, the nodes that got disconnected elect a new master and you end up with two clusters with the same name, with two master nodes. Such a situation is called a split-brain and you must avoid it as much as possible. When split-brain happens, you end up with two (or more) clusters that won't join each other until the network (or any other) problems are fixed. The thing to remember is that split-brain may result in not recoverable errors, such as data conflicts in which you end up with data corruption or partial data loss. That's why it is important to avoid such situations at all costs.

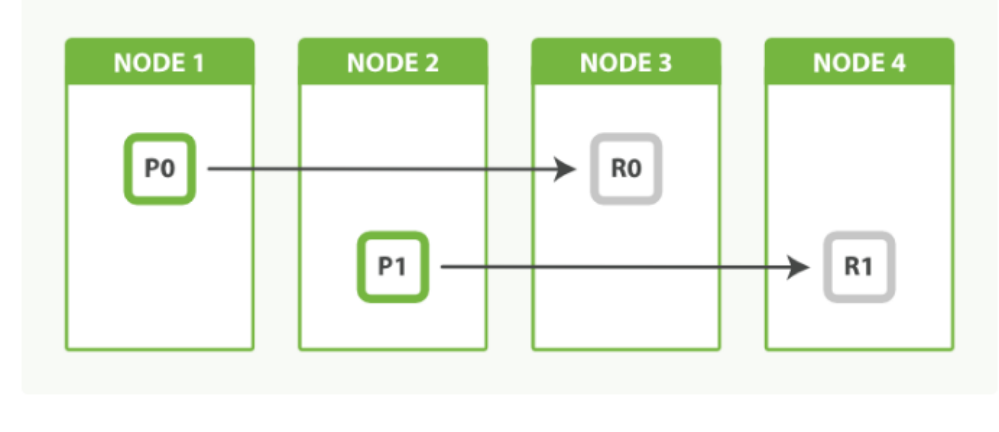
Indexing

However, until now we've used Elasticsearch functionality that allows us not to care about indices, shards, and data structure. This is not something that you may be used to when you are coming from the world of SQL databases, where you need the database and the tables with all the columns created upfront. In general, you needed to describe the data structure to be able to put data into the database. Elasticsearch is schema-less (data-driven schema) and by default creates indices automatically and because of that we can just install it and index data without the need of any preparations.

Shard Rehash

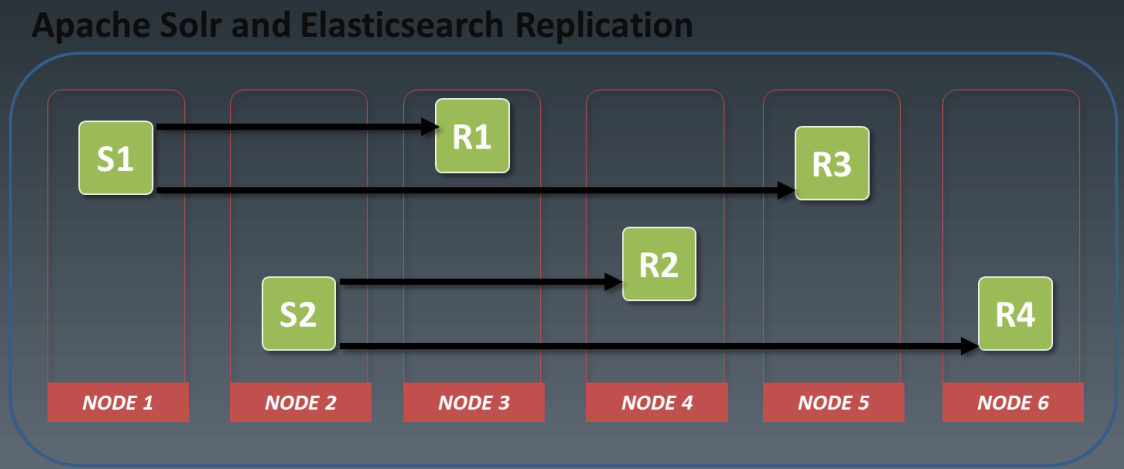
Earlier we told you that indices in Elasticsearch are built from one or more shards. Each of those shards contains part of the document set and each shard is a separate Lucene index. In addition to that, each shard can have replicas – physical copies of the primary shard itself. When we create an index, we can tell Elasticsearch how many shards it should be built from.

When defaults are used, we will end up with five Apache Lucene indices that our Elasticsearch index is built of and one replica for each of those. So, with five shards and one replica, we would actually get 10 shards. This is because each shard would get its own copy, so the total number of shards in the cluster would be 10.



Key Points about Indexing

- Having more shards in the index allows us to spread the index between more servers and parallelize the indexing operations and thus have better indexing throughput.
- Depending on your deployment, having more shards may increase query throughput and lower queries latency – especially in environments that don't have a large number of queries per second.
- Having more shards may be slower compared to a single shard query, because Elasticsearch needs to retrieve the data from multiple servers and combine them together in memory, before returning the final query results.
- Having more replicas results in a more resilient cluster, because when the primary shard is not available, its copy will take that role. Basically, having a single replica allows us to lose one copy of a shard and still serve the whole data. Having two replicas allows us to lose two copies of the shard and still see the whole data.
- The higher the replica count, the higher queries throughput the cluster will have. That's because each replica can serve the data it has independently from all the others.
- The higher number of shards (both primary and replicas) will result in more memory needed by Elasticsearch.



Write Consistency

- Elasticsearch allows us to control the write consistency to prevent writes happening when they should not. By default, Elasticsearch indexing operation is successful when the write is successful on the quorum on active shards – meaning 50% of the active shards plus one.
- We can control this behavior by adding action.write_consitency to our elasticsearch.yml file or by adding the consistency parameter to our index request. The mentioned properties can take the following values:
  - quorum: The default value, requiring 50% plus 1 active shards to be successful for the index operation to succeed
  - one: Requires only a single active shard to be successful for the index operation to succeed
  - all: Requires all the active shards to be successful for the index operation to succeed

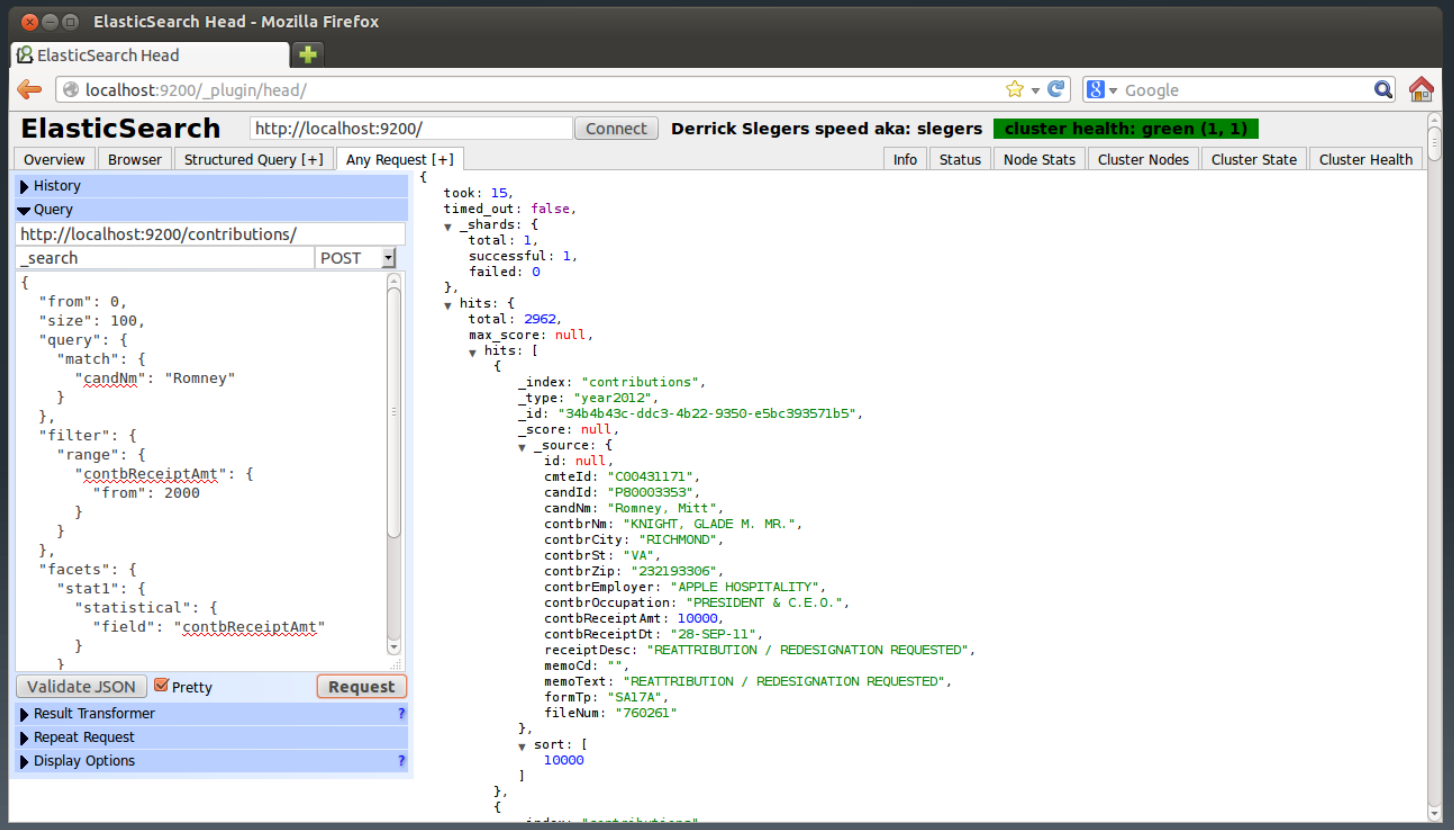
Summary

- Index, documents, fields, and mappings make up core data components of ElasticSearch
- The roles of ES servers are either data, master, or client nodes.
- Clusters are collections of nodes
- Shards are data segments dedicated to one or more nodes.
- Indexing is a way of grouping key terms to enable search

Lab 2 - Installing and Setting up Elasticsearch and associated plugins

- There is a LAB 1 titled PDF file that contains install instructions
- Lab 2 contains some cluster commands that you run when you need to do a mass or a rolling upgrade.

Module 3: Data Importation

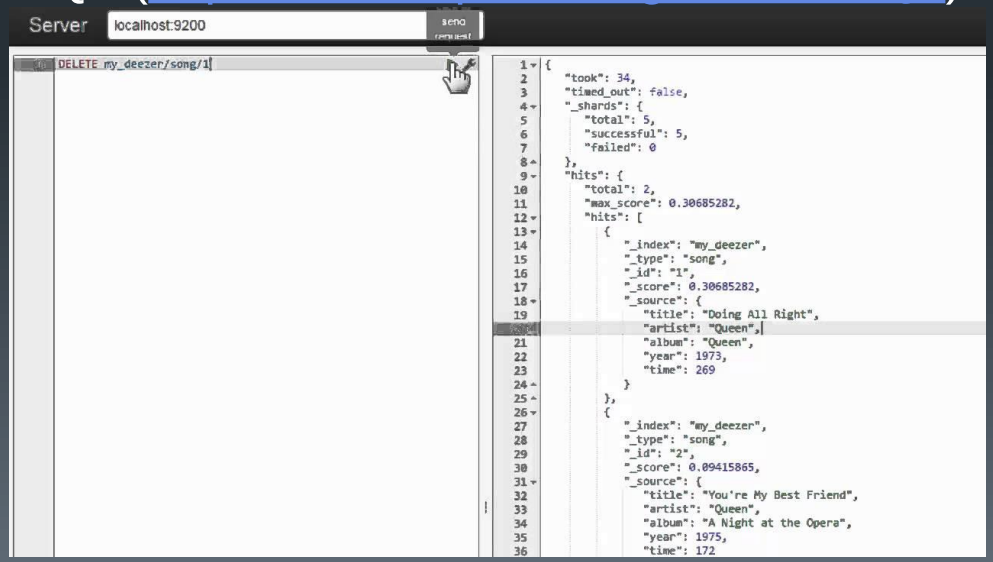


Objectives

- Document and Document Metadata
- CRUD for Documents
- Single versus Bulk data

Data Manipulation with REST

- Elasticsearch exposes a very rich REST API that can be used to search through the data, index the data, and control Elasticsearch behavior. You can imagine that using the REST API allows you to get a single document, index or update a document, get the information on Elasticsearch current state, create or delete indices, or force Elasticsearch to move around shards of your indices. We will concentrate on using the **create**, **retrieve**, **update**, **delete** (**CRUD**) part of the Elasticsearch API (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete), which allows us to use Elasticsearch in a fashion similar to how we would use any other NoSQL (https://en.wikipedia.org/wiki/NoSQL) data store.



Understanding the REST API

- If you've never used an application exposing the REST API, you may be surprised how easy it is to use such applications and remember how to use them. In REST-like architectures, every request is directed to a concrete object indicated by a path in the address. For example, let's assume that our hypothetical application exposes the /books REST end-point as a reference to the list of books. In such case, a call to /books/1 could be a reference to a concrete book with the identifier 1. You can think of it as a data-oriented model of an API. Of course, we can nest the paths—for example, a path such as /books/1/chapters could return the list of chapters of our book with identifier 1 and a path such as /books/1/chapters/6 could be a reference to the sixth chapter in that particular book.

- We talked about paths, but when using the HTTP protocol, (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) we have some additional verbs (such as POST, GET, PUT, and so on.) that we can use to define system behavior in addition to paths. So if we would like to retrieve the book with identifier 1, we would use the GET request method with the /books/1 path. However, we would use the PUT request method with the same path to create a book record with the identifier or one, the POST request method to alter the record, DELETE to remove that entry, and the HEAD request method to get basic information about the data referenced by the path.

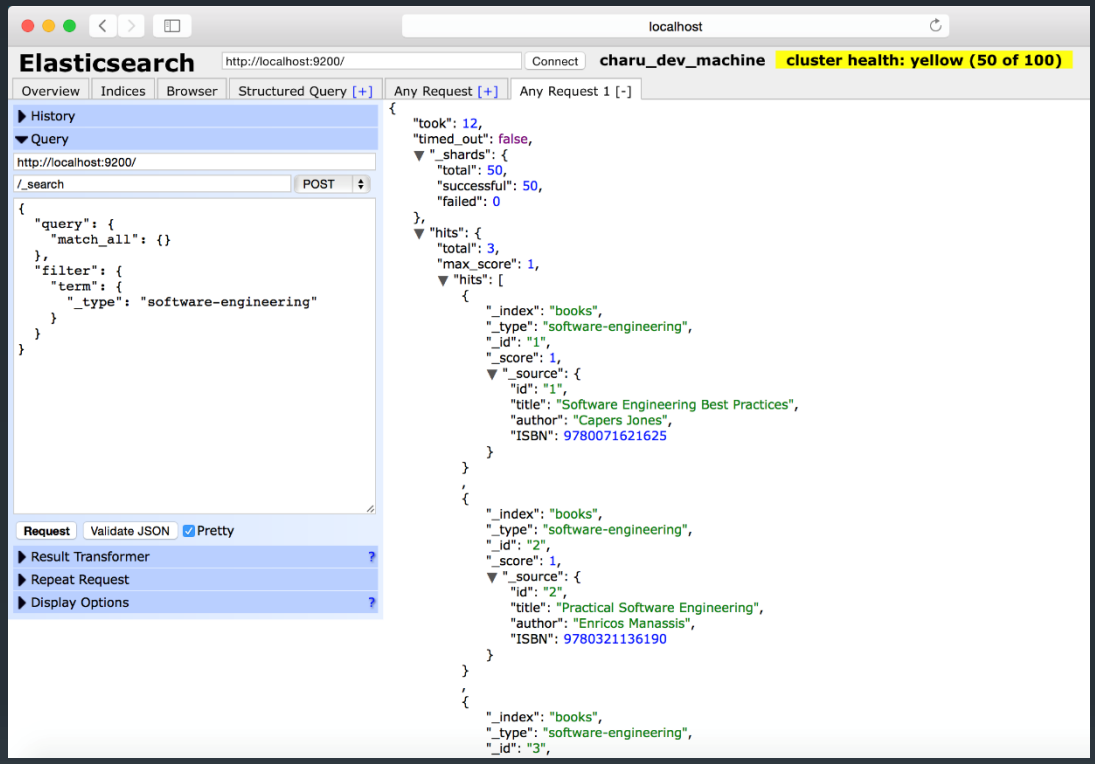
GET `http://localhost:9200/`: This retrieves basic information about Elasticsearch, such as the version, the name of the node that the command has been sent to, the name of the cluster that node is connected to, the Apache Lucene version, and so on.

GET `http://localhost:9200/_cluster/state/nodes/` This retrieves information about all the nodes in the cluster, such as their identifiers, names, transport addresses with ports, and additional node attributes for each node.

DELETE `http://localhost:9200/books/book/123`: This deletes a document that is indexed in the books index, with the book type and an identifier of 123.

Documents

- **DOCUMENT -** The main entity stored in Elasticsearch is a document. A document can have multiple fields, each having its own type and treated differently. Using the analogy to relational databases, a document is a row of data in a database table. All the documents with a field called title need to have the same data type for it, for example, string.
- Documents consist of fields, and each field may occur several times in a single document (such a field is called **multivalued**). Each field has a type (text, number, date, and so on). The field types can also be complex—a field can contain other subdocuments or arrays. The field type is important to Elasticsearch because type determines how various operations such as analysis or sorting are performed.
- Unlike the relational databases, documents don't need to have a fixed structure—every document may have a different set of fields, and in addition to this, fields don't have to be known during application development. Of course, one can force a document structure with the use of schema. From the client's point of view, a document is a JSON object (see more about the JSON format at https://en.wikipedia.org/wiki/JSON). Each document is stored in one index and has its own unique identifier, which can be generated automatically by Elasticsearch, and document type. The thing to remember is that the document identifier needs to be unique inside an index and should be for a given type. This means that, in a single index, two documents can have the same unique identifier if they are not of the same type.



Document Types

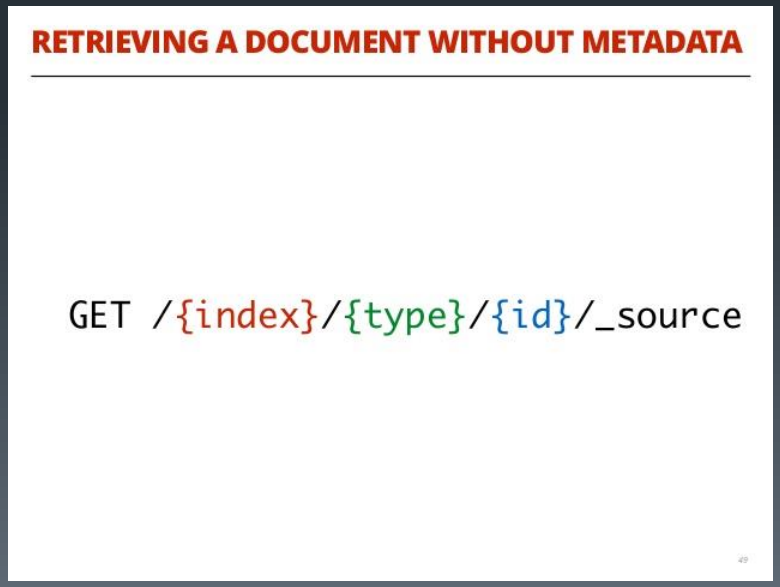
```
{
   "data": {
      "mappings": {
        "_type": {
          "type": "string",
          "index": "not_analyzed"
        },
        "name": {
          "type": "string"
        }
        "address": {
          "type": "string"
        }
        "timestamp": {
          "type": "long"
        }
        "message": {
          "type": "string"
        }
      }
   }
}
```

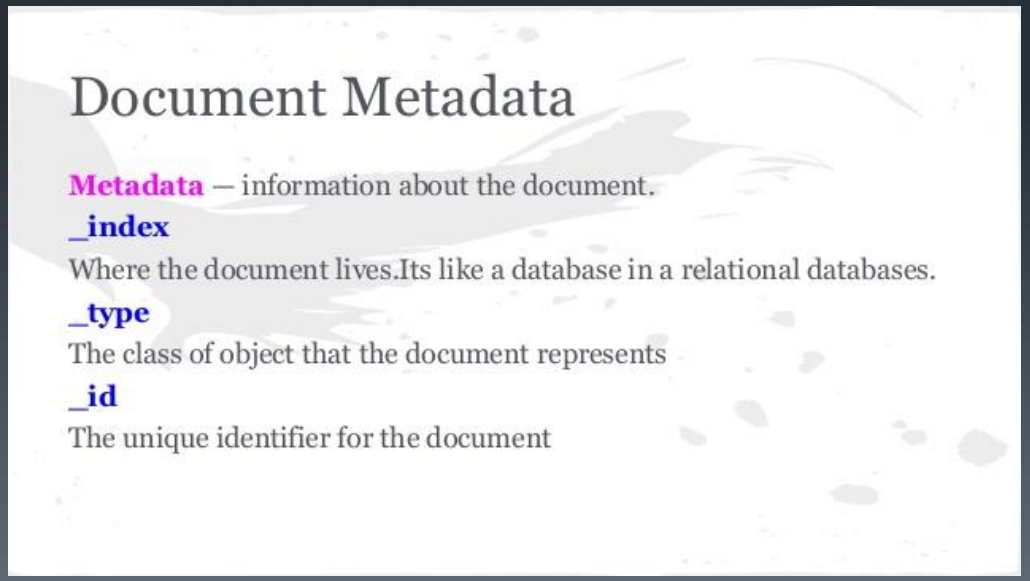
- **DOCUMENT TYPE -** In Elasticsearch, one index can store many objects serving different purposes. For example, a blog application can store articles and comments. The document type lets us easily differentiate between the objects in a single index. Every document can have a different structure, but in real-world deployments, dividing documents into types significantly helps in data manipulation. Of course, one needs to keep the limitations in mind. That is, different document types can't set different types for the same property. For example, a field called title must have the same type across all document types in a given index.

Document Metadata

A document doesn't consist only of its data. It also has metadata—information about the document. The three required metadata elements are as follows:

- _index Where the document lives
- _type The class of object that the document represents
- _id The unique identifier for the document





Creating a Document

We will start learning the Elasticsearch REST API by indexing one document. Let's imagine that we are building a CMS system (http://en.wikipedia.org/wiki/Content_management_system) that will provide the functionality of a blogging platform for our internal users. We will have different types of documents in our indices, but the most important ones are the articles that will be published and are readable by users.Because we talk to Elasticsearch using JSON notation and Elasticsearch responds to us again using JSON, our example document could look as follows:

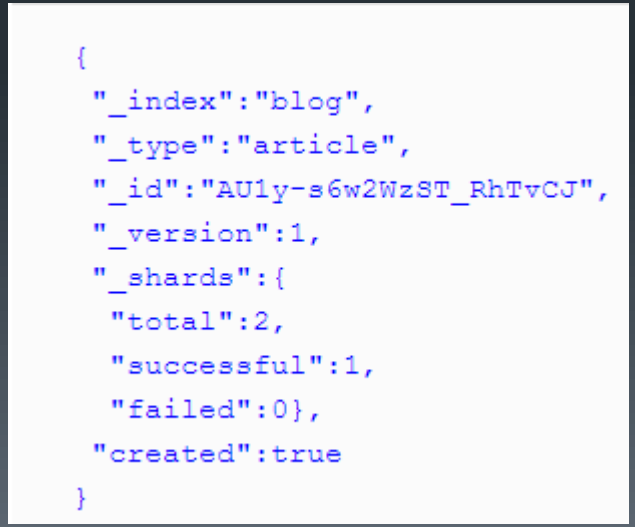
```
{
 "id": "1",
 "title": "New version of Elasticsearch released!",
 "content": "Version 2.2 released today!",
 "priority": 10,
 "tags": ["announce", "elasticsearch", "release"]
}
```

curl -XPUT 'http://localhost:9200/blog/article/1' -d '{"title": "New version of Elasticsearch released!", "content": "Version 2.2 released today!", "priority": 10, "tags": ["announce", "elasticsearch", "release"] }'

Automatic Identifier Creation

In the previous example, we specified the document identifier manually when we were sending the document to Elasticsearch. However, there are use cases when we don't have an identifier for our documents—for example, when handling logs as our data. In such cases, we would like some application to create the identifier for us and Elasticsearch can be such an application. Of course, generating document identifiers doesn't make sense when your document already has them, such as data in a relational database. In such cases, you may want to update the documents; in this case, automatic identifier generation is not the best idea. However, when we are in need of such functionality, instead of using the HTTP PUT method we can use POST and omit the identifier in the REST API path. So if we would like Elasticsearch to generate the identifier in the previous example, we would send a command like this:

```
curl -XPOST 'http://localhost:9200/blog/article/' -d '{"title": "New version of Elasticsearch released!", "content": "Version 2.2 released today!", "priority": 10, "tags": ["announce", "elasticsearch", "release"] }'
```



Retrieving Documents

Let's now try to retrieve one of the documents using its unique identifier. To do this, we will need information about the index the document is indexed in, what type it has, and of course what identifier it has. For example, to get the document from the blog index with the article type and the identifier of 1, we would run the following HTTP GET request:

curl -XGET 'localhost:9200/blog/article/1?pretty'

NOTE

The additional URI property called `pretty` tells Elasticsearch to include new line characters and additional white spaces in response to make the output easier to read for users.

```
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "1",
  "_version" : 1,
  "found" : true,
  "_source" : {
    "title" : "New version of Elasticsearch released!",
    "content" : "Version 2.2 released today!",
    "priority" : 10,
    "tags" : [ "announce", "elasticsearch", "release" ]
  }
}
```

Updating a Document

Updating documents in the index is a more complicated task compared to indexing. When the document is indexed and Elasticsearch flushes the document to a disk, it creates segments—an immutable structure that is written once and read many times. This is done because the inverted index created by Apache Lucene is currently impossible to update (at least most of its parts). To update a document, Elasticsearch internally first fetches the document using the GET request, modifies its _source field, removes the old document, and indexes a new document using the updated content.

```
curl -XPOST 'http://localhost:9200/blog/article/1/_update' -d '{
 "script" : "ctx._source.content = new_content",
 "params" : {
  "new_content" : "This is the updated document"
 }
}'
```

Other document updates

- Upsert – to update or add the document if it doesn't exist
- Partial updates – using POST

```
curl -XPOST 'http://localhost:9200/blog/article/2/_update' -d '{
 "script" : "ctx._source.priority += 1",
 "upsert" : {
  "title" : "Empty document",
  "priority" : 0,
  "tags" : ["empty"]
 }
}'
```

```
curl -XPOST 'http://localhost:9200/blog/article/1/_update' -d '{
  "doc" : {
    "count" : 1
  },
  "doc_as_upsert" : true
}
```

Deleting Documents

- Now that we know how to index documents, update them, and retrieve them, it is time to learn about how we can delete them. Deleting a document from an Elasticsearch index is very similar to retrieving it, but with one major difference—instead of using the HTTP GET method, we have to use HTTP DELETE one. For example, if we would like to delete the document indexed in the blog index under the article type and with an identifier of 1, we would run the following command:

```
curl -XDELETE 'localhost:9200/blog/article/1'
```

```
{
 "found":true,
 "_index":"blog",
 "_type":"article",
 "_id":"1",
 "_version":4,
 "_shards":{
  "total":2,
  "successful":1,
  "failed":0
 }
}
```

```
curl -XDELETE 'localhost:9200/blog'
```

Versioning

- As you may have already noticed, Elasticsearch increments the document version when it does updates to it. We can leverage this functionality and use optimistic locking (http://en.wikipedia.org/wiki/Optimistic_concurrency_control), and avoid conflicts and overwrites when multiple processes or threads access the same document concurrently. You can assume that your indexing application may want to try to update the document, while the user would like to update the document while doing some manual work. The question that arises is: Which document should be the correct one—the one updated by the indexing application, the one updated by the user, or the merged document of the changes? What if the changes are conflicting? To handle such cases, we can use versioning.

```
curl -XPUT 'localhost:9200/blog/article/10' -d '{"title":"Test document"}'
curl -XPUT 'localhost:9200/blog/article/10' -d '{"title":"Updated test document"}'
```

```
curl -XDELETE 'localhost:9200/blog/article/10?version=1'
```

```
{
  "error" : {
    "root_cause" : [ {
      "type" : "version_conflict_engine_exception",
      "reason" : "[article][10]: version conflict, current [2], provided [1]",
      "shard" : 1,
      "index" : "blog"
    } ],
    "type" : "version_conflict_engine_exception",
    "reason" : "[article][10]: version conflict, current [2], provided [1]",
    "shard" : 1,
    "index" : "blog"
  },
  "status" : 409
}
```

Bulk Creation and Updates

- Elasticsearch allows us to merge many requests into one package. This package can be sent as a single request. What's more, we are not limited to having a single type of request in the so called bulk – we can mix different types of operations together, which include:
  - Adding or replacing the existing documents in the index (index)
  - Removing documents from the index (delete)
  - Adding new documents into the index when there is no other definition of the document in the index (create)
  - Modifying the documents or creating new ones if the document doesn't exist (update)

```
{ "index": { "_index": "addr", "_type": "contact", "_id": 1 }}
{ "name": "Fyodor Dostoevsky", "country": "RU" }
{ "create": { "_index": "addr", "_type": "contact", "_id": 2 }}
{ "name": "Erich Maria Remarque", "country": "DE" }
{ "create": { "_index": "addr", "_type": "contact", "_id": 2 }}
{ "name": "Joseph Heller", "country": "US" }
{ "delete": { "_index": "addr", "_type": "contact", "_id": 4 }}
{ "delete": { "_index": "addr", "_type": "contact", "_id": 1 }}
```

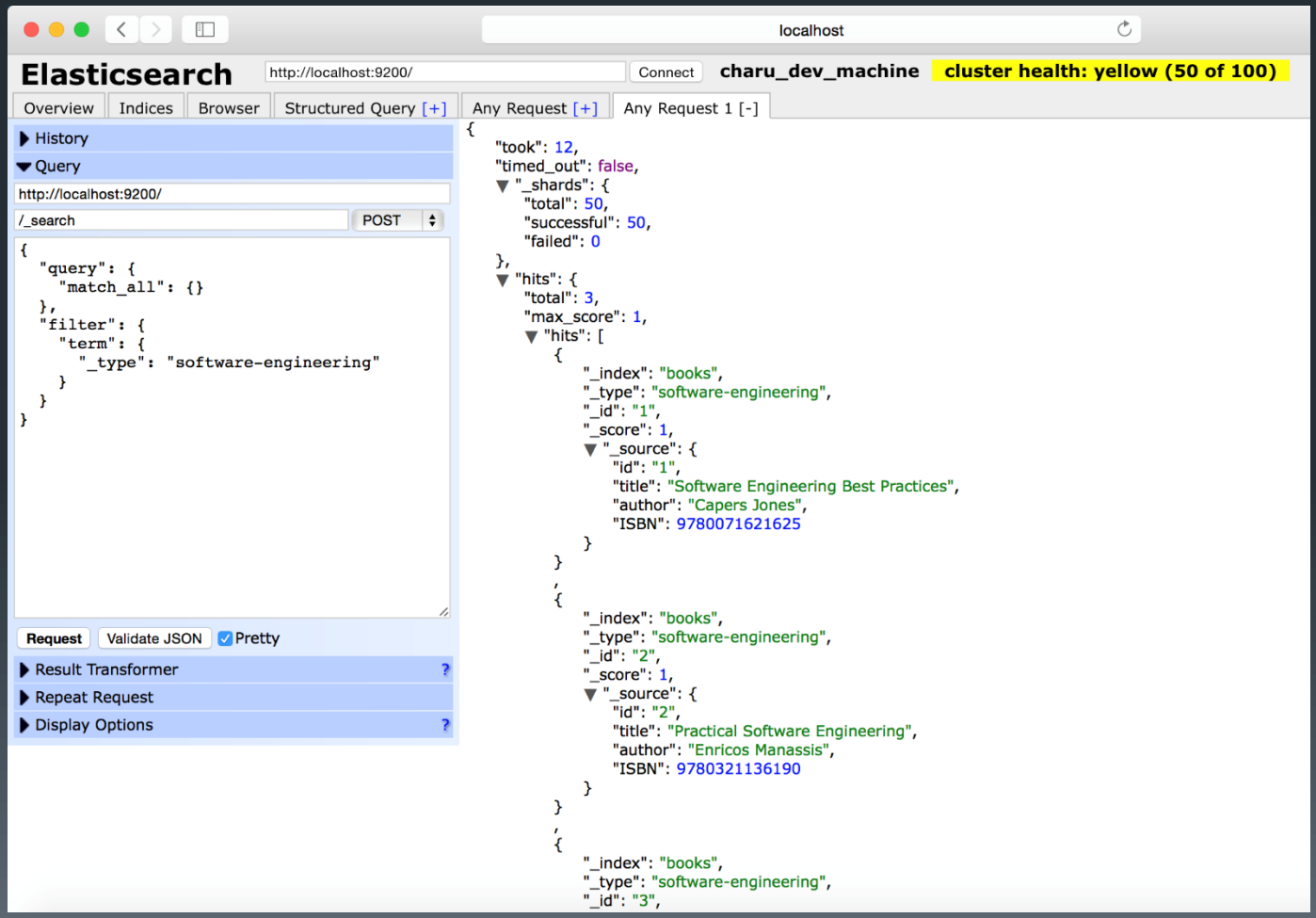
```
curl -XPOST 'localhost:9200/_bulk?pretty' --data-binary @documents.json
```

Summary

- We explored the REST API with various endpoint for mananging ElasticSearch
- Performed Create, Replace, Update, and Delete actions using the RESTful API
- Documents can be partial updated or ‘upsert’ed if they don’t’ already exist.
- Documents are automatically versioned, but can be externally and manually versioned.
- Touched on Bulk Creation and Updates.

Lab 3 - importing data and verifying data

Module 4: Queries



Objectives

- Search,Query,Search
- Basic queries
- DSL queries
- Phrase Search
- Full-text search

URI Request Query

Before getting into the wonderful world of the Elasticsearch query language, we would like to introduce you to the simple but pretty flexible URI request search, which allows us to use a simple Elasticsearch query combined with the Lucene query language.

All queries in Elasticsearch are sent to the _search endpoint. You can search a single index or multiple indices, and you can restrict your search to a given document type or multiple types. For example, in order to search our book's index, we will run the following command:

```
curl -XGET 'localhost:9200/books/_search?pretty'
```


```
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 6,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "books",
      "_type" : "es",
      "_id" : "2",
      "_score" : 1.0,
      "_source" : {
        "title" : "Elasticsearch Server Second Edition",
        "published" : 2014
      }
    }, {
      "_index" : "books",
      "_type" : "es",
      "_id" : "4",
      "_score" : 1.0,
      "_source" : {
        "title" : "Mastering Elasticsearch Second Edition",
        "published" : 2015
      }
```

URI for many indices

We can also run queries against many indices. For example, if we had another index called clients, we could also run a single query against these two indices as follows:

```
curl -XGET 'localhost:9200/books,clients/_search?pretty'
```

We can also run queries against all the data in Elasticsearch by omitting the index names completely or setting the queries to _all:

```
curl -XGET 'localhost:9200/_search?pretty'
curl -XGET 'localhost:9200/_all/_search?pretty'
```

In a similar manner, we can also choose the types we want to use during searching. For example, if we want to search only in the es type in the book's index, we run a command as follows:

```
curl -XGET 'localhost:9200/books/es/_search?pretty'
```

URI Search with Title

Let's assume that we want to find all the documents in our book's index that contain the elasticsearch term in the title field. We can do this by running the following query:

```
curl -XGET 'localhost:9200/books/_search?pretty&q=title:elasticsearch'
```

The response will look like this:


```
{
  "took" : 37,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 0.625,
    "hits" : [ {
      "_index" : "books",
      "_type" : "es",
      "_id" : "1",
      "_score" : 0.625,
      "_source" : {
        "title" : "Elasticsearch Server",
        "published" : 2013
      }
    }, {
      "_index" : "books",
      "_type" : "es",
      "_id" : "2",
      "_score" : 0.5,
      "_source" : {
        "title" : "Elasticsearch Server Second Edition",
        "published" : 2014
      }
    }, {
```

URI Query String Parameters

- There are a few parameters that we can use to control URI query behavior, which we will discuss now. The thing to remember is that each parameter in the query should be concatenated with the & character, as shown in the following example:
- curl -XGET 'localhost:9200/books/_search?pretty&q=published:2013&df=title&explain=true&default_operator=AND'
- The q parameter allows us to specify the query that we want our documents to match.
- Using the df parameter, we can specify the default search field that should be used when no field indicator is used in the q parameter. By default, the _all field will be used.
- The default_operator property that can be set to OR or AND, allows us to specify the default Boolean operator used for our query (http://en.wikipedia.org/wiki/Boolean_algebra).
- If we set the explain parameter to true, Elasticsearch will include additional explain information with each document in the result—such as the shard from which the document was fetched and the detailed information about the scoring calculation

More URI Query String Parameters

- Using the sort parameter, we can specify custom sorting.
- By default, Elasticsearch doesn't have timeout for queries, but you may want your queries to timeout after a certain amount of time (for example, 5 seconds). Elasticsearch allows you to do this by exposing the timeout parameter.
- Elasticsearch allows you to specify the results window (the range of documents in the results list that should be returned). We have two parameters that allow us to specify the results window size: size and from.
- Elasticsearch allows us to specify the maximum number of documents that should be fetched from each shard using terminate_after property and specifying the maximum number of documents.

Full List of URI Parameters

- Ignore_unavailable=true
- Search_type=query_then_fetch
- Lowercase_expanded_terms
- Analyze_wildcard=true
- So many more.

NOTE

If you want to see all the parameters exposed by Elasticsearch as the URI request parameters, please refer to the official documentation available at: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-uri-request.html.

Basic Query String

- When you use the _search endpoint that is a basic URI query

```
curl -XGET 'localhost:9200/library/book/_search?q=title:crime&pretty'
```

This is a simple but limited way to execute search. This query is technically a query_string query using the _search endpoint. It can be rewritten in a more expansive format as follows:

```
curl -XGET 'localhost:9200/library/book/_search?pretty' -d '{
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}'
```

Paging and Result Size

- Elasticsearch allows us to control how many results we want to get (at most) and from which result we want to start. The following are the two additional properties that can be set in the request body:
  - from: This property specifies the document that we want to have our results from. Its default value is 0, which means that we want to get our results from the first document.
  - size: This property specifies the maximum number of documents we want as the result of a single query (which defaults to 10). For example, if we are only interested in aggregations results and don't care about the documents returned by the query, we can set this parameter to 0.If we want our query to get documents starting from the tenth item on the list and fetch 20 documents, we send the following query:

```
curl -XGET 'localhost:9200/library/book/_search?pretty' -d '{
  "from" :  9,
  "size" : 20,
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}'
```

Boosting a Query

- we can also include the boost attribute to our term query, which will affect the importance of the given term. Remember that it changes the importance of the given part of the query. For example, to change a query and give our term query a boost of 10.0, send the following query:

```
{
  "query" : {
  "term" : {
    "title" : {
    "value" : "crime",
    "boost" : 10.0
    }
  }
  }
}
```

Terms Query

- The terms query is an extension to the term query. It allows us to match documents that have certain terms in their contents instead of a single term. The term query allowed us to match a single, not analyzed term and the terms query allows us to match multiple of those. For example, let's say that we want to get all the documents that have the terms novel or book in the tags field. To achieve this, we will run the following query:

```
{
  "query" : {
  "terms" : {
    "tags" : [ "novel", "book" ]
  }
  }
}
```

The preceding query returns all the documents that have one or both of the searched terms in the tags field. This is a key point to remember – the terms query will find documents having any of the provided terms.

Match All Query

- The match all query is one of the simplest queries available in Elasticsearch. It allows us to match all of the documents in the index. If we want to get all the documents from our index, we just run the following query:

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

```
{
  "query" : {
  "match_all" : {
    "boost" : 2.0
  }
  }
}
```

Type Query

- A very simple query that allows us to find all the documents with a certain type. For example, if we would like to search for all the documents with the book type in our library index, we will run the following query:

```
{
  "query" : {
  "type" : {
    "value" : "book"
  }
  }
}
```

Exists Query

- A query that allows us to find all the documents that have a value in the defined field. For example, to find the documents that have a value in the tags field, we will run the following query:

```
{
  "query" : {
  "exists" : {
    "field" : "tags"
  }
  }
}
```

Missing Query

- Opposite to the exists query, the missing query returns the documents that have a null value or no value at all in a given field. For example, to find all the documents that don't have a value in the tags field, we will run the following query:

```
{
  "query" : {
  "missing" : {
    "field" : "tags"
  }
  }
}
```

Phrase Search Query

- Finding individual words in a field is all well and good, but sometimes you want to match exact sequences of words or phrases. For instance, we could perform a query that will match only employee records that contain both “rock” and “climbing” and that display the words next to each other in the phrase “rock climbing.”

- To do this, we use a slight variation of the match query called the match_phrase query:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    }
}
```

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         }
      ]
   }
}
```

Full-Text Search Query

- By default, Elasticsearch sorts matching results by their relevance score, that is, by how well each document matches the query.
- Elasticsearch can search *within* full-text fields and return the most relevant results first. This concept of *relevance* is important to Elasticsearch, and is a concept that is completely foreign to traditional relational databases, in which a record either matches or it doesn't.

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "about" : "rock climbing"
        }
    }
}
```

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.16273327,
      "hits": [
         {
            ...
            "_score":         0.16273327, ❶
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_score":         0.016878016, ❷
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

More Complicated Search…

- Don’t worry about the syntax too much for now; we will cover it in great detail later. Just recognize that we’ve added a filter that performs a range search, and reused the same match query as before.

```
GET /megacorp/employee/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "range" : {
                    "age" : { "gt" : 30 } ❶
                }
            },
            "query" : {
                "match" : {
                    "last_name" : "smith" ❷
                }
            }
        }
    }
}
```

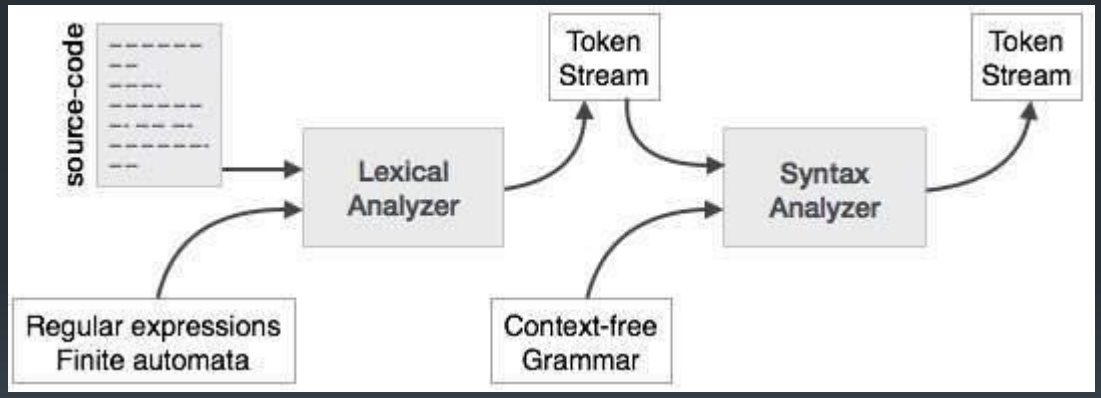
```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

Summary

- Executed basic search
- Explored the basic URI search with terms
- Examined a number of Query terms
- Looked at Phrase Search
- Showcased Full-Text Search
- Touched on Complicated Search

Lab 4 - Let’s Search

Module 5: Analyzers

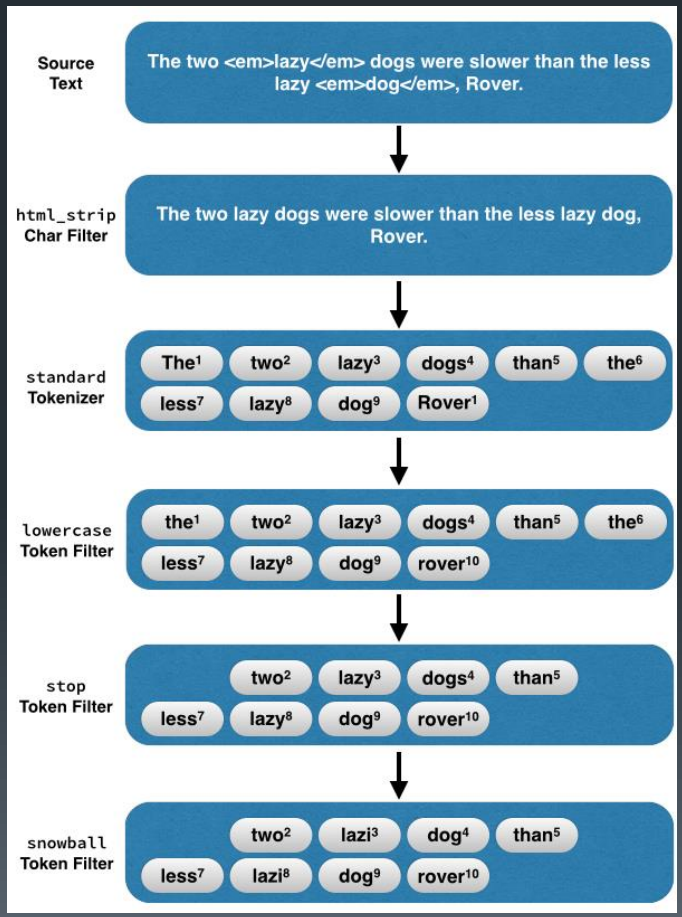


Objectives

- Analysis and Analyzers
- Built-In Analyzers
- Enabling Analyzers
- Testing Analyzers

Analysis and Analyzers

- The index analysis module acts as a configurable registry of Analyzers that can be used in order to both break indexed (analyzed) fields when a document is indexed and process query strings. It maps to the Lucene Analyzer.
- Analyzers are composed of a single Tokenizer and zero or more TokenFilters. The tokenizer may be preceded by one or more CharFilters. The analysis module allows you to register Analyzers under logical names which can then be referenced either in mapping definitions or in certain APIs.
- Elasticsearch comes with a number of prebuilt analyzers which are ready to use. Alternatively, you can combine the built in character filters, tokenizers and token filters to create custom analyzers.



Indexing and Analysis

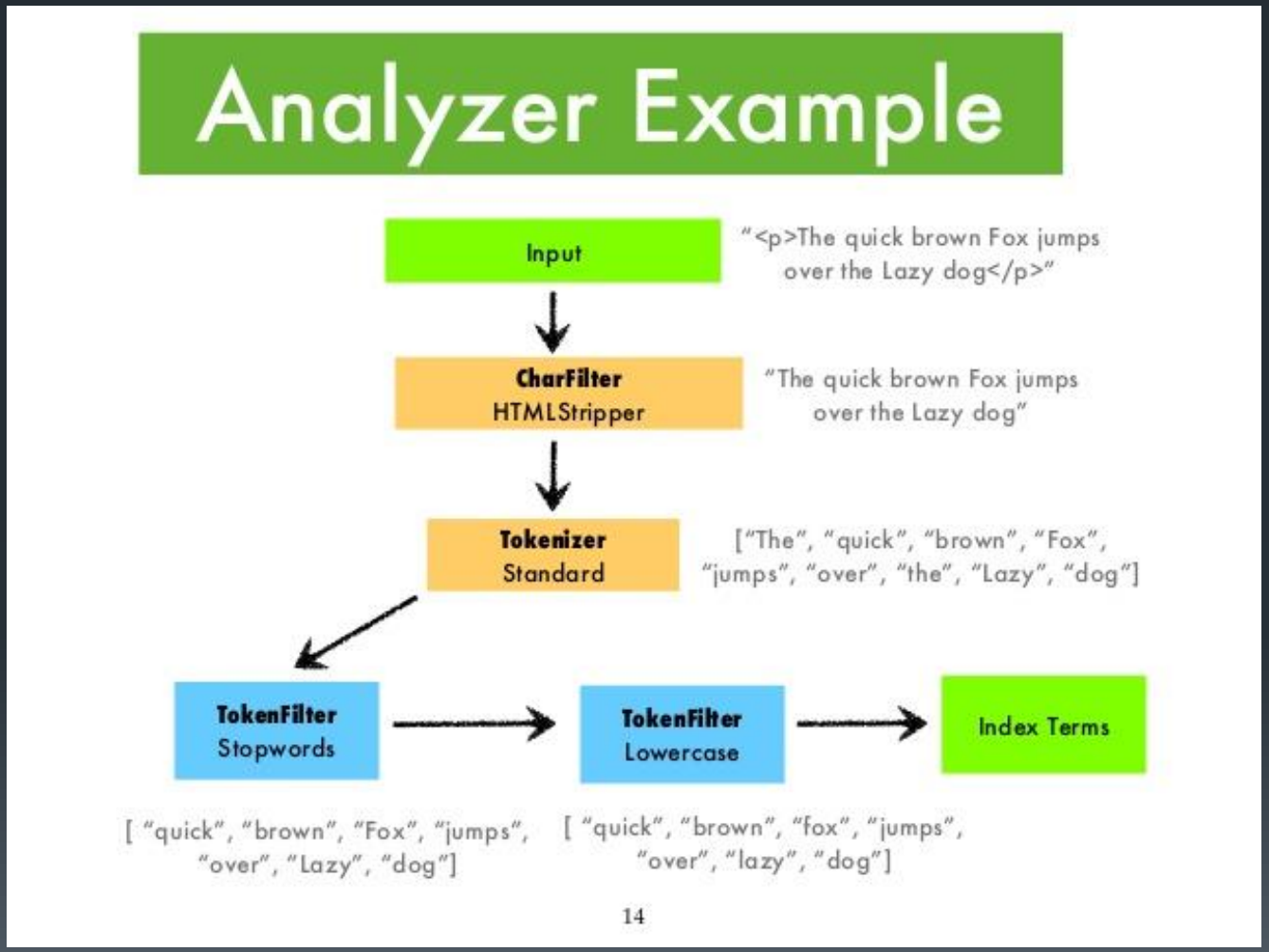
- The goal of indexing is to provide quick matches. Since humans don't always use the same exact words to describe something, the words need to be massaged a bit to normalize case and remove suffixes such as -ed or -ing. This massaging is called analysis, and it is performed on both the documents being searched and the search query itself.

- Analyzing a lot of documents takes time, so it is usually done up front. This process is called indexing. Analyzed documents are stored in a format that is efficient for searching, called an index.

Components of Analysis

- an analyzer is a wrapper that combines three functions into a single package, which are executed in sequence:
  - Character filters
    - Character filters are used to “tidy up” a string before it is tokenized. For instance, if our text is in HTML format, it will contain HTML tags like <p> or that we don’t want to be indexed. We can use the html_strip character filter to remove all HTML tags and to convert HTML entities like Á into the corresponding Unicode character Á.
    - An analyzer may have zero or more character filters.
  - Tokenizers
    - An analyzer must have a single tokenizer. The tokenizer breaks up the string into individual terms or tokens. The standard tokenizer, which is used in the standard analyzer, breaks up a string into individual terms on word boundaries, and removes most punctuation, but other tokenizers exist that have different behavior.
    - For instance, the keyword tokenizer outputs exactly the same string as it received, without any tokenization. The whitespace tokenizer splits text on whitespace only. The pattern tokenizer can be used to split text on a matching regular expression.
  - Token filters
    - After tokenization, the resulting token stream is passed through any specified token filters, in the order in which they are specified.
    - Token filters may change, add, or remove tokens. We have already mentioned the lowercase and stop token filters, but there are many more available in Elasticsearch. Stemming token filters “stem” words to their root form. The ascii_folding filter removes diacritics, converting a term like "très" into "tres". The ngram and edge_ngram token filters can produce tokens suitable for partial matching or autocomplete.

Generic Analysis



- Data is accepted
- The CharFilter removes certain characters
- The standard tokenizer segments each piece of data
- Stopwords removes non-unique words
- Lowercase lowers
- Terms are put into the index

Default Analyzers

- An analyzer is registered under a logical name. It can then be referenced from mapping definitions or certain APIs. When none are defined, defaults are used. There is an option to define which analyzers will be used by default when none can be derived.
- The default logical name allows one to configure an analyzer that will be used both for indexing and for searching APIs. The default_search logical name can be used to configure a default analyzer that will be used just when searching (the default analyzer would still be used for indexing).
- For instance, the following settings could be used to perform exact matching only by default:

```
index :
  analysis :
    analyzer :
      default :
        tokenizer : keyword
```

Aliasing Analyzers

- Analyzers can be aliased to have several registered lookup names associated with them. For example, the following will allow the standard analyzer to also be referenced with alias1 and alias2 values.

```
index :
  analysis :
    analyzer :
      standard :
        alias: [alias1, alias2]
        type : standard
        stopwords : [test1, test2, test3]
```

- There are several built in analyzers that we will review.

Standard Analyzer

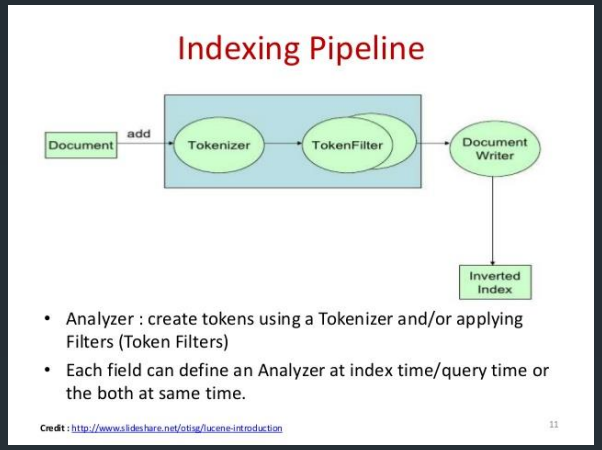
- An analyzer of type standard is built using the Standard Tokenizer with the Standard Token Filter, Lower Case Token Filter, and Stop Token Filter.

- The following are settings that can be set for a standard analyzer type:

Setting	Description
`stopwords`	A list of stopwords to initialize the stop filter with. Defaults to an *empty* stopword list Check Stop Analyzer for more details.
`max_token_length`	The maximum token length. If a token is seen that exceeds this length then it is split at `max_token_length` intervals. Defaults to `255`.

Other Built-in Analyzers

- Simple Analyzer - An analyzer of type simple that is built using a Lower Case Tokenizer.
- Whitespace Analyzer - An analyzer of type simple that is built using a Lower Case Tokenizer.
- Stop Analyzer - An analyzer of type stop that is built using a Lower Case Tokenizer, with Stop Token Filter.
- Keyword Analyzer - An analyzer of type keyword that "tokenizes" an entire stream as a single token. This is useful for data like zip codes, ids and so on. Note, when using mapping definitions, it might make more sense to simply mark the field as not_analyzed.
- Pattern Analyzer - An analyzer of type pattern that can flexibly separate text into terms via a regular expression.
- Language Analyzer - A set of analyzers aimed at analyzing specific language text.
- Snowball Analyzer - An analyzer of type snowball that uses the standard tokenizer, with standard filter, lowercase filter, stop filter, and snowball filter. The Snowball Analyzer is a stemming analyzer from Lucene that is originally based on the snowball project from snowballstem.org.



Sample Analyzer Configuration

- The Analysis module automatically registers (**if not explicitly defined**) built in analyzers, token filters, and tokenizers.
- Here is a sample configuration:

```
index :
    analysis :
        analyzer :
            standard :
                type : standard
                stopwords : [stop1, stop2]
            myAnalyzer1 :
                type : standard
                stopwords : [stop1, stop2, stop3]
                max_token_length : 500
            # configure a custom analyzer which is
            # exactly like the default standard analyzer
            myAnalyzer2 :
                tokenizer : standard
                filter : [standard, lowercase, stop]
        tokenizer :
            myTokenizer1 :
                type : standard
                max_token_length : 900
            myTokenizer2 :
                type : keyword
                buffer_size : 512
        filter :
            myTokenFilter1 :
                type : stop
                stopwords : [stop1, stop2, stop3, stop4]
            myTokenFilter2 :
                type : length
                min : 0
                max : 2000
```

Configuring Analyzers

- Elasticsearch supports multiple indexes per server instance. Analyzers can be configured on a per-index basis — just include the analyzer configuration when you create the index.
- You can create and delete indexes without restarting elasticsearch.
- The following example creates an index called test and configures a custom analyzer as the default for both indexing and searching. In this case, the custom analyzer is similar to the Standard analyzer, except it adds the KStem token filter.
- This is an API call, so there's no warning prompt of any kind. Make sure you type the name of the index correctly, and don't test on a production instance of elasticsearch!

```
curl -XPUT http://localhost:9200/test -d '{
  "settings":{
    "analysis":{
      "analyzer":{
        "default":{
          "type":"custom",
          "tokenizer":"standard",
          "filter":[ "standard", "lowercase", "stop", "kstem" ]
        }
      }
    }
  }
}'
```

Testing Analyzers

- Normally, analysis is an opaque process — text is analyzed and immediately stored as terms in the index or used in a query. Fortunately, elasticsearch provides a helpful endpoint that allows you to run some sample text through an analyzer and inspect the resulting tokens.
- Let's try analyzing a word with a suffix to make sure KStem is correctly configured. (The pretty parameter just tells elasticsearch to format the JSON in a human-readable way. Don't use this parameter in production.)

```
curl 'http://localhost:9200/test/_analyze?text=Searched&pretty'
{
  "tokens" : [ {
    "token" : "search",
    "start_offset" : 0,
    "end_offset" : 8,
    "type" : "<ALPHANUM>",
    "position" : 1
  } ]
}
```

Specifying Analyzers

- You can also specify a specific analyzer to use with the analyzer parameter. This allows you to configure and test a number of different analyzers at once. Just give each one a different name (besides default) in the configuration and reference them in the analyzer parameter.
- The standard analyzer is one of the built-in ones, so let's see what it does with the same text:

```
curl 'http://localhost:9200/test/_analyze?analyzer=standard&text=Searched&pretty'
{
  "tokens" : [ {
    "token" : "searched",
    "start_offset" : 0,
    "end_offset" : 8,
    "type" : "<ALPHANUM>",
    "position" : 1
  } ]
}
```

Specifying Analyzers Part 2

- In the following example, we create a new analyzer called the es_std analyzer, which uses the predefined list of Spanish stopwords:

```
PUT /spanish_docs
{
    "settings": {
        "analysis": {
            "analyzer": {
                "es_std": {
                    "type":      "standard",
                    "stopwords": "_spanish_"
                }
            }
        }
    }
}
```

- The es_std analyzer is not global—it exists only in the spanish_docs index where we have defined it. To test it with the analyze API, we must specify the index name:

```
GET /spanish_docs/_analyze?analyzer=es_std
El veloz zorro marrón
```

```
{
  "tokens" : [
    { "token" :    "veloz",   "position" : 2 },
    { "token" :    "zorro",   "position" : 3 },
    { "token" :    "marrón",  "position" : 4 }
  ]
}
```

Summary

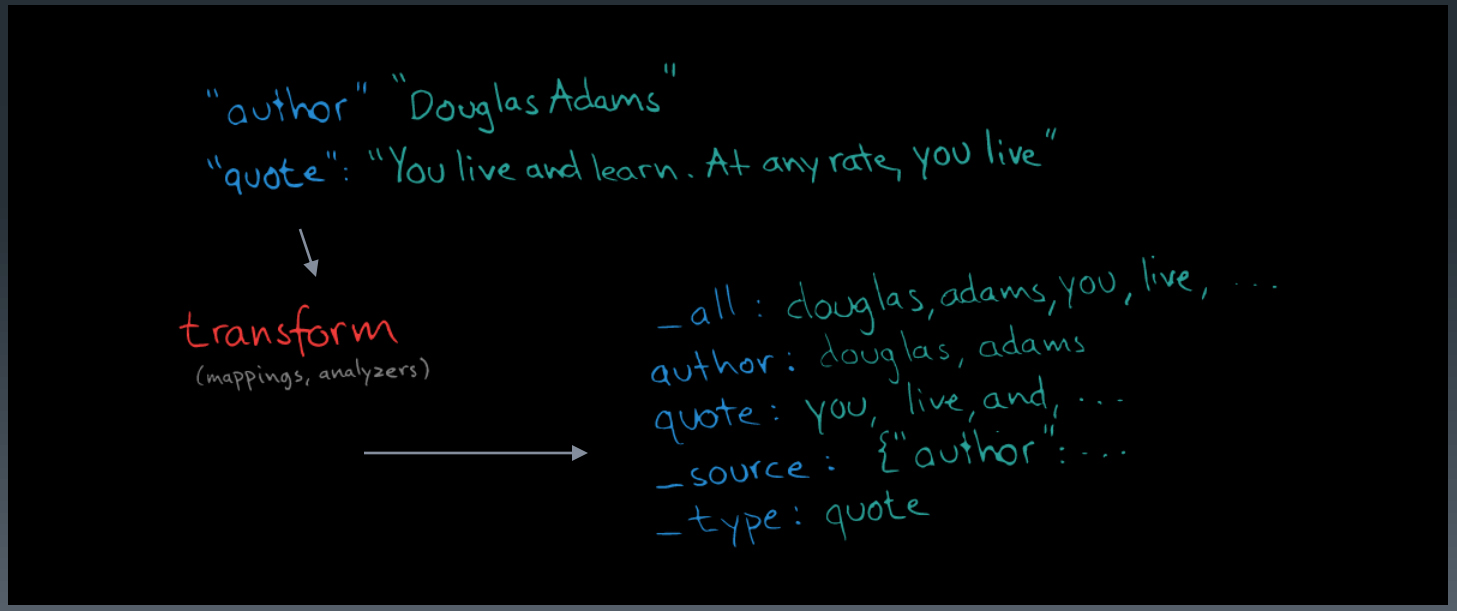
- Deep Dive into the three components of analysis: character filters, tokenization, and token filtering
- The different built-in analyzers: standard, stop, simple, keyword, and more
- Analyzers can be tested by specifying them
- You can create custom analyzers with different combinations of token filters, char filters, and tokenizers.

Lab 5 - Analyzers and Analysis lab

Day 2 Agenda

- Mappings
- Deeper Search
- Suggesters
- Aggregations and Modeling
- Bringing it All Together

Module 6: Mappings



Objectives

- What are Mappings
- Viewing Mappings
- CRUD for Mappings
- Dynamic Mappings
- Testings for Mappings

What are Mappings?

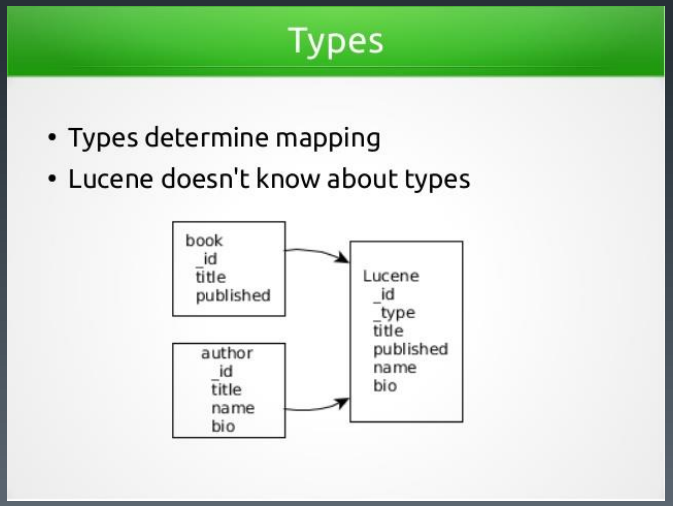
- Mapping is the process of defining how a document, and the fields it contains, are stored and indexed. For instance, use mappings to define:
  - which string fields should be treated as full text fields.
  - which fields contain numbers, dates, or geolocations.
  - whether the values of all fields in the document should be indexed into the catch-all _all field.
  - the format of date values.
  - custom rules to control the mapping for dynamically added fields.

Mapping Types

- Each index has one or more mapping types, which are used to divide the documents in an index into logical groups. User documents might be stored in a user type, and blog posts in a blogpost type.

- Each mapping type has:

  - Meta-fields
  - Meta-fields are used to customize how a document's metadata associated is treated. Examples of meta-fields include the document's _index, _type, _id, and _source fields.
  - Fields or properties
  - Each mapping type contains a list of fields or properties pertinent to that type. A user type might contain title, name, and age fields, while a blogpost type might contain title, body, user_id and created fields. Fields with the same name in different mapping types in the same index must have the same mapping.



Mapping Types

- Mapping types are used to group fields, but the fields in each mapping type are not independent of each other. Fields with:
  - the same name
  - in the same index
  - in different mapping types
  - map to the same field internally,
  - and must have the same mapping.
- If a title field exists in both the user and blogpost mapping types, the title fields must have exactly the same mapping in each type. The only exceptions to this rule are the copy_to, dynamic, enabled, ignore_above, include_in_all, and properties parameters, which may have different settings per field.
- Usually, fields with the same name also contain the same type of data, so having the same mapping is not a problem. When conflicts do arise, these can be solved by choosing more descriptive names, such as user_title and blog_title.

```
PUT my_index ❶
{
  "mappings": {
    "user": { ❷
      "_all":       { "enabled": false  }, ❸
      "properties": { ❹
        "title":    { "type": "string"  }, ❺
        "name":     { "type": "string"  }, ❻
        "age":      { "type": "integer" }  ❼
      }
    },
    "blogpost": { ❽
      "properties": { ❾
        "title":    { "type": "string"  }, ❿
        "body":     { "type": "string"  }, ⓫
        "user_id":  {
          "type":   "string", ⓬
          "index":  "not_analyzed"
        },
        "created":  {
          "type":   "date", ⓭
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
  }
}
```

Type Determining

- Elasticsearch can try guessing the schema for our documents by looking at the JSON that the document is built from. Because JSON is structured, that seems easy to do. For example, strings are surrounded by quotation marks, Booleans are defined using specific words, and numbers are just a few digits.

```
{
  "field1": 10,
  "field2": "10"
}
```

```
curl -XPUT 'localhost:9200/sites' -d '{
  "index.mapper.dynamic": false
}'
```

/Users/dennis/Projects/blog/elasticsearch/curl.json

```
curl -XPUT localhost:9200/documents/blog/one -d'
{
    "title": "Introduction To Elasticsearch",
    "author": {
        "born": "1978-04-28",
        "name": "Dennis Probst"
    },
    "date": "2014-02-05",
    "location" : "blog.codecentric.de",
    "language" : "english",
    "text" : "Case read they must it of cold that. Speaking trifling an to unpac
ked moderate debating learning. An particular contrasted he excellence favourabl
e on. Nay preference dispatched difficulty continuing joy one. Songs it be if ou
ght hoped of. Too carriage attended him entrance desirous the saw. Twenty sister
 hearts garden limits put gay has. We hill lady will both sang room by. Desirous
 men exercise overcame procured speaking her followed.",
    "word_count" : 188
}'
```

U:**- curl.json All (15,0) (JSON Fill)

Tuning Type Determination

- let's assume that we want to create an index called users and we want it to have the user type on which we will want more aggressive numeric fields parsing. To do that, we will use the following command:

```
curl -XPUT http://localhost:9200/users/?pretty -d '{
  "mappings" : {
    "user": {
      "numeric_detection" : true
    }
  }
}'
```

```
curl -XPOST http://localhost:9200/users/user/1 -d '{"name": "User 1", "age": "20"}'
```

With the default settings, the age field would be set to string type. With the numeric_detection property set to true, the type of the age field will be set to long.

```
{
  "users" : {
    "mappings" : {
      "user" : {
        "numeric_detection" : true,
        "properties" : {
          "age" : {
            "type" : "long"
          },
          "name" : {
            "type" : "string"
          }
        }
      }
    }
  }
}
```

Tuning Type Determination

- Another type of data that causes trouble are fields with dates. Dates can come in different flavors, for example, 2015-10-01 11:22:33 is a proper date and so is 2015-10-01T11:22:33+00. Because of that, Elasticsearch tries to match the fields to timestamps or strings that match some given date format. If that matching operation is successful, the field is treated as a date based one. If we know how our date fields look, we can help Elasticsearch by providing a list of recognized date formats using the dynamic_date_formats property, which allows us to specify the formats array. Let's look at the following command for creating an index:

```
curl -XPUT 'http://localhost:9200/blog/' -d '{
  "mappings" : {
    "article" : {
      "dynamic_date_formats" : ["yyyy-MM-dd hh:mm"]
    }
  }
}'
```

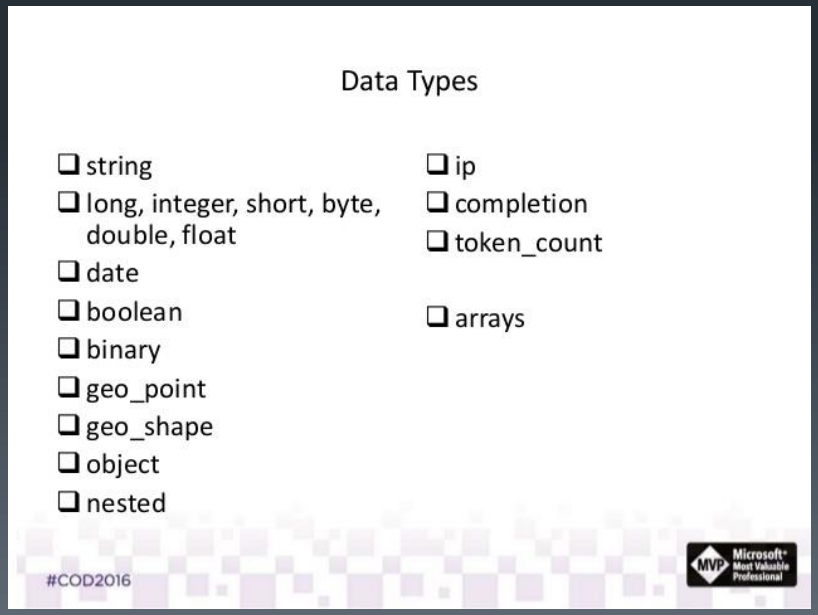
```
{
  "blog" : {
    "mappings" : {
      "article" : {
        "dynamic_date_formats" : [ "yyyy-MM-dd hh:mm" ],
        "properties" : {
          "name" : {
            "type" : "string"
          },
          "test_field" : {
            "type" : "date",
            "format" : "yyyy-MM-dd hh:mm"
          }
        }
      }
    }
  }
}
```

NOTE

Remember that the `dynamic_date_format` property accepts an array of values. That means that we can handle several date formats simultaneously.

Field Datatypes

- Each field has a data type which can be:
  - a simple type like string, date, long, double, boolean or ip.
  - a type which supports the hierarchical nature of JSON such as object or nested.
  - or a specialized type like geo_point, geo_shape, or completion.
- It is often useful to index the same field in different ways for different purposes. For instance, a string field could be indexed as an analyzed field for full-text search, and as a not_analyzed field for sorting or aggregations. Alternatively, you could index a string field with the standard analyzer, the english analyzer, and the french analyzer.
- This is the purpose of multi-fields. Most datatypes support multi-fields via the fields parameter.



Attributes for field types 1

- index_name: This attribute defines the name of the field that will be stored in the index. If this is not defined, the name will be set to the name of the object that the field is defined with. Usually, you don't need to set this property, but it may be useful in some cases; for example, when you don't have control over the name of the fields in the JSON documents that are sent to Elasticsearch.
- index: This attribute can take the values analyzed and no and, for string-based fields, it can also be set to the additional not_analyzed value. If set to analyzed, the field will be indexed and thus searchable. If set to no, you won't be able to search on such a field. The default value is analyzed. In case of string-based fields, there is an additional option, not_analyzed. This, when set, will mean that the field will be indexed but not analyzed. So, the field is written in the index as it was sent to Elasticsearch and only a perfect match will be counted during a search – the query will have to include exactly the same value as the value in the index. If we compare it to the SQL databases world, setting the index property of a field to not_analyzed would work just like using where field = value. Also remember that setting the index property to no will result in the disabling inclusion of that field in include_in_all (the include_in_all property is discussed as the last property in the list).

Attributes for field types 2

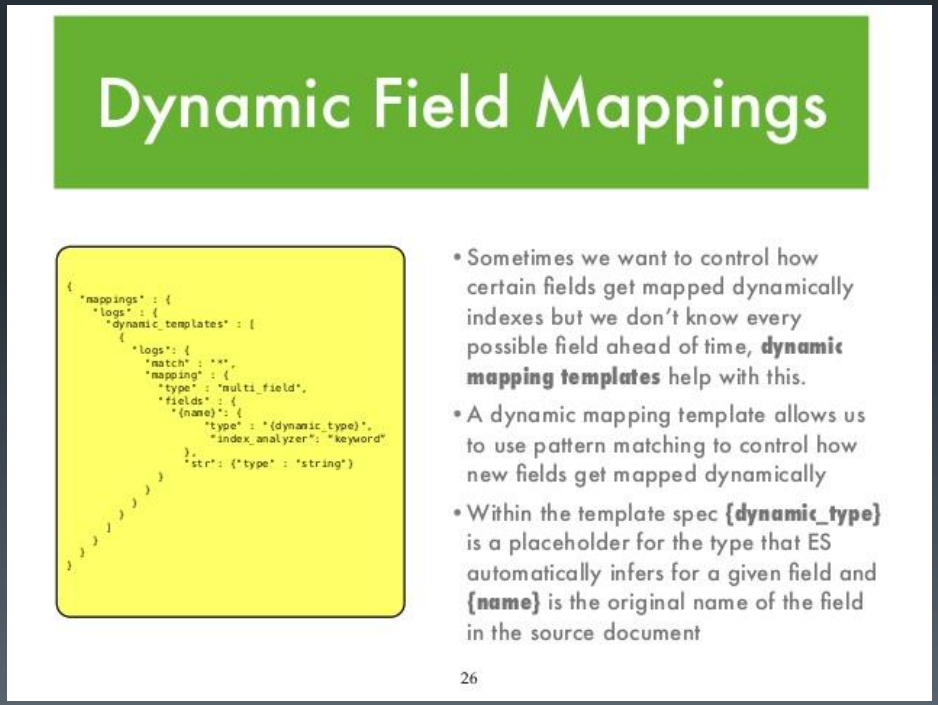
- store: This attribute can take the values yes and no and specifies if the original value of the field should be written into the index. The default value is no, which means that Elasticsearch won't store the original value of the field and will try to use the _source field (the JSON representing the original document that has been sent to Elasticsearch) when you want to retrieve the field value. Stored fields are not used for searching, however they can be used for highlighting if enabled (which may be more efficient that loading the _source field in case it is big).
- doc_values: This attribute can take the values of true and false. When set to true, Elasticsearch will create a special on disk structure during indexation for not tokenized fields (like not analyzed string fields, number based fields, Boolean fields, and date fields). This structure is highly efficient and is used by Elasticsearch for operations that require un-inverted data, such as aggregations, sorting, or scripting. Starting with Elasticsearch 2.0 the default value of this is true for not tokenized fields. Setting this value to false will result in Elasticsearch using field data cache instead of doc values, which has higher memory demand, but may be faster in some rare situations.

Attributes for field types 3

- boost: This attribute defines how important the field is inside the document; the higher the boost, the more important the values in the field are. The default value of this attribute is 1, which means a neutral value – anything above 1 will make the field more important, anything less than 1 will make it less important.
- null_value: This attribute specifies a value that should be written into the index in case that field is not a part of an indexed document. The default behavior will just omit that field.
- copy_to: This attribute specifies an array of fields to which the original value will be copied to. This allows for different kind of analysis of the same data. For example, you could imagine having two fields – one called title and one called title_sort, each having the same value but processed differently. We could use copy_to to copy the title field value to title_sort.
- include_in_all: This attribute specifies if the field should be included in the _all field. The _all field is a special field used by Elasticsearch to allow easy searching in the contents of the whole indexed document. Elasticsearch creates the content of the _all field by copying all the document fields there. By default, if the _all field is used, all the fields will be included in it.

Dynamic Mappings

- Fields and mapping types do not need to be defined before being used. Thanks to dynamic mapping, new mapping types and new field names will be added automatically, just by indexing a document. New fields can be added both to the top-level mapping type, and to inner object and nested fields.



Explicit Mappings

- You know more about your data than Elasticsearch can guess, so while dynamic mapping can be useful to get started, at some point you will want to specify your own explicit mappings.
- You can create mapping types and field mappings when you create an index, and you can add mapping types and fields to an existing index with the PUT mapping API.

Creates an index called twitter with the message field in the tweet mapping type.

Uses the PUT mapping API to add a new mapping type called user.

Uses the PUT mapping API to add a new field called user_name to the tweet mapping type.

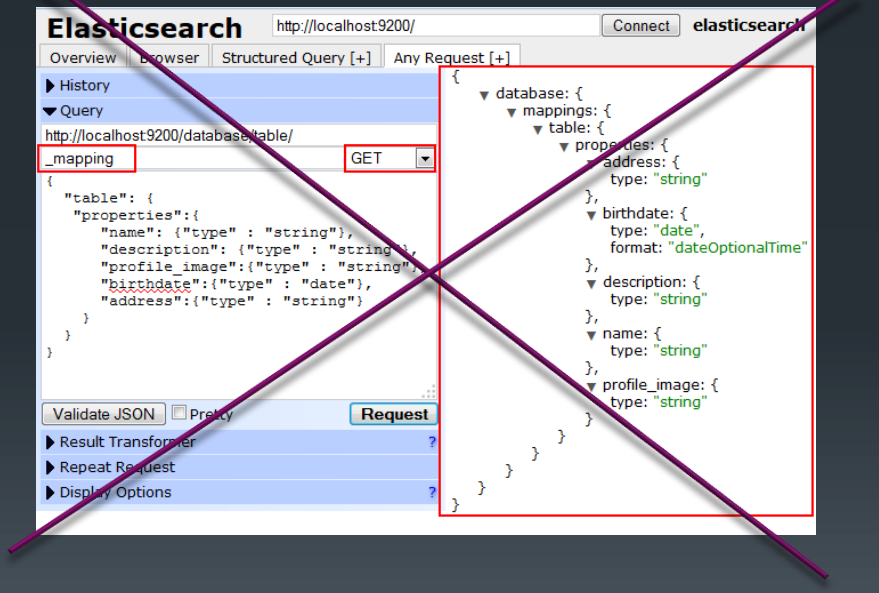
```
PUT twitter ❶
{
  "mappings": {
    "tweet": {
      "properties": {
        "message": {
          "type": "string"
        }
      }
    }
  }
}

PUT twitter/_mapping/user ❷
{
  "properties": {
    "name": {
      "type": "string"
    }
  }
}

PUT twitter/_mapping/tweet ❸
{
  "properties": {
    "user_name": {
      "type": "string"
    }
  }
}
```

Updating Mappings

- Other than where documented, existing type and field mappings cannot be updated. Changing the mapping would mean invalidating already indexed documents. Instead, you should create a new index with the correct mappings and reindex your data into that index.



Creates an index called twitter with the message field in the tweet mapping type.

Uses the PUT mapping API to add a new mapping type called user.

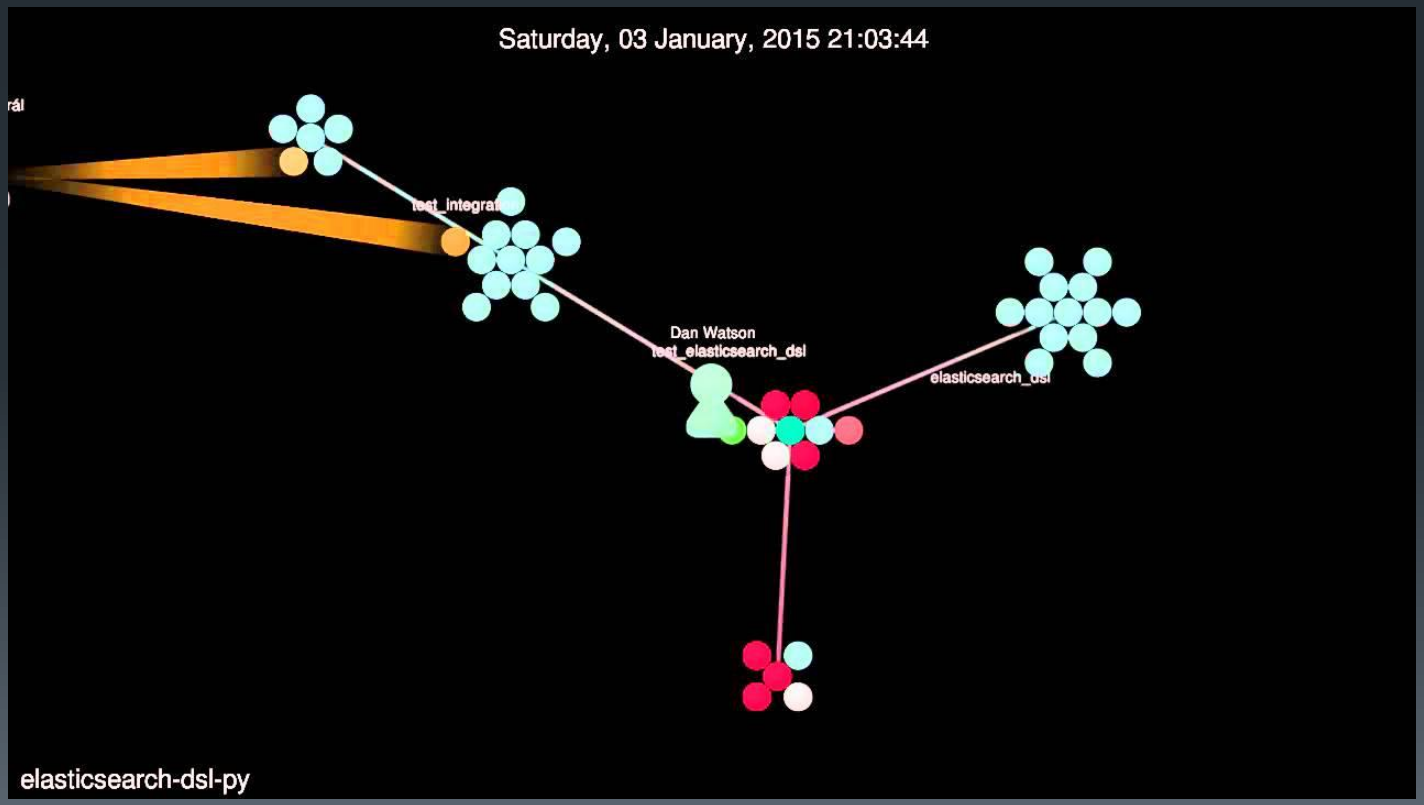
Uses the PUT mapping API to add a new field called user_name to the tweet mapping type.

Summary

- Index, documents, fields, and mappings make up core data components of ElasticSearch
- The roles of ES servers are either data, master, or client nodes.
- Clusters are collections of nodes
- Shards are data segments dedicated to one or more nodes.
- Indexing is a way of grouping key terms to enable search

Lab 5 - A deeper dive on Mappings

Module 7: Deeper Search



Objectives

- Full Text Query
  - Match_all
  - Multi_match
  - Common_terms
  - Query_string
  - Simple query string
- Other Compound Queries

What is DSL Query?

- The query DSL is a flexible, expressive search language that Elasticsearch uses to expose most of the power of Lucene through a simple JSON interface. It is what you should be using to write your queries in production. It makes your queries more flexible, more precise, easier to read, and easier to debug.
- We saw this prior in reviewing basic search.
- To use the Query DSL, pass a query in the query parameter:

```
GET /_search
{
    "query": YOUR_QUERY_HERE
}
```

Structure of Query

- A query clause typically has this structure:

```
{
    QUERY_NAME: {
        ARGUMENT: VALUE,
        ARGUMENT: VALUE,...
    }
}
```

- If it references one particular field, it has this structure:

```
{
    QUERY_NAME: {
        FIELD_NAME: {
            ARGUMENT: VALUE,
            ARGUMENT: VALUE,...
        }
    }
}
```

Match_all Query

- The empty search—{}—is functionally equivalent to using the match_all query clause, which, as the name suggests, matches all documents:

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

Match Query 1

- The empty search—{}—is functionally equivalent to using the match_all query clause, which, as the name suggests, matches all documents:

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

Match Query

- The match query should be the standard query that you reach for whenever you want to query for a full-text or exact value in almost any field.

- If you run a match query against a full-text field, it will analyze the query string by using the correct analyzer for that field before executing the search:

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

If you use it on a field containing an exact value, such as a number, a date, a Boolean, or a not_analyzed string field, then it will search for that exact value:

```
{ "match": { "age":    26           }}
{ "match": { "date":   "2014-09-01" }}
{ "match": { "public": true         }}
{ "match": { "tag":    "full_text"  }}
```

Multi_Match Query

- The multi_match query allows to run the same match query on multiple fields:

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

```
{
  "query" : {
    "multi_match" : {
      "query" : "crime punishment",
      "fields" : [ "title^10", "otitle" ]
  }
  }
}
```

Common Terms Query - Overview

- The common terms query is a modern Elasticsearch solution for improving query relevance and precision with common words when we are not using stop words (http://en.wikipedia.org/wiki/Stop_words). For example, a crime and punishment query results in three term queries and each of them have a cost in terms of performance. However, the and term is a very common one and its impact on the document score will be very low.
  - The solution is the common terms query which divides the query into two groups. The first group is the one with important terms, which are the ones that have lower frequency. The second group is the one with less important terms, which are the ones with high frequency.
  - The first query is executed first and Elasticsearch calculates the score for all of the terms from the first group. This way the low frequency terms, which are usually the ones that have more importance, are always taken into consideration. Then Elasticsearch executes the second query for the second group of terms, but calculates the score only for the documents matched for the first query. This way the score is only calculated for the relevant documents and thus higher performance can be achieved.
  - An example of the common terms query is as follows:

```
{
 "query" : {
  "common" : {
   "title" : {
    "query" : "crime and punishment",
    "cutoff_frequency" : 0.001
   }
  }
 }
}
```

Common Terms – The Problem

- Every term in a query has a cost. A search for "The brown fox" requires three term queries, one for each of "the", "brown" and "fox", all of which are executed against all documents in the index. The query for "the" is likely to match many documents and thus has a much smaller impact on relevance than the other two terms.

- Previously, the solution to this problem was to ignore terms with high frequency. By treating "the" as a stopword, we reduce the index size and reduce the number of term queries that need to be executed.

- The problem with this approach is that, while stopwords have a small impact on relevance, they are still important. If we remove stopwords, we lose precision, (eg we are unable to distinguish between "happy" and "not happy") and we lose recall (eg text like "The The" or "To be or not to be" would simply not exist in the index).

Common Terms – The Solution 1

- The common terms query divides the query terms into two groups: more important (ie low frequency terms) and less important (ie high frequency terms which would previously have been stopwords).

- First it searches for documents which match the more important terms. These are the terms which appear in fewer documents and have a greater impact on relevance.

- Then, it executes a second query for the less important terms — terms which appear frequently and have a low impact on relevance. But instead of calculating the relevance score for all matching documents, it only calculates the _score for documents already matched by the first query. In this way the high frequency terms can improve the relevance calculation without paying the cost of poor performance.

Common Terms – The Solution 2

- If a query consists only of high frequency terms, then a single query is executed as an AND (conjunction) query, in other words all terms are required. Even though each individual term will match many documents, the combination of terms narrows down the resultset to only the most relevant. The single query can also be executed as an OR with a specific minimum_should_match, in this case a high enough value should probably be used.

- Terms are allocated to the high or low frequency groups based on the cutoff_frequency, which can be specified as an absolute frequency (>=1) or as a relative frequency (0.0 .. 1.0). (Remember that document frequencies are computed on a per shard level.)

- Perhaps the most interesting property of this query is that it adapts to domain specific stopwords automatically. For example, on a video hosting site, common terms like "clip" or "video" will automatically behave as stopwords without the need to maintain a manual list.

Common Terms – Examples 1

- In this example, words that have a document frequency greater than 0.1% (eg "this" and "is") will be treated as common terms.

```
{
  "common": {
    "body": {
      "query": "this is bonsai cool",
      "cutoff_frequency": 0.001
    }
  }
}
```

The number of terms which should match can be controlled with the minimum_should_match (high_freq, low_freq), low_freq_operator (default "or") and high_freq_operator (default "or") parameters.

For low frequency terms, set the low_freq_operator to "and" to make all terms required:

```
{
  "common": {
    "body": {
      "query": "nelly the elephant as a cartoon",
      "cutoff_frequency": 0.001,
      "low_freq_operator": "and"
    }
  }
}
```

Common Terms – Examples 2

- which is roughly equivalent to:

```
{
  "bool": {
    "must": [
      { "term": { "body": "nelly"}},
      { "term": { "body": "elephant"}},
      { "term": { "body": "cartoon"}}
    ],
    "should": [
      { "term": { "body": "the"}}
      { "term": { "body": "as"}}
      { "term": { "body": "a"}}
    ]
  }
}
```

Query_string and simple_query_string

- The query_string and simple_query_string queries query the _all field by default, unless another field is specified:

```
GET _search
{
  "query": {
    "query_string": {
      "query": "john smith 1970"
    }
  }
}
```

The same goes for the ?q= parameter in URI search requests (which is rewritten to a query_string query internally):

```
GET _search?q=john+smith+1970
```

Other queries, such as the match and term queries require you to specify the _all field explicitly.

_all field

- The _all field is a special catch-all field which concatenates the values of all of the other fields into one big string, using space as a delimiter, which is then analyzed and indexed, but not stored. This means that it can be searched, but not retrieved.

- The _all field allows you to search for values in documents without knowing which field contains the value. This makes it a useful option when getting started with a new dataset. For instance:

The _all field will contain the terms: ["john", "smith", "1970", "10", "24"]

```
PUT my_index/user/1 ❶
{
  "first_name":    "John",
  "last_name":     "Smith",
  "date_of_birth": "1970-10-24"
}

GET my_index/_search
{
  "query": {
    "match": {
      "_all": "john smith 1970"
    }
  }
}
```

All values treated as strings

The date_of_birth field in the above example is recognised as a date field and so will index a single term representing 1970-10-24 00:00:00 UTC. The _all field, however, treats all values as strings, so the date value is indexed as the three string terms: "1970", "24", "10".

It is important to note that the _all field combines the original values from each field as a string. It does not combine the terms from each field.

Query_string Query 1

- A query that uses a query parser in order to parse its content. Here is an example:

```
{
    "query_string" : {
        "default_field" : "content",
        "query" : "this AND that OR thus"
    }
}
```

Query_string Query 2

- The query_string top level parameters include:

Parameter	Description
`query`	The actual query to be parsed. See Query string syntax.
`default_field`	The default field for query terms if no prefix field is specified. Defaults to the `index.query.default_field` index settings, which in turn defaults to `_all`.
`default_operator`	The default operator used if no explicit operator is specified. For example, with a default operator of `OR`, the query `capital of Hungary` is translated to `capital OR of OR Hungary`, and with default operator of `AND`, the same query is translated to `capital AND of AND Hungary`. The default value is `OR`.
`analyzer`	The analyzer name used to analyze the query string.
`allow_leading_wildcard`	When set, `*` or `?` are allowed as the first character. Defaults to `true`.
`lowercase_expanded_terms`	Whether terms of wildcard, prefix, fuzzy, and range queries are to be automatically lower-cased or not (since they are not analyzed). Defaults to `true`.
`enable_position_increments`	Set to `true` to enable position increments in result queries. Defaults to `true`.
`fuzzy_max_expansions`	Controls the number of terms fuzzy queries will expand to. Defaults to `50`
`fuzziness`	Set the fuzziness for fuzzy queries. Defaults to `AUTO`. See the section called "Fuzziness" for allowed settings.
`fuzzy_prefix_length`	Set the prefix length for fuzzy queries. Default is `0`.
`phrase_slop`	Sets the default slop for phrases. If zero, then exact phrase matches are required. Default value is `0`.
`boost`	Sets the boost value of the query. Defaults to `1.0`.

`analyze_wildcard`	By default, wildcards terms in a query string are not analyzed. By setting this value to `true`, a best effort will be made to analyze those as well.
`auto_generate_phrase_queries`	Defaults to `false`.
`max_determinized_states`	Limit on how many automaton states regexp queries are allowed to create. This protects against too-difficult (e.g. exponentially hard) regexps. Defaults to 10000.
`minimum_should_match`	A value controlling how many "should" clauses in the resulting boolean query should match. It can be an absolute value (`2`), a percentage (`30%`) or a combination of both.
`lenient`	If set to `true` will cause format based failures (like providing text to a numeric field) to be ignored.
`locale`	Locale that should be used for string conversions. Defaults to `ROOT`.
`time_zone`	Time Zone to be applied to any range query related to dates. See also JODA timezone.

Simple Query String Query

- A query that uses the SimpleQueryParser to parse its context. Unlike the regular query_string query, the simple_query_string query will never throw an exception, and discards invalid parts of the query. Here is an example:

```
{
    "simple_query_string" : {
        "query": "\"fried eggs\" +(eggplant | potato) -frittata",
        "analyzer": "snowball",
        "fields": ["body^5","_all"],
        "default_operator": "and"
    }
}
```

Simple Query Parameters 1

- The simple_query_string top level parameters include:

Parameter	Description
`query`	The actual query to be parsed. See below for syntax.
`fields`	The fields to perform the parsed query against. Defaults to the `index.query.default_field` index settings, which in turn defaults to `_all`.
`default_operator`	The default operator used if no explicit operator is specified. For example, with a default operator of `OR`, the query `capital of Hungary` is translated to `capital OR of OR Hungary`, and with default operator of `AND`, the same query is translated to `capital AND of AND Hungary`. The default value is `OR`.
`analyzer`	The analyzer used to analyze each term of the query when creating composite queries.
`flags`	Flags specifying which features of the `simple_query_string` to enable. Defaults to `ALL`.

Simple Query Parameters 2

- The simple_query_string top level parameters include:

`lowercase_expanded_terms`	Whether terms of prefix and fuzzy queries should be automatically lower-cased or not (since they are not analyzed). Defaults to `true`.
`analyze_wildcard`	Whether terms of prefix queries should be automatically analyzed or not. If `true` a best effort will be made to analyze the prefix. However, some analyzers will be not able to provide a meaningful results based just on the prefix of a term. Defaults to `false`.
`locale`	Locale that should be used for string conversions. Defaults to `ROOT`.
`lenient`	If set to `true` will cause format based failures (like providing text to a numeric field) to be ignored.
`minimum_should_match`	The minimum number of clauses that must match for a document to be returned. See the `minimum_should_match` documentation for the full list of options.

Simple Query String Syntax

- The simple_query_string supports the following special characters:
  - + signifies AND operation
  - | signifies OR operation
  - - negates a single token
  - " wraps a number of tokens to signify a phrase for searching
  - * at the end of a term signifies a prefix query
  - (and) signify precedence
  - ~N after a word signifies edit distance (fuzziness)
  - ~N after a phrase signifies slop amount
  - In order to search for any of these special characters, they will need to be escaped with \.

Combining Multiple Clauses

- Query clauses are simple building blocks that can be combined with each other to create complex queries. Clauses can be as follows:
  - Leaf clauses (like the match clause) that are used to compare a field (or fields) to a query string.
  - Compound clauses that are used to combine other query clauses. For instance, a bool clause allows you to combine other clauses that either must match, must_not match, or should match if possible. They can also include non-scoring, filters for structured search:

```
{
    "bool": {
        "must":     { "match": { "tweet": "elasticsearch" }},
        "must_not": { "match": { "name":  "mary" }},
        "should":   { "match": { "tweet": "full text" }},
        "filter":   { "range": { "age" : { "gt" : 30 }} }
    }
}
```

Compound Queries – Bool(ean)

- The bool query
  - The bool query allows us to wrap a virtually unbounded number of queries and connect them with a logical value using one of the following sections:
  - should: The query wrapped into this section may or may not match. The number of should sections that have to match is controlled by the minimum_should_match parameter
  - must: The query wrapped into this section must match in order for the document to be returned.
  - must_not: The query when wrapped into this section must not match in order for the document to be returned.

```
{
  "query" : {
    "bool" : {
      "must" : {
        "term" : {
          "title" : "crime"
        }
      },
      "should" : {
        "range" : {
          "year" : {
            "from" : 1900,
            "to" : 2000
          }
        }
      },
      "must_not" : {
        "term" : {
          "otitle" : "nothing"
        }
      }
    }
  }
}
```

Compound Queries – dis_max

- The dis_max query is very useful as it generates a union of documents returned by all the sub queries and returns it as the result. The good thing about this query is the fact that we can control how the lower scoring sub queries affect the final score of the documents. For the dis_max query, we specify the queries using the queries property (query or an array of queries) and the tie breaker, with the tie_breaker property. We can also include additional boost by specifying the boost parameter.
- The final document score is calculated as the sum of scores of the maximum scoring query and the sum of scores returned from the rest of the queries, multiplied by the value of the tie parameter. So, the tie_breaker parameter allows us to control how the lower scoring queries affect the final score. If we set the tie_breaker parameter to 1.0, we get the exact sum, while setting the tie parameter to 0.1 results in only 10 percent of the scores (of all the scores apart from the maximum scoring query) being added to the final score.
- An example of the dis_max query is as follows:

```
{
  "query" : {
    "dis_max" : {
      "tie_breaker" : 0.99,
      "boost" : 10.0,
      "queries" : [
        {
          "match" : {
            "title" : "crime"
          }
        },
        {
          "match" : {
            "author" : "fyodor"
          }
        }
      ]
    }
  }
}
```

Compound Queries – constant_score

- The constant_score query wraps another query and returns a constant score for each document returned by the wrapped query. We specify the score that should be given to the documents by using the boost property, which defaults to 1.0. It allows us to strictly control the score value assigned for a document matched by a query. For example, if we want to have a score of 2.0 for all the documents that have the term crime in the title field, we send the following query to Elasticsearch:

```
{
  "query" : {
    "constant_score" : {
      "query" : {
        "term" : {
          "title" : "crime"
        }
      },
      "boost" : 2.0
    }
  }
}
```

Compound Queries – indices

- The indices query is useful when executing a query against multiple indices. It allows us to provide an array of indices (the indices property) and two queries, one that will be executed if we query the index from the list (the query property) and the second that will be executed on all the other indices (the no_match_query property). For example, assume we have an alias named books, holding two indices: library and users. What we want to do is use this alias. However, we want to run different queries depending on which index is used for searching. An example query following this logic will look as follows:

```
{
  "query" : {
    "indices" : {
      "indices" : [ "library" ],
      "query" : {
        "term" : {
          "title" : "crime"
        }
      },
      "no_match_query" : {
        "term" : {
          "user" : "crime"
        }
      }
    }
  }
}
```

Compound Queries – boosting

- The boosting query wraps around two queries and lowers the score of the documents returned by one of the queries. There are three sections of the boosting query that need to be defined: the positive section that holds the query whose document score will be left unchanged, the negative section whose resulting documents will have their score lowered, and the negative_boost section that holds the boost value that will be used to lower the second section's query score. The advantage of the boosting query is that the results of both the queries (the negative and the positive ones) will be present in the results, although the scores of some queries will be lowered. For comparison, if we were to use the bool query with the must_not section, we wouldn't get the results for such a query.
- Let's assume that we want to have the results of a simple term query for the term crime in the title field and want the score of such documents to not be changed. However, we also want to have the documents that range from 1800 to 1900 in the year field, and the scores of documents returned by such a query to have an additional boost of 0.5. Such a query will look like the following:

```
{
  "query" : {
    "boosting" : {
      "positive" : {
        "term" : {
          "title" : "crime"
        }
      },
      "negative" : {
        "range" : {
          "year" : {
            "from" : 1800,
            "to" : 1900
          }
        }
      },
      "negative_boost" : 0.5
    }
  }
}
```

Span Queries

Elasticsearch leverages Lucene span queries, which allow us to make queries when some tokens or phrases are near other tokens or phrases. Basically, we can call them position aware queries. When using the standard non span queries, we are not able to make queries that are position aware; to some extent, the phrase queries allow that, but only to some extent. So, for Elasticsearch and the underlying Lucene, it doesn't matter if the term is in the beginning of the sentence or at the end or near another term. When using span queries, it does matter.

Know that they exist and that they are expensive (compute) to use.

```
curl -XPUT 'localhost:9200/spans/book/1' -d '{
"title" : "Test book", "author" : "Test author",
"description" : "The world breaks everyone, and
afterward, some are strong at the broken
places"
```

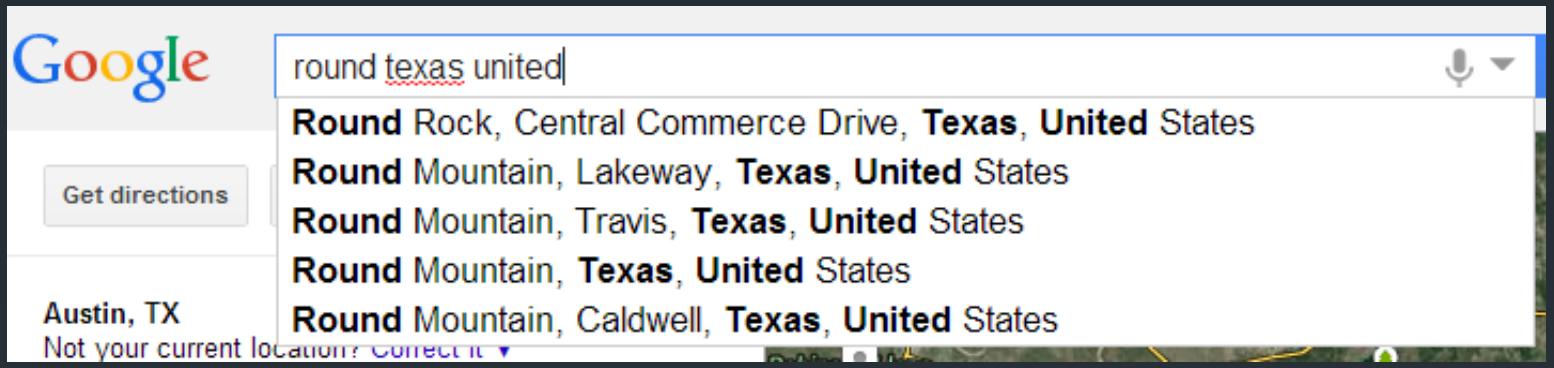
- span term query
- span first query
- span near query
- span or query
- span not query
- span within query
- span containing query
- span multi query

Summary

- Review of match, match_all queries
- Discussed _all field and its use in query_string and simple_query_string
- Explored simple query string parameters
- Covered various types of compound queries
- Mentioned the power of Span queries

Lab 7 – Deeper Search

Module 8: Suggesters



Objectives

- What are Suggesters?
- Term Suggesters
- Phrase Suggesters
- Completion Suggesters
- Context Suggesters

What are Suggesters?


```
curl -s -XPOST 'localhost:9200/_search' -d '{
  "query" : {
    ...
  },
  "suggest" : {
    ...
  }
}'
```

A long time ago, starting from Elasticsearch 0.90 (which was released on April 29, 2013), we got the ability to use so-called suggesters. We can define a suggester as a functionality allowing us to correct the user's spelling mistakes and build autocomplete functionality keeping performance in mind. This section is dedicated to these functionalities and will help you learn about them. We will discuss each available suggester type and show the most common properties that allow us to control them. The suggest feature suggests similar looking terms based on a provided text by using a suggester. Parts of the suggest feature are still under development. The suggest request part is either defined alongside the query part in a _search request or via the REST _suggest endpoint.

Available Suggestor Types

- These have changed since the initial introduction of the Suggest API to Elasticsearch. We are now able to use four type of suggesters:
  - **term**: A suggester returning corrections for each word passed to it. Useful for suggestions that are not phrases, such as single term queries.
  - **phrase**: A suggester working on phrases, returning a proper phrase.
  - **completion**: A suggester designed to provide fast and efficient autocomplete results.
  - **context**: Extension to the Suggest API of Elasticsearch. Allows us to handle parts of the suggest queries in memory and thus very effective in terms of performance.

Including Suggestions

- Let's now try getting suggestions along with the query results. For example, let's use a match_all query and try getting a suggestion for a serlock holnes phrase, which has two terms spelled incorrectly. To do this, we run the following command:

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
 "query" : {
  "match_all" : {}
 },
 "suggest" : {
  "first_suggestion" : {
   "text" : "serlock holnes",
   "term" : {
    "field" : "_all"
   }
  }
 }
}'
```

As you can see, we've introduced a new section to our query – the suggest one. We've specified the text we want to get the correction for by using the text property. We've specified the suggester we want to use (the term one) and configured it specifying the name of the field that should be used for building suggestions using the field property. first_suggestion is the name we give to our suggester; we need to do this because there can be multiple ones used. This is how you send a request for suggestion in general.

Suggesters Response

- Now let's look at the response of the first query we sent. As you can guess, the response includes both the query results and the suggestions:

- The **options** array contains suggestions for the given word and will be empty if Elasticsearch doesn't find any suggestions. Each entry in this array is a suggestion and described by the following properties:
  - text: Text of the suggestion.
  - score: Suggestion score; the higher the score, the better the suggestion.
  - freq: Frequency of the suggestion. The frequency represents how many times the word appears in the documents in the index we are running the suggestion query against.


```
{
  "took" : 10,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [ ... ]
  },
  "suggest" : {
    "first_suggestion" : [ {
      "text" : "serlock",
      "offset" : 0,
      "length" : 7,
      "options" : [ {
        "text" : "sherlock",
        "score" : 0.85714287,
        "freq" : 1
      } ]
    }, {
      "text" : "holnes",
      "offset" : 8,
      "length" : 6,
      "options" : [ {
        "text" : "holmes",
        "score" : 0.8333333,
        "freq" : 1
      } ]
    } ]
  }
}
```

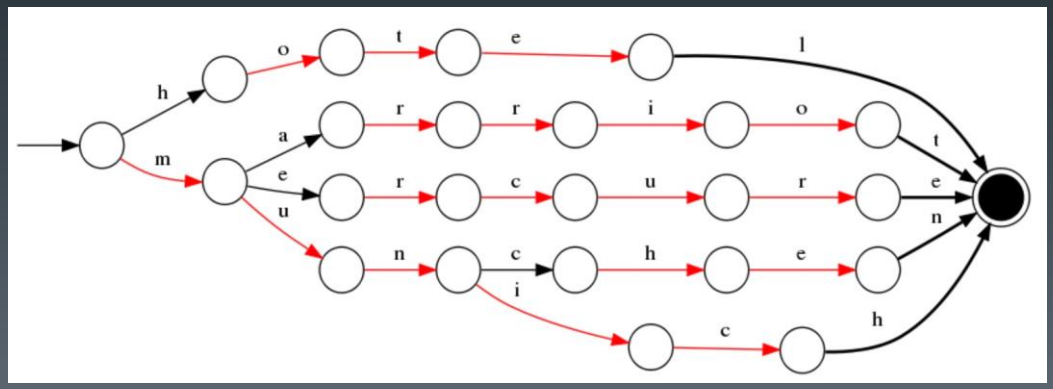
Multiple Suggestions

- If we want to get multiple suggestions for the same text, we can embed our suggestions in the suggest object and place the text property as the suggest object option. For example, if we want to get suggestions for the serlock holnes text for the title field and for the _all field, we run the following command:

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
 "query" : {
  "match_all" : {}
 },
 "suggest" : {
  "text" : "serlock holnes",
  "first_suggestion" : {
   "term" : {
    "field" : "_all"
   }
  },
  "second_suggestion" : {
   "term" : {
    "field" : "title"
   }
  }
 }
}'
```

Term Suggester

- The term suggester works on the basis of string edit distance. This means that the suggestion with the fewest characters that need to be changed, added, or removed to make the suggestion look as the original word, is the best one. For example, let's take the words worl and work. To change the worl term to work, we need to change the l letter to k, so it means a distance of 1. The text provided to the suggester is of course analyzed and then terms are chosen to be suggested.



Term Suggester configuration

- The common and most used term suggester options can be used for all the suggester implementations that are based on the term one. Currently, these are the phrase suggester and of course the base term one. The available options are:

  - `text`: The text we want to get the suggestions for. This parameter is required in order for the suggester to work.
  - `field`: Another required parameter that we need to provide. The `field` parameter allows us to set which field the suggestions should be generated for.
  - `analyzer`: The name of the analyzer which should be used to analyze the text provided in the `text` parameter. If not set, Elasticsearch utilizes the analyzer used for the field provided by the `field` parameter.
  - `size`: Defaults to `5` and specifies the maximum number of suggestions allowed to be returned by each term provided in the `text` parameter.
  - `suggest_mode`: Controls which suggestions will be included and for what terms the suggestions will be returned. The possible options are: `missing` – the default behavior, which means that the suggester will only provide suggestions for terms that are not present in the index; `popular` – means that the suggestions will only be returned when they are more frequent than the provided term; and finally `always` means that suggestions will be returned every time.
  - `sort`: Allows us to specify how the suggestions are sorted in the result returned by Elasticsearch. By default, it is set to `score`, which tells Elasticsearch that the suggestions should be sorted by the suggestion score first, the suggestion document frequency next, and finally by the term. The second possible value is `frequency`, which means that the results are first sorted by the document frequency, then by the score, and finally by the term.

Phrase Suggester

- The term suggester provides a great way to correct user spelling mistakes on per term basis, but it is not great for phrases. That's why the phrase suggester was introduced. It is built on top of the term suggester, but adds additional phrase calculation logic to it.Let's start with an example of how to use the phrase suggester. This time we will omit the query section in our query. We do that by running the following command:

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
 "suggest" : {
  "text" : "sherlock holnes",
  "our_suggestion" : {
   "phrase" : { "field" : "_all" }
  }
 }
}'
```

```
{
  "took" : 24,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [ ... ]
  },
  "suggest" : {
    "our_suggestion" : [ {
      "text" : "sherlock holnes",
      "offset" : 0,
      "length" : 15,
      "options" : [ {
        "text" : "sherlock holmes",
        "score" : 0.12227806
      } ]
    } ]
  }
}
```

Phrase Suggester

- The term suggester provides a great way to correct user spelling mistakes on per term basis, but it is not great for phrases. That's why the phrase suggester was introduced. It is built on top of the term suggester, but adds additional phrase calculation logic to it.Let's start with an example of how to use the phrase suggester. This time we will omit the query section in our query. We do that by running the following command:

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
 "suggest" : {
  "text" : "sherlock holnes",
  "our_suggestion" : {
   "phrase" : { "field" : "_all" }
  }
 }
}'
```

```
{
  "took" : 24,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [ ... ]
  },
  "suggest" : {
    "our_suggestion" : [ {
      "text" : "sherlock holnes",
      "offset" : 0,
      "length" : 15,
      "options" : [ {
        "text" : "sherlock holmes",
        "score" : 0.12227806
      } ]
    } ]
  }
}
```

Phrase Suggester Configuration

- Because the phrase suggester is based on the term suggester, it can also use some of the configuration options provided by it. Those options are: text, size, analyzer, and shard_size. In addition to the mentioned properties, the phrase suggester exposes additional options. Some of these options are:

- `max_errors`: Specifies the maximum number (or percentage) of terms that can be erroneous in order to create a correction using it. The value of this property can be either an integer number, such as `1`, or a float between `0` and `1` which will be treated as a percentage value. By default, it is set to `1`, which means that at most a single term can be misspelled in a given correction.
- `separator`: Defaults to a whitespace character and specifies the separator that will be used to divide the terms in the resulting bigram field.

Completion Suggester

- The completion suggester allows us to create autocomplete functionality in a very performance-effective way, because of storing complicated structures in the index instead of calculating them during query time. We need to prepare Elasticsearch for that by using a dedicated field type called completion. Let's assume that we want to create an autocomplete feature to allow us to show book authors. In addition to author's name we want to return the identifiers of the books she/he wrote. We start with creating the authors index by running the following command:

```
curl -XPOST 'localhost:9200/authors' -d '{
 "mappings" : {
  "author" : {
   "properties" : {
    "name" : { "type" : "string" },
    "ac" : {
     "type" : "completion",
     "payloads" : true,
    "analyzer" : "standard",
    "search_analyzer" : "standard"
    }
   }
  }
 }
}'
```

Completion Suggester - Indexing

- To index the data, we need to provide some additional information along with the ones we usually provide during indexing. Let's look at the following commands that index two documents describing the authors:

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
 "name" : "Fyodor Dostoevsky",
 "ac" : {
  "input" : [ "fyodor", "dostoevsky" ],
  "output" : "Fyodor Dostoevsky",
  "payload" : { "books" : [ "123456", "123457" ] }
 }
}'
curl -XPOST 'localhost:9200/authors/author/2' -d '{
 "name" : "Joseph Conrad",
 "ac" : {
  "input" : [ "joseph", "conrad" ],
  "output" : "Joseph Conrad",
  "payload" : { "books" : [ "121211" ] }
 }
}'
```

Completion Suggester - Querying

- To index the data, we need to provide some additional information along with the ones we usually provide during indexing. Let's look at the following commands that index two documents describing the authors:

```
curl -XGET 'localhost:9200/authors/_suggest?pretty' -d '{
 "authorsAutocomplete" : {
  "text" : "fyo",
  "completion" : {
   "field" : "ac"
  }
 }
}'
```

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 3,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 1.0,
      "payload" : {
        "books" : [ "123456", "123457" ]
      }
    } ]
  } ]
}
```

Completion Suggester – Custom Weights

- By default, the term frequency is used to determine the weight of the document returned by the prefix suggester. However, this may not be the best solution. In such cases, it is useful to define the weight of the suggestion by specifying the weight property for the field defined as completion. The weight property should be set to an integer value. The higher the weight property value, the more important the suggestion. For example, if we want to specify a weight for the first document in our example, we run the following command:

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
 "name" : "Fyodor Dostoevsky",
 "ac" : {
  "input" : [ "fyodor", "dostoevsky" ],
  "output" : "Fyodor Dostoevsky",
  "payload" : { "books" : [ "123456", "123457" ] },
  "weight" : 30
 }
}'
```

```
{
  ...
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 3,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 30.0,
      "payload":{
        "books":["123456","123457"]
      }
    } ]
  } ]
}
```

Context Suggester

- The context suggester is an extension to the suggest API of Elasticsearch. Namely the suggester system provides a very fast way of searching documents by handling these entirely in memory. But this special treatment does not allow the handling of traditional queries and filters, because those would have notable impact on the performance. So the context extension is designed to take so-called context information into account to specify a more accurate way of searching within the suggester system. Instead of using the traditional query and filter system a predefined ``context`` is configured to limit suggestions to a particular subset of suggestions. Such a context is defined by a set of context mappings which can either be a simple category or a geo location. The information used by the context suggester is configured in the type mapping with the context parameter, which lists all of the contexts that need to be specified in each document and in each suggestion request. For instance:

```
PUT services/_mapping/service
{
    "service": {
        "properties": {
            "name": {
                "type" : "string"
            },
            "tag": {
                "type" : "string"
            },
            "suggest_field": {
                "type": "completion",
                "context": {
                    "color": { ❶
                        "type": "category",
                        "path": "color_field",
                        "default": ["red", "green", "blue"]
                    },
                    "location": { ❷
                        "type": "geo",
                        "precision": "5m",
                        "neighbors": true,
                        "default": "u33"
                    }
            }
        }
    }
    }
}
```

Category Context and Mapping

- The category context allows you to specify one or more categories in the document at index time. The document will be assigned to each named category, which can then be queried later. The category type also allows to specify a field to extract the categories from. The path parameter is used to specify this field of the documents that should be used. If the referenced field contains multiple values, all these values will be used as alternative categories.
- The mapping for a category is simply defined by its default values. These can either be defined as list of default categories:

```
"context": {
    "color": {
        "type": "category",
        "default": ["red", "orange"]
    }
}
```

or as a single value

```
"context": {
    "color": {
        "type": "category",
        "default": "red"
    }
}
```

or as reference to another field within the documents indexed:

```
"context": {
    "color": {
        "type": "category",
        "default": "red",
        "path": "color_field"
    }
}
```

Summary

- The different types of Suggesters namely term, phrase, completion, and context, were explored
- The term suggester is used to suggest corrections to terms
- The phrase suggester is similar to the term suggesters, but with phrase logic added in to suggest corrections to phrases
- Completion is used to autocomplete partially input words and phrases.
- Context is an older suggester from ES 2.1 and prior.

Lab 8 - The Power of Suggestion

Module 9: Aggregations

Objectives

- The Aggregations Framework
  - What are aggregations?
  - Elasticsearch aggregation engine
  - Metrics
  - Bucket
  - Pipeline Aggregations
  - Other considerations

What are aggregations?

Introduced in Elasticsearch 1.0, aggregations are the heart of data analytics in Elasticsearch. Highly flexible and performant, aggregations brought Elasticsearch 1.0 to a new position as a full-featured analysis engine. Extended through the life of Elasticsearch 1.x, in 2.x they are yet more powerful, less memory demanding, and faster. With this framework, you can use Elasticsearch as the analysis engine for data extraction and visualization.

```
curl -XPOST 'http://elk.com:81/dblog-2015.11.25/sql/_search?size=0&pretty=1' -d'
{
    "aggs": {
        "site_max_dur": {
            "filter": {
                "term": {
                    "site_id": 1167639
                }
            },
            "aggs": {
                "max_dur": {
                    "max": {
                        "field": "duration"
                    }
                }
            }
        }
    }
}'

result :

{
  "took" : 6121,
  "timed_out" : false,
  "_shards" : {
    "total" : 16,
    "successful" : 16,
    "failed" : 0
  },
  "hits" : {
    "total" : 65773792,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "site_max_dur" : {
      "doc_count" : 6221072,
      "max_dur" : {
        "value" : 1037.0
      }
    }
  }
}
```

Query Structure with Aggs

- To use aggregations, we need to add an additional section in our query. In general, our queries with aggregations look like this:

```
{
    "query": { … },
    "aggs" : {
      "aggregation_name" : {
        "aggregation_type" : {
          ...
        }
      }
    }
}
```

Aggs Property

- In the aggs property (you can use aggregations if you want; aggs is just an abbreviation), you can define any number of aggregations. Each aggregation is defined by its name and one of the types of aggregations that are provided by Elasticsearch. One thing to remember though is that the key defines the name of the aggregation (you will need it to distinguish particular aggregations in the server response). Let's take our library index and create the first query using use aggregations. A command sending such a query looks like this:

```
curl 'localhost:9200/library/_search?search_type=query_then_fetch&size=0&pretty' -d '{
   "aggs": {
      "years": {
         "stats": {
            "field": "year"
         }
      },
      "words": {
         "terms": {
            "field": "copies"
         }
      }
   }
}'
```

Aggs Property

- look at the response returned by Elasticsearch for the preceding query:


```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "words" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [ {
        "key" : 0,
        "doc_count" : 2
      }, {
        "key" : 1,
        "doc_count" : 1
      }, {
        "key" : 6,
        "doc_count" : 1
      } ]
    },
    "years" : {
      "count" : 4,
      "min" : 1886.0,
      "max" : 1961.0,
      "avg" : 1928.0,
      "sum" : 7712.0
    }
  }
}
```

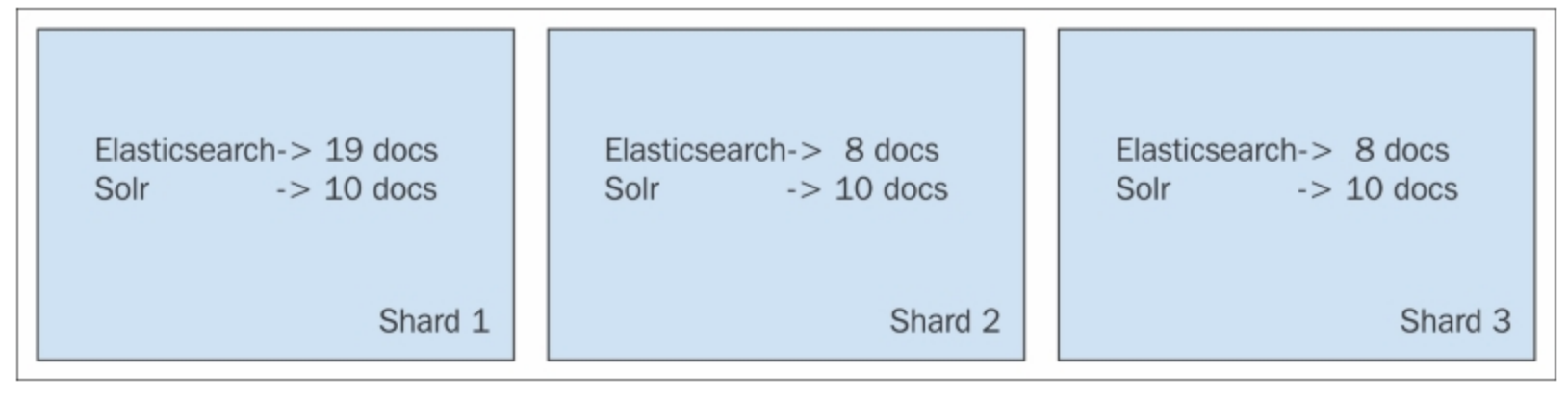
Aggregation Engine

- The great thing about the aggregation engine is that it allows you to have multiple aggregations and that aggregations can be nested. This means that you can have indefinite levels of nesting and any number of aggregations in general. The extended structure of the query is shown next:

```
{
   "query": { … },
   "aggs" : {
     "first_aggregation_name" : {
       "aggregation_type" : {
         ...
       },
     "aggregations" : {
          "first_nested_aggregation" : {
          ...
          },
          .
          .
          .
          "nth_nested_aggregation" : {
          ...
          }
       }
     },
     .
     .
     .
     "nth_aggregation_name" : {
     ...
     }
   }
}
```

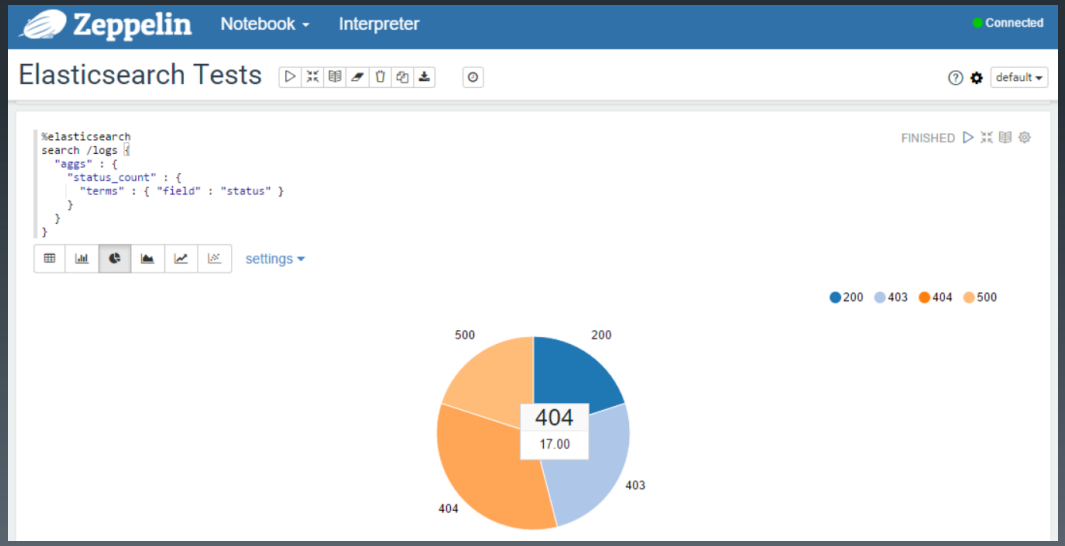
Inside the Aggregation Engine

- Aggregations work on the basis of results returned by the query. This is very handy as we get the information that we are interested in, both from the query as well as the data analysis perspective. So what does Elasticsearch do when we include the aggregation part of the query in the request that we send to Elasticsearch? First of all, the aggregation is executed on each relevant shard and the results are returned to the node that is responsible for running that query. That node waits for the partial results to be calculated; after it gets all the results, it merges the results, producing the final results. This approach is nothing new when it comes to distributed systems and how they work and communicate, but can cause issues when it comes to the precision of the results. In most cases this is not a problem, but you should be aware about what to expect. Let's imagine the following example:



Aggregation types

- Elasticsearch 2.x allows us to use three types of aggregation: metrics, buckets, and pipeline. The metrics aggregations return a metric, just like the stats aggregation we used for the stats field. The bucket aggregations return buckets, the key and the number of documents sharing the same values, ranges, and so on, just like the terms aggregation we used for the copies field. Finally, the pipeline aggregations introduced in Elasticsearch 2.0 aggregate the output of the other aggregations and their metrics, which allows us to do even more sophisticated data analysis. Knowing all that, let's now look at all the aggregations we can use in Elasticsearch 2.x.



Metric Aggregations

- We will start with the metrics aggregations, which can aggregate values from documents into a single metric. This is always the case with metrics aggregations – you can expect them to be a single metric on the basis of the data. Let's now take a look at the metrics aggregations available in Elasticsearch 2.x.

MINIMUM, MAXIMUM, AVERAGE, AND SUM

The first group of metrics aggregations that we want to show you is the one that calculates the basic value from the given documents. These aggregations are:

- `min`: This calculates the minimum value from the given numeric field in the returned documents
- `max`: This calculates the maximum value from the given numeric field in the returned documents
- `avg`: This calculates an average from the given numeric field in the returned documents
- `sum`: This calculates the sum from the given numeric field in the returned documents

Metric Aggregations in action

- We will start with the metrics aggregations, which can aggregate values from documents into a single metric. This is always the case with metrics aggregations – you can expect them to be a single metric on the basis of the data. Let's now take a look at the metrics aggregations available in Elasticsearch 2.x.

```
{
 "aggs" : {
  "avg_copies" : {
   "avg" : {
    "field" : "copies"
   }
  }
 }
}
```

```
{
  "took" : 5,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "avg_copies" : {
      "value" : 1.75
    }
  }
}
```

Other Metric Aggregations

- Value count
- Field
- Percentiles
- Percentile Ranks
- Top hits aggregation
- Geobounds aggregation
- Scripted Metric Aggregations

```
{
 "aggs" : {
  "copies_percentile_ranks" : {
   "percentile_ranks" : {
    "field" : "year",
    "values" : [ "1932", "1960" ]
   }
  }
 }
}
```

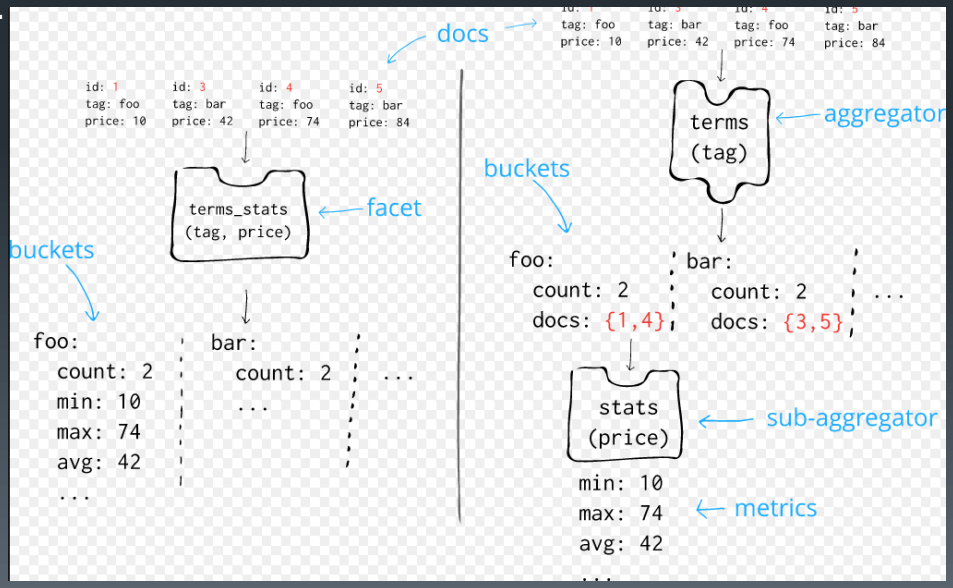
```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "copies_percentile_ranks" : {
      "values" : {
        "1932.0" : 49.5,
        "1960.0" : 61.5
      }
    }
  }
}
```

Bucket Aggregations

- The second type of aggregations that we will discuss are the buckets aggregations.
- Bucket aggregations don't calculate metrics over fields like the metrics aggregations do, but instead, they create buckets of documents. Each bucket is associated with a criterion (depending on the aggregation type) which determines whether or not a document in the current context "falls" into it. In other words, the buckets effectively define document sets. In addition to the buckets themselves, the bucket aggregations also compute and return the number of documents that "fell into" each bucket.

- Bucket aggregations, as opposed to metrics aggregations, can hold sub-aggregations. These sub-aggregations will be aggregated for the buckets created by their "parent" bucket aggregation.

- There are different bucket aggregators, each with a different "bucketing" strategy. Some define a single bucket, some define fixed number of multiple buckets, and others dynamically create the buckets during the aggregation process.



Filter (Bucket) Aggregation

- The filter aggregation is a simple bucketing aggregation that allows us to filter the results to a single bucket. For example, let's assume that we want to get a count and the average copies count of all the books that are novels, which means they have the term novel in the tags field. The query that will return such results looks as follows:

```
{
 "aggs" : {
  "novels_count" : {
   "filter" : {
    "term": {
     "tags": "novel"
    }
   },
   "aggs" : {
    "avg_copies" : {
     "avg" : {
      "field" : "copies"
     }
    }
   }
  }
 }
}
```

```
{
  "took" : 13,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "novels_count" : {
      "doc_count" : 2,
      "avg_copies" : {
        "value" : 3.5
      }
    }
  }
}
```

FilterS (Bucket) Aggregation

- The second bucket aggregation we want to show you is the filters aggregation. While the previously discussed filter aggregation resulted in a single bucket, the filters aggregation returns multiple buckets – one for each of the defined filters. Let's extend our previous example and assume that, in addition to the average number of copies for the novels, we also want to know the average number of copies for the books that are available.

```
{
 "aggs" : {
  "count" : {
   "filters" : {
    "filters" : {
     "novels" : {
      "term" : {
       "tags" : "novel"
      }
     },
     "available" : {
      "term" : {
       "available" : true
      }
     }
    }
   },
   "aggs" : {
    "avg_copies" : {
     "avg" : {
      "field" : "copies"
     }
    }
   }
  }
 }
}
```

```
},
"aggregations" : {
  "count" : {
    "buckets" : {
      "novels" : {
        "doc_count" : 2,
        "avg_copies" : {
          "value" : 3.5
        }
      },
      "available" : {
        "doc_count" : 2,
        "avg_copies" : {
          "value" : 0.5
        }
      }
    }
  }
}
```

Terms (Bucket) Aggregation

- One of the most commonly used bucket aggregations is the terms aggregation. It allows us to get information about the terms and the count of documents having those terms. For example, one of the simplest uses is getting the count of the books that are available and not available. We can do that by running the following query:

```
{
 "aggs" : {
  "counts" : {
   "terms" : {
    "field" : "available"
   }
  }
 }
}
```

```
},
"aggregations" : {
  "counts" : {
    "doc_count_error_upper_bound" : 0,
    "sum_other_doc_count" : 0,
    "buckets" : [ {
      "key" : 0,
      "key_as_string" : "false",
      "doc_count" : 2
    }, {
      "key" : 1,
      "key_as_string" : "true",
      "doc_count" : 2
    } ]
  }
}
```

Range (Bucket) Aggregation

- The range aggregation allows us to define one or more ranges and Elasticsearch calculates buckets for them. For example, if we want to check how many books were published in a given period of time, we create the following query:

```
{
 "aggs": {
  "years": {
   "range": {
    "field": "year",
    "ranges": [
     { "to" : 1850 },
     { "from": 1851, "to": 1900 },
     { "from": 1901, "to": 1950 },
     { "from": 1951, "to": 2000 },
     { "from": 2001 }
    ]
   }
  }
 }
}
```

```
  },
  "aggregations" : {
    "years" : {
      "buckets" : [ {
        "key" : "*-1850.0",
        "to" : 1850.0,
        "to_as_string" : "1850.0",
        "doc_count" : 0
      }, {
        "key" : "1851.0-1900.0",
        "from" : 1851.0,
        "from_as_string" : "1851.0",
        "to" : 1900.0,
        "to_as_string" : "1900.0",
        "doc_count" : 1
      }, {
        "key" : "1901.0-1950.0",
        "from" : 1901.0,
        "from_as_string" : "1901.0",
        "to" : 1950.0,
        "to_as_string" : "1950.0",
        "doc_count" : 2
      }, {
        "key" : "1951.0-2000.0",
        "from" : 1951.0,
        "from_as_string" : "1951.0",
        "to" : 2000.0,
        "to_as_string" : "2000.0",
        "doc_count" : 1
      }, {
        "key" : "2001.0-*",
        "from" : 2001.0,
        "from_as_string" : "2001.0",
        "doc_count" : 0
      } ]
    }
  }
}
```

IPv4 (Bucket) Aggregation

- A very interesting aggregation is the ip_range one as it works on Internet addresses. It works on the fields defined with the ip type and allows defining ranges given by the IP range in CIDR notation (http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing). An example usage of the ip_range aggregation looks as follows:

```
{
 "aggs": {
  "access": {
   "ip_range": {
    "field": "ip",
    "ranges": [
     { "from": "192.168.0.1", "to": "192.168.0.254" },
     { "mask": "192.168.1.0/24" }
    ]
   }
  }
 }
}
```

```
"access": {
   "buckets": [
      {
         "from": 3232235521,
         "from_as_string": "192.168.0.1",
         "to": 3232235774,
         "to_as_string": "192.168.0.254",
         "doc_count": 0
      },
      {
         "key": "192.168.1.0/24",
         "from": 3232235776,
         "from_as_string": "192.168.1.0",
         "to": 3232236032,
         "to_as_string": "192.168.2.0",
         "doc_count": 4
      }
   ]
}
```

Histogram (Bucket) Aggregation

- The histogram aggregation is an interesting one because of its automation. This aggregation defines buckets itself. We are only responsible for defining the field and the interval, and the rest is done automatically. The simplest form of a query that uses this aggregation looks as follows:

```
{
 "aggs": {
  "years": {
   "histogram": {
    "field" : "year",
    "interval": 100
   }
  }
 }
}
```

```
},
"aggregations" : {
  "years" : {
    "buckets" : [ {
      "key" : 1800,
      "doc_count" : 1
    }, {
      "key" : 1900,
      "doc_count" : 3
    } ]
  }
}
```

Geo Distance (Bucket) Aggregation

- The next two aggregations are connected with maps and spatial searches.

```
{
 "aggs": {
  "neighborhood": {
   "geo_distance": {
    "field": "location",
    "origin": [-0.1275, 51.507222],
    "ranges": [
     { "to": 1200 },
     { "from": 1201 }
    ]
   }
  }
 }
}
```

```
"neighborhood": {
   "buckets": [
      {
         "key": "*-1200.0",
         "from": 0,
         "to": 1200,
         "doc_count": 1
      },
      {
         "key": "1201.0-*",
         "from": 1201,
         "doc_count": 4
      }
   ]
}
```

Geohash Grid (Bucket) Aggregation

- The second aggregation related to geographical analysis is based on grids and is called geohash_grid. It organizes areas into grids and assigns every location to a cell in such a grid. To do this efficiently, Elasticsearch uses Geohash (http://en.wikipedia.org/wiki/Geohash), which encodes the location into a string.
- The longer the string is, the more accurate the description of a particular location. For example, one letter is sufficient to declare a box of about five thousand square kilometers and 5 letters are enough to increase the accuracy to five square kilometers.

```
{
 "aggs": {
  "neighborhood": {
   "geohash_grid": {
    "field": "location",
    "precision": 5
   }
  }
 }
}
```

Significant Terms (Bucket) Aggregation

- The significant_terms aggregation allows us to get the terms that are relevant and probably the most significant for a given query. The good thing is that it doesn't only show the top terms from the results of the given query, but also the one that seems to be the most important one.The use cases for this aggregation type can vary from finding the most troublesome server working in your application environment, to suggesting nicknames from text. Whenever Elasticsearch sees a significant change in the popularity of a term, such a term is a candidate for being significant.

```
curl -XGET 'localhost:9200/interns/_search?size=0&pretty' -d '{
 "query" : {
  "match" : {
   "intern" : "Richard"
  }
 },
 "aggregations" : {
  "description" : {
   "significant_terms" : {
    "field" : "grade"
   }
  }
 }
}'
```

```
    },
    "aggregations" : {
      "description" : {
        "doc_count" : 5,
        "buckets" : [ {
          "key" : "bad",
          "doc_count" : 3,
          "score" : 0.84,
          "bg_count" : 3
        } ]
      }
    }
  }
```

Choosing Significant Terms

- To calculate significant terms, Elasticsearch looks for data that reports a significant change in their popularity between two sets of data: the foreground set and the background set. The foreground set is the data returned by our query, while the background set is the data in our index (or indices, depending on how we run our queries). If a term exists in 10 documents out of one million indexed, but appears in 5 documents from the 10 returned, then such a term is definitely significant and worth concentrating on.

```
/interns/review/1' -d '{"intern" : "Richard", "grade" : "bad", "type" : "grade"}'
/interns/review/2' -d '{"intern" : "Ralf", "grade" : "perfect", "type" : "grade"}'
/interns/review/3' -d '{"intern" : "Richard", "grade" : "bad", "type" : "grade"}'
/interns/review/4' -d '{"intern" : "Richard", "grade" : "bad", "type" : "review"}'
/interns/review/5' -d '{"intern" : "Richard", "grade" : "good", "type" : "grade"}'
/interns/review/6' -d '{"intern" : "Ralf", "grade" : "good", "type" : "grade"}'
/interns/review/7' -d '{"intern" : "Ralf", "grade" : "perfect", "type" : "review"}'
/interns/review/8' -d '{"intern" : "Richard", "grade" : "medium", "type" : "review"}
/interns/review/9' -d '{"intern" : "Monica", "grade" : "medium", "type" : "grade"}'
/interns/review/10' -d '{"intern" : "Monica", "grade" : "medium", "type" : "grade"}'
/interns/review/11' -d '{"intern" : "Ralf", "grade" : "good", "type" : "grade"}'
/interns/review/12' -d '{"intern" : "Ralf", "grade" : "good", "type" : "grade"}'
```

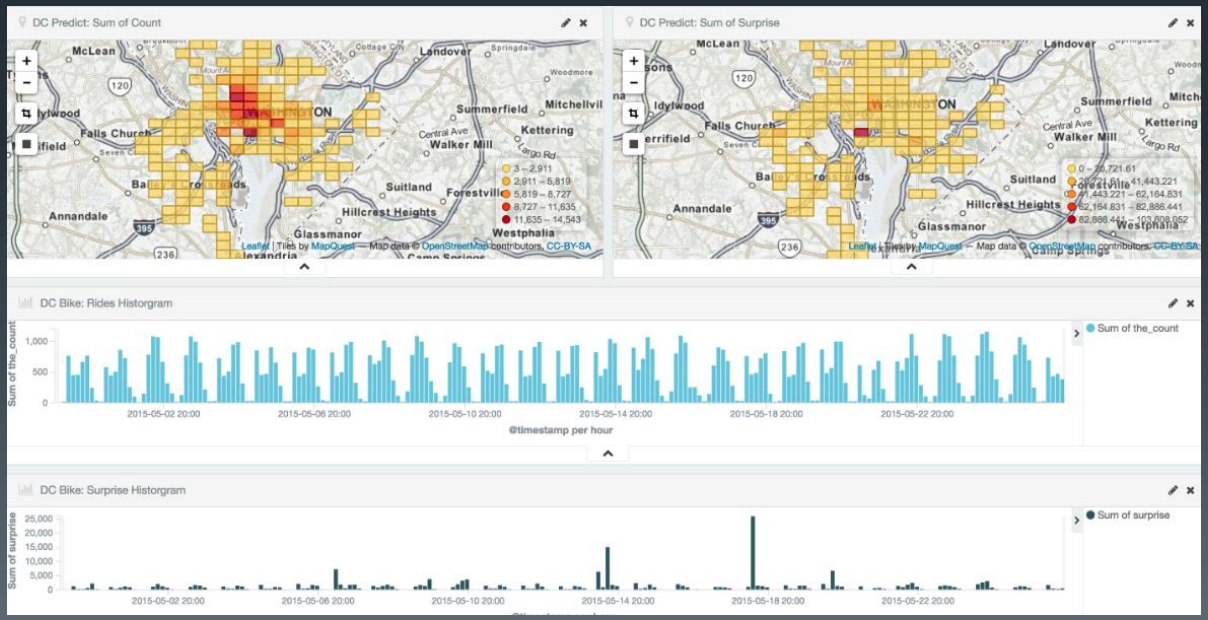
So many more Bucket aggs

- Children
- Sampler
- Global
- Nested
- Reverse Nested
- Missing
- Date Range

This functionality is experimental and may be changed or removed completely in a future release.

Pipeline Aggregations

- Pipeline aggregations work on the outputs produced from other aggregations rather than from document sets, adding information to the output tree. There are many different types of pipeline aggregation, each computing different information from other aggregations, but these types can be broken down into two families:
  - Parent - A family of pipeline aggregations that is provided with the output of its parent aggregation and is able to compute new buckets or new aggregations to add to existing buckets.
  - Sibling - Pipeline aggregations that are provided with the output of a sibling aggregation and are able to compute a new aggregation which will be at the same level as the sibling aggregation.



Buckets_path syntax

- Most pipeline aggregations require another aggregation as their input. The input aggregation is defined via the buckets_path parameter, which follows a specific format:

```
AGG_SEPARATOR       :=  '>'
METRIC_SEPARATOR    :=  '.'
AGG_NAME            :=  <the name of the aggregation>
METRIC              :=  <the name of the metric (in case of multi-value metrics aggregation)>
PATH                :=  <AGG_NAME>[<AGG_SEPARATOR><AGG_NAME>]*[<METRIC_SEPARATOR><METRIC>]
```

```
{
    "my_date_histo":{
        "date_histogram":{
            "field":"timestamp",
            "interval":"day"
        },
        "aggs":{
            "the_sum":{
                "sum":{ "field": "lemmings" } ❶
            },
            "the_movavg":{
                "moving_avg":{ "buckets_path": "the_sum" } ❷
            }
        }
    }
}
```

Sibling pipeline aggregations

- buckets_path is also used for Sibling pipeline aggregations, where the aggregation is "next" to a series of buckets instead of embedded "inside" them. For example, the max_bucket aggregation uses the buckets_path to specify a metric embedded inside a sibling aggregation:

```
{
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                }
            }
        },
        "max_monthly_sales": {
            "max_bucket": {
                "buckets_path": "sales_per_month>sales"
            }
        }
    }
}
```

Special Paths

- Instead of pathing to a metric, buckets_path can use a special "_count" path. This instructs the pipeline aggregation to use the document count as it's input. For example, a moving average can be calculated on the document count of each bucket, instead of a specific metric:

```
{
    "my_date_histo":{
        "date_histogram":{
            "field":"timestamp",
            "interval":"day"
        },
        "aggs":{
            "the_movavg":{
                "moving_avg":{ "buckets_path": "_count" }
            }
        }
    }
}
```

Avg Bucket Aggregation

- A sibling pipeline aggregation which calculates the (mean) average value of a specified metric in a sibling aggregation. The specified metric must be numeric and the sibling aggregation must be a multi-bucket aggregation.

```
{
    "avg_bucket": {
        "buckets_path": "the_sum"
    }
}
```

Table 1. `avg_bucket` Parameters

Parameter Name	Description	Required	Default Value
`buckets_path`	The path to the buckets we wish to find the average for (see the section called "`buckets_path` Syntax" for more details)	Required	
`gap_policy`	The policy to apply when gaps are found in the data (see the section called "Dealing with gaps in the data" for more details)	Optional, defaults to `skip`	
	`format`	format to apply to the output value of this aggregation	Optional, defaults to `null`

Avg Bucket Aggregation Example

- The following snippet calculates the average of the total monthly sales:

```
{
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                }
            }
        },
        "avg_monthly_sales": {
            "avg_bucket": {
                "buckets_path": "sales_per_month>sales"
            }
        }
    }
}
```


```
{
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "sales": {
                  "value": 550
               }
            },
            {
               "key_as_string": "2015/02/01 00:00:00",
               "key": 1422748800000,
               "doc_count": 2,
               "sales": {
                  "value": 60
               }
            },
            {
               "key_as_string": "2015/03/01 00:00:00",
               "key": 1425168000000,
               "doc_count": 2,
               "sales": {
                  "value": 375
               }
            }
         ]
      },
      "avg_monthly_sales": {
         "value": 328.33333333333333
      }
   }
}
```

Derivative Aggregation

- A parent pipeline aggregation which calculates the derivative of a specified metric in a parent histogram (or date_histogram) aggregation. The specified metric must be numeric and the enclosing histogram must have min_doc_count set to 0 (default for histogram aggregations).

```
{
    "derivative": {
        "buckets_path": "the_sum"
    }
}
```

Table 2. `derivative` Parameters

Parameter Name	Description	Required	Default Value
`buckets_path`	The path to the buckets we wish to find the derivative for (see the section called "`buckets_path` Syntax" for more details)	Required	
`gap_policy`	The policy to apply when gaps are found in the data (see the section called "Dealing with gaps in the data" for more details)	Optional, defaults to `skip`	
`format`	format to apply to the output value of this aggregation	Optional, defaults to `null`	

First Order Derivative Aggregation

- The following snippet calculates the derivative of the total monthly sales:

```
{
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                },
                "sales_deriv": {
                    "derivative": {
                        "buckets_path": "sales" (1)
                    }
                }
            }
        }
    }
}
```

```
"aggregations": {
   "sales_per_month": {
      "buckets": [
         {
            "key_as_string": "2015/01/01 00:00:00",
            "key": 1420070400000,
            "doc_count": 3,
            "sales": {
               "value": 550
            } (1)
         },
         {
            "key_as_string": "2015/02/01 00:00:00",
            "key": 1422748800000,
            "doc_count": 2,
            "sales": {
               "value": 60
            },
            "sales_deriv": {
               "value": -490 (2)
            }
         },
         {
            "key_as_string": "2015/03/01 00:00:00",
            "key": 1425168000000,
            "doc_count": 2, (3)
            "sales": {
               "value": 375
            },
            "sales_deriv": {
               "value": 315
```

Second Order Derivative Aggregation

- A second order derivative can be calculated by chaining the derivative pipeline aggregation onto the result of another derivative pipeline aggregation as in the following example which will calculate both the first and the second order derivative of the total monthly sales:

```
"aggs" : {
    "sales_per_month" : {
        "date_histogram" : {
            "field" : "date",
            "interval" : "month"
        },
        "aggs": {
            "sales": {
                "sum": {
                    "field": "price"
                }
            },
            "sales_deriv": {
                "derivative": {
                    "buckets_path": "sales"
                }
            },
            "sales_2nd_deriv": {
                "derivative": {
                    "buckets_path": "sales_deriv" (1)
                }
```

```
"aggregations": {
  "sales_per_month": {
    "buckets": [
      {
        "key_as_string": "2015/01/01 00:00:00",
        "key": 1420070400000,
        "doc_count": 3,
        "sales": {
          "value": 550
        } (1)
      },
      {
        "key_as_string": "2015/02/01 00:00:00",
        "key": 1422748800000,
        "doc_count": 2,
        "sales": {
          "value": 60
        },
        "sales_deriv": {
          "value": -490
        } (2)
      },
      {
        "key_as_string": "2015/03/01 00:00:00",
        "key": 1425168000000,
        "doc_count": 2,
        "sales": {
          "value": 375
        },
        "sales_deriv": {
          "value": 315
        },
        "sales_2nd_deriv": {
          "value": 805
```

Units

- The derivative aggregation allows the units of the derivative values to be specified. This returns an extra field in the response normalized_value which reports the derivative value in the desired x-axis units. In the below example we calculate the derivative of the total sales per month but ask for the derivative of the sales as in the units of sales per day:

```
{
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                },
                "sales_deriv": {
                    "derivative": {
                        "buckets_path": "sales",
                        "unit": "day" ❶
```

```
"aggregations": {
  "sales_per_month": {
    "buckets": [
      {
        "key_as_string": "2015/01/01 00:00:00",
        "key": 1420070400000,
        "doc_count": 3,
        "sales": {
          "value": 550
        } ❶
      },
      {
        "key_as_string": "2015/02/01 00:00:00",
        "key": 1422748800000,
        "doc_count": 2,
        "sales": {
          "value": 60
        },
        "sales_deriv": {
          "value": -490, ❷
          "normalized_value": -17.5 ❸
        }
      },
      {
        "key_as_string": "2015/03/01 00:00:00",
        "key": 1425168000000,
        "doc_count": 2,
        "sales": {
          "value": 375
        },
        "sales_deriv": {
          "value": 315,
          "normalized_value": 10.16129032258065
```

Additional Pipeline Aggregations

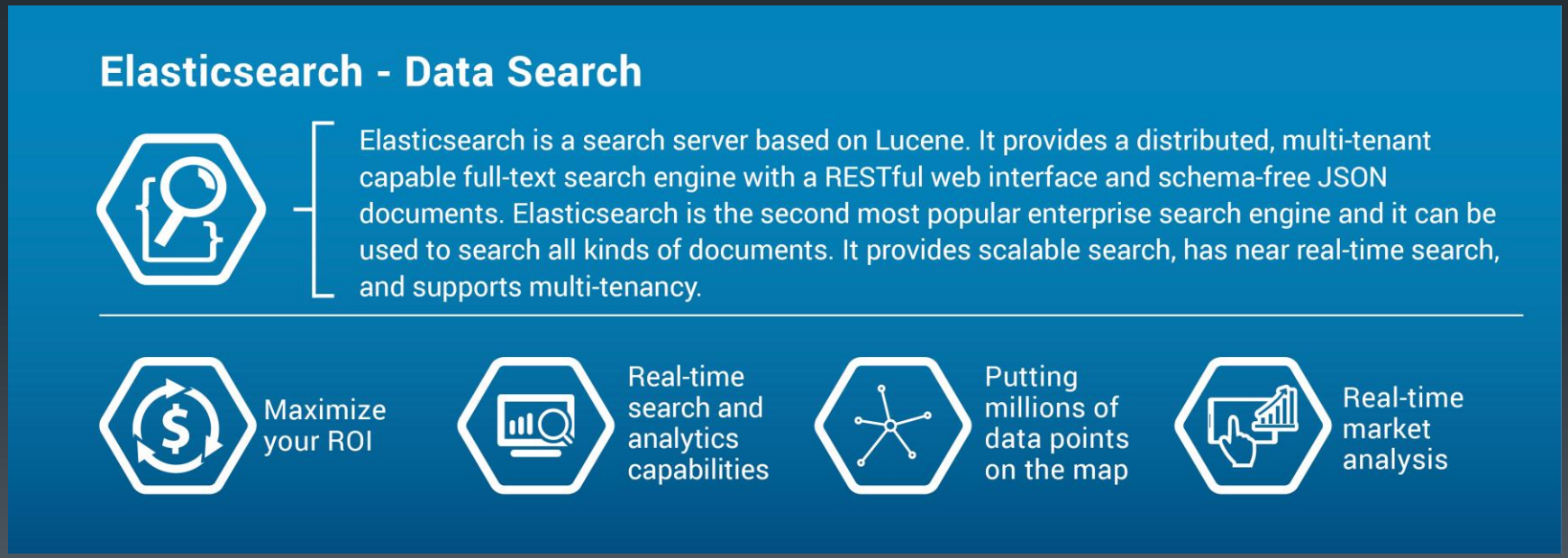
- Max/Min Bucket Aggregation
- Sum Bucket Aggregation
- Stats Bucket Aggregation
- Extended Stats Bucket Aggregation
- Percentiles Bucket Aggregation
- Moving Average Bucket Aggregation
- Cumulative Sum Aggregation
- Bucket/Bucket Script Aggregation
- Serial Differencing Aggregation

Summary

- We learned that there are three major types of aggregations: metrics, bucket, and pipeline
- The metrics aggregations return a metric, just like the stats aggregation we used for the stats field.
- The bucket aggregations return buckets, the key and the number of documents sharing the same values, ranges, and so on, just like the terms aggregation we used for the copies field.
- The pipeline aggregations introduced in Elasticsearch 2.0 aggregate the output of the other aggregations and their metrics, which allows us to do even more sophisticated data analysis.
- The pipeline aggregations are experimental and may be retired in future releases.

Lab 9 - Aggregation

Module 10: Bringing It All Together



Objectives

- Review of all major components
- Review of all best practices and tips
- Search, Search, Search
- Resources that may help
- What next for learning ElasticSearch?

Glossary of Terms

- analysis
- Analysis is the process of converting full text to terms. Depending on which analyzer is used, these phrases: FOO BAR, Foo-Bar, foo,bar will probably all result in the terms foo and bar. These terms are what is actually stored in the index. A full text query (not a term query) for FoO:bAR will also be analyzed to the terms foo,bar and will thus match the terms stored in the index. It is this process of analysis (both at index time and at search time) that allows elasticsearch to perform full text queries. Also see text and term.

- cluster
- A cluster consists of one or more nodes which share the same cluster name. Each cluster has a single master node which is chosen automatically by the cluster and which can be replaced if the current master node fails.

- document
- A document is a JSON document which is stored in elasticsearch. It is like a row in a table in a relational database. Each document is stored in an index and has a type and an id. A document is a JSON object (also known in other languages as a hash / hashmap / associative array) which contains zero or more fields, or key-value pairs. The original JSON document that is indexed will be stored in the _source field, which is returned by default when getting or searching for a document.

- id
- The ID of a document identifies a document. The index/type/id of a document must be unique. If no ID is provided, then it will be auto-generated. (also see routing)

Glossary of Terms

- field
- A document contains a list of fields, or key-value pairs. The value can be a simple (scalar) value (eg a string, integer, date), or a nested structure like an array or an object. A field is similar to a column in a table in a relational database. The mapping for each field has a field type (not to be confused with document type) which indicates the type of data that can be stored in that field, eg integer, string, object. The mapping also allows you to define (amongst other things) how the value for a field should be analyzed.

- index
- An index is like a database in a relational database. It has a mapping which defines multiple types. An index is a logical namespace which maps to one or more primary shards and can have zero or more replica shards.

- mapping
- A mapping is like a schema definition in a relational database. Each index has a mapping, which defines each type within the index, plus a number of index-wide settings. A mapping can either be defined explicitly, or it will be generated automatically when a document is indexed.

Glossary of Terms

- node
- A node is a running instance of elasticsearch which belongs to a cluster. Multiple nodes can be started on a single server for testing purposes, but usually you should have one node per server. At startup, a node will use unicast to discover an existing cluster with the same cluster name and will try to join that cluster.

- primary shard
- Each document is stored in a single primary shard. When you index a document, it is indexed first on the primary shard, then on all replicas of the primary shard. By default, an index has 5 primary shards. You can specify fewer or more primary shards to scale the number of documents that your index can handle. You cannot change the number of primary shards in an index, once the index is created. See also routing

- replica shard
- Each primary shard can have zero or more replicas. A replica is a copy of the primary shard, and has two purposes:

  - increase failover: a replica shard can be promoted to a primary shard if the primary fails
  - increase performance: get and search requests can be handled by primary or replica shards. By default, each primary shard has one replica, but the number of replicas can be changed dynamically on an existing index. A replica shard will never be started on the same node as its primary shard.

Glossary of Terms

- routing
- When you index a document, it is stored on a single primary shard. That shard is chosen by hashing the routing value. By default, the routing value is derived from the ID of the document or, if the document has a specified parent document, from the ID of the parent document (to ensure that child and parent documents are stored on the same shard). This value can be overridden by specifying a routing value at index time, or a routing field in the mapping.

- shard
- A shard is a single Lucene instance. It is a low-level “worker” unit which is managed automatically by elasticsearch. An index is a logical namespace which points to primary and replica shards. Other than defining the number of primary and replica shards that an index should have, you never need to refer to shards directly. Instead, your code should deal only with an index. Elasticsearch distributes shards amongst all nodes in the cluster, and can move shards automatically from one node to another in the case of node failure, or the addition of new nodes.

- source field
- By default, the JSON document that you index will be stored in the _source field and will be returned by all get and search requests. This allows you access to the original object directly from search results, rather than requiring a second step to retrieve the object from an ID. Note: the exact JSON string that you indexed will be returned to you, even if it contains invalid JSON. The contents of this field do not indicate anything about how the data in the object has been indexed.

Glossary of Terms

- term
- A term is an exact value that is indexed in elasticsearch. The terms foo, Foo, FOO are NOT equivalent. Terms (i.e. exact values) can be searched for using term queries. See also text and analysis.

- text
- Text (or full text) is ordinary unstructured text, such as this paragraph. By default, text will be analyzed into terms, which is what is actually stored in the index. Text fields need to be analyzed at index time in order to be searchable as full text, and keywords in full text queries must be analyzed at search time to produce (and search for) the same terms that were generated at index time. See also term and analysis.

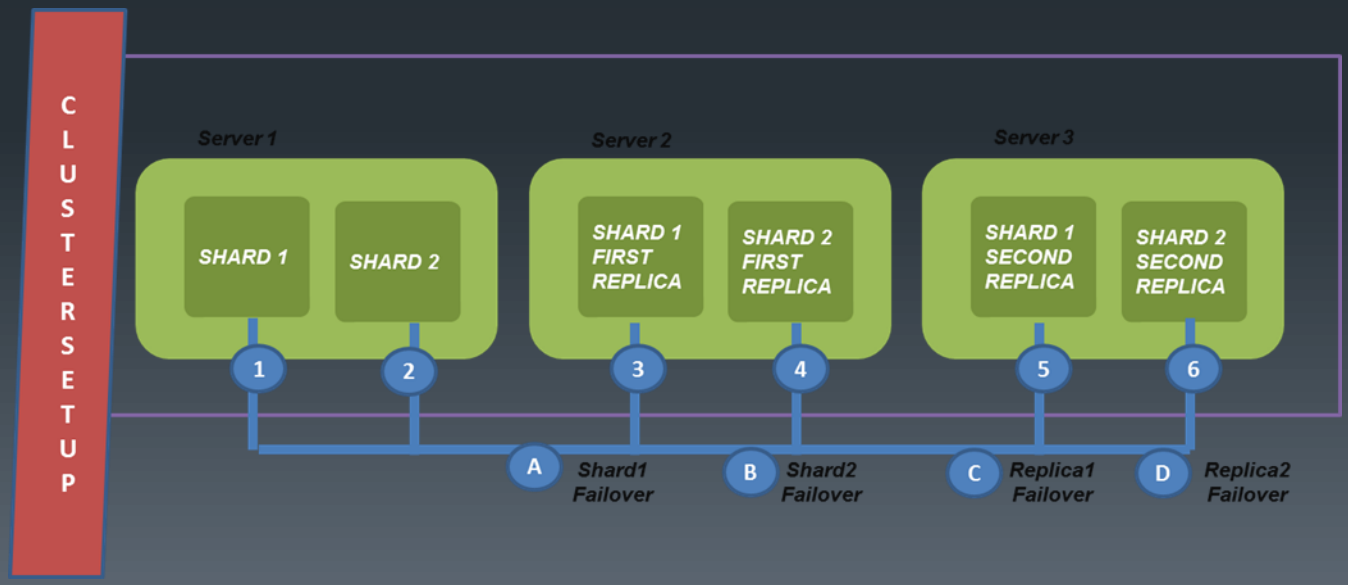
- type
- A type is like a table in a relational database. Each type has a list of fields that can be specified for documents of that type. The mapping defines how each field in the document is analyzed.

Module 1 Summary

- Elasticsearch is an open-source, broadly-distributable, readily-scalable, enterprise-grade search engine
- Managed through an HTTP RESTful API
- Schema less with a data driven schema
- Written in Java and backed by Apache Lucene
- Used by many major tech players in the market
- Near Real Time Search
- Runs supported on Windows and Linux in production, but on anything that runs in a JVM.

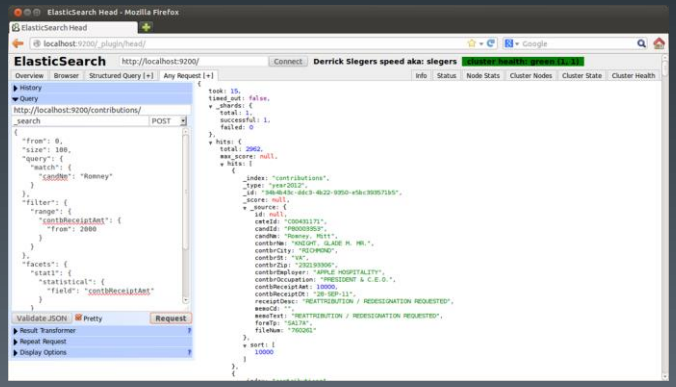
Module 2 Summary

- Index, documents, fields, and mappings make up core data components of ElasticSearch
- The roles of ES servers are either data, master, or client nodes.
- Clusters are collections of nodes
- Shards are data segments dedicated to one or more nodes.
- Indexing is a way of grouping key terms to enable search



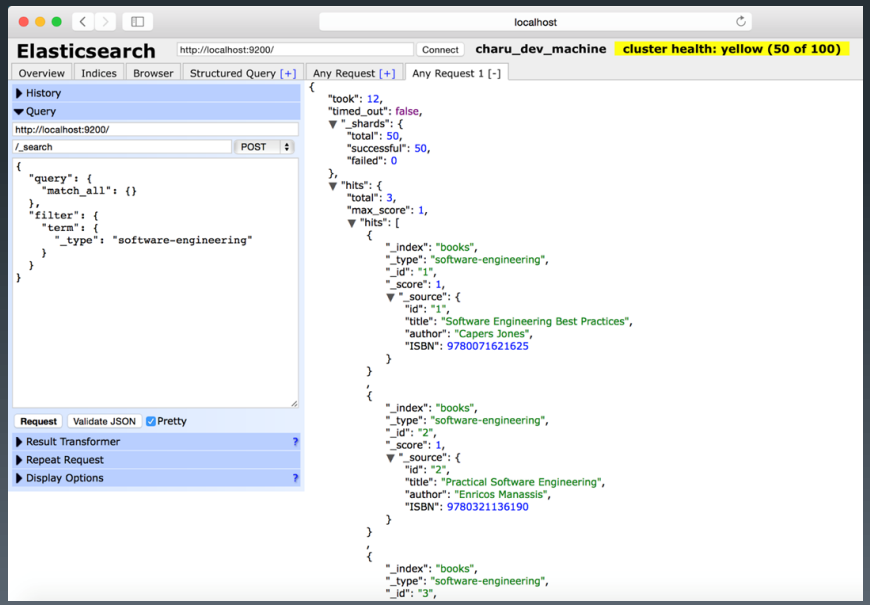
Module 3 Summary

- We explored the REST API with various endpoint for mananging ElasticSearch
- Performed Create, Replace, Update, and Delete actions using the RESTful API
- Documents can be partial updated or ‘upsert’ed if they don’t’ already exist.
- Documents are automatically versioned, but can be externally and manually versioned.
- Touched on Bulk Creation and Updates.



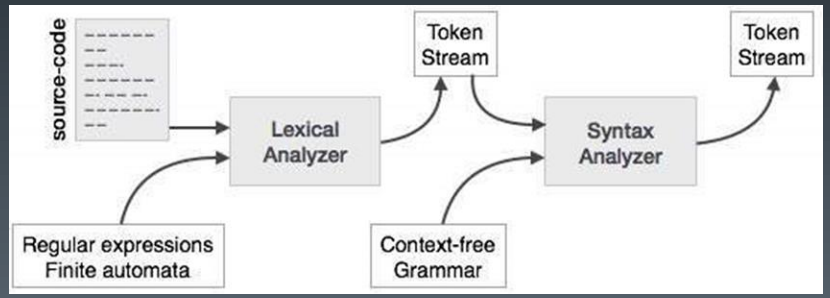
Module 4 Summary

- Executed basic search
- Explored the basic URI search with terms
- Examined a number of Query terms
- Looked at Phrase Search
- Showcased Full-Text Search
- Touched on Complicated Search



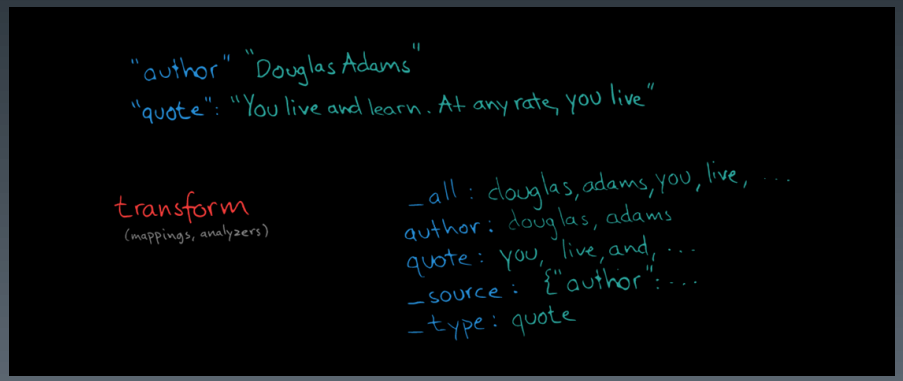
Module 5 Summary

- Deep Dive into the three components of analysis: character filters, tokenization, and token filtering
- The different built-in analyzers: standard, stop, simple, keyword, and more
- Analyzers can be tested by specifying them
- You can create custom analyzers with different combinations of token filters, char filters, and tokenizers.



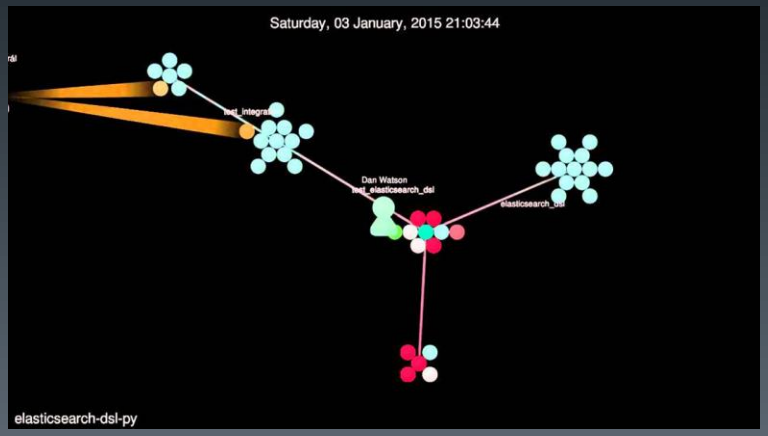
Module 6 Summary

- Index, documents, fields, and mappings make up core data components of ElasticSearch
- The roles of ES servers are either data, master, or client nodes.
- Clusters are collections of nodes
- Shards are data segments dedicated to one or more nodes.
- Indexing is a way of grouping key terms to enable search



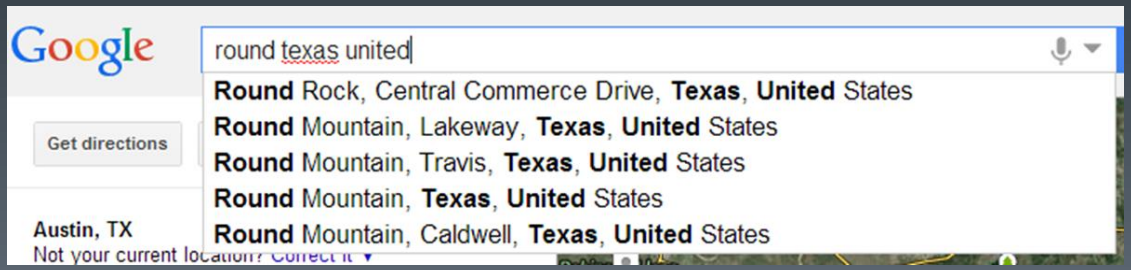
Module 7 Summary

- Review of match, match_all queries
- Discussed _all field and its use in query_string and simple_query_string
- Explored simple query string parameters
- Covered various types of compound queries
- Mentioned the power of Span queries



Module 8 Summary

- The different types of Suggesters namely term, phrase, completion, and context, were explored
- The term suggester is used to suggest corrections to terms
- The phrase suggester is similar to the term suggesters, but with phrase logic added in to suggest corrections to phrases
- Completion is used to autocomplete partially input words and phrases.
- Context is an older suggester from ES 2.1 and prior.



Module 9 Summary

- We learned that there are three major types of aggregations: metrics, bucket, and pipeline
- The metrics aggregations return a metric, just like the stats aggregation we used for the stats field.
- The bucket aggregations return buckets, the key and the number of documents sharing the same values, ranges, and so on, just like the terms aggregation we used for the copies field.
- The pipeline aggregations introduced in Elasticsearch 2.0 aggregate the output of the other aggregations and their metrics, which allows us to do even more sophisticated data analysis.
- The pipeline aggregations are experimental and may be retired in future releases.

Best Practices

- A Test cluster is essential
- Choose the right number of primary shards
- Size your cluster carefully
- Be generous with memory
- Pay attention to your queries and aggs
- Use visual tools/plugins
- Practice failure scenarios
- Avoid complicated routing and jurying rigging

Search, Search, Search

- Search much
- Search often
- Misspell your searchesx
- Search HigH
- Search lOw
- Suggest your searches
- Aggregate your searches
- Map your searches
- Monitor your cluster

Learning Resources

- At the end of this slide deck
- Covers articles
- Books
- Be mindful of 2.x versus older content. 2.x came out in October of 2015 so keep that in mind
- Glossary of Terms

Next Steps to Learn

- Set up a test cluster
- ES Training?

The End

Many thanks for attending!

Appendix

- Additional topics of interest for reference
  - Links
  - Books
  - APIs
  - GUIs
  - Other topics

Links to Getting Started

- **Optimizing Elasticsearch: How Many Shards per Index?**.
- **Thoughts on Launching and Scaling Elasticsearch**.
- **How-to: Quick-and-Easy Resizing of your Elasticsearch Cluster**
- **Choosing a Size for Your Nodes**
- A Useful Elasticsearch Cheat Sheet in Times of Trouble
- ElasticSearch pre-flight checklist

Links to operations links

- **Super-easy Cluster Provisioning**
- Docker – Using Compose to scale up elasticsearch cluster
- https://www.elastic.co/blog/setting-up-elasticsearch-for-a-blog
- https://www.elastic.co/blog/a-heap-of-trouble
- http://insightdataengineering.com/blog/elasticsearch-crud/
- https://www.spinn3r.com/blog/elasticsearch-at-scale.html
- https://www.elastic.co/blog/hot-warm-architecture

Links to Lucene

- https://www.elastic.co/blog/lucene-points-6.0
- https://www.elastic.co/blog/store-compression-in-lucene-and-elasticsearch
- https://www.elastic.co/blog/elasticsearch-storage-the-true-story
- http://logz.io/blog/elasticsearch-queries/

Links to Cloud

- The definitive guide for Elasticsearch on Windows Azure
- https://aws.amazon.com/blogs/aws/cloudwatch-logs-subscription-consumer-elasticsearch-kibana-dashboards/
- https://aws.amazon.com/blogs/aws/category/amazon-elasticsearch-service/
- https://azure.microsoft.com/en-us/blog/archive-elasticsearch-indices-to-azure-blob-storage-using-the-azure-cloud-plugin/

Links to misc

- Docker – Using Compose to scale up elasticsearch cluster
- https://www.elastic.co/v5
- https://www.elastic.co/blog/category/engineering
- https://blog.codecentric.de/en/2016/02/elasticsearch-cluster-in-a-jiffy-2/
- http://rubytester.github.io/2016/03/22/elasticsearch-mac-osx

Books

- Elasticsearch 2.3 Elastic.co Documentation
- Elasticsearch 2.x the definitive guide
- Elasticsearch Server
- Elasticsearch Essentials
- Elasticsearch Indexing