

Full Text Search Engine Using ElasticSearch

Rishabh Jain

College of Computing
Illinois Institute of Technology
Chicago, IL, USA, 60616
rjain35@hawk.iit.edu
A20495530

Siddhant Bhatia

College of Computing
Illinois Institute of Technology
Chicago, IL, USA, 60616
sbhatia14@hawk.iit.edu
A20500508

Utkarsha Malegaonkar

College of Computing
Illinois Institute of Technology
Chicago, IL, USA, 60616
umalegaonkar@hawk.iit.edu
A20493621

Abstract— Every day humans are generating billions of logs and all of them may or may not be important. Each log contains some key elements that are unique and will not be the same in the next one. Logs are having vital information in them, and each piece of information is not required every time. So, we will build a tool that can easily obtain data from them according to our dataset using AWS ElasticSearch. For extracting the required text from a chunk of the data ElasticSearch will be used by hosting elastic search service on AWS integrated with Kibana for more convenience while handling and understanding the logs. The sorted data will be used for other purposes too. For example, it can be used for the text auto completion feature, which helps the user while they are searching for a specific query.

Keywords—AWS, Kibana, Logs.

I. INTRODUCTION

In modern times, all the processed information is necessary, and we cannot lose it at any cost. But, sometimes when a system failure occurs, we might be at the risk of losing data. Therefore, to make sure that we never lose the data even in the time of system failure, a concept named ‘Logs’ was derived in the market back in the early 1990s.

Now looking at today, we realize that each and every second we are generating billions of logs and each of them is required to be safely stored for at least 24 hrs. After that, administrators can remove them as they might be consuming a huge amount of storage, increasing the payment amount for more resource usage.

In case of emergency, when a transaction is lost, then we have logs for the same. To recover that transaction, we need to search for the relevant logs, which is quite impossible without using any text search tools. If we have efficient text search tools, we can get relevant in a fraction of a second. But, if there is no text search tool then it will take a huge amount of time to find the accurate logs as they might be in millions or billions in the count.

As per the market trends, one of the most efficient text search tools is ElasticSearch, it is a modern search and analytics engine which is based on Apache Lucene. It is completely an open-source project and built with Java. Elasticsearch is a NoSQL database. That means it stores data in an unstructured way and that you cannot use SQL to query it.

It is super-efficient because, unlike most NoSQL databases, Elasticsearch has a strong focus on search capabilities and features. That's why the easiest way to get data from ES is to search for it using the extensive Elasticsearch API. In the context of data analysis, Elasticsearch is used together with the other components in the ELK Stack.

II. PROPOSED WORK

We have built a search engine that has the capacity of an auto-complete search. Apart from the ElasticSearch, its libraries and Kibana, we will be using other technologies like react and flask to create a simple frontend and backend for our project. We chose to use the Netflix dataset for this project. We have performed automation on the Netflix dataset, ex. if we are searching for a movie that starts with the letter A, then our system will automatically show all the other relevant movies as a result that starts with the letter A, and it will also save the selected movie into our search so that the next time when we search, the previous search will be automatically pre-populated.

Below are main components that are required as prerequisite to move ahead with the project.

1) *ElasticSearch*

Elasticsearch is a full-text search and analysis engine based on Apache Lucene. With Elasticsearch, you can easily perform data aggregation operations on data from multiple sources or perform unstructured queries such as fuzzy searches on stored data. Like MongoDB, it saves data in a document-like format. The data is serialized in JSON format. This adds non-relational characters and can also be used as a NoSQL/non-relational database.

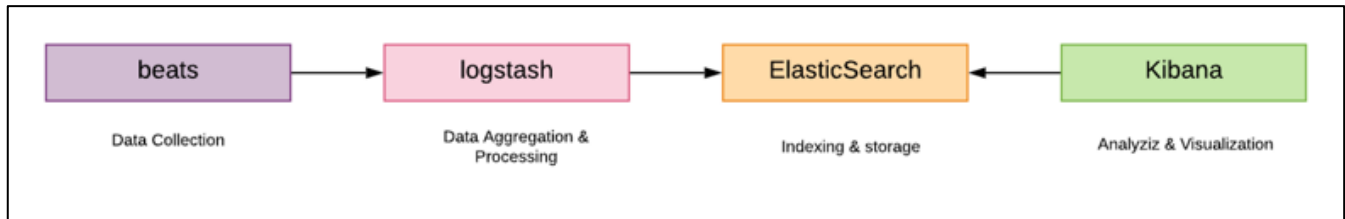
2) *Kibana*

Kibana is a free, open front-end application that sits on top of Elastic Stack that provides searchability and data visualization for data indexed in Elasticsearch. Commonly known as the graph engine for Elastic Stacks. Kibana also serves as a user interface for monitoring, managing, and securing an Elastic Stack cluster, as well as a centralized hub for integrated solutions built on Elastic Stack.

III. METHODOLOGY

In this section we will be explaining the detailed steps to complete our objective including high level steps.

A. High Level steps



1) Data Collection

Beats are used for data collection. Beats is a lightweight data sender. This is a free open platform for single purpose data senders. Send data from hundreds or thousands of machines and systems to Logstash or Elasticsearch.

2) Data Aggregation & Processing

Logstash is used to centralize, transform, and store data. Logstash is a free and open server-side data processing pipeline that retrieves data from a variety of sources, transforms it, and sends it to your favorite stash.

3) Indexing & Storage

Elasticsearch is a decentralized RESTful search and analytics engine for an ever-growing number of use cases. At the heart of the Elastic Stack, this data is centrally stored for ultra-fast searches, fine-grained relevance, and powerful analysis that can be easily scaled.

4) Analysis & Visualization

Kibana is a free, open front-end application built on Elastic Stack that provides Elasticsearch-indexed data search and data visualization capabilities. Known as the Elastic Stack charting tool, Kibana also serves as a user interface for monitoring, managing, and protecting Elastic Stack clusters, as well as a central hub for integrated solutions built on Elastic Stack.

B. Terminologies

1) *Cluster* - A cluster is a collection of Elasticsearch-enabled systems that collaborate and work near store data and resolve queries. These are further divided into categories based on their function within the cluster.

2) *Node* - A node is a JVM Process that runs an Elasticsearch instance and is accessible over the network by other machines or nodes in a cluster.

3) *Index*: An index in Elasticsearch is analogous to tables in relational databases.

4) *Mapping* - Each index is coupled with a mapping, which is a schema-definition of the data that each individual document in the index can carry. This can be done manually for each index, or it can be done automatically when data is uploaded into the index.

5) *Document* - A JSON document. In relational terms, this would represent a single row in a table.

6) *Shard* - Shards are data blocks that may or may not be associated with the same index. Vertically scaling storage is impossible since data pertaining to a single index can get quite large, say a few hundred GBs or even a few TBs. Instead, data is conceptually separated into shards that are stored on multiple nodes and operate on their own data. Horizontal scaling is possible because of this.

7) *Replicas*: Each shard in a cluster can be replicated on one or more cluster nodes. This provides for a backup failover. If one of the nodes fails or can no longer use its resources, a replica of the data is always available to work with. Each shard has one replica by default, although the number can be changed. The usage of replicas, in addition to Failover, improves search performance.

C. Technology Used – NodeJS, Docker, React, ElasticSearch, Kibana, LogStash.

D. Implementation

1. Project Set up

- 1.1 Docker: We'll be using Docker to manage the environments and dependencies for this project. Docker is a containerization engine that allows applications to be run in isolated environments, unaffected by the host operating system and local development environment.
- 1.2 Installing Docker and Docker Compose
Install Docker - <https://docs.docker.com/engine/installation/>
Install Docker Compose - <https://docs.docker.com/compose/install/>
- 1.3 Add Docker-Compose Config
We'll create a docker-compose.yml file to define each container in our application stack.
This is for the connection to the node.js

```
version: '3'

services:
  api: # Node.js App
    container_name: gs-api
    build: .
    ports:
      - "3000:3000" # Expose API port
      - "9229:9229" # Expose Node process debug port (disable in production)
    environment: # Set ENV vars
      - NODE_ENV=local
      - ES_HOST=elasticsearch
      - PORT=3000
    volumes: # Attach local book data directory
      - ./netflix:C:\Users\bhati\Downloads\elasticsearch\netflix
```

- 1.4 This is for the connection to the ElasticSearch.

```
17     frontend: # Nginx Server For Frontend App
18         container_name: gs-frontend
19         image: nginx
20         volumes: # Serve local "public" dir
21             - ./public:/usr/share/nginx/html
22         ports:
23             - "8080:80" # Forward site to localhost:8080
24
25     elasticsearch: # Elasticsearch Instance
26         container_name: gs-search
27         image: docker.elastic.co/elasticsearch/elasticsearch:7.15.2
28         volumes: # Persist ES data in separate "esdata" volume
29             - esdata:/usr/share/elasticsearch/data
30         environment:
31             - bootstrap.memory_lock=true
32             - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
33             - discovery.type=single-node
34         ports: # Expose ElasticSearch ports
35             - "9300:9300"
36             - "9200:9200"
37
38     volumes:
39         esdata:
```

1.5 Add DockerFile

We are using official prebuilt images for Nginx and Elasticsearch, but we'll need to build our own image for the Node.js app.

Define a simple Dockerfile configuration in the application root directory.

```
1  # Use Node v8.9.0 LTS
2  FROM node:carbon
3
4  # Setup app working directory
5  WORKDIR /usr/src/app
6
7  # Copy package.json and package-lock.json
8  COPY package*.json ./
9
10 # Install app dependencies
11 RUN npm install
12
13 # Copy sourcecode
14 COPY . .
15
16 # Start app
17 CMD [ "npm", "start" ]
```

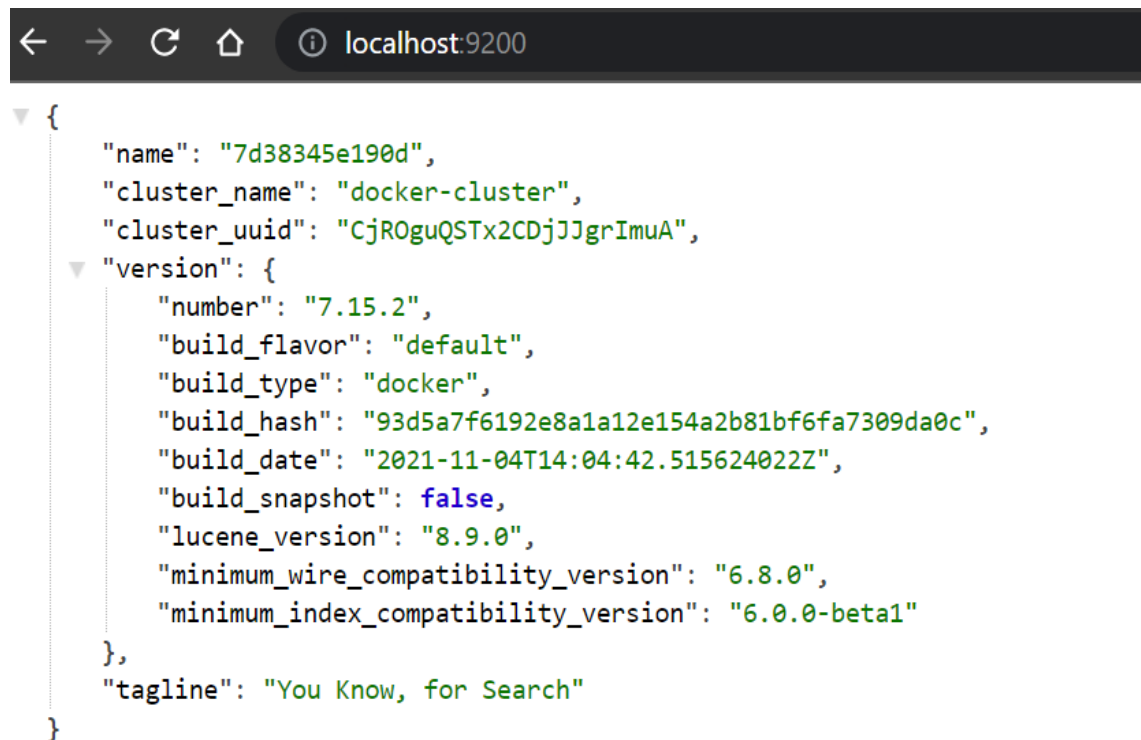
1.6 Add Base Files

Add the placeholder Node.js app file at server/app.js.

```
1  const Koa = require('koa')
2  const Router = require('koa-router')
3  const joi = require('joi')
4  const validate = require('koa-joi-validate')
5  const search = require('./search')
6  const app = new Koa()
7  const router = new Router()
8
9  // Log each request to the console
10 app.use(async (ctx, next) => {
11   const start = Date.now()
12   await next()
13   const ms = Date.now() - start
14   console.log(`${ctx.method} ${ctx.url} - ${ms}`)
15 })
16
17 // Log percolated errors to the console
18 app.on('error', err => {
19   console.error('Server Error', err)
20 })
21
22 // Set permissive CORS header
23 app.use(async (ctx, next) => {
24   ctx.set('Access-Control-Allow-Origin', '*')
25   return next()
26 })
27
```

1.7 Try to run Docker

Running localhost:9200 to verify that Elasticsearch is running.

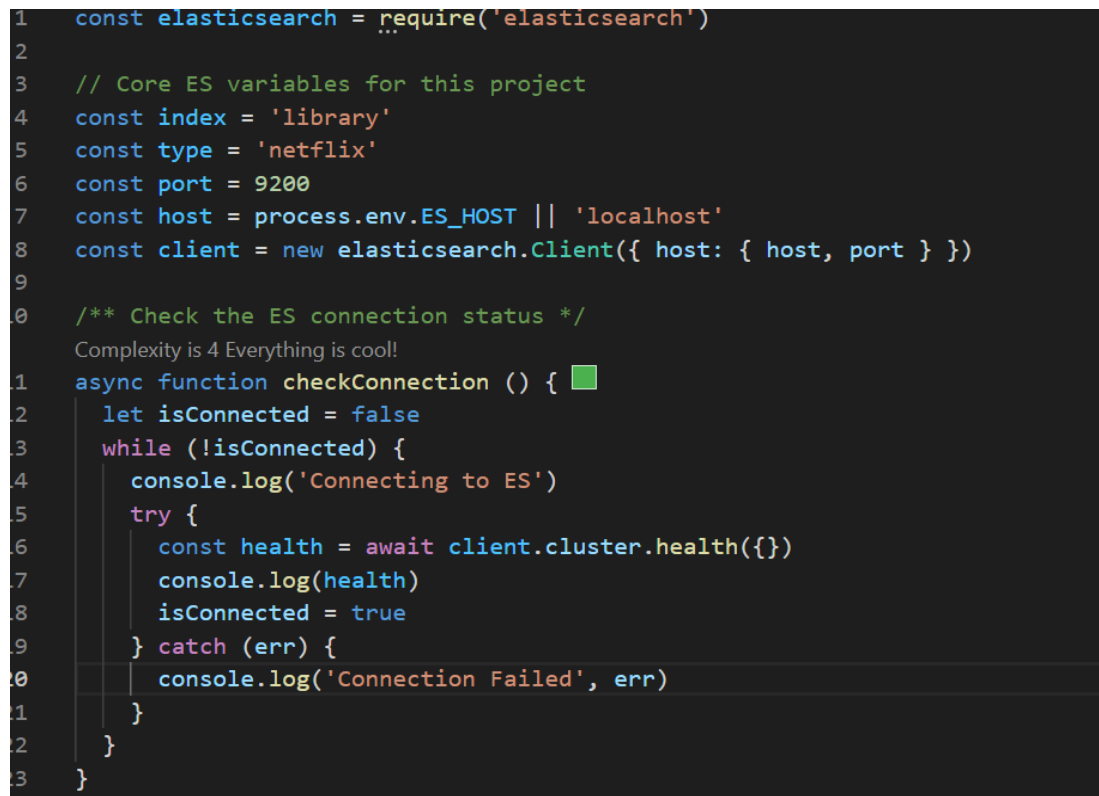


```
{
  "name": "7d38345e190d",
  "cluster_name": "docker-cluster",
  "cluster_uuid": "CjROguQSTx2CDjJJgrImuA",
  "version": {
    "number": "7.15.2",
    "build_flavor": "default",
    "build_type": "docker",
    "build_hash": "93d5a7f6192e8a1a12e154a2b81bf6fa7309da0c",
    "build_date": "2021-11-04T14:04:42.515624022Z",
    "build_snapshot": false,
    "lucene_version": "8.9.0",
    "minimum_wire_compatibility_version": "6.8.0",
    "minimum_index_compatibility_version": "6.0.0-beta1"
  },
  "tagline": "You Know, for Search"
}
```

1.8 Connect to Elasticsearch

Add ES Connection Module:

Add the following Elasticsearch initialization code to a new file server/connection.js.



```
1  const elasticsearch = require('elasticsearch')
2
3  // Core ES variables for this project
4  const index = 'library'
5  const type = 'netflix'
6  const port = 9200
7  const host = process.env.ES_HOST || 'localhost'
8  const client = new elasticsearch.Client({ host: { host, port } })
9
10 /** Check the ES connection status */
11 Complexity is 4 Everything is cool!
12 async function checkConnection () {
13   let isConnected = false
14   while (!isConnected) {
15     console.log('Connecting to ES')
16     try {
17       const health = await client.cluster.health({})
18       console.log(health)
19       isConnected = true
20     } catch (err) {
21       console.log('Connection Failed', err)
22     }
23   }
24 }
```

Let's rebuild our Node app now that we've made changes, using docker-compose build. Next, run docker-compose up -d to start the application stack as a background daemon process.

With the app started, run docker exec gs-api "node" "server/connection.js" on the command line to run a script within the container. You should see some system output like the following.

```
{ cluster_name: 'docker-cluster',
  status: 'yellow',
  timed_out: false,
  number_of_nodes: 1,
  number_of_data_nodes: 1,
  active_primary_shards: 1,
  active_shards: 1,
  relocating_shards: 0,
  initializing_shards: 0,
  unassigned_shards: 1,
  delayed_unassigned_shards: 0,
  number_of_pending_tasks: 0,
  number_of_in_flight_fetch: 0,
  task_max_waiting_in_queue_millis: 0,
  active_shards_percent_as_number: 50 }
```

1.9 Add helper function to Reset Index

In server/connection.js add the following function below checkConnection, to provide an easy way.

```
/** Clear the index, recreate it, and add mappings */
async function resetIndex () {
  if (await client.indices.exists({ index })) {
    await client.indices.delete({ index })
  }

  await client.indices.create({ index })
  await putBookMapping()
}
```

1.10 Add netflix schema

Add a "mapping" for the book data schema. Add the following function below resetIndex in server/connection.js.

```

/** Add netflix section schema mapping to ES */
async function putNetflixMapping () {
  const schema = {
    show_id: {Integer},
    type: {type: 'keyword'},
    title: { type: 'keyword' },
    director: { type: 'keyword' },
    cast: { type: 'keyword' },
    country: {type: 'keyword'},
    date_added: {Integer},
    releaseYear: {Integer},
    genre: {type: 'keyword'},
    description: { type: 'text' }
  }

  return client.indices.putMapping({ index, type, body: { properties: schema } })
}

module.exports = {
  client, index, type, checkConnection, resetIndex
}

```

1.10 Load the data

We will be using Netflix data for our project.

1.12 Downloading the data

We will be downloading the “netflix.csv” file from Kaggle. Extract this file into the /netflix directory into our project.

1.13 Read Data Dir

We will be writing a script to read the content of each book and to add that data to Elasticsearch. We'll define a new Javascript file server/load_data.js to perform these operations.

First, we'll obtain a list of every file within the netflix/ data directory.

server/load_data.js:

```

async function readAndInsert () {
  await esConnection.checkConnection()

  try {
    // Clear previous ES index
    await esConnection.resetIndex()

    // Read netflix directory
    let files = fs.readdirSync('./netflix').filter(file => file.slice(-4) === '.txt')
    console.log(`Found ${files.length} Files`)

    // Read each netflix file, and index each paragraph in elasticsearch
    for (let file of files) {
      console.log(`Reading File - ${file}`)
      const filePath = path.join('./netflix', file)
      const { show_id, type, title, director, cast, country, date_added, ReleaseYear, description } = parseBook
      await insertBookData(show_id, type, title, director, cast, country, date_added, ReleaseYear, genre, descrip
    }
  } catch (err) {
    console.error(err)
  }
}

readAndInsertBooks()

```

1.14 Index datafile in ES

insertNetflixData function to load_data.js

```
async function insertNetflixData (show_id,type,title, director, cast, country,date_added,ReleaseYear,genre) {
  let bulkOps = [] // Array to store bulk operations

  // Add an index operation for each section in the book
  for (let i = 0; i < paragraphs.length; i++) {
    // Describe action
    bulkOps.push({ index: { _index: esConnection.index, _type: esConnection.type } })

    // Add document
    bulkOps.push({
      show_id,
      type,
      director,
      title,
      cast,
      country,
      date_added,
      ReleaseYear,
      genre,
      text: description[i]
    })
  }

  if (i > 0 && i % 500 === 0) { // Do bulk insert after every 500 paragraphs
    await esConnection.client.bulk({ body: bulkOps })
    bulkOps = []
    console.log(`Indexed Paragraphs ${i - 499} - ${i}`)
  }
}
```

1.15 API Server

server/app.js

```
1  const Koa = require('koa')
2  const Router = require('koa-router')
3  const joi = require('joi')
4  const validate = require('koa-joi-validate')
5  const search = require('./search')
6  const app = new Koa()
7  const router = new Router()
8
9  // Log each request to the console
10 app.use(async (ctx, next) => {
11   const start = Date.now()
12   await next()
13   const ms = Date.now() - start
14   console.log(`${ctx.method} ${ctx.url} - ${ms}`)
15 })
16
17 // Log percolated errors to the console
18 app.on('error', err => {
19   console.error('Server Error', err)
20 })
21
22 // Set permissive CORS header
23 app.use(async (ctx, next) => {
24   ctx.set('Access-Control-Allow-Origin', '*')
25   return next()
26 })
27
```



```
const port = process.env.PORT || 3000

app
  .use(router.routes())
  .use(router.allowedMethods())
  .listen(port, err => {
    if (err) console.error(err)
    console.log(`App Listening on Port ${port}`)
  })
```

This code will import our server dependencies and set up simple logging and error handling for a Koa.js Node API server

1.16 Link endpoint with queries

Next, we'll add an endpoint to our server in order to expose our Elasticsearch query function.

server/app.js

```
router.get('/search',
  async (ctx, next) => {
    const { term, offset } = ctx.request.query
    ctx.body = await search.queryTerm(term, offset)
  }
)
```

1.17 Input validation This endpoint is still not complete- We'll add some middleware to the endpoint in order to validate input parameters using Joi and the Koa-Joi-Validate library.

```
router.get('/search',
  validate({
    query: {
      term: joi.string().max(60).required(),
      offset: joi.number().integer().min(0).default(0)
    }
  }),
  async (ctx, next) => {
    const { term, offset } = ctx.request.query
    ctx.body = await search.queryTerm(term, offset)
  }
)
```

Now, if you restart the server and make a request with a missing term(<http://localhost:3000/search>), you will get back an HTTP 400 error with a relevant message, such as Invalid URL Query - child "term" fails because ["term" is required].

To view live logs from the Node app, you can run `docker-compose logs -f api`.

1.18 Front End Application

Now that our `/search` endpoint is in place, let's wire up a simple web app to test out the API.

6.0 - Vue.js App

We'll be using Vue.js to coordinate our frontend.

Add a new file, `/public/app.js`, to hold our Vue.js application code.

```
const vm = new Vue ({
  el: '#vue-instance',
  data () {
    return {
      baseUrl: 'http://localhost:3000', // API url
      searchTerm: 'Hello World', // Default search term
      searchDebounce: null, // Timeout for search bar debounce
      searchResults: [], // Displayed search results
      numHits: null, // Total search results found
      searchOffset: 0, // Search result pagination offset

      selectedParagraph: null, // Selected paragraph object
      bookOffset: 0, // Offset for book paragraphs being displayed
      paragraphs: [] // Paragraphs being displayed in book preview window
    }
  },
  async created () {
    this.searchResults = await this.search() // Search for default term
  },
  methods: {
    /** Debounce search input by 100 ms */
    onSearchInput () {
      clearTimeout(this.searchDebounce)
      this.searchDebounce = setTimeout(async () => {
        this.searchOffset = 0
        this.searchResults = await this.search()
      }, 100)
    },

    /** Call API to search for inputted term */
    async search () {
      const response = await axios.get(`${this.baseUrl}/search`, { params: { term: this.searchTerm, offset: this.searchOffset } })
      this.numHits = response.data.hits.total
      return response.data.hits.hits
    },

    /** Get next page of search results */
    async nextResultsPage () {
      if (this.numHits > 10) {
        this.searchOffset += 10
        if (this.searchOffset + 10 > this.numHits) { this.searchOffset = this.numHits - 10 }
        this.searchResults = await this.search()
        document.documentElement.scrollTop = 0
      }
    },

    /** Get previous page of search results */
    async prevResultsPage () {
      this.searchOffset -= 10
      if (this.searchOffset < 0) { this.searchOffset = 0 }
      this.searchResults = await this.search()
      document.documentElement.scrollTop = 0
    }
  }
})
```

We're just defining some shared data properties, and adding methods to retrieve and paginate through search results. The search input is debounced by 100ms, to prevent the API from being called with every keystroke.

1.19 HTML

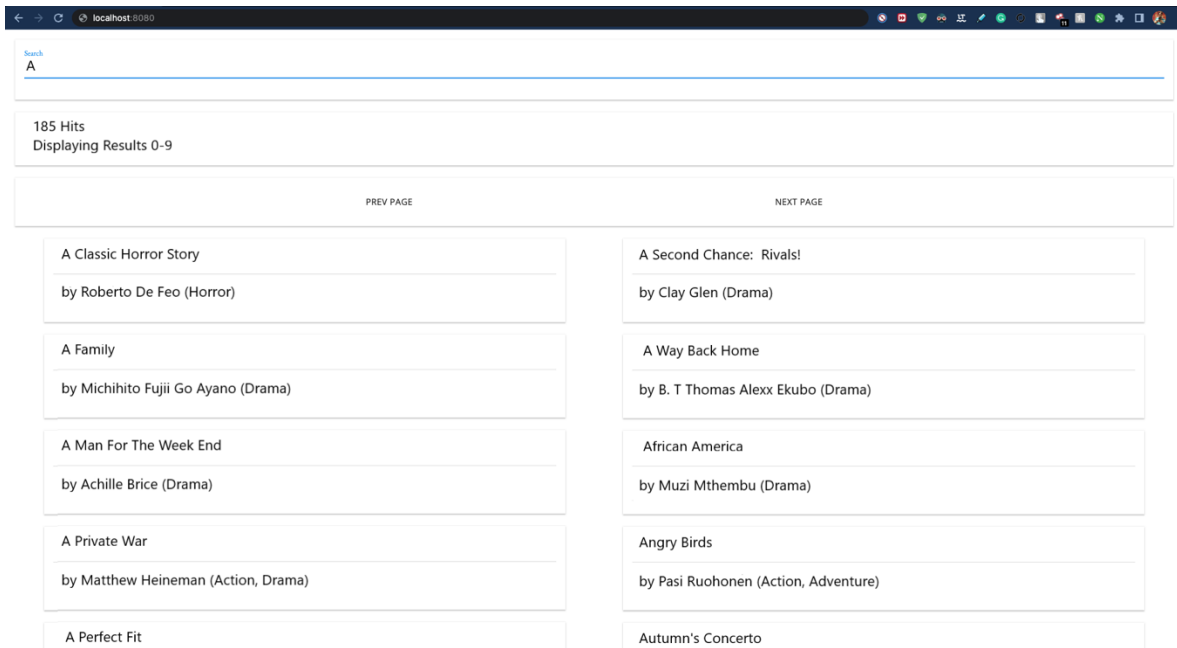
Replace our placeholder `/public/index.html` file with the following contents, in order to load our Vue.js app and to lay out a basic search interface.

CSS

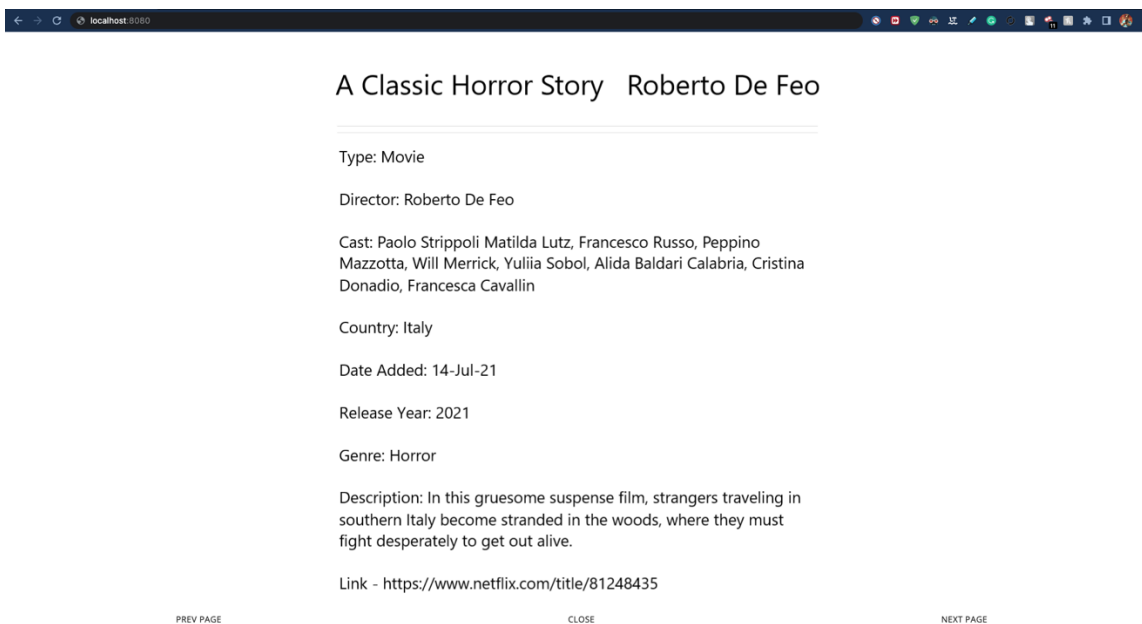
Run the app

Open `localhost:8080` in your web browser, you should see a simple search interface with paginated results. Try typing in the top search bar to find matches from different terms.

Case 1 – When a user entered a character A, he/she/they can saw the relevant result.



Case 2 – When a user clicks on a relevant result tile, he/she/they will be further redirected to the more information page.



Once you have docker installed, we can run ElasticSearch by the following command:

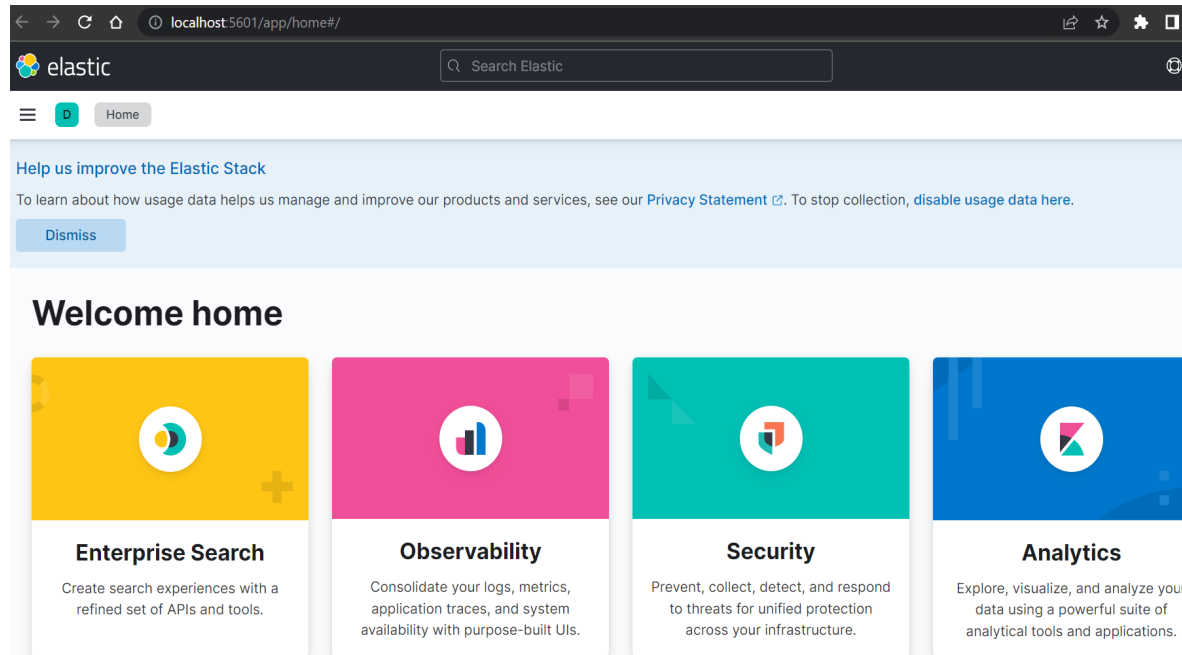
```
docker network create elastic
docker pull docker.elastic.co/elasticsearch/elasticsearch:7.15.2
docker run --name es01-test --net elastic -p 127.0.0.1:9200:9200 -p 127.0.0.1:9300:9300 -e "discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:7.15.2.
```

Let's Connect Kibana now:

```
docker pull docker.elastic.co/kibana/kibana:7.15.2
docker run --name kib01-test --net elastic -p 127.0.0.1:5601:5601 -e "ELASTICSEARCH_HOSTS=http://es01-test:9200" docker.elastic.co/kibana/kibana:7.15.2
```

If the above commands are running then we should be able to access the following ElasticSearch URL.

<http://localhost:5601/app/home#/>



Now let's try to analyze our Netflix data using Kibana

netflix_titles.csv

File contents

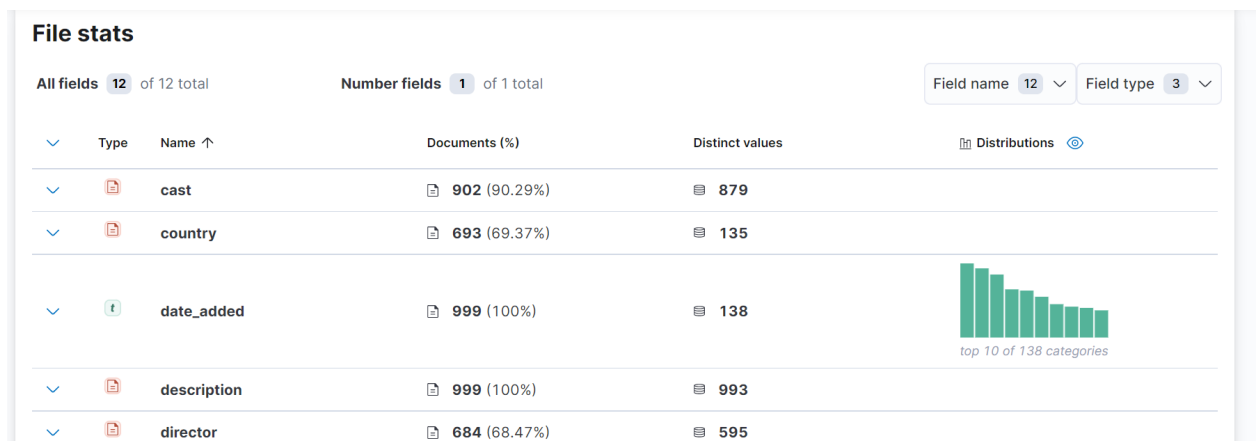
First 1,000 lines

```
5 s4,TV Show,Jailbirds New Orleans,,,,,"September 24, 2021",2021,TV-MA,1 Season,"Docuseries, Reality TV","Feuds, flirtations and toilet talk go down among the
6 s5,TV Show,Kota Factory,,Mayur More, Jitendra Kumar, Ranjan Raj, Alam Khan, Ahsaas Channa, Revathi Pillai, Urvi Singh, Arun Kumar",India,"September 24, 2021"
7 s6,TV Show,Midnight Mass,Mike Flanagan,"Kate Siegel, Zach Gilford, Hamish Linklater, Henry Thomas, Kristin Lehman, Samantha Sloan, Igby Rigney, Rahul Kohli,
8 s7,Movie,My Little Pony: A New Generation,"Robert Cullen, José Luis Ucha","Vanessa Hudgens, Kimiko Glenn, James Marsden, Sofia Carson, Liza Koshy, Ken Jeong,
9 s8,Movie,Sankofa,Haile Gerima,"Kofi Ghanaba, Oyafunmike Ogunlano, Alexandra Duah, Nick Medley, Mutabaruka, Afemo Omilami, Reggie Carter, Mzuri","United States,
```

Summary

| | |
|--------------------------|-----------|
| Number of lines analyzed | 1000 |
| Format | delimited |
| Delimiter | , |
| Has header row | true |

[Override settings](#) [Analysis explanation](#)



IV. COMPARATIVE STUDY

In this section of paper, we have mentioned differences between Elasticsearch, Solr and Algolia based on multiple parameters.

A. *ElasticSearch vs Solr*

- Analytics Engine for Solr have facets and powerful streaming aggregations while Elasticsearch has sophisticated and highly flexible aggregations.
- Solr has no Optimized Query Execution, but Elasticsearch has Faster range queries depending on the context.
- Search Speed of Solr is best for static data, because of caches and un-inverted reader. Elasticsearch has Very good for rapidly changing data, because of per-segment caches.
- Analysis Engine Performance for Solr is Great for static data with exact calculations while Elasticsearch Exactness of the results depends on data placement.
- Full text search features for Solr are Language analysis based on Lucene, multiple suggesters, spell checkers, rich highlighting support while Language analysis based on Lucene, single suggest API implementation, highlighting rescoring for Elasticsearch.
- Machine Learning for Solr is Built-in – on top of streaming aggregations focused on logistic regression and learning to rank contrib module while for Elasticsearch it is Commercial feature, focused on anomalies and outliers and time-series data

Architectural Differences

| Tools integrate with Elasticsearch | Tools integrate with Solr |
|------------------------------------|---------------------------|
| Kibana | Datadog |
| Logstash | Netdata |
| Datadog | Lucene |
| Contentful | StreamSets |

B. *ElasticSearch vs Algolia*

1. *Search Indexing* - Elasticsearch has More complex approach to index management. Requires advanced planning, expertise, and optimization. Clients must already know their search needs and how they'll evolve. Algolia is Worry free, low maintenance, and delivers performance to scale. Better organizes large data volumes (for example, product SKUs). Easy configurability helps teams fine-tune for a better search experience.

2. *Search Relevance* - Elasticsearch Algorithms are more complex, unpredictable, and harder to control. Optimizing one set of search results may hurt others, potentially cutting into revenue. It's hard to see where changes make the biggest impact. Algolia Built on transparent search rules that are simple and easy to manage. Utilizes business insights to improve accuracy of search results order. Integrates with AI to help engineers fine-tune search performance

3. *Front End* - Elasticsearch Offers only one basic UI library. Requires investment in an additional layer of security. Limited UX features limit analytical insights and opportunity for optimization. Algolia Comes prebuilt with 6 rich UI libraries. Fully customizable and features built-in security. Delivers a consumer-grade search experience.

4. *Personalization* - Elasticsearch Build-it-yourself model requires extensive coding. Implementation is always reliant on development team speed and search expertise. May take weeks for developers to build sophisticated tools. Algolia Leverages AI to create a more personalized user experience. Built-in integrations with Relevance and Analytics. Instantly implements changes up to 100x faster than alternatives.

5. *Analytics* - ElasticSearch Requires data engineering to extract information and build visualizations. Success predicated on do-it-yourself reporting tools. No business insights provided; additional tools needed. Algolia Bolsters search experience with preloaded user and performance insights. Leverages KPIs to optimize search and discovery. Surfaces opportunities to improve the search experience

Architectural Differences

| Tools integrate with ElasticSearch | Tools integrate with Algolia |
|------------------------------------|------------------------------|
| Kibana | JavaScript |
| Logstash | React |
| Datadog | Java |
| Contentful | WordPress |

V. CONCLUSION

After doing this project, we can conclude that Search is a core part of Elasticsearch. If the analysis process is defined on the index, the retrieval process would be more efficient. Elasticsearch is a good technology of big data for searching operation. It is near real time. Elasticsearch uses inverted index for search. Elasticsearch can search full text fields and most relevant result is returned. The search becomes quicker.

VI. REFERENCES

- [1] amazon OpenSearch Service Documentation [Link](#)
- [2] Amazon Web Services, Amazon Elasticsearch Service: Developer Guide, Kindle Edition
- [3] Alberto Paro, Elasticsearch 7.0 Cookbook: Over 100 recipes for fast, scalable, and reliable search for your enterprise, 4th Edition
- [4] Anurag Srivastava, Learning Elasticsearch 7.x: Index, Analyze, Search and Aggregate Your Data Using Elasticsearch, 1st Edition
- [5] Darshita Kalyani, Dr. Devarshi Mehta, Paper on Searching and Indexing Using Elasticsearch
- [6] Urvi Thacker ; Manjusha Pandey ; Siddharth S. Rautaray Electrical, Electronics, and Optimization Techniques (ICEEOT), International Conference
- [7] Use of Elastic Search for Intelligent Algorithms to Ease the Healthcare Industry C. Bhadane, H. A. Mody, D. U. Shah, P. R. Sheth
- [8] Full-Text Search on Data with Access Control Ahmad Zaky, Rinaldi Munir, S.T., M.T. School of Electrical Engineering and Informatics Institut Teknologi Bandung