Site Search Engines

A Comparative Survey

Aumüller Thomas, Liegl Daniel, and Platzer Fabian

706.041 Information Architecture and Web Usability 3VU WS 2023/2024
Graz University of Technology

23 Jan 2024

Abstract

Site search engines are used by people interacting with the internet almost daily. Users today type search queries into text fields on web sites across the World Wide Web and do not overthink about how and why it works, even less so comparing different web sites based on their provided search functionality. Three different site search engines were selected based on their provided functionality and limitations and compared according to a compiled list of criteria. To compare the setup, as well as the back-end aspect of the three site search engines, a web site was created to not only provide the option to explore and inspect the search functionality but also to open up the possibility of comparing the site search engines in a direct side-by-side manner. This approach revealed numerous differences between the site search engines, for example, regarding indexing, security, the activity of their communities, as well as their offered search functionality.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

A number of possibilities are available for web site owners to include local site search into their web site, enabling end users to search for content on the site. One of those is to integrate a site search engine (SSE) into the web site.

There are many SSEs to choose from, both free and commercial, each providing a wide and almost identical feature pallete with a multitude of options for the user. This survey explored and compared three such SSEs: Typesense, OpenSearch, and Algolia. A list of criteria was created as a basis to assess and compare the SSEs.

In order to gain experience and directly observe similarities and differences with each of the chosen three SSEs, a self-hosted website was created with Metalsmith. The dataset of choice is the "Movies and TV shows on Netflix" dataset from Bansal [2021], which provides a sufficient number of data records and is from an everyday setting. This dataset was individually indexed for each SSE. In the frontend, the user can type queries in a single search box and the matching results for each SSE are displayed in three columns side by side for the user to see.

Self-hosting an SSE on one's own hardware has some advantages and disadvantages, as can be seen in Table 1.1. Alternatively, a cloud-based hosting provider can be used. Again, there are advantages and disadvantages, as can be seen in Table 1.2.

The three SSEs covered in this survey are:

- *Typesense*: Typesense is an open-source and typo-tolerant SSE [Typesense 2023c]. Typesense was chosen as the first open-source and self-hosted option, because it provides a simple developer-friendly setup and comes with a wide array of features and plugins.
- *OpenSearch*: OpenSearch is a community-driven and open-source SSE, that not only provides search capabilities but also some analytical functionality [OpenSearch 2023d]. It is the second open-source and self-hosted SSE in the survey.
- *Algolia*: Algolia is a commercial, cloud-based SSE [Algolia 2023]. A free trial version of Algolia was used for the comparison. Algolia provides a powerful search implementation with numerous features and a variety of sophisticated search analysis tools. It hides a great deal of the complexity behind its web interface and is easy to setup.

Positive Aspects	Negative Aspects
Open-source.	Own server needed.
More control.	Complex setup.
	Heavy maintenance.

Table 1.1: Positive and negative aspects of self-hosted SSEs.

Positive Aspects	Negative Aspects
No hardware needed.	Lack of control.
Fewer limitations (e.g. computational power).	Data privacy.
Less maintenance.	Network bandwidth limits.
Hidden complexity.	

Table 1.2: Positive and negative aspects of cloud-based SSEs.

Chapter 2

Backend

This chapter describes how the backend setup of the three site search engines, including the dataset used and how it was prepared and indexed for each of the SSEs.

2.1 Self-Hosted SSE Setup

Typesense and OpenSearch were the two Self-Hosted SSEs of choice. An Apache 2.0 web server and the servers for OpenSearch and Typesense were installed on a Thinkpad W540 with an Intel i7-4600M (4) @ 3.600GHz and 16 gigabytes of RAM. This machine is running Ubuntu 22.04.3 LTS x86_64.

To make the process of setting up the backend easier, Docker was chosen as the platform of choice:

> "Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host." [Docker 2023]

These properties and the fact that Docker Compose templates are available for OpenSearch (refer to OpenSearch [2023b]) and Typesense (refer to Typesense [2023b]), made it easy to setup the servers without problems.

2.1.1 Typesense Self-Hosting

Typesense can be setup on many different platforms, DEB, RPM and prebuilt binaries can be found on their downloads page, which can be found in Typesense [2023a]. The platforms supported by Typesense are:

- Docker
- Kubernetes
- macOS (Binary and Homebrew)
- Ubuntu/Debian (deb package)
- CentOS/RHEL (rpm package)
- Windows (within WSL)

The Docker Compose template provided for Typesense (refer to Listing 2.1) creates a simple setup, to start working with Docker immediately.

```
version: '3.4'
services:
  typesense:
    image: typesense/typesense:0.25.1
    restart: on-failure
    ports:
      - "8108:8108"
    volumes:
      - ./typesense-data:/data
    command: '--data-dir /data --api-key=xyz --enable-cors'
```

Listing 2.1: Typesense Docker compose template.

```
- http.cors.allow-origin:"http://<yourdomain>"
- http.cors.enabled:true
- http.cors.allow-headers:X-Requested-With,X-Auth-Token,Content-Type,Content-Length,
    Authorization
- http.cors.allow-credentials:true
```

Listing 2.2: OpenSearch settings to enable CORS

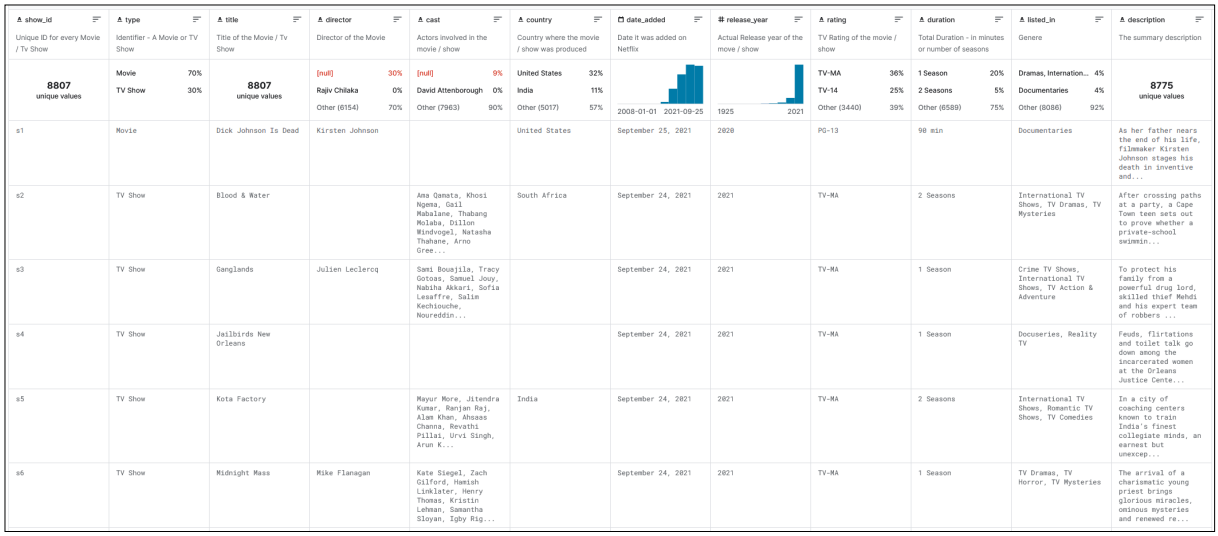
2.1.2 OpenSearch Self-Hosting

OpenSearch was setup using their Docker Compose template, made available in their documentation (refer to OpenSearch [2023b]). The platforms supported by OpenSearch are:

- RHEL/CentOS
- Rocky Linux
- CentOS/RHEL
- Windows Server 2019

The Docker Compose template creates a setup with two nodes running OpenSearch and includes the OpenSearch Dashboard. One thing that needed setup from our side was Cross-Origin-Resource-Sharing (CORS). “CORS is a part of HTTP that lets servers specify any other hosts from which a browser should permit loading of content.” [MDN 2023]. Due to CORS not being enabled by default, no search requests could be made. The settings shown in Listing 2.2 had to be changed to enable CORS and to start working with OpenSearch. This problem could have been prevented if a proper rerouting setup for the API calls was made by using Nginx as a proxy server. This way API calls do not interfere with the same-origin policy [F5 2023].

During the survey work, there were three incidents regarding OpenSearch where the index data was lost without apparent reason. It is our theory, that it happened due to enabling CORS and OpenSearch not requiring an API key or any other authentication out of the box. When using OpenSearch, security settings have to be enabled and set up by the backend engineer [OpenSearch 2023a].



show_id	type	title	director	cast	country	date_added	release_year	rating	duration	listed_in	description
Unique ID for every Movie / Tv Show	Identifier - A Movie or TV Show	Title of the Movie / Tv Show	Director of the Movie	Actors involved in the movie / show	Country where the movie / show was produced	Date it was added on Netflix	Actual Release year of the move / show	TV Rating of the movie / show	Total Duration - in minutes or number of seasons	Genere	The summary description
8807 unique values	Movie 70% TV Show 30%	8807 unique values	[null] 30% Rajiv Chilaka 0% Other (6154) 70%	[null] 9% David Attenborough 0% Other (7963) 90%	United States 32% India 11% Other (5017) 57%	2008-01-01 2021-09-25	1925 2021	TV-MA 36% TV-14 25% Other (3440) 39%	1 Season 20% 2 Seasons 5% Other (6589) 75%	Dramas, Internation... 4% Documentaries 4% Other (8086) 92%	8775 unique values
s1	Movie	Dick Johnson Is Dead	Kirsten Johnson		United States	September 25, 2021	2020	PG-13	90 min	Documentaries	As her father nears the end of his life, filmmaker Kirsten Johnson stages his death in inventive and...
s2	TV Show	Blood & Water		Ama Qamata, Khosi Ngema, Gail Mabalane, Thabang Molaba, Dillon Windvogel, Natasha Thahane, Arno Gree...	South Africa	September 24, 2021	2021	TV-MA	2 Seasons	International TV Shows, TV Dramas, TV Mysteries	After crossing paths at a party, a Cape Town teen sets out to prove whether a private-school swimmin...
s3	TV Show	Ganglands	Julien Leclercq	Sami Bouajila, Tracy Gotoas, Samuel Jouy, Nabiha Akkari, Sofia Lesaffre, Salim Kechiouche, Noureddin...		September 24, 2021	2021	TV-MA	1 Season	Crime TV Shows, International TV Shows, TV Action & Adventure	To protect his family from a powerful drug lord, skilled thief Mehdi and his expert team of robbers ...
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 incarcerated women at the Orleans Justice Cente...
s5	TV Show	Kota Factory		Mayur More, Jitendra Kumar, Ranjan Raj, Alam Khan, Ahsaas Channa, Revathi Pillai, Urvi Singh, Arun K...	India	September 24, 2021	2021	TV-MA	2 Seasons	International TV Shows, Romantic TV Shows, TV Comedies	In a city of coaching centers known to train India's finest collegiate minds, an earnest but unexcep...
s6	TV Show	Midnight Mass	Mike Flanagan	Kate Siegel, Zach Gilford, Hamish Linklater, Henry Thomas, Kristin Lehman, Samantha Sloyan, Igby Rig...		September 24, 2021	2021	TV-MA	1 Season	TV Dramas, TV Horror, TV Mysteries	The arrival of a charismatic young priest brings glorious miracles, ominous mysteries and renewed re...

Figure 2.1: Dataset: Movies and TV shows on Netflix, `netflix_titles.csv` (3.4 MB) from Bansal [2021].

2.2 Dataset

An important part of the survey website was the dataset used to compare the individual SSEs. Without data, comparing the search and indexing aspects of our criteria sections and showcasing the differences in a meaningful way would not be possible.

Web sites like Huggingface [2023] or Kaggle [2023] offer a vast array of datasets to choose from with the "License CC0: Public Domain", indicating that the creator of this dataset placed the work in the public domain. Thus, it is possible to copy, modify, or even use the dataset for commercial purposes, without asking for permission from the creator.

The chosen dataset contains metedata about movies and TV shows on Netflix comprising roughly 3.4 MB of data [Bansal 2021]. These listings are provided in a comma-separated file format and consist of 12 columns and 8807 rows, as can be seen in Figure 2.1. Movies and TV shows are encountered by most people in their daily lives, and hence need little further explanation.

2.3 Indexing

An index is a database containing metadata about a website. This data must be generally be provided by the owner of the site, for example a list of products. Algolia also provides built-in site crawling, which creates this metadata for a site. Indexing for OpenSearch and Typesense was done using Python 3.11.6 running on Windows 10 Home Version 22H2 and the `opensearchpy` and `typesense` packages.

2.3.1 Indexing for Typesense

The index for Typesense was created using the `typesense` Python package, as shown in Listing 2.3. The data was read and turned into a Pandas `DataFrame`. Each of the rows was first converted to a dictionary, and then converted to the JSON Lines format [Ward 2023].

Afterwards an instance of the `typesense.Client` class was created. It is required to pass the admin API key, the host name, and the port of the Typesense server. As can be seen in line 18 of Listing 2.3, it is possible to use multiple Typesense server nodes at the same time. SSL is also supported.

The index (in Typesense these are called collections) is then created using the `create()` method in line 26. It is required to define all fields of the documents in advance. Using the `retrieve()` method, we tested if the index works as intended.

To add the documents, the `import_()` function was used in line 50. In comparison to OpenSearch, the action has to be defined for all the documents passed to the function. Different actions cannot be created for single documents in the request. Instead of the `create` action, `upsert` can be used to update if the document exists, or create a new document, if it does not.

2.3.2 Indexing for OpenSearch

Indexing for OpenSearch was done using the `opensearchpy` Python package. First, the client was created using the OpenSearch class, as shown in Listing 2.5. This class contains all the functionality needed to interface with the OpenSearch server. When creating the client, the host name and port must be provided. SSL encryption is supported by this client, but was not used during our survey. Afterwards, the CSV file is read using Pandas' `read_csv()` function and a Pandas DataFrame is created [pandas 2020].

Once imported, the DataFrame needs to be converted into the format required by the bulk import function of OpenSearch. This format consists of two JSON objects. The first contains the action, in our case `index`, the index the document belongs to (`_index`) and a unique identifier for the document (`_id`). Other actions which can be used include `create`, `update`, and `delete`. The second JSON object contains the actual data of the document. The backend engineer can insert fields as required into this object. An example of these JSON objects can be found in Listing 2.4. There are probably better ways to create the JSON string needed for this API call.

After creating the JSON string, the last thing to do is to make the API call using the `OpenSearch.bulk()` method. This is just a matter of passing the string as an argument and checking the response for any errors.

2.3.3 Indexing for Algolia

Algolia provides a web interface for indexing, which is the most convenient way of indexing of the three SSEs discussed. When logged in to the web-interface, navigate to "Data sources" → "Indices" to access the Indices interface. After clicking "Create Index", type in a name for the index, and the the index is created. Data can be added either manually or by uploading a CSV or JSON file, as shown in Figure 2.2. There is no need to define the data fields in advance.

At first, uploading data as JSON was attempted. However, the trial license provided by Algolia only included a limited number of API calls, so this did not succeed. However, uploading a CSV file containing the Netflix movie data worked instantly. Once data has been uploaded, the web interface allows the index to be browsed and entries can be manually added, edited, and removed, as shown in Figure 2.3.

```
import pandas as pd
import jsonlines
import typesense
import os

df = pd.read_csv("netflix_processed.csv")
df = df.astype(str)
json_out = []

for index, row in df.iterrows():
  json_out.append(row.to_dict())

with jsonlines.open('typesense_documents.jsonl', 'w') as writer:
  writer.write_all(json_out)

client = typesense.Client({
  'api_key': 'Your Admin API Key',
  'nodes': [{
    'host': "Your Hostname",
    "port": "8108",
    "protocol": "http"
  }],
  'connection_timeout_seconds': 2
})

create_response = client.collections.create({
  "name": "netflix",
  "fields": [
    {"name": "id", "type": "string", "index": True},
    {"name": "type", "type": "string"},
    {"name": "title", "type": "string"},
    {"name": "director", "type": "string"},
    {"name": "cast", "type": "string"},
    {"name": "country", "type": "string"},
    {"name": "date_added", "type": "string"},
    {"name": "release_year", "type": "string"},
    {"name": "rating", "type": "string"},
    {"name": "duration", "type": "string"},
    {"name": "listed_in", "type": "string"},
    {"name": "description", "type": "string"},
    {"name": "url", "type": "string"},
    {"name": "url_title", "type": "string"}
  ],
})

retrieve_response = client.collections['netflix'].retrieve()

with open("typesense_documents.jsonl", encoding="utf-8") as jsonl_file:
  response = client.collections['netflix'].documents
    .import_(jsonl_file.read(), {"action": "create"})
  print(response)
```

Listing 2.3: Indexing for Typesense using the `typesense` package in Python.

```
{"index" : { "_index" : "netflix", "_id" : "5940" } }
{
  "title":"Breaking Bad",
  "description":"A high school chemistry teacher dying of cancer teams with a
    former student to secure his family's future by manufacturing and
    selling crystal meth.",
  "cast": "Bryan Cranston, Aaron Paul, Anna Gunn, Dean Norris, Betsy Brandt,
    R.J. Mitte, Bob Odenkirk, Steven Michael Quezada, Jonathan Banks,
    Giancarlo Esposito",
  "listed_in":"Crime TV Shows, TV Dramas, TV Thrillers"
}
```

Listing 2.4: Example of the two JSON objects needed for the bulk operation of OpenSearch.

```
from opensearchpy import OpenSearch
import pandas as pd

client = OpenSearch(
    hosts = [{'host': "Your Hostname", 'port': 9200}],
    http_compress = True,
    use_ssl = False,
)

df = pd.read_csv("./netflix_processed.csv", index_col=0)

data = ""

#convert the dataframe to the JSON format required by OpenSearch
for index, row in df.iterrows():
  description = row['description'].replace("\n", " ")
  description = description.replace('"', "'")
  cast = row["cast"].replace('"', "'")
  director = row["director"].replace('"', "'")
  title = row['title'].replace("\n", " ")
  title = title.replace('"', "'")
  data = data + '''{{"index" : {{ "_index" : "netflix", "_id" : "{}" }} }}
  {{"cast": "{}", "country":"{}", "date_added":"{}", "description":"{}",
    "director":"{}", "duration":"{}", "listed_in":"{}", "rating":"{}",
    "release_year":"{}", "title":"{}", "type":"{}", "url":"{}",
    "url_title":"{}"}}\n'''
      .format(index, cast, row["country"], row["date_added"], description,
        director, row["duration"], row['listed_in'], row['rating'],
        row['release_year'], title, row['type'], row['url'], row['url_title']
      )

response = client.bulk(data, timeout=100)
print(response)
```

Listing 2.5: Indexing for OpenSearch using the `opensearchpy` package in Python.

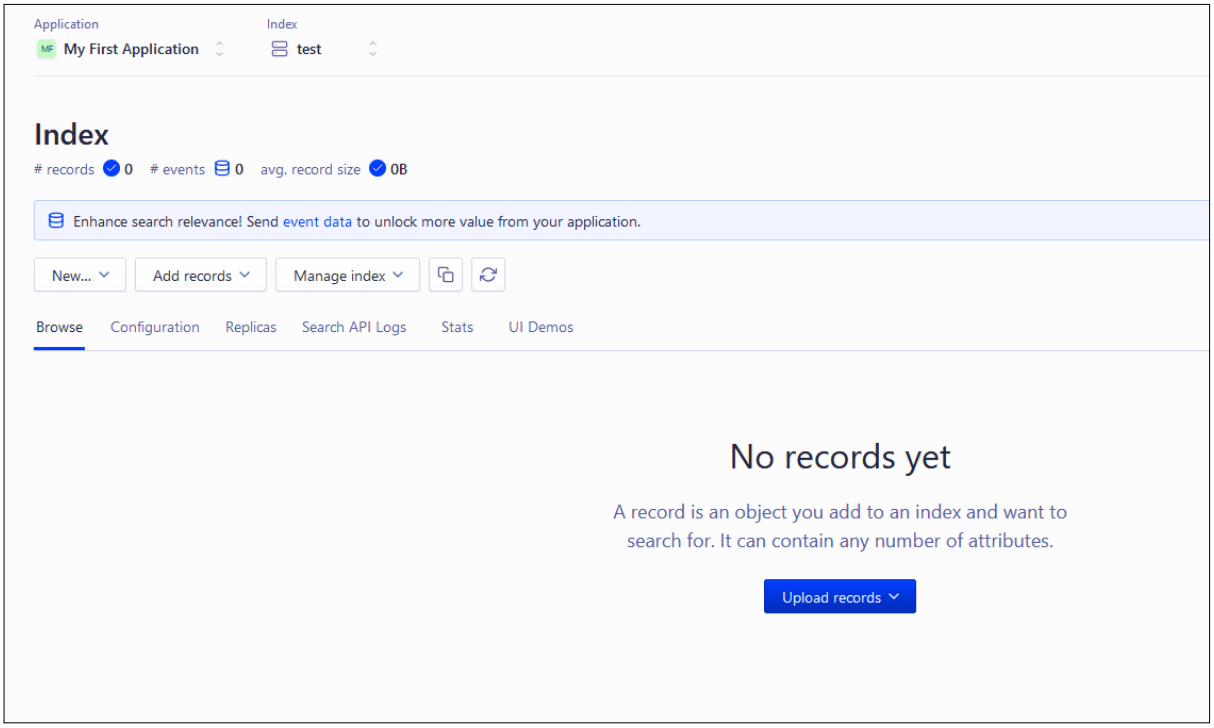


Figure 2.2: Algolia indexing interface.

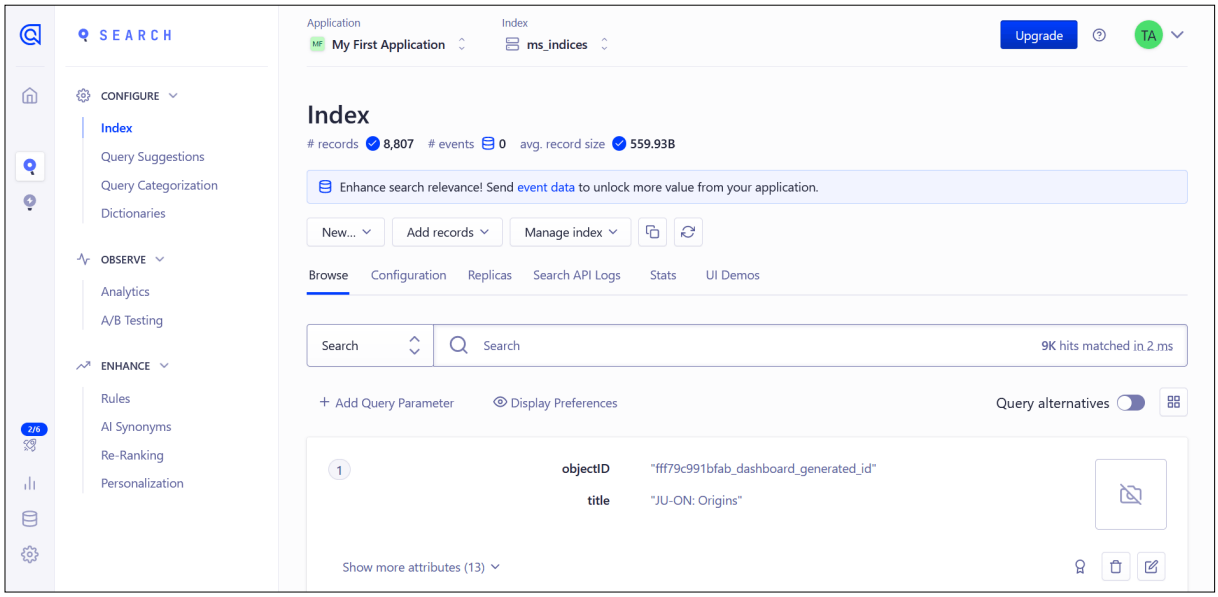


Figure 2.3: Algolia indexing interface after adding documents to the index.

Chapter 3

Frontend

In order to showcase the different SSEs, a web site with a frontend was needed. The site is hosted on an Apache 2.0 web server and is generated using the Metalsmith static site generator. Metalsmith is a simple, pluggable static site generator built in JavaScript. It is designed to be minimalistic and highly flexible, allowing developers to create static websites by chaining together various plugins to process and transform files. Specifically, the bare-bones starter by Werner Glinka was used for this project. The Nunjucks templating engine was used to construct the individual HTML pages.

For the integration of the SSEs into the frontend, a single search bar was used. A search query can be entered, and is then sent to all three SSEs at the same time. The resulting JSONs from the SSEs were converted into HTML to show them on the web page, as shown in Figure 3.1.

3.1 Integration of Typesense

Integrating Typesense turned out to be relatively easy. The `<script>` element was used to include the necessary library, as shown in Listing 3.1, and then the backend was queried using the respective request function, as shown in Listing 3.2. The resulting JSON was converted into HTML and displayed, as shown in Listing 3.3.

3.2 Integration of OpenSearch

Integrating OpenSearch turned out to be the most challenging out of all three SSEs. It only has a node.js library and does not support vanilla JavaScript. Hence, a manual `fetch()` query had to be written, as shown in Listing 3.4. Then, JavaScript was used to convert the resulting JSON into HTML for display, as shown in Listing 3.5.

3.3 Integration of Algolia

Integrating Algolia turned out to be similar to Typesense. The `<script>` element was used to include the necessary library, as shown in Listing 3.6, Then, the backend was queried using the respective request function, as shown in Listing 3.7. Finally, the JSON results were converted into HTML for display, as shown in Listing 3.8.

```
<head>
<script src="https://cdn.jsdelivr.net/npm/typesense@1/dist/typesense.min.js">
</script>
// ... (rest of your scripts)
</head>
```

Listing 3.1: Typesense integration into the web page template.

```
var typesense = new Typesense.Client({
  'nodes': [
      {
      'host': 'YourHostname',
      'port': '8108',
      'protocol': 'http'
      },
  ],
  'apiKey': 'Your Search-Only API Key',
  'numRetries': 3, // A total of 4 tries (1 original try + 3 retries)
  'connectionTimeoutSeconds': 10,
  'retryIntervalSeconds': 0.1,
  'healthcheckIntervalSeconds': 2,
  'logLevel': 'debug'
})

function searchTypesense (searchTerm) {
  typesense.collections('netflix').documents().search({
      'q': searchTerm,
      'query_by': ["title", "director", "cast", "description", "listed_in",
                   "country", "release_year", "rating", "duration"]
  }).then(function (searchResults) {
      this.createHTMLFromJSON(searchResults.hits)
  }).catch(function (error) {
      document.getElementById('typesense-results').innerHTML = error
  })
}
```

Listing 3.2: Typesense search query using the JavaScript client.

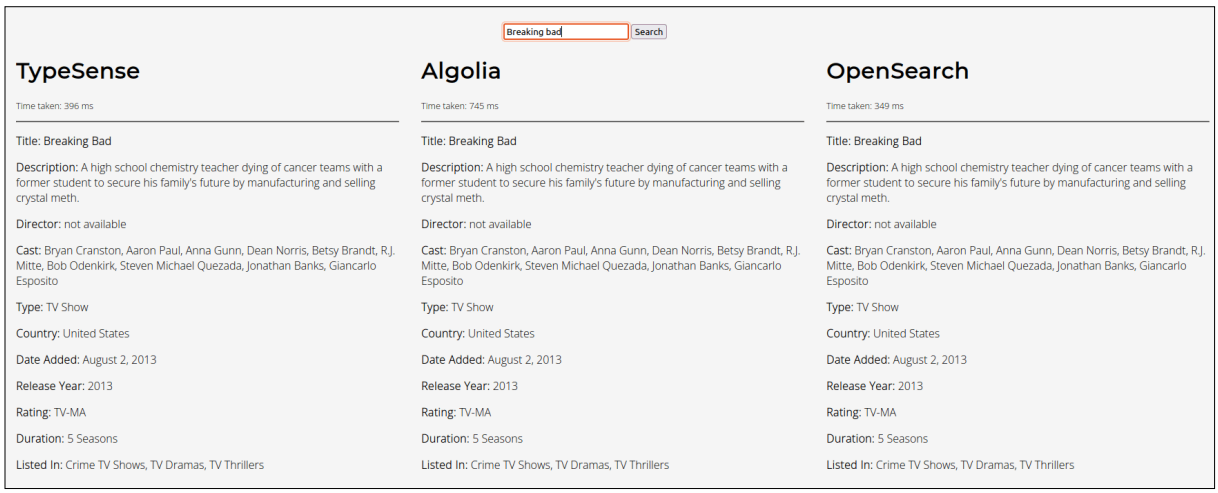


Figure 3.1: A single search bar is used to search all three search engines.

```
function createHTMLFromTypesenseJSON(data, timeTaken) {
  var htmlContent =
    `<h1>TypeSense</h1><p class="time-taken">Time taken: ${timeTaken} ms</p><hr>`
  for (let i = 0; i < Math.min(3, data.length); i++) {
    htmlContent = htmlContent.concat(
      `<p><strong>Title:</strong> <strong>${data[i].document.title}</strong></p>
      <p><strong>Description:</strong> ${data[i].document.description}</p>
      <p><strong>Director:</strong> ${data[i].document.director}</p>
      <p><strong>Cast:</strong> ${data[i].document.cast}</p>
      <p><strong>Type:</strong> ${data[i].document.type}</p>
      <p><strong>Country:</strong> ${data[i].document.country}</p>
      <p><strong>Date Added:</strong> ${data[i].document.date_added}</p>
      <p><strong>Release Year:</strong> ${data[i].document.release_year}</p>
      <p><strong>Rating:</strong> ${data[i].document.rating}</p>
      <p><strong>Duration:</strong> ${data[i].document.duration}</p>
      <p><strong>Listed In:</strong> ${data[i].document.listed_in}</p>
      </br><hr>`);
  }
  document.getElementById('typesense-results').innerHTML = htmlContent;
  return htmlContent;
}
```

Listing 3.3: Converting the Typesense JSON results into HTML.

```
const searchQuery = {
  query: {
    multi_match: {
      query: searchTerm,
      fuzziness: "AUTO",
      fields: ["title", "director", "cast", "description",
               "listed_in", "country", "rating", "type"]
    },
  },
};

fetch(search_api_url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(searchQuery),
})
.then(response => {
  if (!response.ok) {
    throw new Error(`HTTP error! Status: ${response.status}`);
  }
  return response.json();
})
```

Listing 3.4: OpenSearch Query using the fetch() function.

```
function createOpenSearchHTMLFromJSON(data, timeTaken) {
  var htmlContent =
    `<h1>OpenSearch</h1><p class="time-taken">Time taken: ${timeTaken} ms</p><hr>`
  console.log(data)
  for (let i = 0; i < Math.min(3, data.length); i++) {
    htmlContent = htmlContent.concat(
      `<p><strong>Title:</strong> <strong>${data[i]._source.title}</strong></p>
      <p><strong>Description:</strong> ${data[i]._source.description}</p>
      <p><strong>Director:</strong> ${data[i]._source.director}</p>
      <p><strong>Cast:</strong> ${data[i]._source.cast}</p>
      <p><strong>Type:</strong> ${data[i]._source.type}</p>
      <p><strong>Country:</strong> ${data[i]._source.country}</p>
      <p><strong>Date Added:</strong> ${data[i]._source.date_added}</p>
      <p><strong>Release Year:</strong> ${data[i]._source.release_year}</p>
      <p><strong>Rating:</strong> ${data[i]._source.rating}</p>
      <p><strong>Duration:</strong> ${data[i]._source.duration}</p>
      <p><strong>Listed In:</strong> ${data[i]._source.listed_in}</p>
      </br><hr>`);
  }
  document.getElementById('opensearch-results').innerHTML = htmlContent;
  return htmlContent;
}
```

Listing 3.5: Converting the OpenSearch JSON results into HTML.

```
<head>
<script
src="https://cdn.jsdelivr.net/npm/algoliasearch@4.20.0/dist/algoliasearch.umd.js"
integrity="sha256-g/utnPLPYCY4MUPmsSC3/SyX889kVpSgdd0+ySDMjo4="
crossorigin="anonymous"
></script>
// ... (rest of your scripts)
</head>
```

Listing 3.6: Algolia integration into web page template.

```
const algolia_client = algoliasearch("Your Application ID", "Your Search-Only API
    Key")
const algolia_index = algolia_client.initIndex("ms_indices")

function searchAlgolia(searchTerm)
{
  var startTime = performance.now()
  algolia_index.search(searchTerm)
    .then(function (response) {
      createAlgoliaHTMLFromJSON(response.hits)
    })
}
```

Listing 3.7: Algolia Search Query using the JavaScript client.

```
function createAlgoliaHTMLFromJSON(data, timeTaken) {
  var htmlContent =
    `<h1>Algolia</h1><p class="time-taken">Time taken: ${timeTaken} ms</p><hr>`

  for (let i = 0; i < Math.min(3, data.length); i++) {
    console.log(data);

    htmlContent = htmlContent.concat(
      `<p><strong>Title:</strong> <strong>${data[i].title}</strong></p>
      <p><strong>Description:</strong> ${data[i].description}</p>
      <p><strong>Director:</strong> ${data[i].director}</p>
      <p><strong>Cast:</strong> ${data[i].cast}</p>
      <p><strong>Type:</strong> ${data[i].type}</p>
      <p><strong>Country:</strong> ${data[i].country}</p>
      <p><strong>Date Added:</strong> ${data[i].date_added}</p>
      <p><strong>Release Year:</strong> ${data[i].release_year}</p>
      <p><strong>Rating:</strong> ${data[i].rating}</p>
      <p><strong>Duration:</strong> ${data[i].duration}</p>
      <p><strong>Listed In:</strong> ${data[i].listed_in}</p>
      </br><hr>`)
    }
    document.getElementById('algolia-results').innerHTML = htmlContent;
    return htmlContent;
}
```

Listing 3.8: Converting the Algolia JSON results into HTML.

Chapter 4

Search Feature Implementation

A number of advanced search features can be implemented in the three SSEs. For this survey, three commonly used search features were explored: curation of search results, phrase search, and fuzzy search. Showcase videos were created to explain these features [G2 2023b].

4.1 Curation of Search Results

The operators of a SSE can use curation to manipulate the search results of their users to fit specific needs (for example, sponsored search results). For this survey, curation was implemented in Typesense and Algolia:

- **Typesense**: Overrides allow the curation of search results when using Typesense. These are rules set in the backend, that activate when certain strings are provided in the query. Creating an override is done by sending a query to the backend. Typesense allows the operator to include and exclude search results for given strings. This feature does not appear to support fuzzy search, however. An example of creating an override using the Python API can be found in Listing 4.1.
- **Algolia**: The web interface of Algolia allows the backend engineer to create rules for search result curation. Figure 4.1 shows Algolia's Rules interface. From this interface, new rules can be created, exported, and imported. When creating a rule, the interface presents a search bar and the usual results for the given search string, as seen in Figure 4.2. The operator can then pin results to a certain ranking, which sets the rule for this query. After clicking review and publish, the rule is in effect. Fuzzy search is supported by this feature.

For demonstration, we implemented curation in Typesense and Algolia in such a way, that a search for the actor "Bryan Cranston" always yields "You don't mess with the Zohan" as the first result.

4.2 Phrase Search

Phrase search allows users to search for exact combinations, spelling, and order of words, by enclosing their phrase inside double quotes. All three SSEs support phrase search:

- **Typesense**: Phrase search is supported by default.
- **OpenSearch**: The match phrase query of OpenSearch enables the use of phrase search. An example can be found in Listing 4.2.
- **Algolia**: Phrase search is supported by default.

```
override = {
  "rule": {
    "query": "bryan cranston",
    "match": "contains"
  },
  "includes": [
    {"id": "5940", "position": 2},
    {"id": "8790", "position": 1}
  ]
}

client.collections['netflix'].overrides.upsert('curate-breaking-bad', override)
```

Listing 4.1: Creating an override in Typesense.

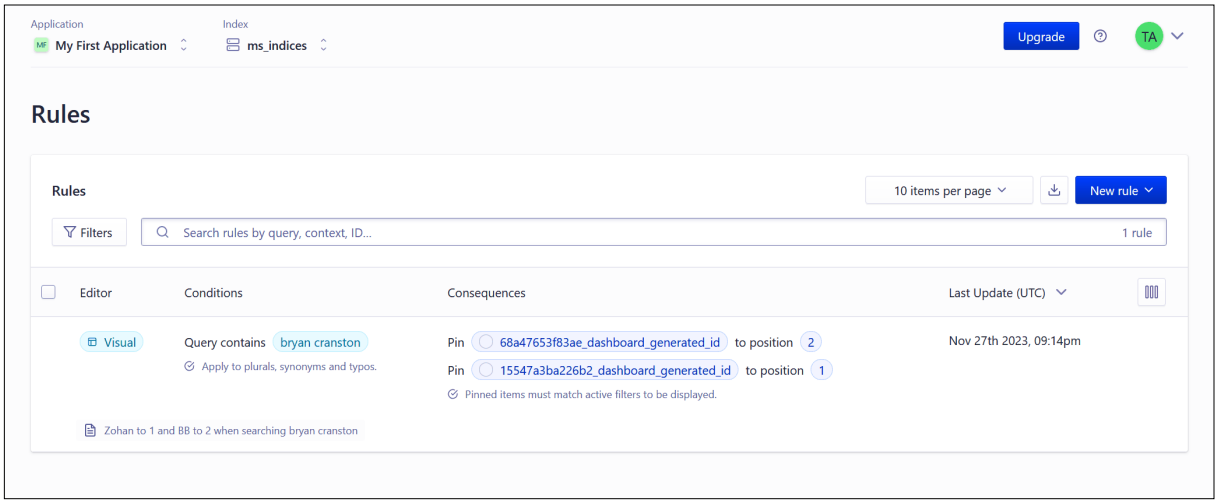


Figure 4.1: Algolia rules interface.

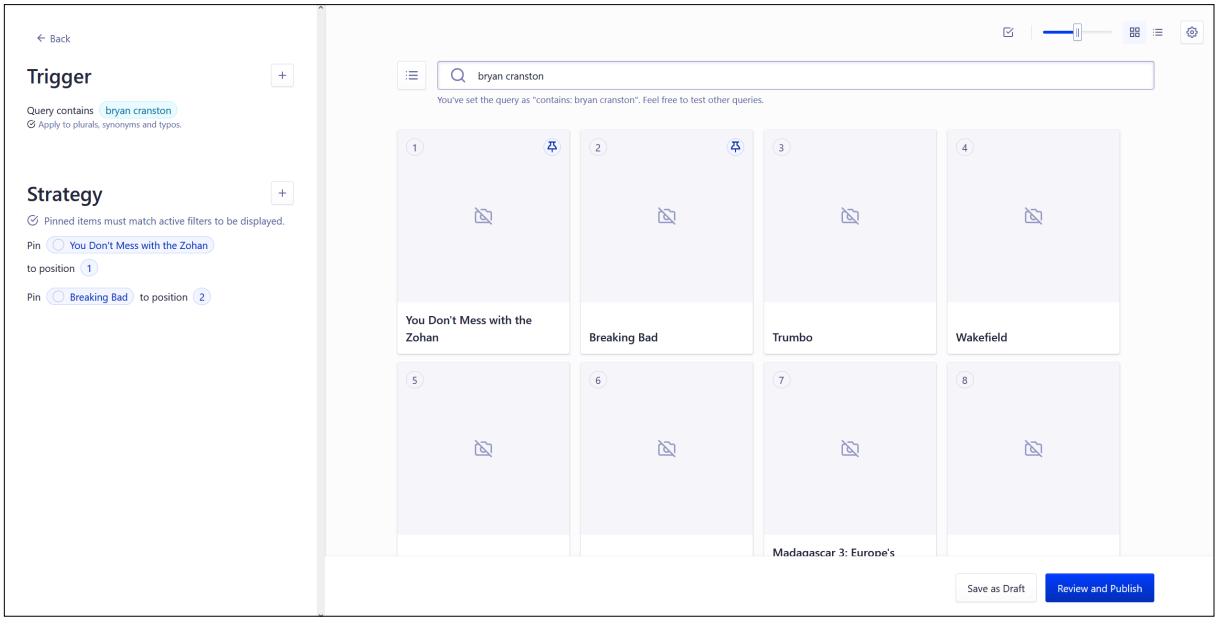


Figure 4.2: Creating a rule in Algolia.

```
GET _search
{
  "query": {
    "match_phrase": {
      "title": "in the jungle"
    }
  }
}
```

Listing 4.2: Match phrase query of OpenSearch.

4.3 Fuzzy Search

Users of search engines do not always type their queries correctly. Either due to random spelling mistakes or not knowing the correct spelling of the word. Fuzzy search allows for typo-tolerant searching, using a metric of how similar two different words are. Such a metric is called the *edit distance*. There are different ways of calculating the edit distance, but the most popular of these metrics is called the *Levenshtein distance*. This metric counts the number of deletions, insertions or substitutions needed to transform one string to another [Typesense 2023e]. All three SSEs support fuzzy search:

- **Typesense**: Fuzzy Search is supported out of the box. There are multiple parameters that can be set: maximum number of typos, minimum word length for 1-typo and 2-typo correction, whether space should be treated as a typo (e.g. `q = basket ball` instead of `q = basketball`), and the typo token threshold, which is the minimum number of search results, where Typesense does not search for typo-corrected variations [Typesense 2023d].

- **OpenSearch**: The match query of OpenSearch can have a fuzziness parameter specified to enable fuzzy search, as shown in Listing 4.3. When using the `AUTO` setting for fuzziness, the following rules apply: Strings of 0-2 characters must match exactly. Strings of 3-5 characters allow 1 edit. Strings longer than 5 characters allow 2 edits [OpenSearch 2023c].

- **Algolia**: This feature is enabled by default in Algolia. The web interface also provides ways to customize how fuzzy search works. It is possible to change the minimum length needed for allowing one or two typos. Typo tolerance can also be disabled for certain attributes of documents or for certain words.

```
GET testindex/_search
{
  "query": {
    "match": {
      "title": {
        "query": "wnid",
        "fuzziness": "AUTO"
      }
    }
  }
}
```

Listing 4.3: Match query with fuzziness parameter of OpenSearch.

Chapter 5

Comparison

In order to conduct a meaningful comparison of the three selected SSEs, a list of 109 individual criteria was compiled. The criteria are structured into seven groups, namely Backend Setup, Indexing, API Clients/Interfaces, Search, Limits, Analytics, and General Criteria. The full resulting comparision was compiled in a spreadsheet [G2 2023a], the main results are presented in this chapter.

The examination of the differences between the SSEs regarding the implementation was started with the assessment of the Backend Setup criteria. For this section of the criteria, it was beneficial that a website was developed and it was possible to distinctly differentiate the three SSEs based on how they were implemented into the website.

5.1 Backend Setup

A comparison of the thre SSEs' characteristics in the Backend Setup category can be seen in Table 5.1. It includes crucial characteristics of the SSE such as whether the SSE is self-hosted or cloud-based, or if the SSE provides a built-in web interface to hide complexity and allow for a more streamlined user experience when setting up the SSE to the desired standards.

5.2 Indexing

Indexing criteria include the difficulty of indexing the given data, whether a GUI is provided, support for batch indexing, and support for bulk edits. Table 5.2 compares the three SSEs in this regard. Algolia does not give information regarding the backwards compatibility of its generated index.

5.3 Search

The criteria in terms of search functionality include support for filtering, faceted search, result ranking, taxonomy integration, phrase search, and fuzzy search. The comparison is shown in Table 5.3. All three SSEs support the majority of characteristics, but Typesense and OpenSearch often require plugins or query parameter modification to do so.

5.4 Limits

The limits category includes limits on the number of documents, size of index, size of records, and number of API keys. The comparision is shown in Table 5.4. As expected, Algolia, being a paid service, restricts its service in some respects. Typesense has no limitations as such, but OpenSearch is constrained in places by the hardware it runs on.

Backend Setup Criteria	Typesense	OpenSearch	Algolia
Self-Hosted	Yes	Yes	No
Cloud-Based	Yes	No	Yes
Other Ways of setting up	Linux, MacOS, Windows	Linux, Windows	N/A
Difficulty	Easy (Docker)	Easy (Docker)	N/A
Docker Compose Templates	Yes	Yes	N/A
Builtin Web Interface	No	Yes	Yes
GPU Acceleration Support	Yes	Yes	N/A
Built Using	C++	Java	C++
Primary Index Location	RAM	Disk, with RAM cache	RAM

Table 5.1: Backend setup criteria.

Indexing Criteria	Typesense	OpenSearch	Algolia
Difficulty	Easy	Complex	Very Easy
Visual Interface	No	No	Yes
Multiple indices	Yes	Yes	Yes
How to index	HTML Request	HTML Request	Web interface, HTML Req.
Batch indexing	Yes	Yes	Yes
Indexing file formats	JSONL	JSON	JSON, CSV, TSV
Basic Unit of Data	JSON Object	2 JSON Objects	1 line, 1 JSON Obj.
Flexibility in fields	Yes	Yes	Yes
Updating	Yes	Yes	Yes
Bulk Edits	Yes	Yes	Yes
Backwards compatibility	Full backwards compatibility	1 major version	N/A

Table 5.2: Indexing criteria.

Search Criteria	Typesense	OpenSearch	Algolia
Search Filtering Options	Yes (param)	Yes (param)	Yes
Faceted Search	Yes (param)	Yes (plugin)	Yes
Result Ranking	Yes	Yes (plugin)	Yes
Advanced Search	Yes	Yes	Yes
Taxonomies	Yes (plugin)	Yes	Yes
Query Suggestions	Yes	Yes	Yes
Dictionaries	No	Yes	Yes
Fuzzy Search	Yes	Yes (query)	Yes
Personalized Search Results	Yes	Yes (plugin)	Yes (premium)
Negative Keyword Search	Yes	Yes	Yes
Phrase Search	Yes	Yes	Yes
Vector Search	Yes	Yes	No
Semantic Search	Yes	Yes	No
Search UI Component Library	Yes, Requires hosted search	Yes, Instantsearch.js	Yes, Instantsearch.js
Visual Search	Yes	Yes	Yes
Voice Search	No	No	Yes

Table 5.3: Search criteria.

Limits Criteria	Typesense	OpenSearch	Algolia
Number of documents	No limitation	Constrained by RAM	Unknown
Maximum number of indices	No limitation	No limitation	No limitation
Maximum index size	No limitation	Constrained by RAM	128GB
Maximum words per field	No limitation	No limitation	No limitation
Maximum record size	No limitation	No limitation	10KB
Number of API keys	No limitation	No limitation	5.000

Table 5.4: Limits criteria.

Major Criteria	Typesense	OpenSearch	Algolia
Paid	No	No	Yes
Faceted Search	Yes (parameter)	Yes (plugin)	Yes
Advanced Search	Yes	Yes	Yes
Query Suggestion	Yes	Yes	Yes
Fuzzy Search	Yes	Yes (in query)	Yes
Taxonomies	Yes (plugin)	Yes	Yes
Dictionaries	No	Yes	Yes
Security (Out-of-the-box)	Seach only; Admin API keys	Manual setup	Seach only; Admin API keys
Personalized Results	Yes (plugin)	Yes (plugin)	Yes (premium tier)

Table 5.5: Comparison of the three SSEs.

5.5 Analytics

Analytics are of lesser importance in this survey, since there is not enough traffic on the demo web site for a meaningful comparison. Rather than inspecting the performance and utility of the analytical tools by each SSE, it was decided to only list the available analytic tools. The comparison can be found in the accompanying spreadsheet [G2 2023a]. Surprisingly enough, all of the SSEs offered analytical tools to observe the infrastructure, analyse performance or specific metrics, and Typesense and Algolia even support A/B Testing.

5.6 General Criteria

General Criteria encompasses metrics such as official support, support channels, community, and documentation. The comparison can be found in the accompanying spreadsheet [G2 2023a].

5.7 Overall Observations

Some of the most important differences between the three SSEs are highlighted in Table 5.5. All three SSEs support advanced search and fuzzy search. Similarly, all three SSEs support more sophisticated functionality like faceted search and query suggestions, even though these are not supported by all SSEs on the market.

One of the most striking differences between the three SSEs is that Typesense does not support dictionaries as of now, while the other two support dictionaries out of the box. There were serious security issues with OpenSearch, where the indices were deleted by an unknown actor. This did not

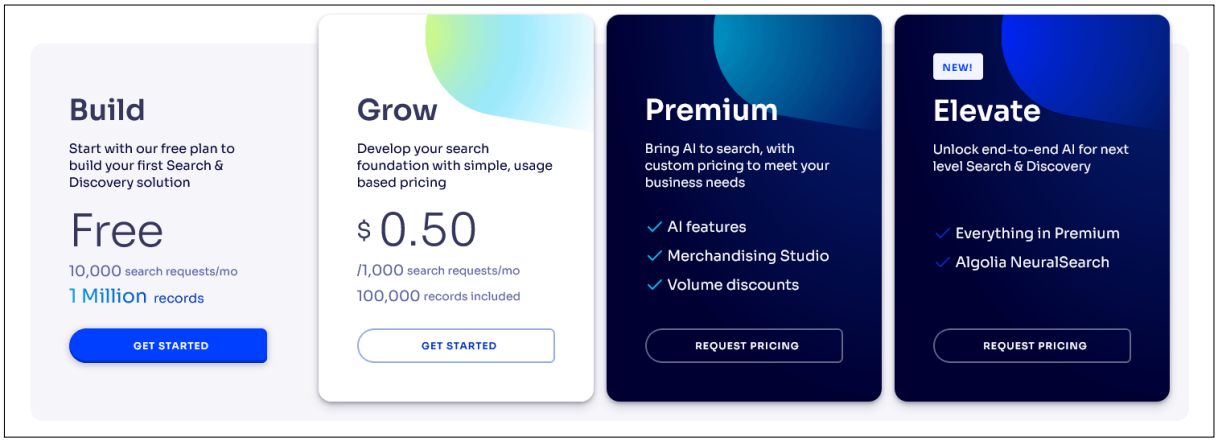


Figure 5.1: Algolia pricing model.

happen with Typesense and Algolia, because both of them offer out-of-the-box security using search and admin API keys. OpenSearch needs manual security setup.

Typesense and OpenSearch are both free and open-source. Algolia is a commercial product and offers a variety of pricing models, shown in Figure 5.1. However the higher tier plans are not transparently priced, and pricing has to be requested.

Chapter 6

Concluding Remarks

This survey compares three Site Search Engines (SSEs): Typesense, OpenSearch, and Algolia. With the the created demo website, it was possible to explore all three SSEs side by side, which in turn enabled parallel observation and comparing the three efficiently.

The accompanying spreadsheet [G2 2023a] provides a comprehensive comparison of the three SSEs. The main similarities and differences were presented in Chapter 5. The comparison also unveiled some hidden peculiarities of the three SSEs such as OpenSearch having some security issues leading to the involuntary deletion of indices and Algolia having an intransparent pricing model.

Overall, in case the pricing model is not a hindrance, Algolia is the most streamlined and consumer-friendly SSE. It includes many important criteria out of the box and provides a web interface for the user which hides many of its complexities.

If an open-source option is preferred, Typesense is strongly recommended as the site search engine of choice. It offers countless options for customisation, and is easier to set up and use than OpenSearch. Typesense supports a wide array of programming languages and frameworks, providing a very good developer experience. Furthermore, the strikingly active community adds another layer of support to the official paid support and the quite extensive documentation provided by the Typsense team. Finally, Typesense is the only SSE which does not collect private user-related data.

Bibliography

Algolia [2023]. *Algolia*. 2023. https://algolia.com/ (cited on page 1).

Bansal, Shivam [2021]. *Netflix Movies and TV Shows*. 27 Sep 2021. https://kaggle.com/datasets/shivambansal/netflix-shows (cited on pages 1, 5).

Docker [2023]. *Docker overview*. 28 Nov 2023. https://docs.docker.com/get-started/overview/ (cited on page 3).

F5 [2023]. *NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy* (29 Nov 2023). https://nginx.com/ (cited on page 4).

G2 [2023a]. *Criteria Spreadsheet*. Information Architecture and Web Usability, WS 2023, G2. 28 Nov 2023. https://docs.google.com/spreadsheets/d/1G1l-4-Oi-zswkyT_IvtEECdnMQlUqo92hm6VAxsr2-k/ (cited on pages 21, 23, 25).

G2 [2023b]. *Showcase Videos: Site Search Engines*. Information Architecture and Web Usability, WS 2023, G2. 10 Dec 2023. https://youtube.com/playlist?list=PLsp4BtuSXH9nkw7SUff_0SsnASeRGWmM1 (cited on page 17).

Huggingface [2023]. *Huggingface Datasets*. 2023. https://huggingface.co/datasets/ (cited on page 5).

Kaggle [2023]. *Kaggle Datasets*. 2023. https://kaggle.com/datasets/ (cited on page 5).

MDN [2023]. *Same-origin policy*. Mozilla Developer Network. 29 Nov 2023. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#cross-origin_network_access (cited on page 4).

OpenSearch [2023a]. *About Security - OpenSearch documentation*. 10 Dec 2023. https://opensearch.org/docs/latest/security/index/ (cited on page 4).

OpenSearch [2023b]. *Docker - OpenSearch documentation*. 28 Nov 2023. https://opensearch.org/docs/latest/install-and-configure/install-opensearch/docker/ (cited on pages 3–4).

OpenSearch [2023c]. *Fuzziness - OpenSearch documentation*. 2023. https://opensearch.org/docs/latest/query-dsl/full-text/match/#fuzziness (cited on page 19).

OpenSearch [2023d]. *OpenSearch*. 2023. https://opensearch.org/ (cited on page 1).

pandas [2020]. *pandas-dev/pandas: Pandas*. Version latest. Feb 2020. doi:10.5281/zenodo.3509134 (cited on page 6).

Typesense [2023a]. *Downloads | Typesense*. 2023. https://typesense.org/downloads/ (cited on page 3).

Typesense [2023b]. *Install Typesense | Typesense*. 2023. https://typesense.org/docs/guide/install-typesense.html#option-2-local-machine-self-hosting (cited on page 3).

Typesense [2023c]. *Typesense*. 2023. https://typesense.org/ (cited on page 1).

Typesense [2023d]. *Typo-Tolerance Parameters | Typesense*. 2023. https://typesense.org/docs/0.25.1/api/search.html#typo-tolerance-parameters (cited on page 19).

Typesense [2023e]. *What is fuzzy search? | Typesense*. 2023. https://typesense.org/learn/fuzzy-search/ (cited on page 19).

Ward, Ian [2023]. *JSON Lines*. 10 Dec 2023. https://jsonlines.org/ (cited on page 5).