**Approach of the project**

Our aim was to build a search engine with a touch of personalization, so our approach was decided on the lines of this aim. Our main approach was decided after a lot of trial and error and availability of resources for achieving the task in hand.

Workflow of the project can be depicted in the following diagram

High level representation of our approach

So, our workflow consists of series of operations namely:

1. Full text Search
2. Ranking of Search results
3. Boosting of search results (Personalization Logic)

**Full text Search:**

In text recovery, full-text scan alludes to methods for looking through a solitary PC put away report or an assortment in a full-text information base. Full-text search is recognized from look through dependent on metadata or on pieces of the first messages addressed in data sets (like titles, abstracts, chosen areas, or bibliographical references).

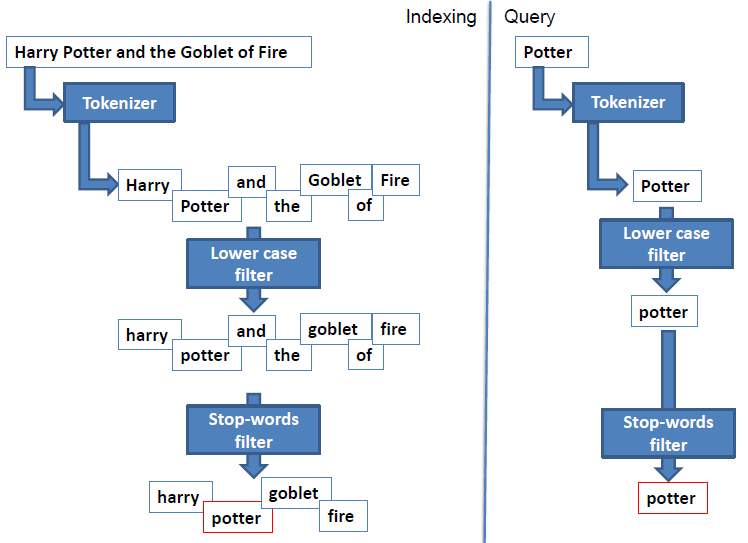
In a full-text search, a web crawler looks at all the words in each put away archive as it attempts to coordinate with search measures (for instance, text indicated by a client). Full-text-looking through strategies got normal in online bibliographic information bases in the 1990s. Many sites and application programs, (for example, word preparing programming) give full-text-search capacities. Some web search tools, like AltaVista, utilize full-text-search strategies, while others file just a segment of the website pages analyzed by their ordering frameworks.

How it works?

For a small amount of simple textual data, it’s possible to provide basic search functionality via simple string matching. For example, using JavaScript you could store product data for a small online shop as an array of objects in a JSON file, then fetch the file and iterate over each object to find matches.

This simplistic approach can be better than nothing, but successful search needs more flexible functionality to find relevant results:

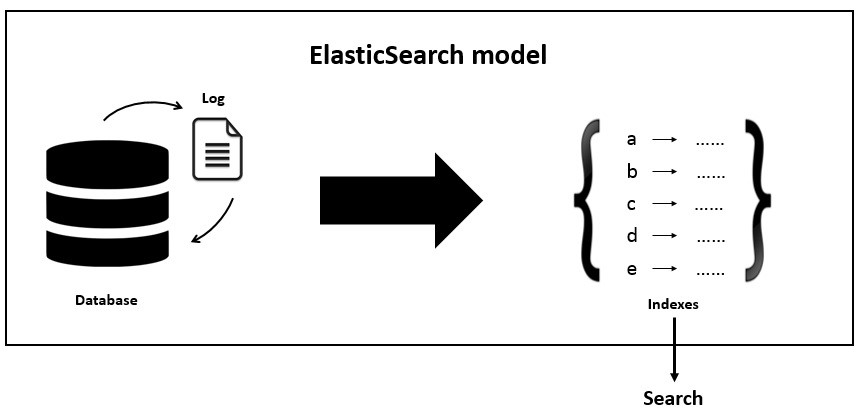
* **Stemming**: *developing for mobile*matches results for *develop for mobile* and vice versa.
* **Stopword handing**: Search engines need to avoid irrelevant results caused by matching common words such as *a* and *the.* (Conversely, ignoring stopwords can also cause problems for queries such as *The Who* or *To be or not to be*.)
* **Basic fuzzy matching**: *service workers* matches results for *Service Worker.*



Full text Search Example

Most of the present-day search engine frameworks provide an abstraction over this process and save developers time in creating this from scratch. Some of the advantages using search engine framework is as follows:

* Speed: Using Full-text search guarantee speed in recovery of the outcomes for huge quantities of reports of an enormous book information base.
* Effectiveness: Accurate exact list items in all fields.
* Configurable inquiry standards and rationale.
* Static sites support: Many static sites utilize level records approach like JSON or Markdown organized documents. A few libraries and systems are utilizing
* Versatile applications support



Frameworks add abstraction over comprehensive full text search process

How does elastic search implement full text search?

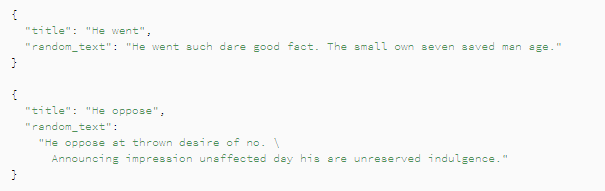
Elastic search implements the full text queries which enable you to search analyzed text fields such as the body of an email. The query string is processed using the same analyzer that was applied to the field during indexing.

The process involves couple of steps to implement the full text search capabilities:

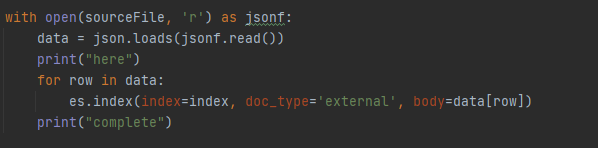
1. Indexing documents –

ElasticSearch is record arranged. It stores and files reports. Ordering makes or updates archives. Subsequent to ordering, you can look, sort, and channel total reports—not lines of columnar information. This is an in a general sense diverse perspective about information and is one reason ElasticSearch can play out a perplexing full-text search.

Reports are addressed as JSON objects. JSON serialization is upheld by most programming dialects and has become the standard arrangement utilized by the NoSQL development. It is basic, brief, and simple to peruse.



Some Random documents in json format



Python code to upload data

1. Retrieving Documents

After adding all the documents, we can retrieve it with the following code

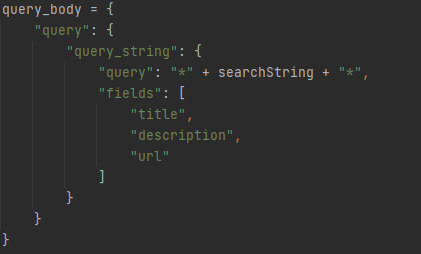


DSL Query for retrieving all documents

Pass in the search keyword in match\_all or leave it blank tor retrieve all results

1. Querying Documents

Documents can be queried based on any search keyword. Elastic Search needs to be instructed about the search keyword, where to search (target) and what kind of search strategy needs to be defined based on preference. The Sample code for DSL query for full text search is given below



DSL query for searching bidirectional in fields like title description, url etc.

By default, ElasticSearch sorts matching results by their relevance score, that is, by how well each document matches the query.

This is a complete process implemented for elastic search for full text search functionality.

**Ranking of Search Results**

Web indexes show their outcomes as an arranged rundown. The rundown is 'positioned' with the most applicable sites (as controlled by the web index's positioning calculation) most noteworthy on the main page of profits. There are numerous components engaged with deciding the position request of indexed lists. It is significant for the genuine searcher to comprehend the fundamental elements of the cycle.

How Elastic search help implement this feature?

Elastic search helps boosts the relevance score of documents based on the numeric value of a rank\_feature or rank\_features field.

The rank\_feature inquiry is regularly utilized in the should proviso of a bool question, so its importance scores are added to different scores from the bool inquiry.

With positive\_score\_impact set to bogus for a rank\_feature or rank\_features field, we suggest that each record that takes an interest in an inquiry has an incentive for this field. Something else, if a rank\_feature question is utilized in the should provision, it doesn't add anything to a score of a record with a missing worth however adds some lift for a report containing an element. This is in opposition to what we need – as we consider these highlights negative, we need to rank records containing them lower than archives missing them.

Dissimilar to the function\_score question or alternate approaches to change importance scores, the rank\_feature inquiry effectively skirts non-cutthroat hits when the track\_total\_hits boundary isn't accurate. This can drastically improve inquiry speed.

The rank feature in elastic search is used in the following manner

1. Index Setup:

To use the rank\_feature query in elastic search, your index must include a rank\_feature or rank\_features field mapping.



Example Mapping defining rank features for an index

1. Querying with ranking:

Instead of querying the DB for just term-based document matching, we can even ask Elasticsearch to query as well as rank the documents based on ranking features. The same thing can be implemented in elastic search as follows



DSL query for implementing querying and ranking

**Boosting of search results**

This covers the last step of our approach for personalized search engine. After deriving the ranked results, which might be popular among general masse but not with specific people. This step helps us to increase the relevancy of the search results for certain user. Boosting is nothing but finding most relevant search results (personalized) based on certain attributes or feature tags (what we call them). Each user is having certain feature tags and same feature tags we derived from resources. Our boosting logic help us find matching documents for user and based on our analysis we concluded that these are the most personalized results for the user.

Boosting your inquiry enables you to numerically focus on explicit reports as more significant than others by assessing how question terms are coordinated in fields. For instance, in the event that you had a formula web search tool, you might need to help the title field so that questions that match terms in the title field are shown higher than say those archives that match terms in the remarks. Boosting can require some manual tweaking. Adding too huge of a lift can come up short of your client's inquiry. Adding excessively little of a lift can clean out the impact of having a lift.

Diagram

Description automatically generated

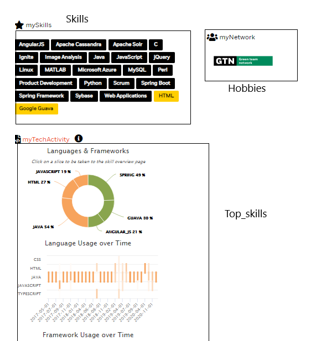
Boosting workflow

What is feature tag?

In the context of our application, we have defined feature tags to define user attributes or interests. These feature tags are labels defined to each of the attributes of user egs skills, education etc. These labels are accepted all through out the system. These feature tags classification is carried out for both user and resources available in our system. Some of the common feature tag used in our system are:

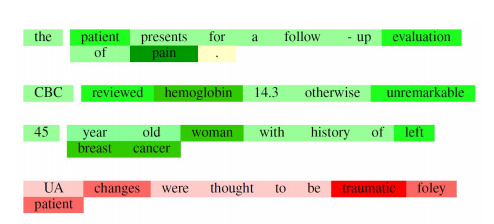
1. Skills: Tools/Technologies in the general space
2. Education: Subject preference, research domain etc.
3. City: Work location
4. Team: User/Resource associated with an internal team

The attributes of user are equivalent to user search history, their preferences and interests. These attributes define everything about the user and help us recommend resource based on matching feature tags with resources available in our database. The user attributes are derived from their profile which includes their skills, education, team they are working with, location of work, past employment etc. These details are scraped from users’ profile and a feature tag label is given based on an algorithmic logic.



Feature tags defined for Users

The resources available in our system consists of news articles, blog article, communication letters, videos etc. These resources are sometimes relevant to certain people or an entire team or for everyone. These resources contain keywords in subject/heading or body which help us understand to whom it is directed to. These keywords are classified with a feature tag like users feature tags. The thing to note here is for both the user and resource feature tags label should be identical. The keyword extraction algorithm can be used to defined keywords for resources and they are classified into feature tags based on knowledge base acquired from the system.



Sample Example for keyword extraction

These feature tags derived from users and resources are matched and based on threshold value as decided after various round of testing. The documents with maximum matched feature tags and passing certain threshold are qualified as personalized search document and is highly recommended to the user.

How elastic search works for boosting?

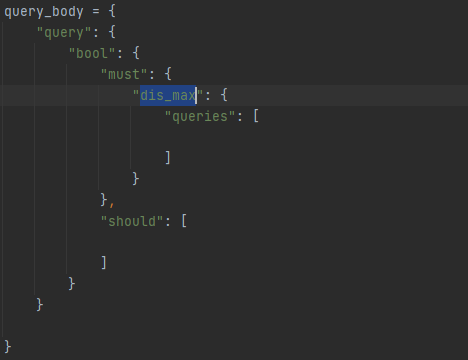
Boosting is a follow up of ranking process implemented by elastic search. Elastic search internal boosting mechanism is not used here (just to avoid confusion). There are certain prerequisites for implementing boosting feature with Elasticsearch, these are:

* Resource feature tag index should be created in the elastic search and same for user feature tag as well



Feature tag json document in Elasticsearch

* Prepare query for multiple keyword matching query which gives back boosted result. FO making query we have used dis\_max feature which returns documents matching one or more wrapped queries, called query clauses or clauses. If a returned document matches multiple query clauses, the dis\_max query assigns the document the highest relevance score from any matching clause, plus a tie breaking increment for any additional matching subqueries.



DSL based disjoint max query