ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΜΣ Τεχνολογίες Πληροφορικής και Επικοινωνιών

Μ151 Συστήματα και Εφαρμογές στον Παγκόσμιο Ιστό

Project Report

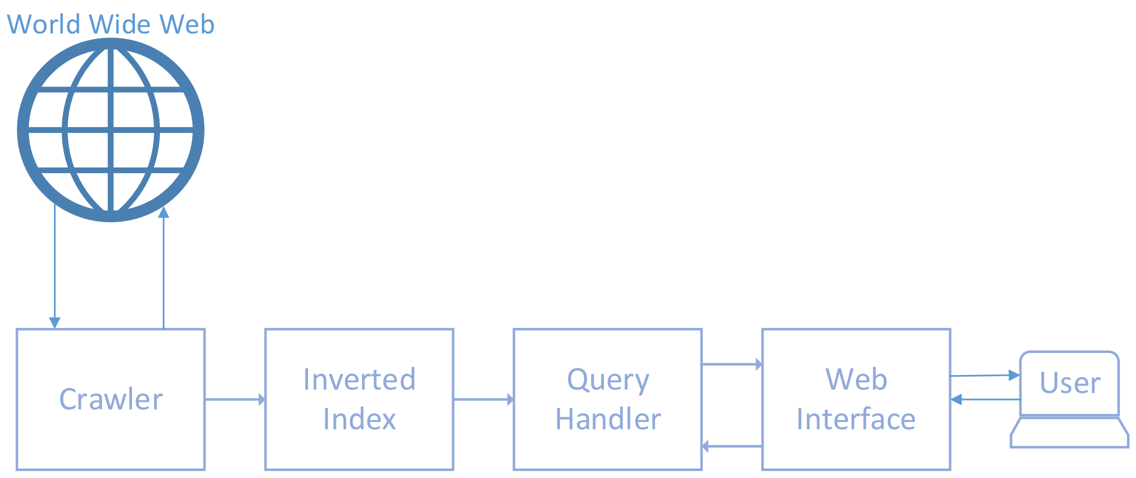
Γεώργιος Χαραλαμπίδης
Αριθμός Μητρώου: IC1180018

ΑΘΗΝΑ

Ιούνιος 2021

Music Search Engine

The project is about building a Music Search Engine. The main infrastructure of the engine is presented below.



Search Engine Infastructrue

The search engine consists of four main building blocks, which are:

- Crawler to discover and download web documents.
- Indexer to make inverted index.
- Query Handler which is responsible for accepting user queries from the web interface, produce search results base on the inverted index and serve them back to the user.
- Web interface is the user friendly tool for communication between the user and the search engine.

All four components are built from scratch and implementation has been tested and operates correctly within rational acceptance levels for an engine build as a project.

1. Crawler (Crawler.py)

```
if __name__ == '__main__':
    domains = 'https://www.allmusic.com'
    start_url = 'https://www.allmusic.com/artist'
    headers = {'user-agent': 'Mozilla/5.0 (Linux; U; Android 4.1.1; en-
gb; Build/KLP) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0
Safari/534.30'}
    urls = []
    visited_urls = []
    urls.append(start_url)
    output = []

    for link in urls:
        if (start_url in link) and (link not in visited_urls):
            crawl(link)

    with open('output.json', 'w') as output_file:
        output_file.write(json.dumps(output))

    # Create Inverted Index
    print('Creating inverted index')
    index()
    print('Inverted index is ready')

    # Start Web Server
    start_server()
```

Before initiation, crawler must get a setup by determining the domain we want to crawl and the **start_url** which is the beginning point of the crawling process. In our case, the **start_url** also limits the crawler to download sites that are below the **start_url**. Domain variable is used for link retrieval which is described below at crawling function. Finally, headers can be customized if needed to trigger servers return documents as they would in a normal web browser.

After program execution, 3 lists are initiated. **Urls** list is used for storing all links that are discovered, and the first link that is added in the list is the **start_url**. The **visited_urls** list stores all the links that the crawler has visit an is used to avoid crawling the same page in the future as the crawling process progresses. Finally, the **output** list stores in memory all the documents that are crawled until they are extracted in a file on disk.

Next the program initiates a loop function which takes links from **urls** list, inspects if the link is below the **start_url** and also is not in **visited_urls** list, and if both restrictions are true it initiates the crawl function. If restrictions are not met, next link in list is inspected. The loop function continues until all links in the list are checked.

With every crawl call more and more links are added in the list. The program is not recursive, rather it visits all links of the first page and then all of the second and so on so forth. The program can become recursive if the crawl function is called within the crawl function every time a link is discovered. We believe that the way we implemented it makes it easier for the algorithm to scale.

After all documents of a domain are crawled, the program serializes and writes the output list in json file on disk for further processing.

Finally, the program calls the indexer (Indexer.py) which creates the inverted index, and upon completion, program calls the start_server function that start the web server.

Crawler consists of two functions, **clean(text)** and **crawl(url).**

Clean(text) function

The first function, **clean(text)**, has task to clean a string which is passed as parameter and return it. To accomplish this, the function uses multiple regular expression for cleaning the string from symbols, new lines, trim spaces and more.

```
def clean(text):
    text = (text.encode('ascii', 'ignore')).decode("utf-8")
    text = re.sub("&.*?;", "", text)
    text = re.sub(">", "", text)
    text = re.sub("[\]\|\[\@\,\$\%\*\&\\\(\)\":]", "", text)
    text = re.sub("-", " ", text)
    text = text.replace("'", "")
    text = text.replace("|", "")
    text = text.replace("\n", " ")
    text = re.sub("\.+", "", text)
    text = re.sub("^\s+", "", text)
    text = text.lower()
    text = text.replace("/", "")
    while "  " in text:
        text = text.replace("  ", " ")
    return text
```

Note:

Although it is common for large search engines to implement cleaning at index creating stage, to reduce delays in crawling and avoid switches between network and cpu workloads, in our case where we only crawl a small set of sites, there is not enough flexibility to switch between the sites we crawl to avoid overloading servers. So instead of adding inert delay time, we prefer to clean the crawled data before initiating the next crawl.

Crawl(url) function

```
def crawl(url):
    visited_urls.append(url)
    response = requests.get(url, headers=headers)
    content = BeautifulSoup(response.text, 'lxml')
    content_text = clean(content.get_text())
    output.append({'url': url, 'page': content_text})

    # Get href links of the document (absolute and relative)
    href_links = content.find_all('a', href=True)
    for link in href_links:
        link_absolute = urllib.parse.urljoin(domains, link.get('href'))
        if (start_url in link_absolute) and (link_absolute not in
urls):
            urls.append(link_absolute)

    # Get links from document text (capture dynamic JavaScript links)
    content_string = str(content)
    content_links = re.findall(r'(\b' + domains + '.*?)[\'"]',
content_string)
    for link in content_links:
        if (start_url in link) and (link not in urls):
            urls.append(link)
    return
```

The second function crawl(url) is the actual crawling operation. The function takes as parameter the url to be crawled. Function inserts the url to visited_urls list and downloads the document. A copy of the document is created where function extracts the text of the document (removes all tags and html, css code) and calls the clean function on the text. The cleaned text of the document that is returned from the clean function is the actual document information to be stores.

Every document that is crawled is represented as a dictionary with two keys. The url key contains the url of the document and the 'page' kay contains the cleaned text of the document. The dictionary is inserted to the output list. Output lists contains multiple dictionaries where each represents a document and its respective url.

The next job of the function is to extract all links from the document. For this purpose, two procedures are used, one from extracting absolute and relative links from the <a> tag of the HTML where href exists, and one for extracting dynamic links from JavaScript parts of the document.

The first procedure finds all <a> tags where href exists and extracts all links. If the link is relative to the domain, the domain is used to produce the absolute link. Next the procedures iterate all extracted links, and adds to the urls list all those links that are below start_url and do not already exists in the urls list.

Next procedure tries to extract dynamic links, links which are under JavaScript scripts (like links which are on buttons)

Note:
JavaScript link extraction is best implemented by frameworks such as **Selenium**. Selenium uses an actual web browser like Google Chrome to actually load the site and execute JavaScript code and extract links. The framework has one main problem which is its speed. Although it extracts more links (Including jQuery and Ajax) it is very slow for our purposes.

One more problem with JavaScipt content is that to extract links correctly, the crawler must be configured for the specific website and thus become very specialized.

To maintain the crawler as a general purpose tool, we selected a different method. Specifically, the second procedure inspects all the downloaded document and search for links that may not be under the <a> tag or use href attribute, by looking for patters. It uses the following regular expression command which does exactly that.

```
content_links = re.findall(r'(\b' + domains + '.*?)[\'"]',
content_string)
```

The above regular expression isolates all strings that start with the domain and finish either with single or double quotes (‘ or “) and extracts the links. Finally, like the first procedure only links that are under start_url and not already in the urls list are added.

Important Note:
Crawler starts with a domain and start_url variables. The algorithm uses general approach while crawling. This means that no use of tag selectors for cleaning content is used which would make the crawler site specific. Thus by changing the two variables one can use the tool to build a search engine for a different context. Without the use of domain limitation, a general search engine can also be produced. Some of the disadvantages of this approach while constructing thematic search engines without dedicated crawler are: the increased amount of data that is stored to the index as no content selector is uses, the consequent, due to large index, increase in computational resources demands and finally some errors may be introduced to the query answers as a result of the trash containing index. Of course TF-IDF technique that is described below takes care of the most mistakes which never reach the user.

2. Indexer

```
import json
import math

def index():
    data = open("output.json").read()
    # list_data = data.split("\n")
    list_data = json.loads(data)
    tf = {}
    idf = {}
    documents_with_term = {}  # Store the number of documents that each
term exists
    tf_idf = {}
    for page in list_data:
        temp_tf = {}
        line_terms = page['page'].split(' ')
        for term in line_terms:
            if term in temp_tf:
                temp_tf[term] += 1
            else:
                temp_tf[term] = 1
                if term in documents_with_term:
                    documents_with_term[term] += 1
                else:
                    documents_with_term[term] = 1
        total_words_in_document = len(line_terms)
        for term in temp_tf:
            temp_tf[term] = temp_tf[term] / total_words_in_document
        tf[page['url']] = temp_tf

    n = len(list_data)  # Total number of crawled web pages
    for df_term in documents_with_term:
        tf_idf[df_term] = []
        idf[df_term] = 1 + math.log10(n / (documents_with_term[df_term]
+ 0.1))
        for url, content in tf.items():
            if df_term in content.keys():
                score = content[df_term] * idf[df_term]
                tf_idf[df_term].append({'url': url, 'score': score})

    with open('inverted_index.json', 'w') as output:
        output.write(json.dumps(tf_idf))
    return
```

Indexer can either run as a standalone program (given that the output file on the crawler exists) or can be directly called from the crawler.

Indexer reads the output.json file, deserialize it and recreates the initial list. Next it creates four dictionaries, (tf, idf, documents_with_term and tf_idf) which are used to produce the TF-IDF score of each term and for each document.

TF: Term frequency, is the number of times a term appears in a document divided by the total terms of the document. Each document has a different set of terms and TF.

DF: Document frequency, is the number of documents that a term appears. For this metric the time a term appears in a document is irrelevant.

IDF: Inverse document frequency is produced from the following equation after DF is calculated.

$$IDF = 1 + \log\left(\frac{Total\ number\ of\ documents\ in\ the\ set}{Total\ number\ of\ documents\ containing\ the\ term + 0.1}\right)$$

Note that the 0.1 value that is added to the total number of documents containing the term is to avoid dividing by zero in case a term does not appear while the quantity is small enough to not affect the output of the equation The 1 at the beginning of the equation is to keep output of the equation always positive. It is added to all calculations an it affects the output at the same rate for all terms. It can also be omitted without a problem as the scoring that is produced can also be negative and it’s still easy to sort results based on score.

TF-IDF: Term frequency – inverse document frequency is calculated by multiplying TF with IDF and constitutes the actual score for every term in every document.

The algorithm, using loop functions, calculates TF for every document and for every term in the document. DF is also calculated while looping through documents. Finally, TF-IDF for every term is calculated. Tf_idf is a dictionary containing as key the terms from all the documents and the value of each term (key) is a list of dictionaries where each dictionary is a document along with its score. In inverted index each term can have multiple documents and each document has a score for the specific term.

After the tf-idf score calculation the data are serialized and written on the disk in ‘inverted_index.json’ file.

3. Query Handler

```python
import json
import re


def clean(text):
    text = (text.encode('ascii', 'ignore')).decode("utf-8")
    text = re.sub("&.*?;", "", text)
    text = re.sub(">", "", text)
    text = re.sub("[\]\|\[\@\,\$\%\*\&\\\(\)\":]", "", text)
    text = re.sub("-", " ", text)
    text = text.replace("'", "")
    text = text.replace("|", "")
    text = text.replace("\n", " ")
    text = re.sub("\.+", "", text)
    text = re.sub("^\s+", "", text)
    text = text.lower()
    text = text.replace("/", "")
    while "  " in text:
        text = text.replace("  ", " ")
    return text


def find_similar_term(query_term, index):  # Compare query term letters
```

```
to index terms letters to find close match.
    similar_term = ''
    similar_term_score = 0
    query_letters = list(query_term)
    for index_term in index:
        matching_ordered_letters = 0
        matching_unordered_letters = 0
        index_term_letters = list(index_term)
        for position, letter in enumerate(index_term_letters):
            try:
                if letter in query_letters[position]:
                    matching_ordered_letters += 1
            except:
                continue
            if letter in query_letters:
                matching_unordered_letters += 1
        ordered_score = matching_ordered_letters / len(query_letters)
        unordered_score = matching_unordered_letters /
len(query_letters) / 2
        if max(ordered_score, unordered_score) > similar_term_score:
            similar_term = index_term
            similar_term_score = max(ordered_score, unordered_score)
    return (similar_term, similar_term_score)


def search(query):
    result_list = {}
    quality_score = 0.002
    quality_results = {}
    suggestion = []
    query = str(query)
    query = clean(query).split(' ')
    query = list(filter(None, query))
    index = json.load(open('inverted_index.json'))
    i = -1 # Index i is used to create new result list for every query
term which includes the results that are common with the previous term.
Final results come from the last result list created
    for term in query:
        i += 1
        result_list[i] = {}
        if len(term) < 2:
            continue
        if term not in index:
            new_term = find_similar_term(term, index)
            if new_term[1] > 0.5:
                suggestion.append(new_term[0])
            else:
                suggestion.append(term)
        else:
            suggestion.append(term)
        try:
            for document in index[term]:
                if i == 0:
                    result_list[i].update({document['url']:
document['score']})
                else:
                    if document['url'] in result_list[i - 1]:
                        result_list[i].update({document['url']:
document['score']})
                        result_list[i][document['url']] +=
result_list[i-1][document['url']]
        except:
            continue
    for url, score in result_list[i].items():
        if score > quality_score:
            quality_results[url] = score
    # Sort results by score in descending order
    quality_results = dict(sorted(quality_results.items(),
reverse=True, key=lambda item: item[1]))
    if suggestion == query:
        suggestion.clear()
    return (quality_results, suggestion)
```

Query handler is used for parsing user queries through web interface, produce results, sort them by ranking, produce suggestions if needed be for terms that could not be found and return them to the user.

The program consists of three functions.

Clean(text) function

This function is exactly the same as the one used in crawling. Its use is to clean the user query the same way as terms are cleared from the documents so as matches can be made.

find_similar_term(query_term, index) function

The purpose of this function is to get a term, search the index for similar term, evaluate them and if a quality match exists return it. Function takes as parameters the term and the inverted index.

Function uses similarity score to evaluate the matches. It operates by comparing the letters of the term with every term in the index as well as their length. For index terms that are close to query term, one or more letters will be similar. For every similar letter the score increases. Score is divided by length of letters of index term to reduce the score according to the difference between the index and query term. Function produces two scores. The first is for terms that match in both the letters and the position while the second considers position of the letters irrelevant. The second score is divided by 2 to give priority to matches in both letter and position. Finally, the match with the highest score (either coming from first or second scoring method) is returned along with the respective score. This method is developed by the author of the project from scratch and it is to my best knowledge that this method has never presented before.

Search(query) function

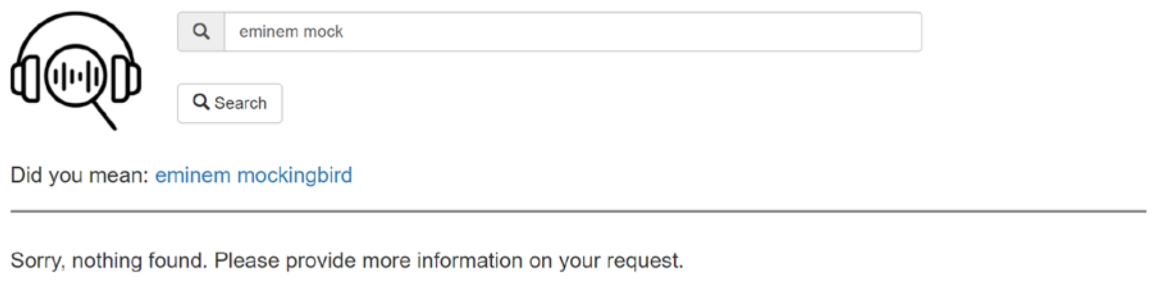
As its name suggests, this function is responsible for taking the user query, searching the index, produce score and return results.

The function takes the user query as parameter, cleans it and splits it to terms. Index is loaded and for every term in query a dictionary of urls and documents is produced. With every term and every iteration, a new result dictionary is generated and only documents that match both the term and the previous term are inserted into it. Scores of the documents are increased accordingly. With this method the results represent the query as a whole and not as individual terms. All terms with length of 1 (All single letters and numbers) are ignored as they match almost every document. For every term that does not exist in the inverted index, the find_similar_term function is called, and a similar term along with the respective score is produced. In this case no results are return. If the similar term is of high quality i.e. meeting score demand, then the user is getting a suggestion of the similar term. If all terms exist in the inverted index, then all quality results that meet the score demand are returned to the user sorted in highest to lowest score.

4. Website Interface

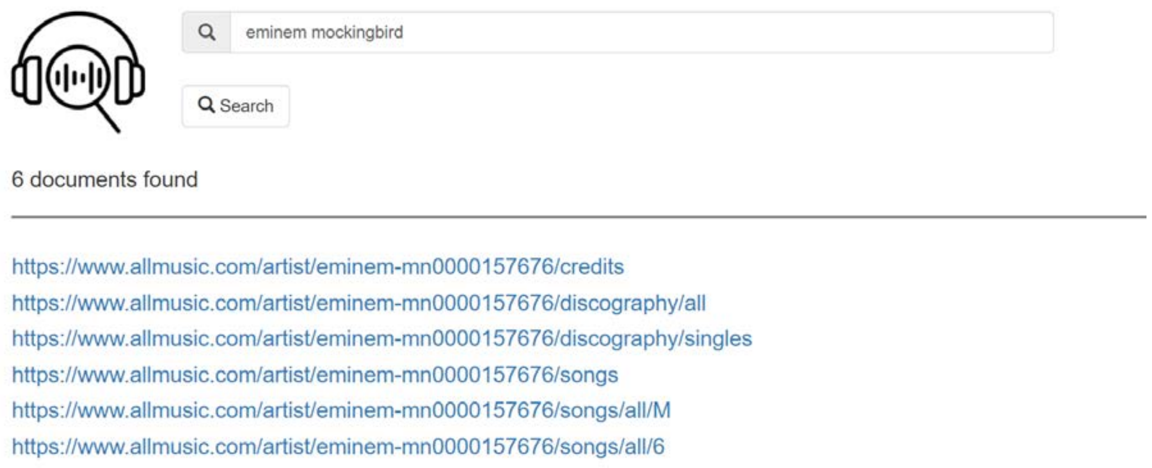
Website interface is the user friendly human machine interface and allows the user to easily input queries and get results. The website is built using html language, CSS language for styling, Bootstrap front end framework and Flask web application framework which connects front and back end. Server is automatically started from the Crawler.py after indexing is finished or can be manually started running the main.py script.

User queries are submitted using GET method and flask initiates the Query Handler and returns the list of results or suggestions to user. Below are some screenshots of the search engine in action.



Did you mean: eminem mockingbird

Sorry, nothing found. Please provide more information on your request.



6 documents found

https://www.allmusic.com/artist/eminem-mn0000157676/credits
https://www.allmusic.com/artist/eminem-mn0000157676/discography/all
https://www.allmusic.com/artist/eminem-mn0000157676/discography/singles
https://www.allmusic.com/artist/eminem-mn0000157676/songs
https://www.allmusic.com/artist/eminem-mn0000157676/songs/all/M
https://www.allmusic.com/artist/eminem-mn0000157676/songs/all/6

Future Work

Of course the search engine of the project is not perfect. The good thing is we didn’t expect to build the next Google Search. Project is developed by the author alone in limited time in the context of a university course. For this reason, there are some important elements that have not been developed but can add significant value to the efficiency and quality of the search engine. Most important of them (but not only) are the following:

Use JavaScript Ajax technology to offer suggestions to the user as they type their query. Although this feature does not improve the search engine itself, will make the web interface more useful. This feature is a front end development and can easily get integrated to the engine as the suggestion mechanism already exists.

Both crawler and Query handler, if they are to serve multiple documents and users respectively, must be converted to parallel software running multiple instances on a single machine. Furthermore, multiple machines (cluster) must run the algorithms and of course the algorithm must be able to synchronize and support such operation.

PageRank is an important score mechanism that have proved to be very useful. Our project does not use PageRank which can increase the quality of the result if built in the search engine. PageRank count the number and quality of links to a page, is not very computationally intensive and converges predictably in about 100 iterations. Of course the ratio of PageRank and TF-IDF scores is up to the developer to balance.

Another, rather easy, addition to the search engine, although a very basic one, is the use of robots.txt protocol with the crawler. Failing to obey those rules can lead to IP bans. Along with that, for a search engine to use in professional level, a better crawling strategy must be used to avoid overloading servers.

Another important feature for the search engine is the use of N-Grams. N-Grams is the tokenization of sentences of terms. Tokenizing each term and indexing words that come from but are smaller than the term can help improve term suggestion as the user types a query. Tokenizing sentences by using words before and after a term can produce better suggestion at sentence level, meaning that suggestion for a term can take place even before the user starts typing his next word in a sentence query. Of course this feature comes with the disadvantage of increase inverted index size and increases computational demands.

Scaling study

Except from the above features, if it is for a search engine to operate at production level, there are some other aspects of the operation to consider. Below we study the case that such a search engine has to crawl a very large set of websites and at the same time to serve to a large number of customers that produce queries in excessive rate. In this case the search engine will run in cluster.

One thing to consider is that in a large scale search engine, multiple crawlers are used and there must exist synchronization between them regarding which crawler visits which site and also avoid overlaps by having sites visited from more than one crawler of the same engine. Database systems are crucial for this task. Search engines replicate the inverted index to multiple machines in order to achieve the required throughput to serve user requests.

One more thing to take care of is the strategy we implement regarding the rate at which the engine visits domains so as not to overload the servers. Given that multiple crawler exists, they have to synchronize so as not to make requests at the same domain at the same time.

Production search engine must have the ability to download web content very fast and maintain their index fresh and serve newly added content to the users as soon as possible, especially in the case of news. Selection of pages to crawl is another very important aspect so as to maintain freshness and use the limited, compared to the massive total web content of the web, resources of the search engine as efficient as possible. To solve this issue, sampling methods are used for selecting where to allocate the download resources of the engine and at the same time cover as much changes as possible. There are multiple policies for sampling including: Proportion, Greedy, Adaptive, Change-Frequency, Round Robin and more. For a well-developed search engine, historical data related to the changes detected in the past, can be used to predict future download allocation.

Additionally, the inverted index of a search engine must be able to update along with the crawling process. We cannot expect from a production search engine to wait a crawling round to finish and update the index upon that.

Spam detection is also very important for a search engine to maintain the quality of its results. Although, high accuracy spam detection is not an easy task, building a method that uses artificial intelligence classifiers along with multiple spam detection heuristics such as Number of words, Number of words in title, Average length of words, Amount of anchor text, Fraction of visible content, Compressibility and more, can produce great results at avoiding indexing spam content and save computational resources.

Regarding the query serving engine that has to deal with a large amount of users, a technique that is used is the two tier architecture. First tier maintains a pruned index that contains documents that are requested frequently and appear on higher places of results. Second tier is the full index and is used when first tier fails to service a user or user gets in greater depth than first tier can serve. Pruned index requires significant lower computation resources. Pruning policies are based on keywords pruning and/or document pruning. There are techniques to know if the results produced from the pruned index will be the same as if the query was served from full index, thus maintain results quality intact.

Of course running such a search engine requires low latency and high bandwidth of internet connection. For this reason, it is wise to scatter computational resources around the globe and placing them to different places according to the expected traffic of each area. Using index replication along with pruned index, a search engine can scale to serve millions of users at the same time.