Group Project Report

Text Search Based on Hadoop & Spark

Subject: Data Processing Workshop II (1002)

Course Teacher: Dr. Changjiang ZHANG

Group 36

薛劭杰	1930026143
马铭泽	1930006138
朱奕卓	1930004048
林晓涛	1930026081

Abstract

Overall

In the group project, we make a Text Search system based on Hadoop/Spark which includes two major components, Text files collection and indexing and ranking. And implement the methods of Hadoop, HDFS, Spark, web crawling, text processing, indexing, ranking, scoring, map-reduce, web design, and third-party library. Finally, achieve the goal that retrieving text contents related to the query and ordering them according to relevance.

Role distribution

Justin majored in web crawling. Jack implements the function of text processing, indexing, and ranking based on Hadoop and HDFS with the third-part library. Bruce is in charge of the overall web page frame connection and David implements the web function details in searching and timing.

Workflow

Firstly, collecting the novel from the Gutenberg website with the crawling method. After we get the files. We build a master and worker network basic on Spark and Hadoop to process the data separately. Then we go through all the files and create an index file containing all the words with Elasticsearch to do the indexing job. Then we implement the TD-IDF ranking algorithm to rank the result for our query depending on the relevant keyword. Finally, we build and website and interface with Django to show the result.

Web Crawling

Data collection

Firstly, we ought to obtain the data from Project Gutenberg (http://www.gutenberg.org/ebooks/) which is a free e-book database. The content we get is consists of the Name of the book, author, type, and the content of the text file.

And the content that we want is on the page directly, and the information we want is located in a certain keyword like the Author name stays after the keyword "*Author*: ".

```
Title: Christmas Eve

Author: Robert Browning

Posting Date: March 16, 2014 [EBook #6670]
Release Date: October, 2004
First Posted: January 12, 2003
Last Updated: February 4, 2008

Language: English


*** START OF THIS PROJECT GUTENBERG EBOOK CHRISTMAS EVE ***
```

We store the data of text, title, and author into both CSV and the txt file. The CSV save the table consisting of the index, author, book name, content, and the text file which is named as index just store the content.

And to continue getting a lot of text files, we have the find how to confirm the same information getting method from one to another one. After we notice the URL of the website, we can find that it can be written by: (*BookID* is an integer)

"https://www.gutenberg.org/cache/epub/" + BookID + "/pg" + BookID + ".txt"

The code for how we get the content:

```
try:
    error_count=0
    url='https://www.gutenberg.org/cache/epub/'+str(current)+'/pg'+str(current)+'.txt'
    r = requests.get(url, headers=headers,proxies=random.choice(http_proxy))
    # beautiful soup
    plain_text = r.text
    #print(plain_text)
    current+=1

    if(len(re.findall(r"<title>(.+?) ",plain_text))== 0 and re.findall(r"Language: (.+?)\r",plain_text))[0]=="English":
        with open('./book/' + str(ID) + '.txt', "w", encoding='utf-8') as f:
            f.write(plain_text)
        f.close()
        author_all=re.findall(r"Author: (.+?)\r",plain_text)

        b_title=re.findall(r"Title: (.+?)\r",plain_text)

        print("Finish the book of "+str(ID))
        ID_list.append(ID)
        author_list.append(author_all)
        book_n_list.append(b_title[0])
        text_list.append(plain_text)
        ID+=1
    else:
        pass
except:
    print('error')
```

After we get all the data for each book. We rename the text by the author's name and

the book name. Since some punctuation is not allowed to use in the name of the text file, we replace all of them in the space. And then put them together which separated by "_". As for multiple authors, we use the "&" to separate them. As for the convenience of processing the data. We set the text type from UTF-8 with BOM to the UTF-8. the book after rename is shown as follow:

The Enemies of Books_lades&William.txt

The code for rename the file:

```
# changing the name of the txt file
import os
count_at=0
count_low=0
replace_list=[ '\\' ,"/" ,':',"?", "#" ,'"','<','>',"|"]
for index,row in df.iterrows():
    name=str(row['Book_name'])
#     try:
    #print(index+1,row['Author'],row['Book_name'])
    string=name+"_"
    for i in eval(row['Author']):
        string+=str(i)[1:]+"&"
    string=string[:-1]
    string+=".txt"
    #print(string)
    ori=str(index+1)+".txt"
    new=string
    for each in replace_list:
        new=new.replace(each," ")
#     print(index)
    error_time=0
    neww=new
    while(1):
        try:
            os.rename("C:\\Users\\Lenovo\\jupyter\\cralw\\book\\"+ori,"C:\\Users\\Lenovo\\jupyter\\cralw\\book\\"+neww)
            print("index done")
            break
        except:
            error_time+=1
            neww=new[:-4]+" V"+str(error_time)+".txt"
            print(neww)
            continue
```

Since this website may ban our IP from time to time. We manage an IP pool from the Internet. If our IP is a ban by the server, we can randomly select another IP for use.

Code for maintaining the IP pool:

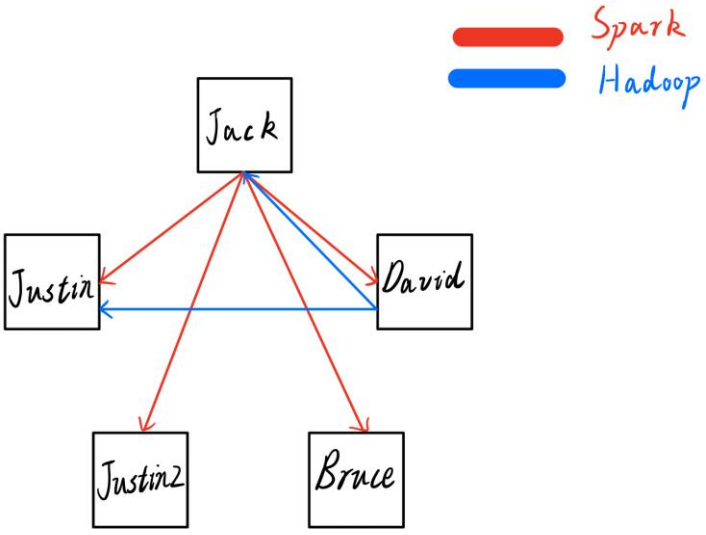
```
# grab proxy
import random
page=999
url = 'http://www.xiladaili.com/gaoni/'

headers = {'User-Agent':'user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
           ' (KHTML, like Gecko) Chrome/81.0.4044.138 Safari/537.36','Cookie': 'session_id=1b1f4756d9d79803e06391a7699920ee686e75f8'}
http_proxy = []
num=0
while num<64:
    page+=1
    current_url=url+str(page)
    page_text = requests.get(url, headers=headers).text
    html_soup = BeautifulSoup(page_text, "html.parser")
    tables = html_soup.find("tbody")
    lines = tables.find_all("tr")

    for line in lines:
        each=line.find_all("td")
        if each[1].text=="HTTPS代理" or each[1].text=="HTTP,HTTPS代理":
            dic = {'https': each[0].text}
            http_proxy.append(dic)
            num+=1
print(len(http_proxy))
#print(http_proxy)
```

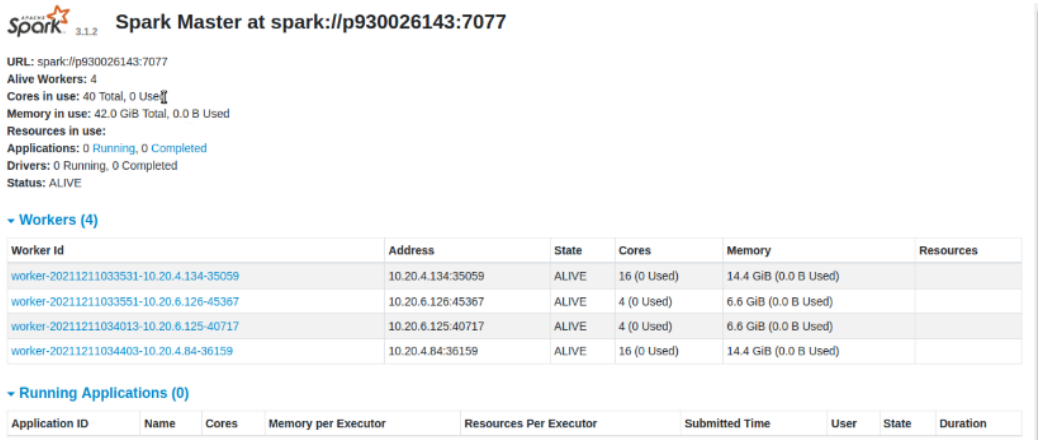
If we want to design the system collect and analyze the text data in real-time. We can set the constant time which determines the cycle time for us to crawl data. After comparing with the book name, if new data appears, it will be put into a new folder, and then an elastic search will read the corresponding new data and convert it into the index for quick search when opening the website. Whenever the users open our website, there will be updated content presented to the user. And we finish the update of the results dynamically.

Inverted Index & TFIDF Implement

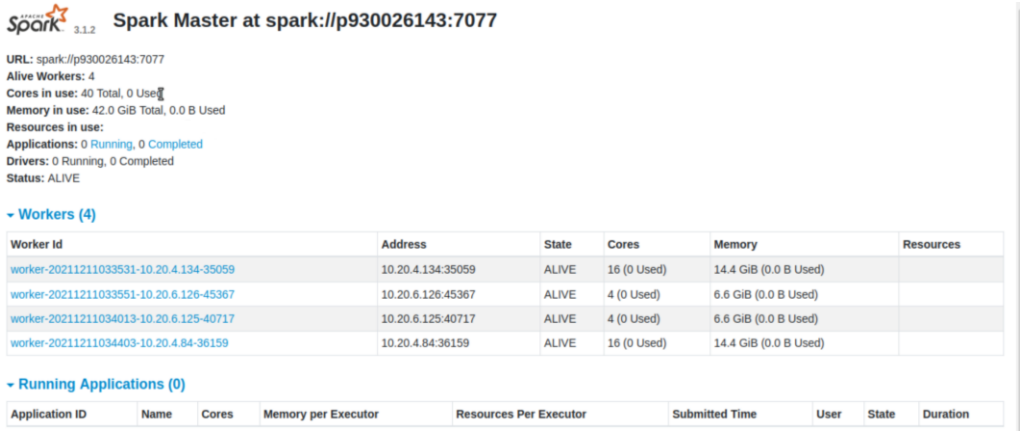
Hadoop/Spark structure cluster

We have five machines to process the data in Hadoop/Spark cluster. For each cluster, we have a different master. Jack's machine as the Spark master and David's as the Hadoop master. Others will be the slave if it's not the master in the cluster. To conclude, we have five machines for Spark cluster and three machines for the Hadoop cluster.





The Configured Capacity is 38.42 GB. Our DFS Used is 3.08 GB, which occupy by 8.02%. And the Live Nodes are two, it means that set up Hadoop cluster successfully.



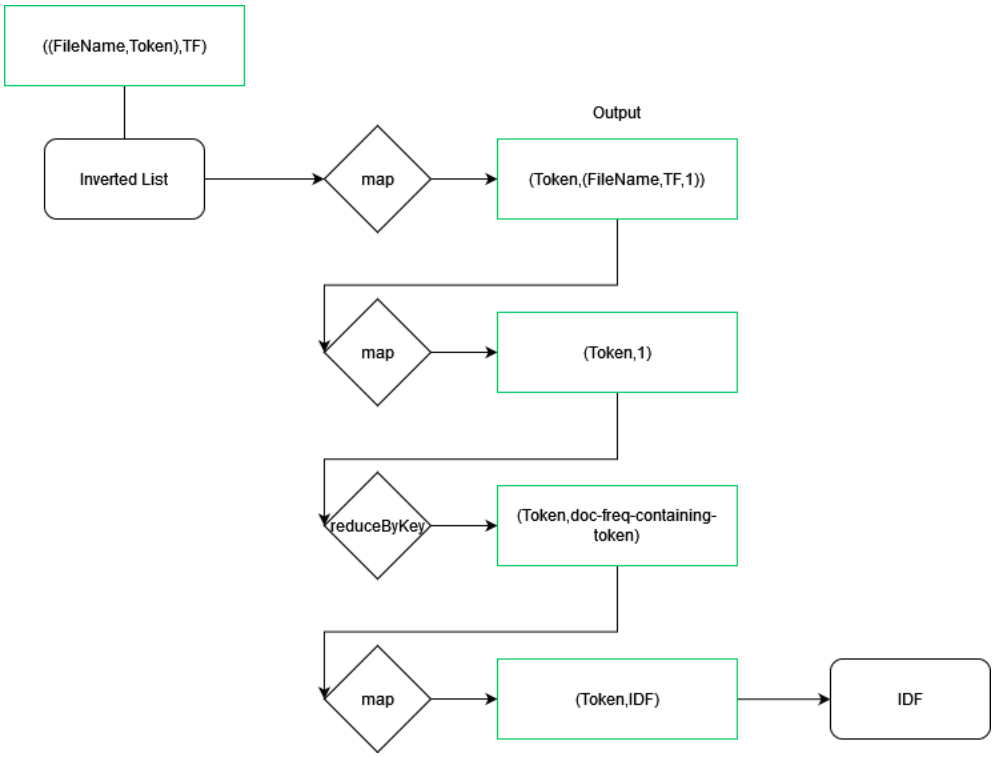
The master's URL of Spark is spark://p930026143:7077. For our Spark cluster, we have total 40 cores and 42 GB memories. And in workers table, it is obvious that our four workers are alive. It means the Spark cluster is set up successfully.

Indexing

- Definition:

  The inverted index is an indexing method that is used to store a mapping of the storage location of a word in a document or a group of documents under a full-text search. It is the most commonly used data structure in document retrieval systems. Through the inverted index, you can quickly get a list of documents containing this word according to the word. The inverted index is mainly composed of two parts: "word dictionary" and "inverted file".

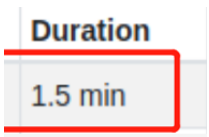
Process



Firstly, prepare the files you need to import. Then use $wholeTextFiles$ to import the full text of a text file. Next use $flatMap$ to transform the new keys and values. Keys are changed from $FileName$ to ($FileName$,Token). Values are changed form FullText to 1. After changing, use reduceByKey to groupby key and sum the value. The last step is also the most important step, using map to transform new keys and values. Keys are changed from ($FileName$,Token) to Token. Values are changed from num to (FileName, Count). The result is Inverted List.

Time cost

Cluster



100M

▾ Finished Executors (1)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs
0	KILLED	16	8.0 GiB		**ID:** app-20211211035725-0000 **Name:** inv.py **User:** root	stdout stderr

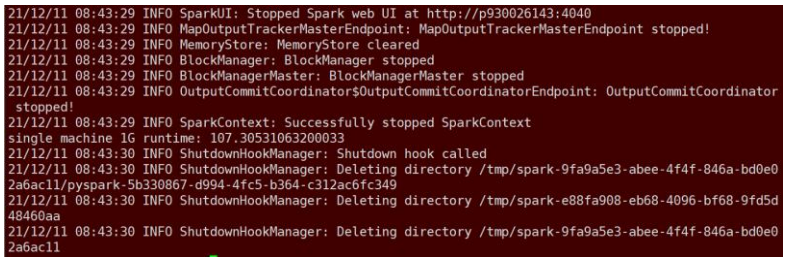
Duration
15 min

1G

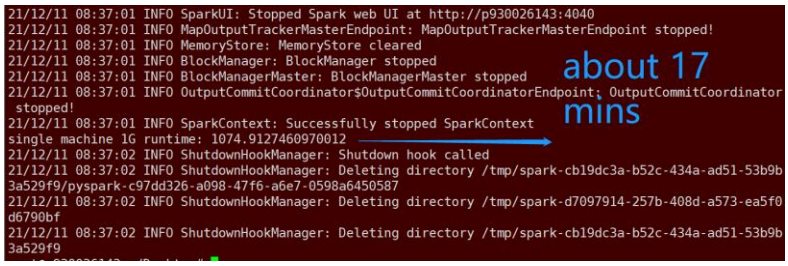
The Spark cluster running 100M files spends 1.5 minutes.

The Spark cluster running 1G files spends 15 minutes.

Single machine



100M



1G

One machine running 100M files spends about 1.78 minutes.

One machine running 1G files spends about 17.91 minutes.

After comparing, we can find the Spark cluster has stronger ability of calculation.

Ranking

- Definition: TF-IDF (term frequency–inverse document frequency) is a commonly used weighting technique for information retrieval and data mining. TF is Term Frequency, and IDF is Inverse Document Frequency. TF-IDF is a statistical method

used to evaluate the importance of a word to a document set or a document in a corpus. The importance of a word increases in proportion to the number of times it appears in the document, but at the same time it decreases in inverse proportion to the frequency of its appearance in the corpus.

- Process

$$TF - IDF = TF * IDF$$

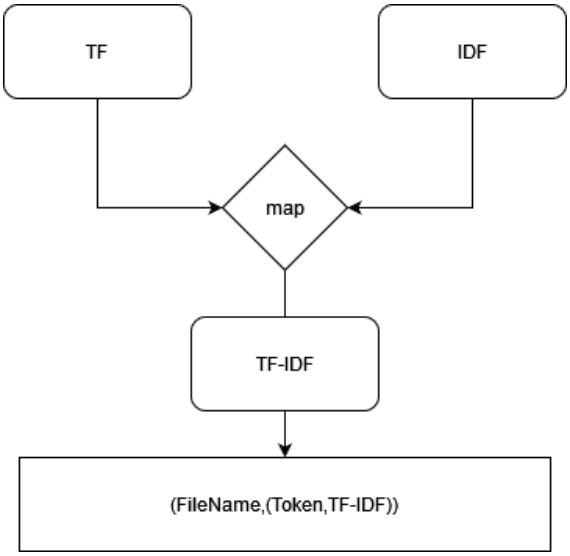
$$IDF_i = \lg \frac{|D|}{|\{j:t_i \in d_j\}|}$$

|D|: Number of Total Files

$|\{j: t\_i \in d\_j\}|$: Number of files which includes Token.

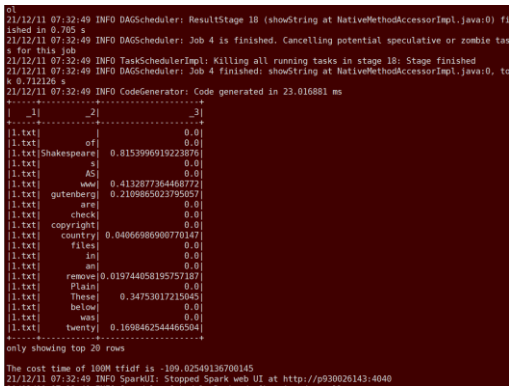
In inverted index, we have already introduced how to calculate the TF. So now we focused on how to calculate IDF.

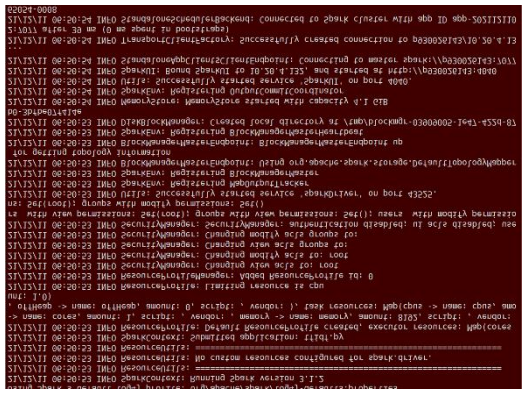
Firstly, use map to change keys and values. Keys are changed from ($FileName$,Token) to Token. Values are changed from TF to ($FileName$,TF,1). Then still use map to transform. Keys are not changed. Values are changed from ($FileName$,TF,1) to 1. Next use $reduceByKey$ to calculate the total number of tokens. Finally, use map to transform. Values are calculated by $\lg \frac{|D|}{|\{j:t_i \in d_j\}|}$. The results are IDF. After you get TF and IDF, the last step is joining them.



Result

Cluster:



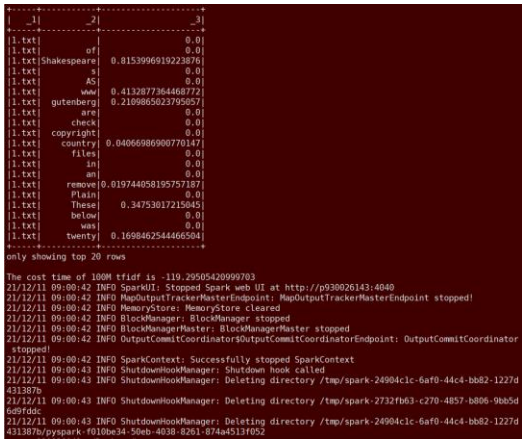


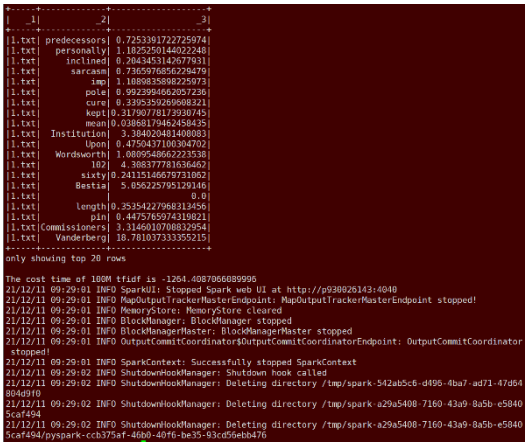
100M 1G

The Spark cluster running 100M files spends 1.82 minutes.

The Spark cluster running 1G files spends 20 minutes.

Single machine:





100M 1G

One machine running 100M files spends about 1.98 minutes.

One machine running 1G files spends about 21.1 minutes.

After comparing, we can find the Spark cluster has stronger ability of calculation.

Problem

- Memory Problem:

  Then, we encountered some problems during the operation. First, we run 100M on *Jupyter* and there will be no problems, but when your file size reaches 1G, we will report an error. The default running memory of spark is only 2g, but it is not

enough for map reduce of large files. It is easy to cause memory overload, so we transfer the file to the local and use the command line to run the following command to submit the task, here we set the running memory and successfully solved this problem.

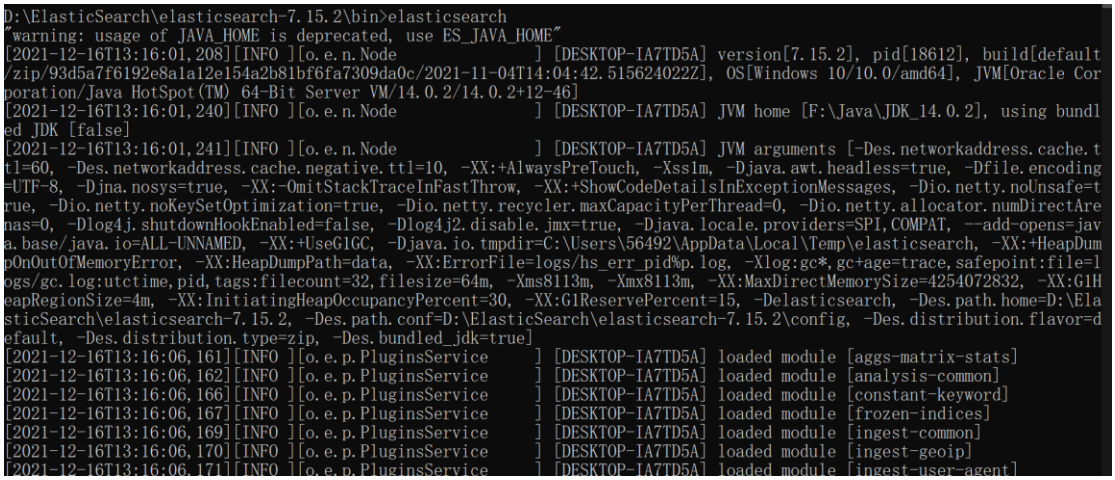
- Time Problem

  The effect of our map reduces on $django$ is very poor. Although it has processed the data into a very considerable form, it still needs to traverse the huge rdd table structure when the user searches, and its index is not obtained. Very good optimization, so the search takes a long time.

Index by ElasticSearch

Principle of ElasticSearch:

1. Firstly, we ought to download Elastic Search Tool in the official website, and then Start the search service in the bin directory of this folder. Because this indexing tool is developed based on the java language, we need to configure it with JVM environment (the version is preferably newer than JDK10) and some basic packages.



2. read file function will initialize the information list, and then we can make some process like split, merge and other some operatiosn to get the feature we want. For our purpose, we just get the bookname, author, and the contect of text.

```
def readfile():
    docx_name=[], filename=[], all_file=[], content=[]
    for root, dirs, files in os.walk(PATH_DATA):
        temp_content = []
        for every_docx_name in files:
            every_file_path = os.path.join(root, every_docx_name)
        with codecs.open(every_file_path, encoding='utf-8') as f:
            temp = f.read()
            attribute=every_docx_name.split("_")
            book_name=attribute[0]
            author_lst=attribute[1][:-4]
            wy.Index_Data(filename[x],book_name,author_lst,temp,cishu)
```

The Elastic Search class & Essential function:

1. Constructor

Create the name of your indexer, and specify the local service address of *ElasticSearch*. The default address is this machine. Because the data volume is only 1G, there is no need to create an *ElasticSearch* cluster. Then the default port number is 9200, we can also see the details on the web page.

2. *create_index* function

Define the index type and we should also use creation for the mapping of index. We can define the search mode result by the analyzer and *search_analyzer*, set it to *ik_max_word* which means that we sort by the max top TFIDF.

```
class ElasticWy:
    def __init__(self, index_name,ip ="localhost"):
        self.index_name = index_name
        self.es = Elasticsearch([ip], port=9200)
    def create_index(self,index_name):
        _index_mappings = {
            "mappings": {
                    "properties": {
                        "source": {
                            "type": "text",
                            "index": True,
                            "analyzer": "ik_max_word",
                            "search_analyzer": "ik_max_word"
                        }
                    }
            }
        }
        if self.es.indices.exists(index=self.index_name) is not True:
            res = self.es.indices.create(index=self.index_name, body=_index_mappings)
```

3. Search function

Get the input from the user, and we define the way for searching which means that query by which feature. In the blue box, we set the content is $True$, it will set the mode for true TDIDF and return the 20 items with the highest scores. If it does not be set, it may return the random and duplicative items. As a result, the output is just the inverted index but not TFIDF. Here is an example for searching by author of the book.

```
    def Search_author(self,input_text):
        doc = {
            "query": {
                "match":{
                    "author_lst": {
                            "query": input_text,
                            "operator": "and"
                    }
                }
                'content':True,
                'topN':20,
            }
        }
        _searched = self.es.search(
            index=self.index_name,
            body=doc)
;
```

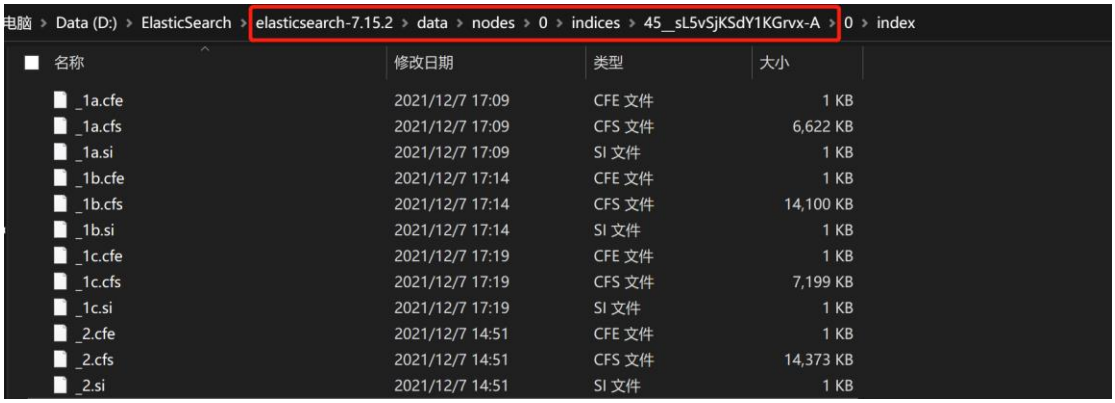
After insert index and search by user, the output items' attributes information will store in a two-dimension dictionary. we can get the same tuple of the search item, include other attributes, TFIDF score and the word position in the text. Finally, it will return the result to the users.

```
for hit in _searched['hits']['hits']:
    print (hit['_source'])
    last_category.append(hit['_source']['category'])
    last_docxname.append(hit['_source']['name'])
    last_author.append(hit['_source']['author_lst'])
    last_sentence.append(hit['_source']['title'])
```

Instantiate the class and we can use the function on the $ElasticWy$ class.

```
# instantiation the class
wy = ElasticWy("Jack", ip="localhost")
wy.create_index(index_name="Jack")
```

After inserting the index, it will create a node folder to store the index. The more files you have, the larger the size of the index file. It is worth noting th at this is not a database, but an index saved in the form of $.cfs$,$.cfe$ and $.si$ class files.

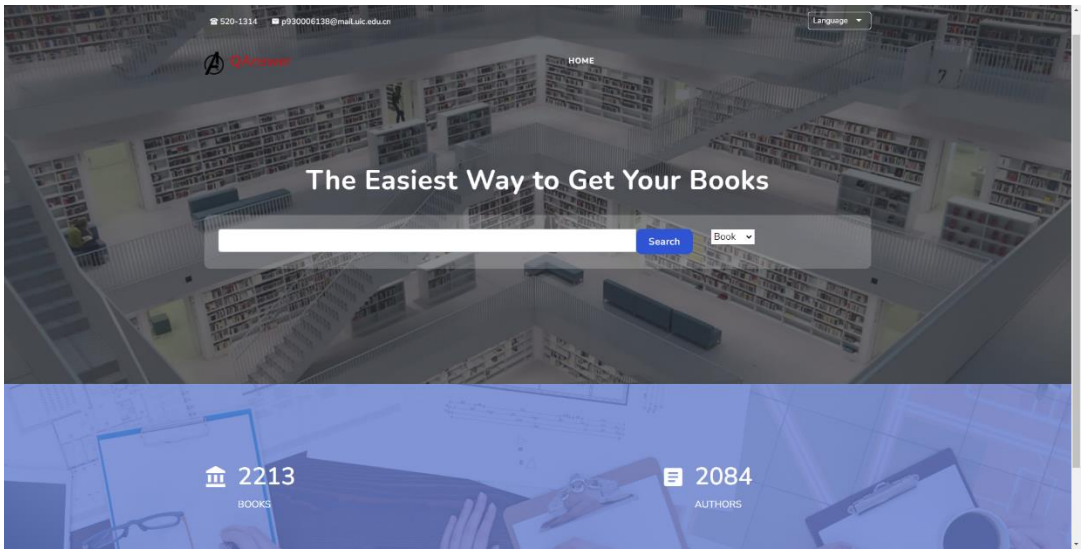


After all, we can combine the function with the Django structure, put them into a view.py so that supporting Searcher Experience Optimization to users.

Connect with the front end with Django

Website Design

Following is the overview all our website. First is the home page. Below the home page shows the total amount of book and author stored in our library. For the main search function, we implement using three methods to be choose, which are search by book, word and author. Thus our user can enjoy surfing the website with what exactly type of result they want. After input the key word will jump to the search result page.



Here stores the top 20 TF-IDF result among our library. In each Browse box we place the title of the book, author and abstract which makes user easily access the book. After getting the result, user can also choose back to the home page and continue searching. From the select box, users can select what feature they want to search by. They can select by the Author, detail word TFIDF and the book title. Any time there will show the time cost and the TFIDF score, and users can click the $'detail'$ button to see all the context of the book. Most importantly, the time cost i's usually less than 0.2 seconds.