Text Search Engine

Group 11

张世辰 1830026154

魏焕哲 1830026116

何其睿 1830026032

Table of Contents

[Data Collection 1](#_Toc61344370)

[Data Indexing and Ranking 4](#_Toc61344371)

[Inverted Index by Hadoop MapReduce 4](#_Toc61344372)

[Inverted Index without MapReduce 7](#_Toc61344373)

[Compare the performance between using MapReduce and without using MapReduce 9](#_Toc61344374)

[Ranking algorithm 10](#_Toc61344375)

[TFIDF-ranking implementation 11](#_Toc61344376)

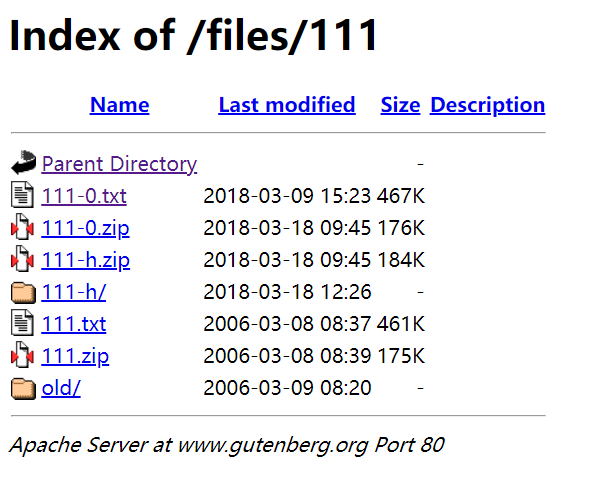
[Web interface and database 13](#_Toc61344377)

# Data Collection

Our data is text data, we collected our data from Project Gutenberg website, the url is “http://www.gutenberg.org/”, this is a library of over 60,000 free eBooks, so the contents of our text data are the contents of many books.

We use BeautifulSoup library in python to crawl the data we need.

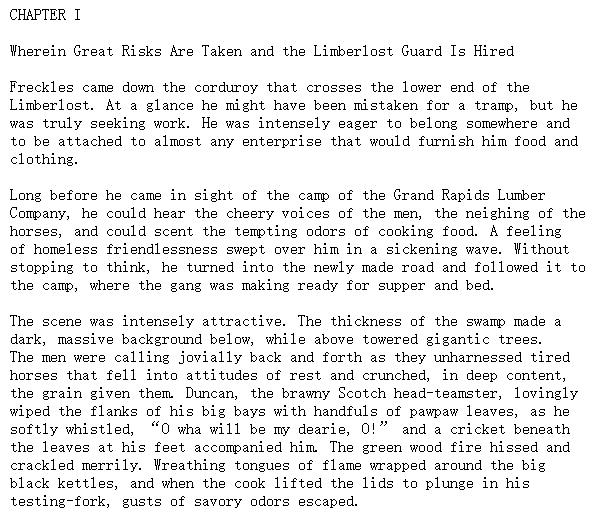
In Gutenberg website, for each book, we found that there is a webpage looks like this, the sample url is “http://www.gutenberg.org/files/111/”.



This page stores the related files of this book, and we found that the txt file stores the textual content of the book. Therefore, firstly we collect the url of the txt file for each book, the code is shown below. We use a for loop to iterate the webpages mentioned before of 500 books, then we use BeautifulSoup to get find the corresponding url of the txt file, if the webpage response with error code 404 or the txt file cannot be found in the webpage, skip current book and go to next book.

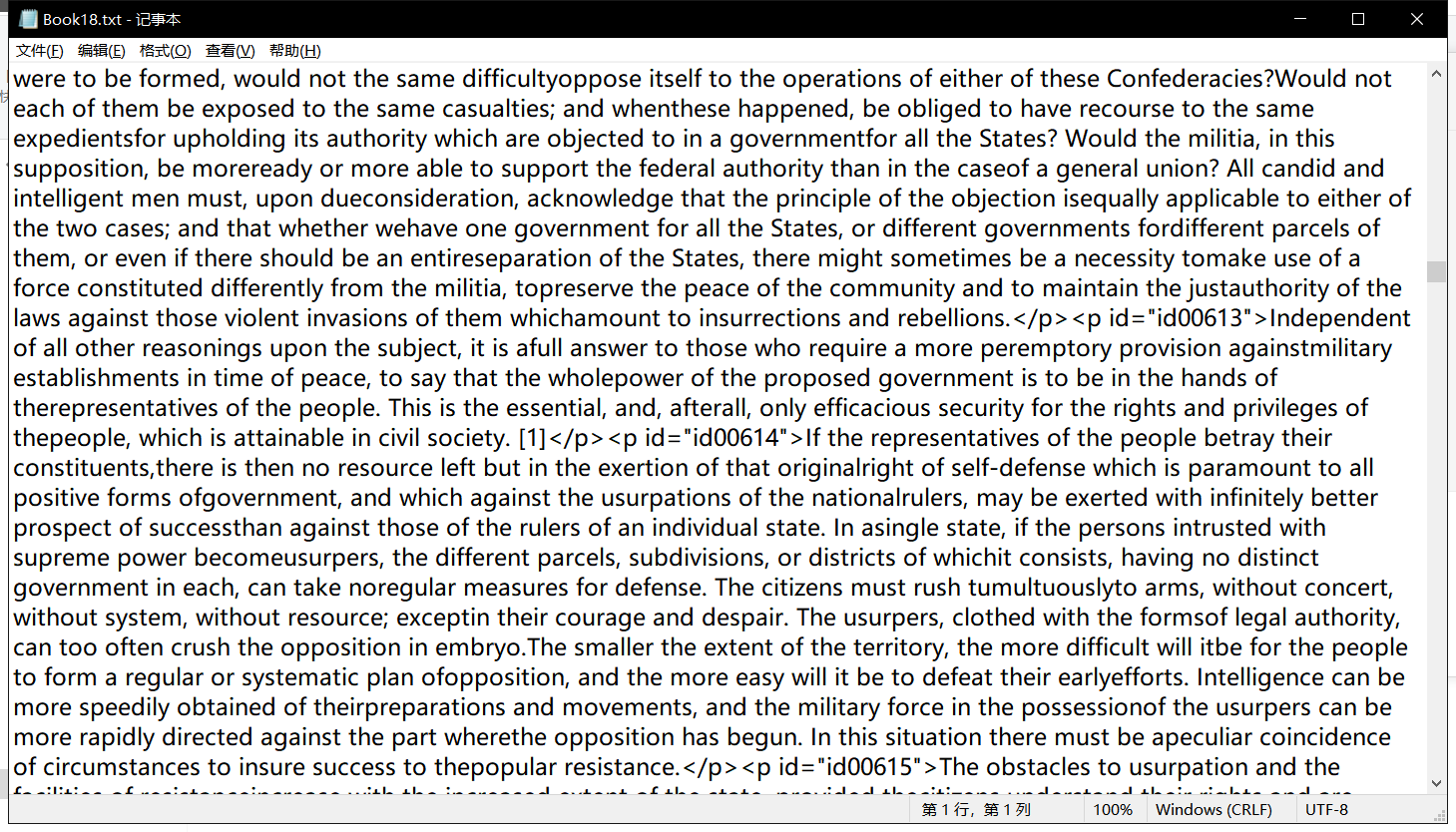


The content of each txt file webpage and the code used to crawl it are shown below, we just get the html contents of this page and store them in a new local txt file. We did not select the tags in the html contents because many different webpages have different tags, we cannot find the rules in them; directly using html contents as text may lead to some tags like <p> and illegal words such as “aaaa” and “ILOVEYOU” appearing in our final txt files, since the project mainly focus on indexing and ranking of text, we think this kind of error is acceptable, we can just treat the illegal words as a word that we have never seen.

文本

描述已自动生成

Eventually, we have crawled 500 books from this website to do the indexing and ranking, and 800 books for the comparison of indexing in different ways. The total file size is more than 200 MB, we can crawl more files, but the performance of indexing and ranking was not good, it takes hours to map and reduce the files, therefore, we just use 200MB files to finish the job. The sample txt file we finally got is shown below, we can see that the text content is generally as expected, which means our data collection is completed.



Additionally, we have implemented the code for crawling the books along with their titles, as shown below, it would be useful in practice, but we encountered some problems when putting these text files named with book titles into hadoop, so, we did not use these files with titles, but we will submit them with the code.



# Data Indexing and Ranking

In data indexing and ranking part, we mainly focusing on using the data we crawled from the website to do the inverted index and the TFIDF ranking for the files. We do two parts to compare the time when we do the inverted index part. And we add the ranking part in inverted index.

# Inverted Index by Hadoop MapReduce

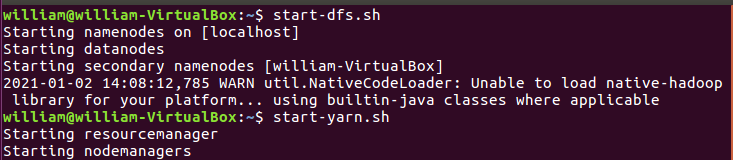
The first part is using Hadoop MapReduce to do the inverted index.

We get the class from the java code using

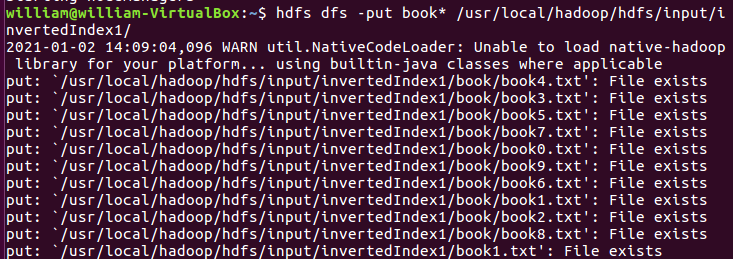
$ hadoop com.sun.tools.javac.Main TFIDF\_InvertedIndex.java

And we put all the classes into a jar file. This jar file is used to run our java program on Hadoop MapReduce.

We start Hadoop system to run it.

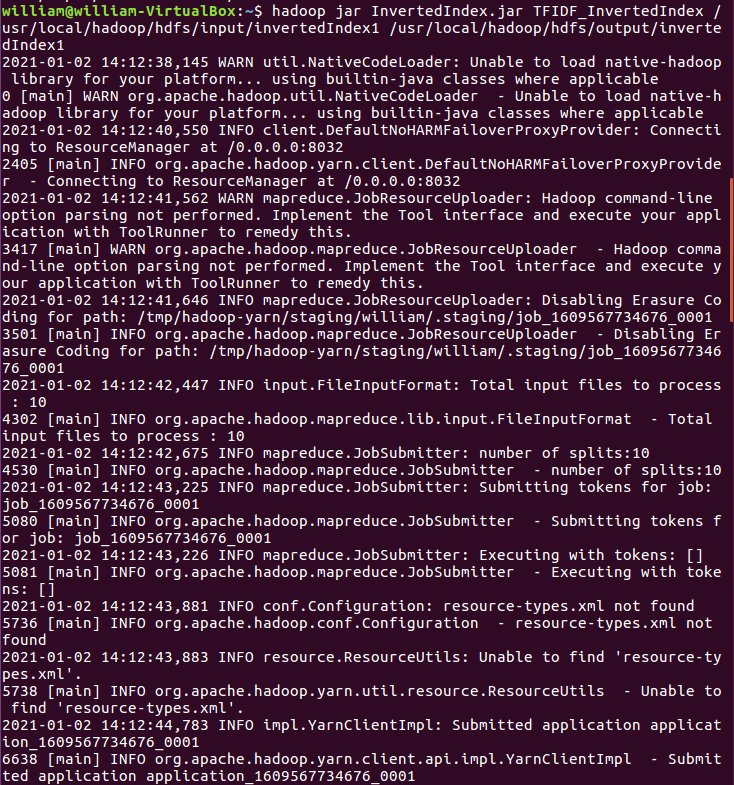


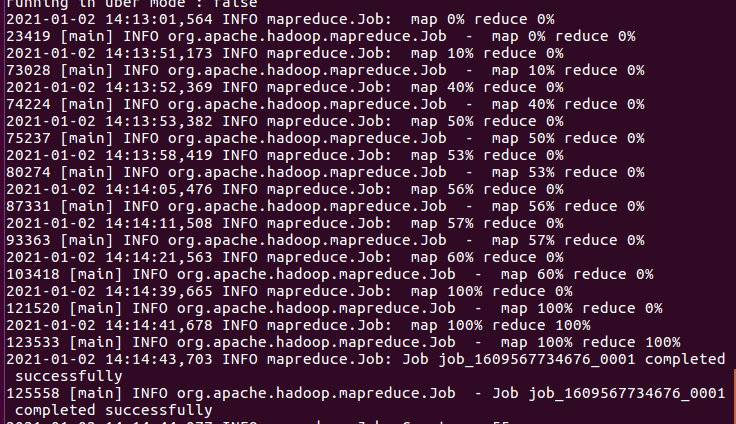
Then we put all the book.txt files into the directory in hdfs we create.



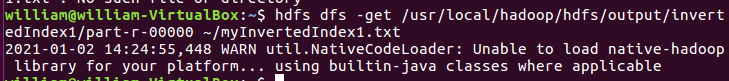
The above file is smaller than 10 m. We also have the file which is around 400 m in the next part.

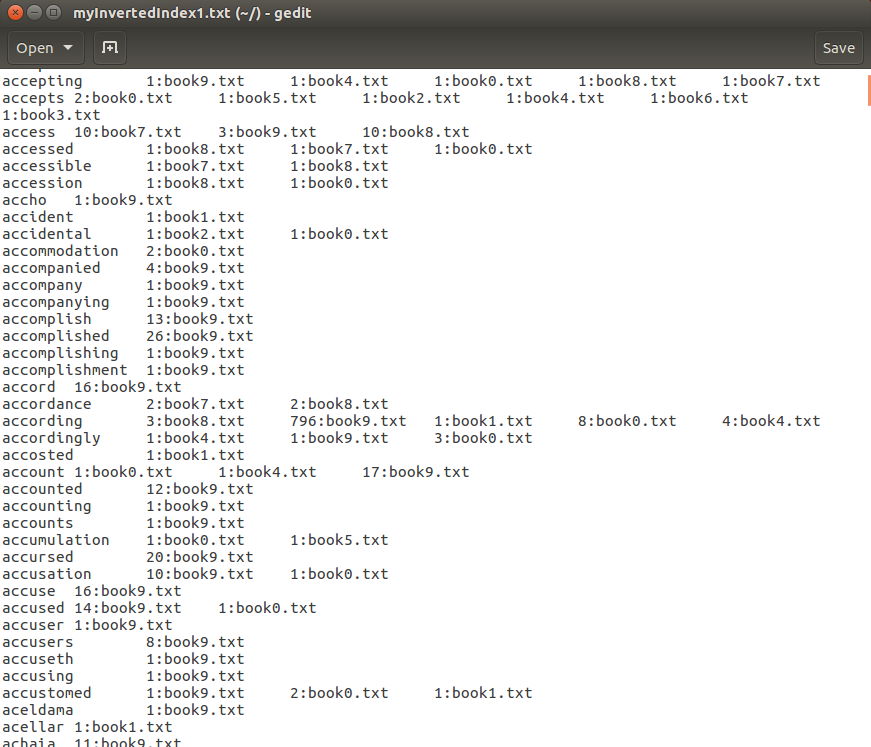
After putting the file into the directory, we start Hadoop to run our java program and do the inverted index job.



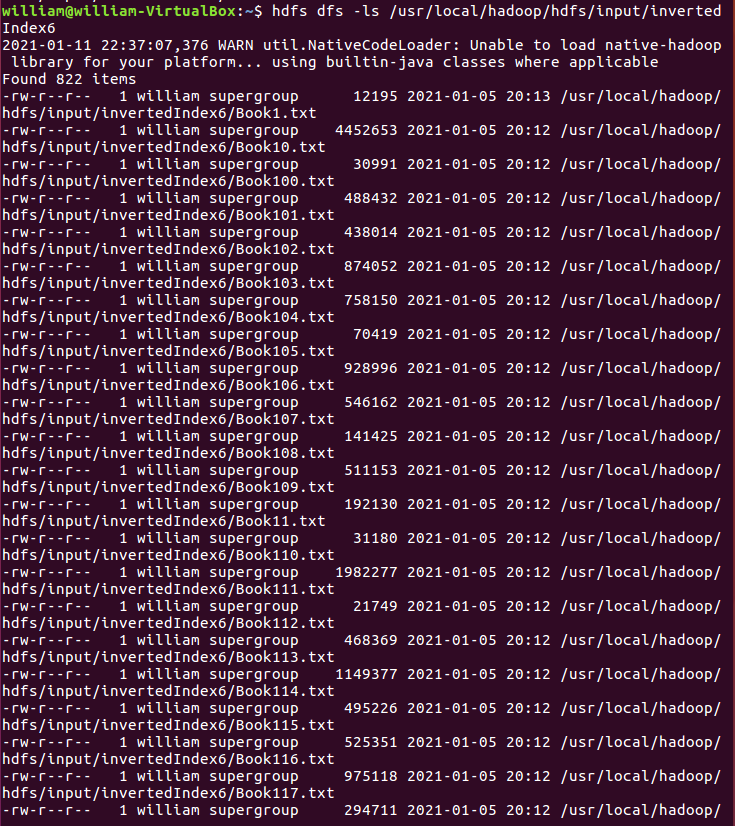


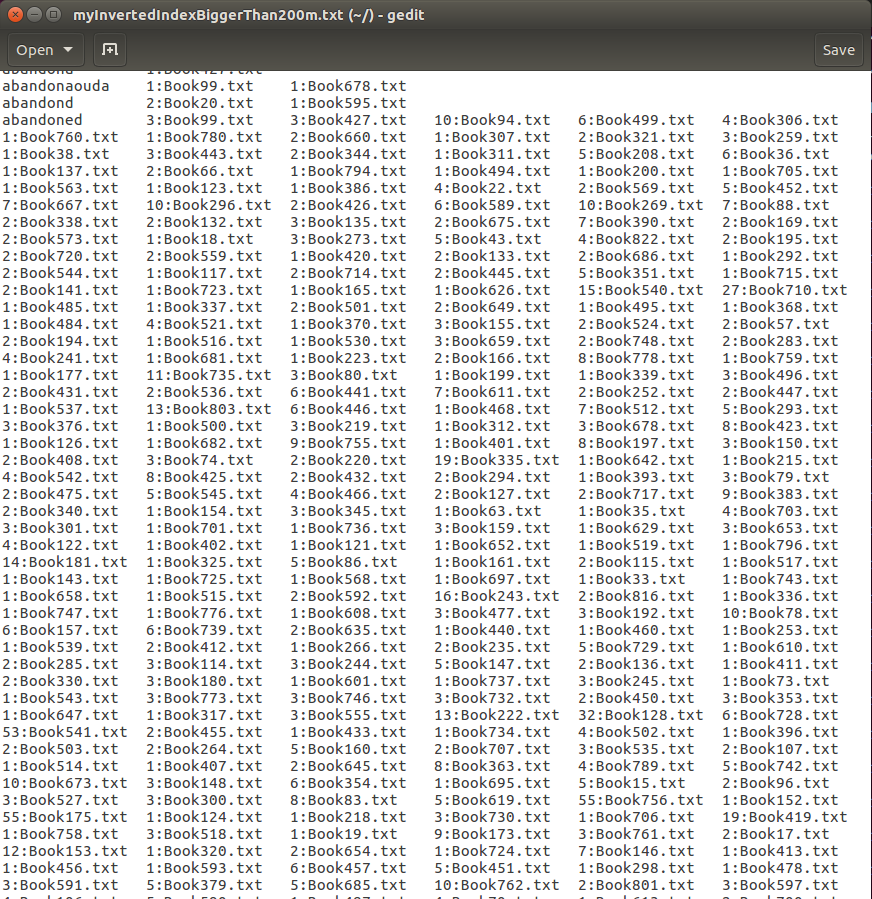
We can see that the job is completed successfully. Then we put the results into the txt file. This txt file can be used in the search engine since we can put it into the database. But we choose to use the final files which is bigger than 200 m and do the TFIDF ranking to our search engine.





The inverted index which is larger than m is shown below.





# Inverted Index without MapReduce

We also do the inverted index without using MapReduce, we directly use a Java program to do the inverted index. And we do comparison of time between with and without using MapReduce.

The code construction for our Java program is shown below.

Code construction:

invertedIndex.java(build HashMap to produce dictionary Index.dict)

run.java (main function)

2. Implementation ideas:

invertedIndex. Java

(1) Loop through the file

If map does not contain this word item, store DocFreq as 1 and DocId in SMAP.

Store the word and Smap in a map

(2) Otherwise, it indicates that the word has already appeared in the Map, then it determines the word item and the word item read so far

Whether a word item that is already in a file is in a file or not

Increases DocFreq by 1 and adds an ID to the array of doCID.Store in a map.

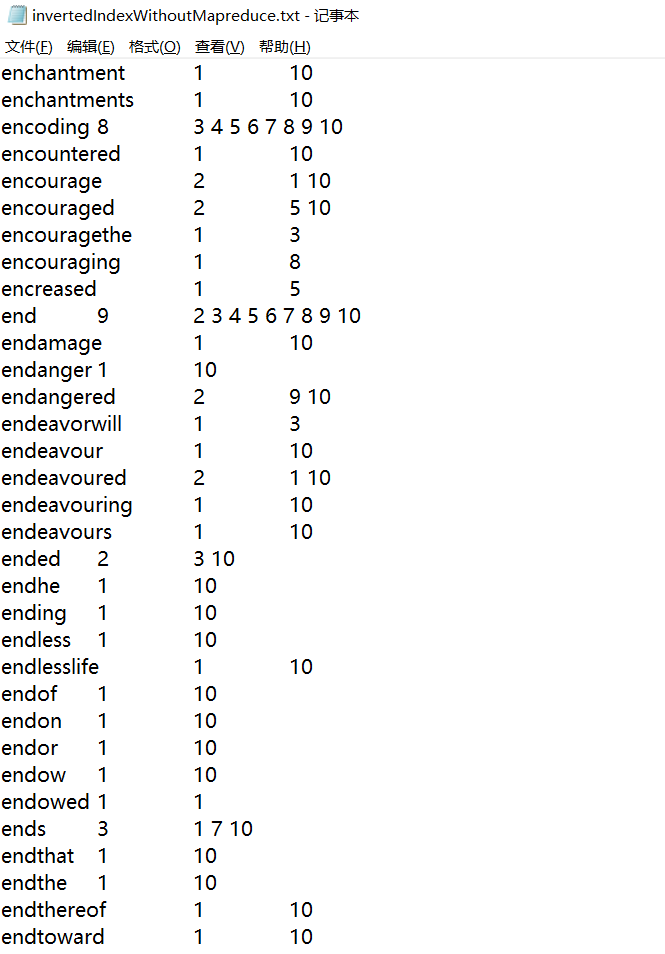
(3) Writes the HashMap to the file using iterators.

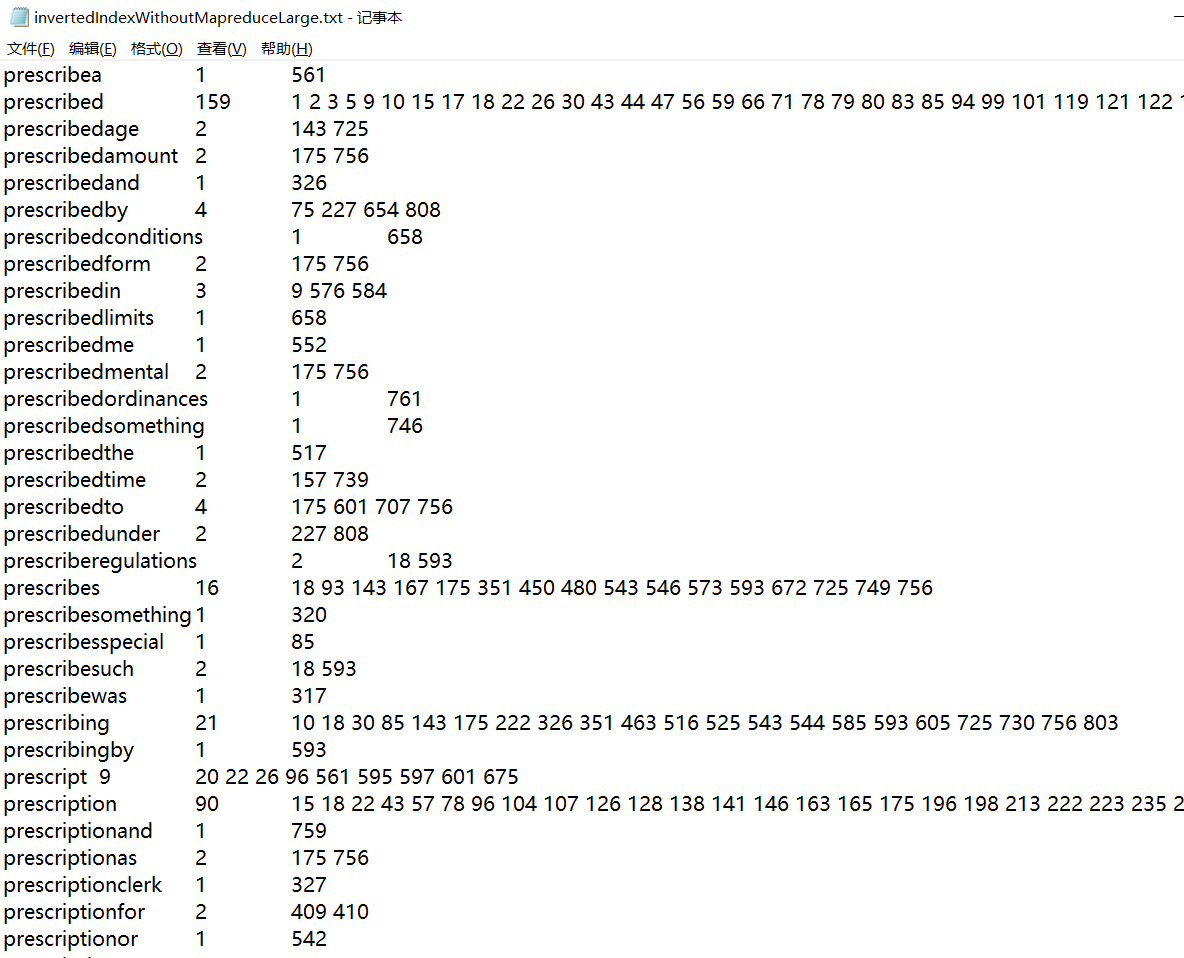
(4) return map.

Run.java

Create a tmpmap by calling invertedIndex.

The running results are shown below (both <10m and >200m)



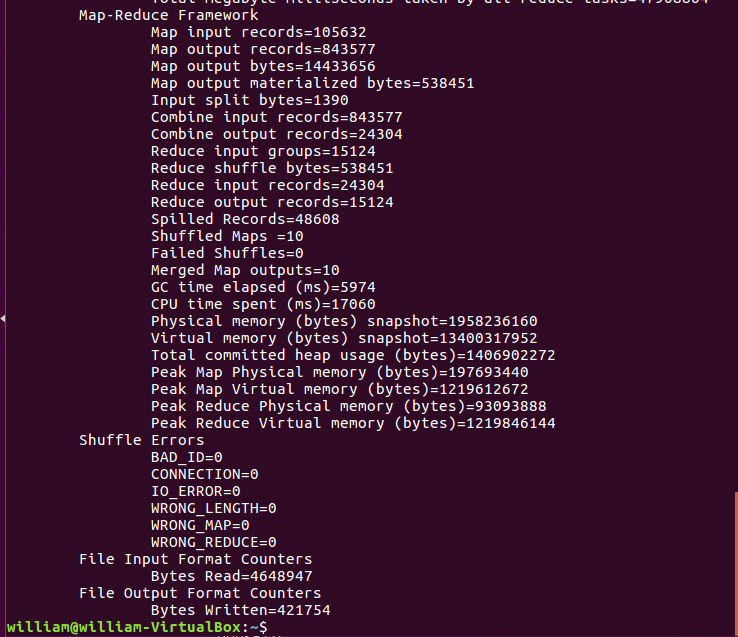


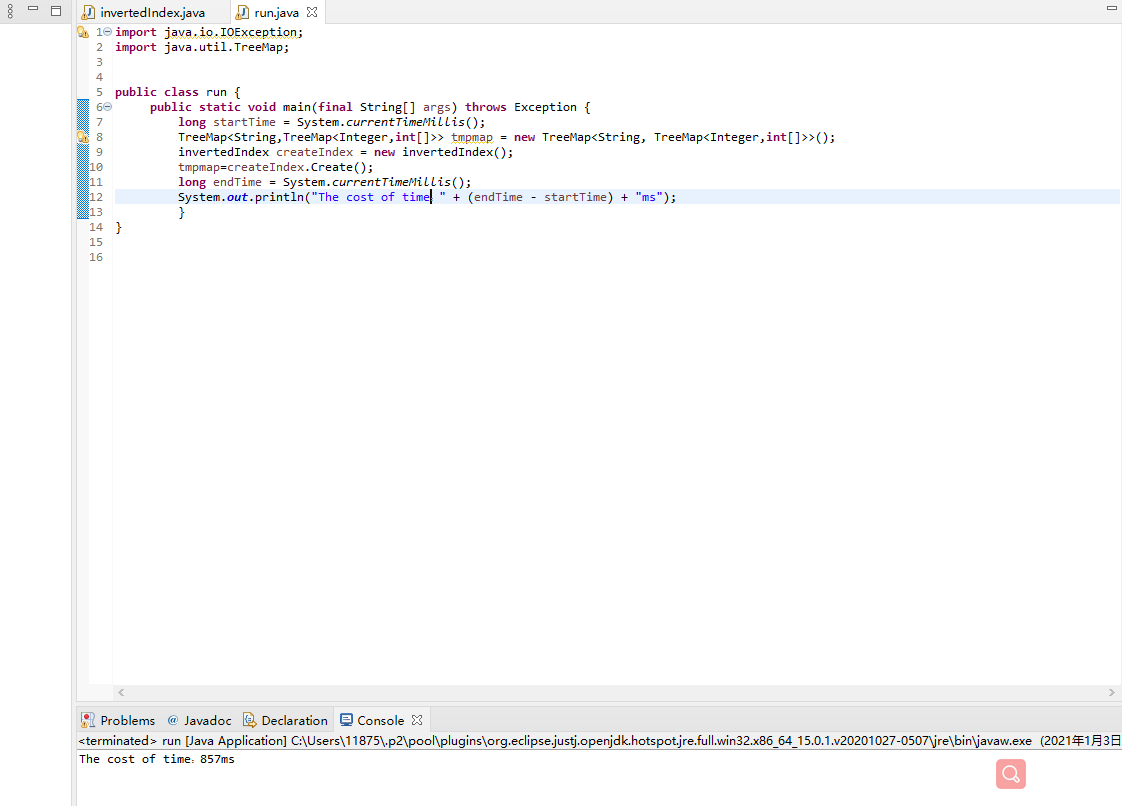
# Compare the performance between using MapReduce and without using MapReduce

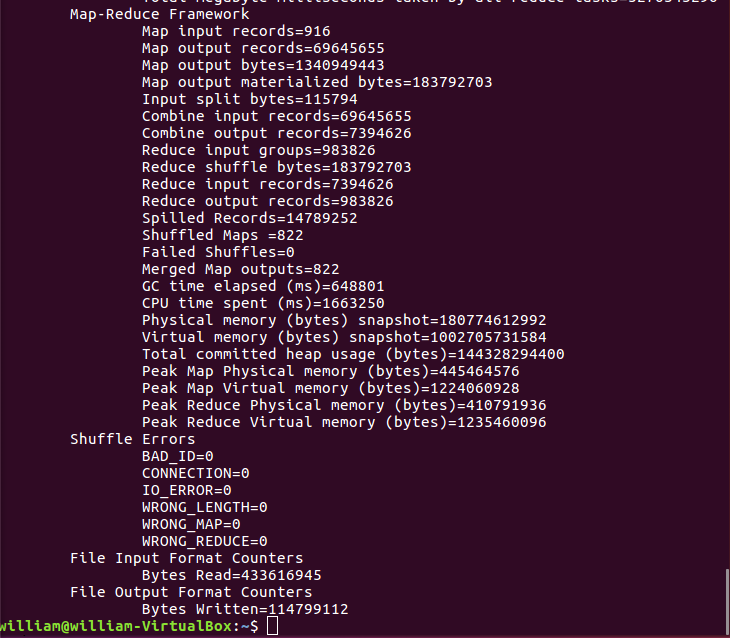
From the picture shown below, we can see that from the CPU time spent. It cost 17060ms to finish the inverted index. And from the Java we implement’s picture, we can see that it only cost 857ms. And this is for the input file which is smaller than 10m.

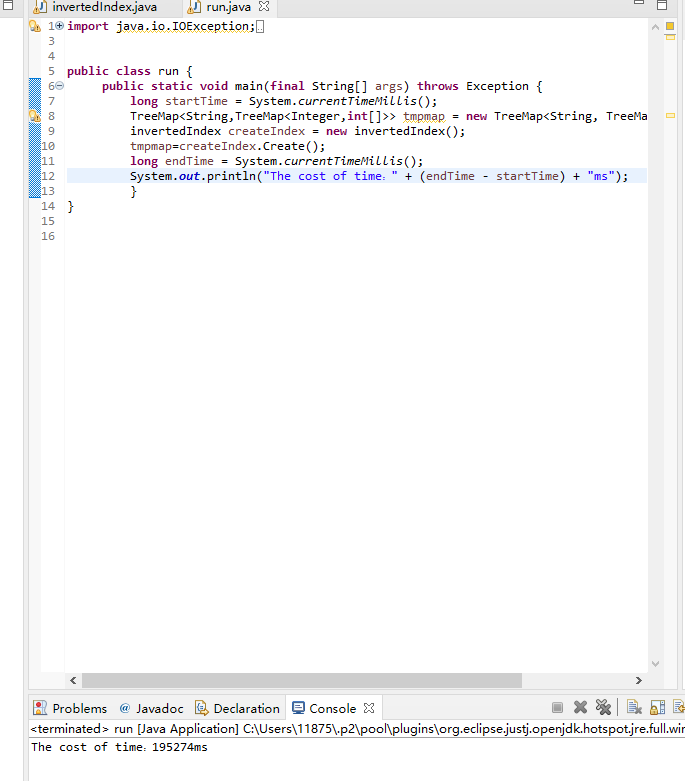
For the file which is larger than 200m, it cost 1663250ms. It is about 4.5 hours. And for Java, it takes about 195274ms. It takes about half an hour.

So, we can see that inverted index runs in a single computer is somehow faster.

****

****

****

****

# Ranking algorithm

Since our data is textual data, the ranking algorithm we chose is TF-IDF, the full name is Term Frequency-Inverse Document Frequency. TF-IDF is a statistical method used to evaluate the importance of a word to one of the documents in a set of documents or a corpus. The importance of a word increases directly with the number of times it appears in the document, but meanwhile, it decreases inversely with the frequency of its appearance in the corpus. In one word, the more times a word appears in a document, and the less times it appears in all documents, the more representative it is for this the document.

TF: Term Frequency, measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Therefore, the term frequency is often divided by the document length (the total number of terms in the document) as a way of normalization:

IDF: Inverse Document Frequency, which measures how important a term is. While computing TF, all terms are considered equally important. However, it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus, we need to weigh down the frequent terms while scale up the rare ones, by computing the following:

Finally, we have:

# TFIDF-ranking implementation

We add TFIDF-ranking into the inverted index. The Implementation is shown below.

1. Get the global Configuration from the main class through the code in invertedIndex.

Get the Hadoop file system instance from the global configuration

3. through the file input interface FileInputFormat and context configuration to obtain the input directory (main method into the parameter directory);

4. Define variable docs, which represents the total number of documents or articles in the input directory; if the value of the variable is 0, the total number of files in the input directory is scanned,

Put it into a globally configured custom variable; If so, it represents the total number of files already retrieved, directly from the global configuration, and only scans the directory once.

Improve efficiency;

5. Define a map to store the name of the word and the TF-IDF value

6. traverse the word in the document (1 or more) fileList.

(1) First, get whether there is a variable named by the file name in the global configuration. If the value is 0, it means there is no total number of entries in the file.

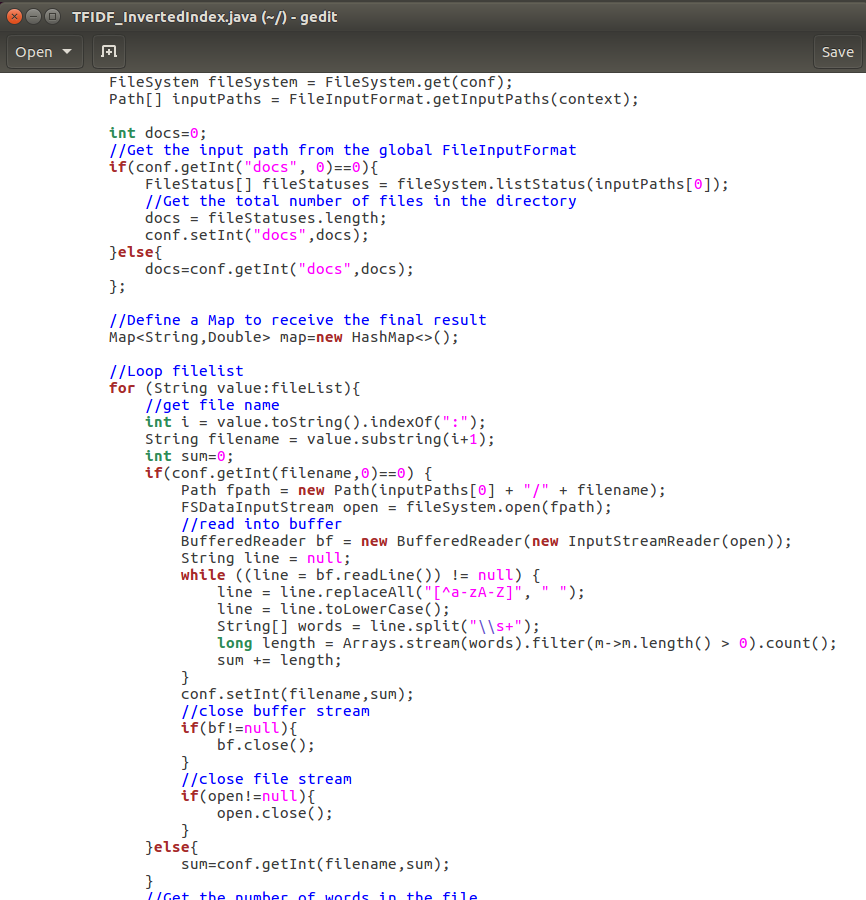
Just get the file from the input directory, create the IO stream, write the file to the BufferedReader, write to the buffer, and get the entry for the map

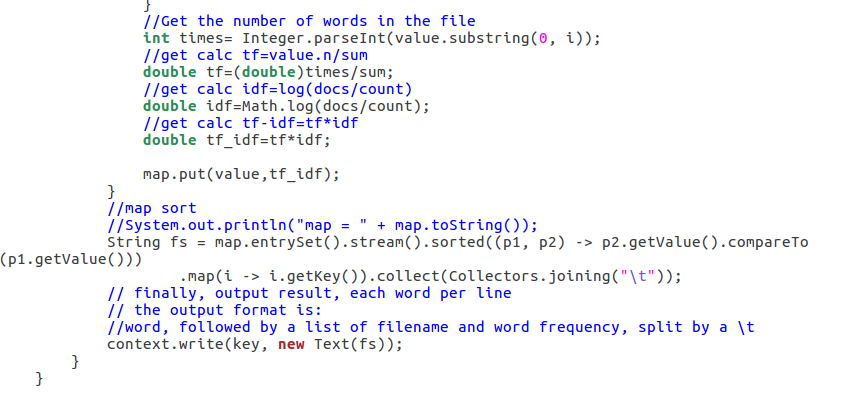
Keep the rules consistent, get the total entry, write to the global variable;

(2) If a global variable has been written, the value can be taken directly from the global configuration to ensure that the file is scanned only once and improve efficiency;

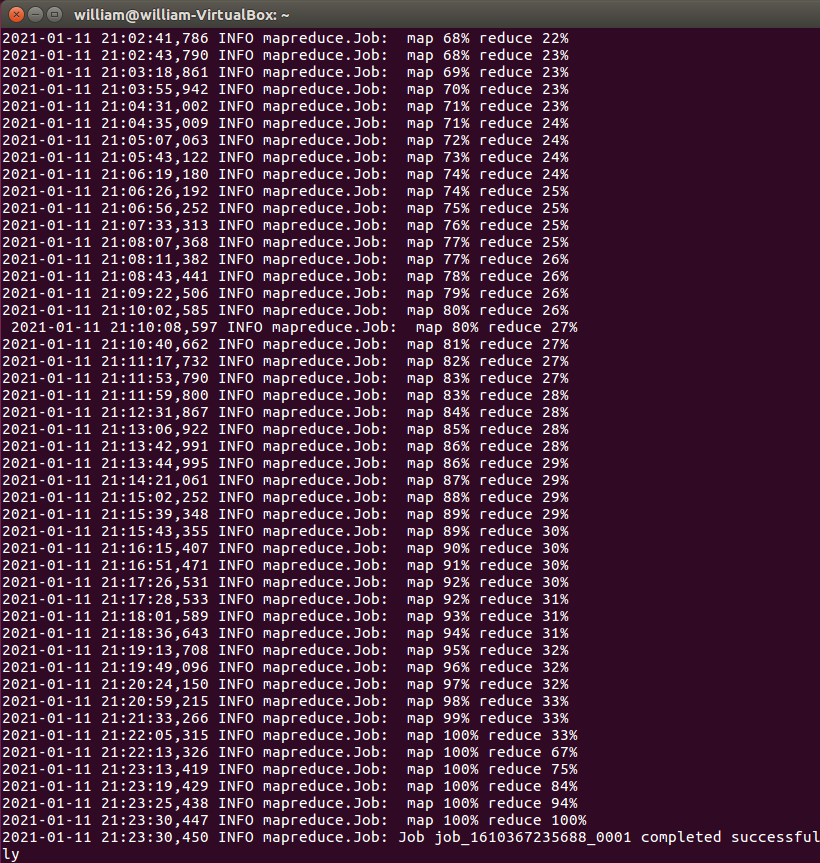
(3) Close the IO stream and clear the buffer.

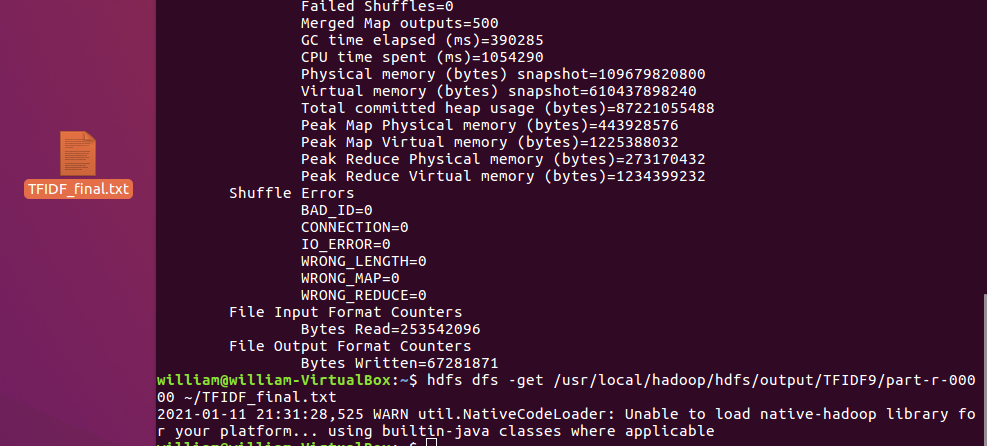
7. Calculate TF IDF and TF-IDF

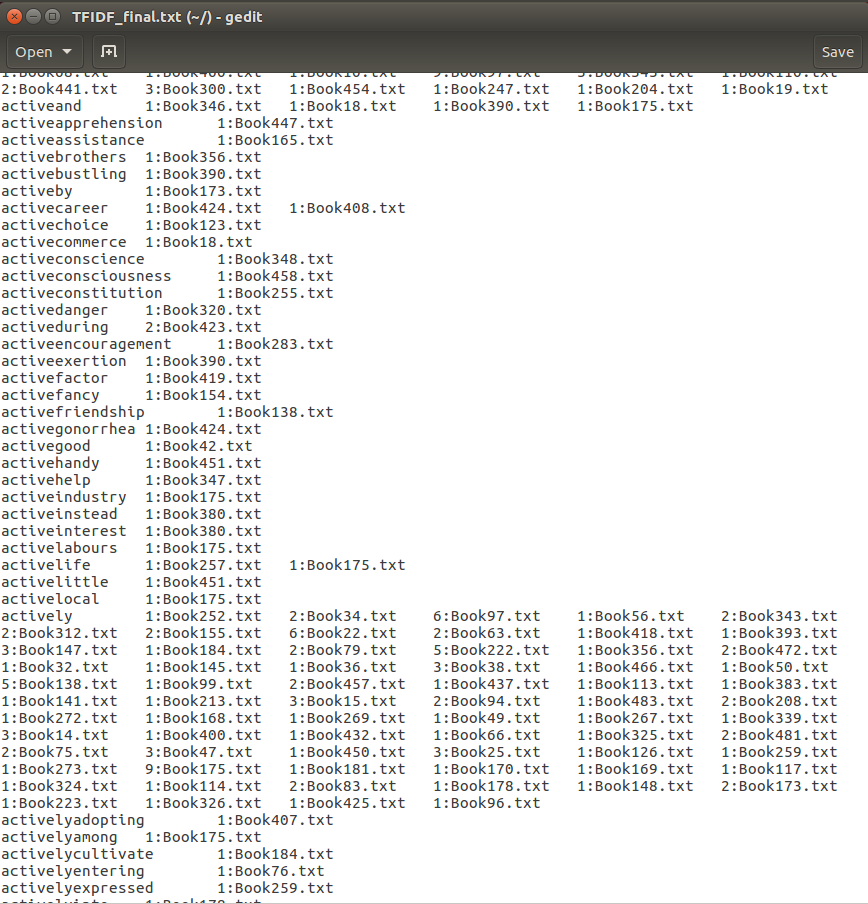




We run this java program using MapReduce and output the result in a txt file. And this is the file we will implement in our search engine.







# Web interface and database

The following part talk about how we use the help of Django and Python to complete an interface that allow the user to search our database.

We use Django and Python to help us create the MTV model.

Here is the code for the views part.

文本

描述已自动生成

We define two functions to allow the user to search our database which is the search function that allow the user to input the searching key word and the handle function that read the searching key word, search in the database and return the result to the user.

After the user press the search button, the handle function check whether the searching key word of the user appears in our database.

图形用户界面, 应用程序

描述已自动生成

If the searching key word exists in our database, the website turns to the result page and posts the relating appearance of this word.  
图片包含 矩形

描述已自动生成

图片包含 信件

描述已自动生成

To avoid the appearance of the useless information, we just show the first ten books that contains the searching key word to the user.

Since our text file doesn’t have list separator, we use Python to read and preprocess the dataset so that we can divide the consecutive string into separate tokens with the appear times and the appearance of the correlate books. By using the functions in Python, we can split the dataset by using the ‘\t’ sign, here we also add a comma behind each book name in order to make it more convenience for us to divide the string to show the dataset to the user in a more obviously way.

文本

描述已自动生成

Then we can use Python to import the data from the text file to the database.

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

Here we connect to the localhost Mysql database and we already create a table and give three attributes to it (nid, the searching key word, the correlate books which contain the searching key word) before we import the dataset.

After importing all the data into our database we can simply use the Mysql workbench to do a simple check. The dataset of our project is called website7 and the table of our dataset called data includes two attributes, the words in the book and their correlate appearance and appear times in the book. We store all the appearance of a word in one attribute and since in the previous step we leave a comma at the end of each book name and in the views part of our Django program we will use this comma to divide the string into several parts while showing the result to the user.

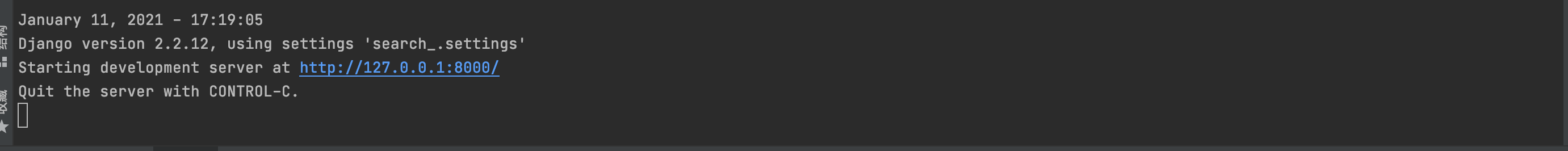
图形用户界面, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

Then we can use the terminal of the Pycharm to connect to our localhost and get to the interface that allows the user to search the appearance of a word in our database like the previous sections.



图形用户界面, 应用程序

描述已自动生成

Now we have successfully implemented the Text Search Engine!