CS 242 Project Report Part A

Group Number: 20

Group Members: Yifan Yu, Tianyang Li,

Qian Xiang, Xinle Chen, Jiang Zhu

Date: Feb 14, 2023

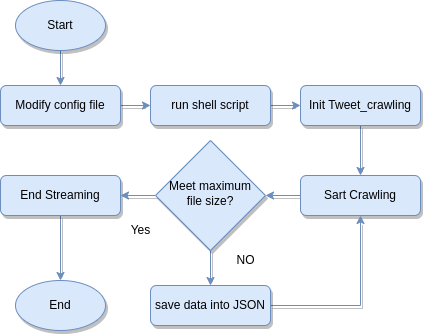
### 

**1. Collaboration Details: Description of the contribution of each team member**

* Yifan Yu collected tweets using Tweepy, wrote parts about the crawling system and instructions for deploying the crawler in the report.
* Xinle Chen initialized the Jupyter environment, wrote the initial and standard indexing program, and implemented the MultiFieldQueryParser hashtag solution.
* Qian Xiang, Tianyang Li, and Jiang Zhu together tested and implemented additional indexing features, collected test results and wrote the rest of the report.

**2. Overview of the crawling system**

**a. Architecture**



The architecture of this crawling system is very straightforward.

By using the Tweepy library, the code is able to crawl tweets related to our job-related topics and save tweets into a JSON file. The main class 'Tweet\_crawling' inherits from the 'tweepy.StreamingClient' class, which provides the functionalities for streaming tweets. The Class we added several attributes like 'filename', 'chunk\_size', 'tweets', 'file\_size', and 'num\_workers' that are used to configure the behavior of the stream. In ‘Tweet\_carwling’ class, we also implemented the 'on\_tweet' method, which is called whenever a new tweet is received. In this method, the tweet text is processed and its relevant information is extracted and added to the 'tweets' list. For future usage, we decided to include “Id”, “create time”, author id, the text of the tweet, Geo data, and the hashtags we extract from the text. Then, the '\_save\_chunk' method is used to save a chunk of tweets to the specified JSON file, ensuring that the file size does not exceed the specified limit.

**b. The Crawling Strategy**

There are several strategies we have been considering.

* We choose to use the Tweepy library's streaming API to crawl the data. Since we are looking for new job-related tweets, the best way to gain information about job info is to use the streaming API and add job-related rules, so that the instance of the crawler will filter the tweets in real-time by using the rules we just set up. Before the programming part, we did some research and found how the job-related hashtag is on Twitter and used the most relevant words to be the rules.
* To make indexing easier, we keep the time of that tweet, author’s id, hashtags, and tweet’s id and save them into a JSON file.
* Before saving into the JSON file, we need to make sure the text of the tweet is encoded under utf-8.
* To maximize the usage of the CPU, we decided to use the chunk method and ThreadPoolExecutor to maximize the workload, so that we have multithread features in the crawling and can let the user decide how many workers are working in the config file.
* To avoid the wrong JSON format, we first check if the JSON file already exists or not, if not, we will create a new one, else will first load the data and continue to dump the new tweet data, If meet the maximum file size, we will disconnect the streaming and end the program.

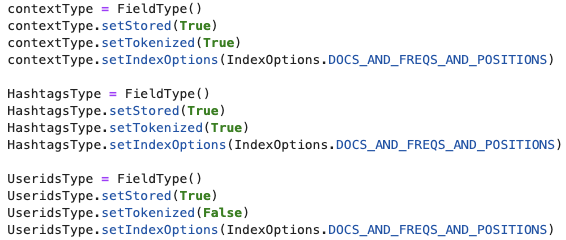
**3. Overview of the Lucene indexing strategy, including (but not limited to).**

**a. Fields in the PyLucene index, with justification (e.g., indexing hash tags separately due to their special meaning in Twitter).**

Besides the context of the tweets, when considering the indexing strategy of tweets, hashtags are an important consideration because they provide valuable insights into the topics, themes, and sentiments that are being discussed on Twitter. Hashtags can be used to identify trending topics, track the popularity of specific events, or measure the sentiment surrounding a particular issue or brand. Therefore, our major indexing concentration is to give hashtags a boosted weight other than the context of the tweets.

Other than hashtags, username is also really important when indexing tweets. For instance, say in real-life, if we were searching for a specific person or a firm on Twitter, we would definitely want the tweets that were sent from that particular person/firm to show on the top of the query along with those tweets containing their names. Thus, we also implemented a username component which gives username a boosted weight as well.

Below is a snippet of the codes that set up hashtags and username as the fieldtype. Note that we are currently using user-id instead of username because we did not collect username during the crawling process and this problem will be discussed later. We set "Tokenized" of user-id to be “True” because we do not want to separate the username into individual terms where it should be treated as one single string during indexing.

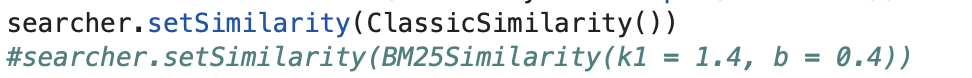


**b. Text analyzer choices, with justification (e.g., removing stop words from web documents; using separate analyzers for hashtags and keywords).**

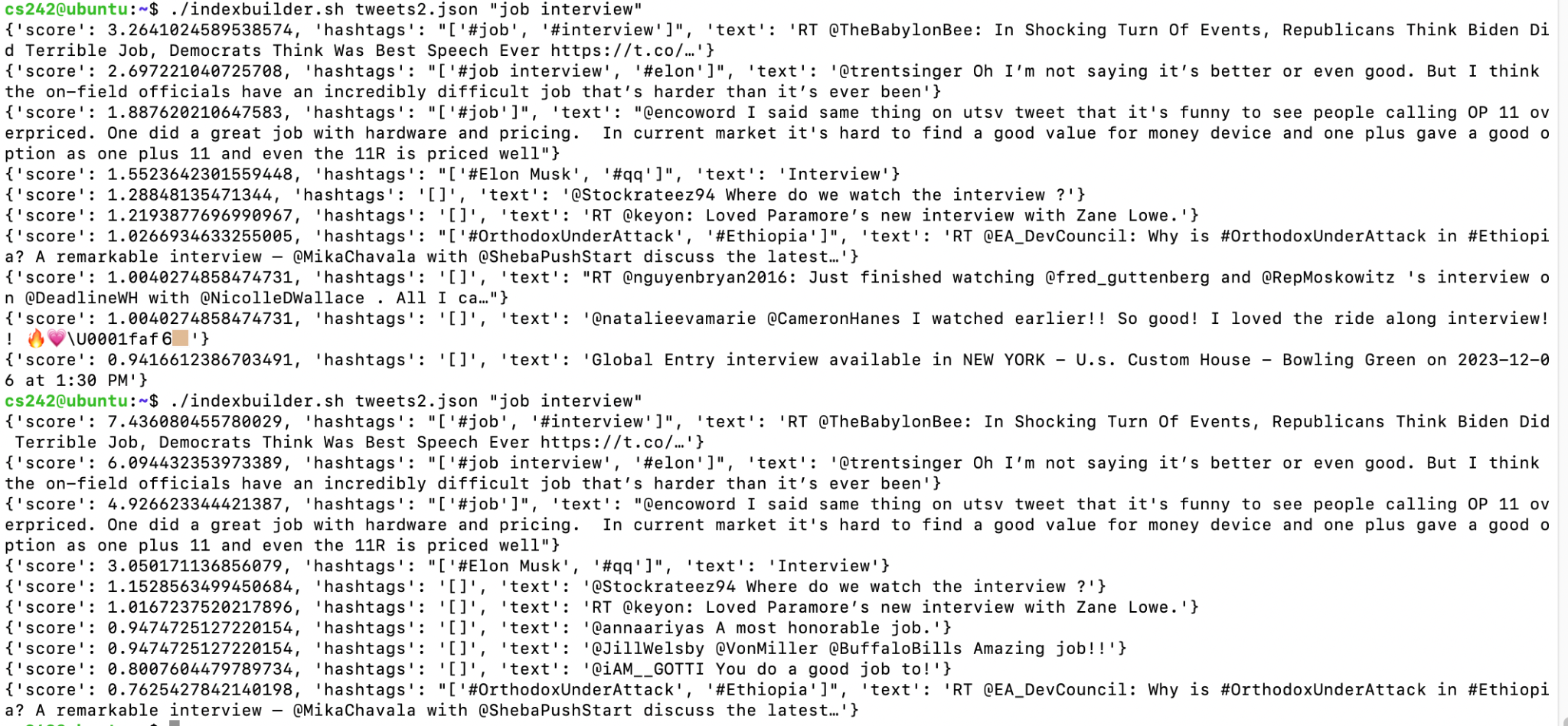
Below is a snippet of the analyzer function. As you can see, we give hashtags a boost of 2 and username a boost of 2.5 because we think during a query, when the hashtag of a tweet matches the query, that tweet has a greater chance of containing the relevant information that the user wants. Moreover, as we explained earlier, username is even more important than hashtags in a lot of scenarios where the query is a specific name other than a random word. Therefore, we set the username to have a boost of 2.5.

#### 

Below is a snippet of the different analyzer model we tried besides the standard model.



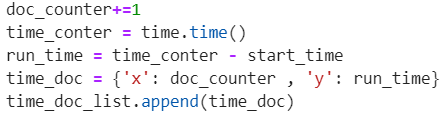
Tweets tend to be short and concise, which means it’s a good choice to use scoring models like VSM(Vector Space Model) and BM25(Best Matching 25) for indexing or retrieving tweets. We tested VSM as implemented by ClassicSimilarity() and BM25 that is implemented by BM25Similarity() separately. The results are shown below.



The scores in the upper part are scored by BM25 and the scores in the lower part are scored by VSM. As we can see, the top ten texts are similar, and it’s hard to say which model is better. We think one of the reasons is that the data we crawled from Twitter are all about “job” or “interview”, and as a result the collection of tweets contains mostly short documents with a similar distribution of terms. Though VSM is a simple model that doesn’t have any parameters to tune, the tasks(tweets) we want to retrieve are not very complex. And it can capture the order of words in tweets or queries which means it can result in better ranking of tweets that contain relevant terms in the right order. So we chose VSM for the subsequent experiments.

**c. Report the run time of the Lucene index creation process. E.g., a graph with run time on the y axis and number of documents on the x axis.**

During the index process, we use the time.time() function to keep track of the time that the current program is taking and save the number of files processed at that moment. Below is our code of collecting the data.



Finally, we output all of them to a csv file, and use matplotlib to draw a graph with run time on the y axis and number of documents on the x axis.



Plots of Runtime Analysis:

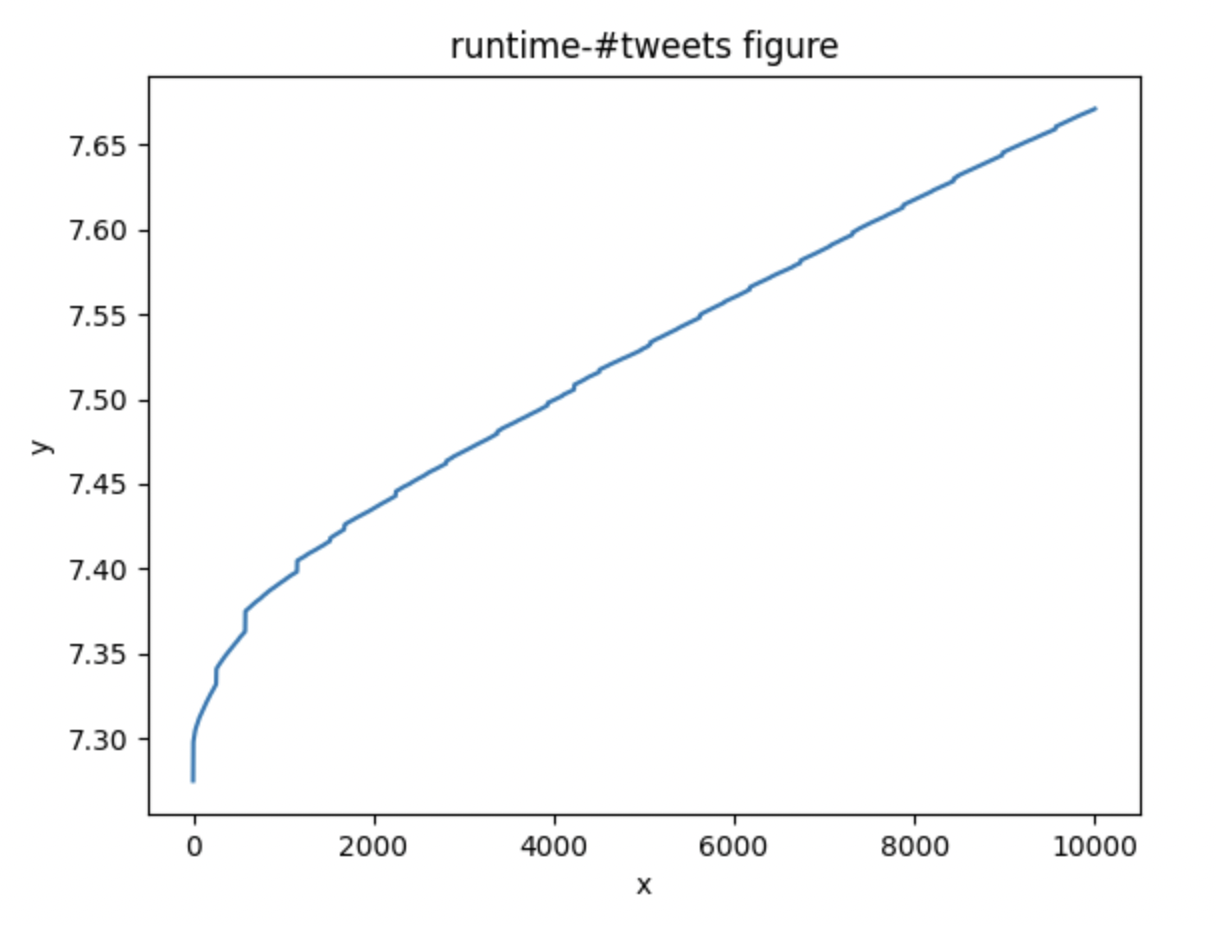
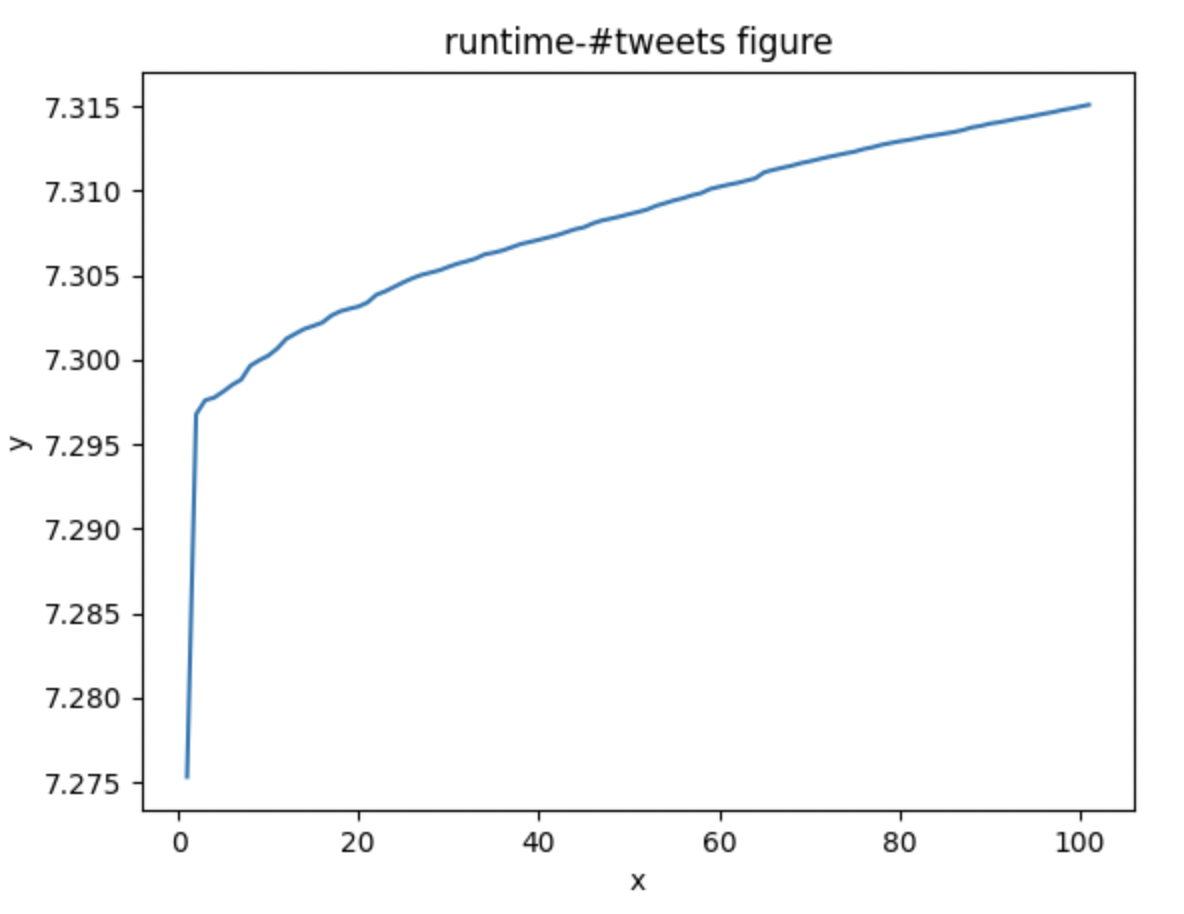


Fig 1. Indexing 100 tweets from a 500mb file Fig 2. Indexing 10000 tweets from a 500mb file

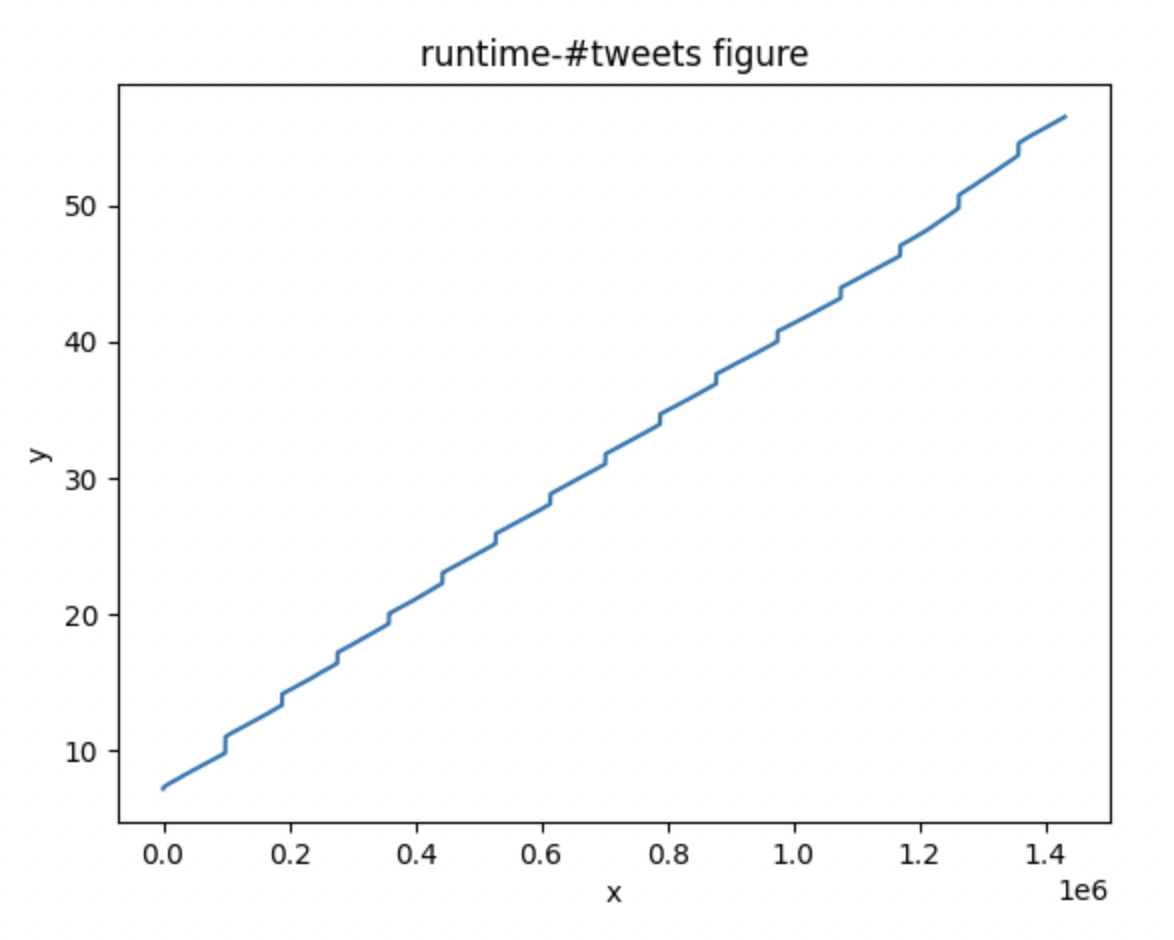
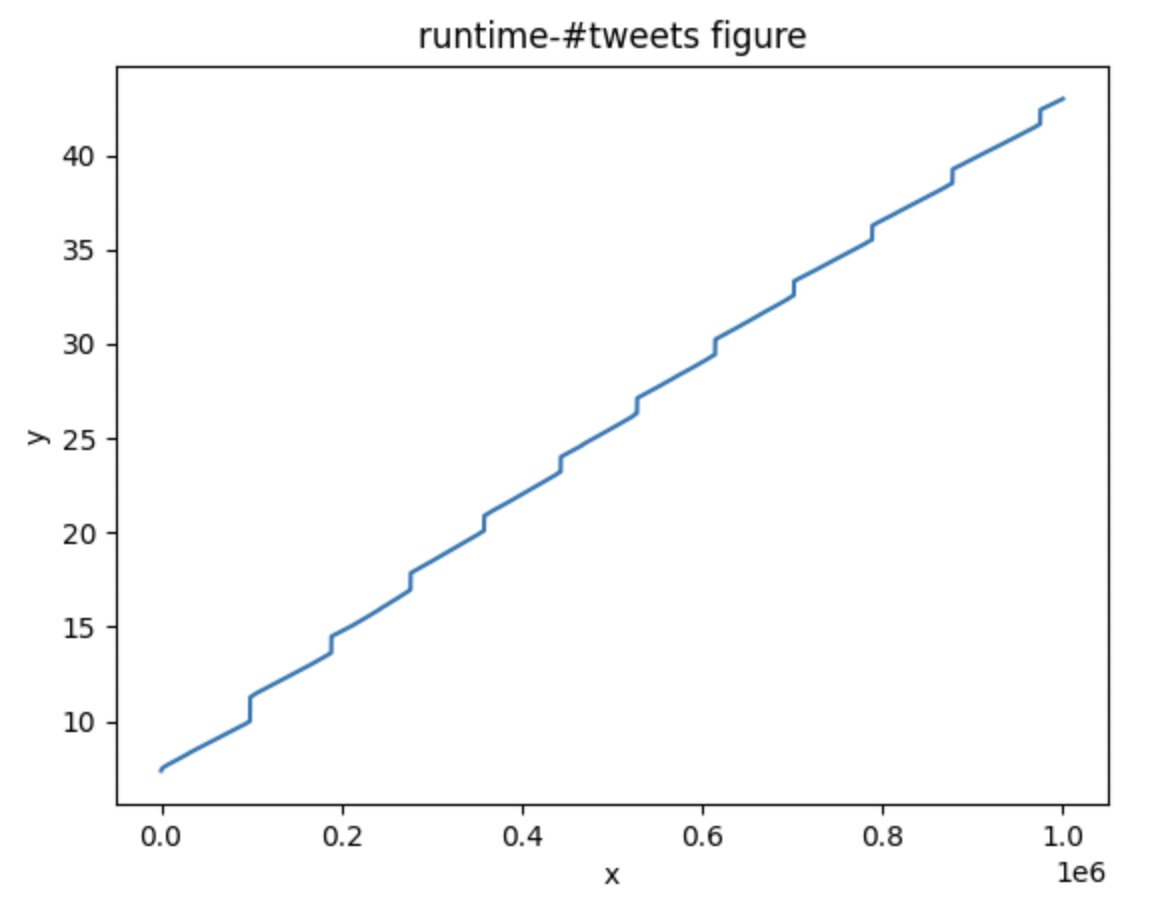


Fig 3. Indexing 1 million tweets from 500mb Fig 4. Indexing all 1.43 million tweets from 500mb

These four figures are the graphs plotted to analyze the runtime of the Lucene index creation process. It is clear that in every graph, before the first index has been created, there is a time gap of about 7.3 seconds after the program starts. This is due to the fact that our program takes about that amount of time to read the entire 500mb of data, which is pretty enormous.

Moreover, it is clear that the rest of figure 1 is pretty smooth and rather linear compared to the other three graphs. On the other hand, we can see that there are stair-shaped patterns in the other three graphs. In figures 3 and 4, for about every 90000 indexing, there is about 1 second time gap in between. However, in figure 2, while we could still notice a slight and subtle stair-shaped pattern in the graph, we can tell that the gap is between about 500 indexing with a 0.01 second gap, which is much smaller than the gap in figure 3 and 4.

For 100 indexes, excluding the initialization time, our program took about 0.017 seconds. When it comes to 10000 indexes, the program took about 0.36 seconds. For 1 million indexes, it took about 35.2 seconds. For the entire 1.43 million indexes, it took about 50 seconds.

| # of index | Time(s) | Indexes per second |
| --- | --- | --- |
| 100 | 0.017 | 5882 |
| 10000 | 0.36 | 27778 |
| 1 million | 35.2 | 28409 |
| ~1.43 million | 50 | 28618 |

From the table above where the indexes per second for each trial is calculated, we can see that when indexes exceed 10000, the efficiency is rather high and stable. When only 100 indexes are processed, our program is rather slow. This might be due to the fact that since 100 indexes only took 0.017 seconds, there might be errors during reading. Moreover, there could actually be a small time gap in the 100-indexes trial just like the stair-shaped pattern in other trials but it is just too small to be noticed. And that caused the indexes per second to be calculated much lower than others.

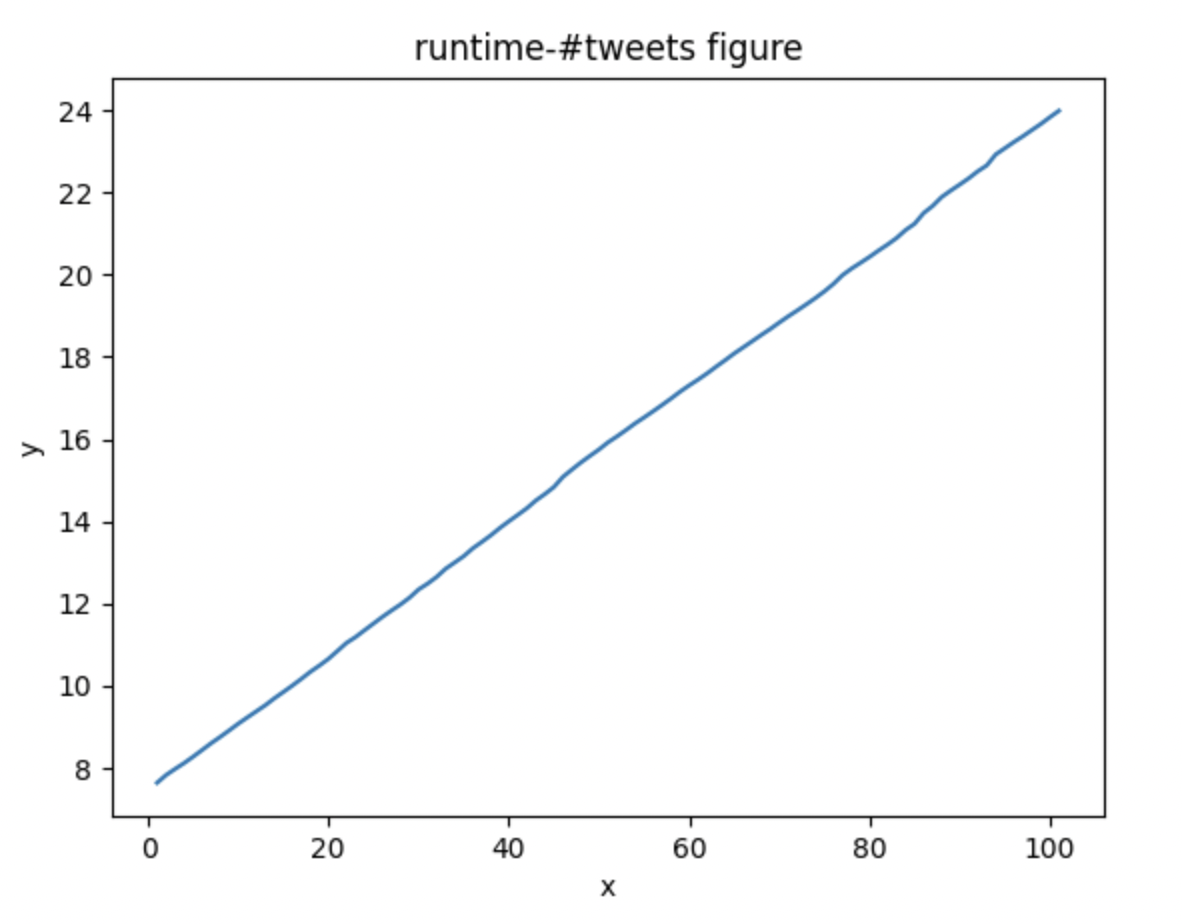


Fig 5. Indexing 100 tweets with usernames from a 500mb file

Figure 5 is the graph plotted showing the runtime of indexing 100 tweets with the username feature turned on. As always, there is also about 7 more seconds needed to open the data file. Excluding that, this trial took about 16 seconds to finish indexing for 100 tweets, which takes much longer than 0.017 seconds when indexing without username. It is almost 1000 times longer. There are lots of reasons leading to this huge difference. One could be that when we use username for indexing, we actually use the user-id to retrieve username using Twitter’s API and that must take a good amount of time.

4. **Limitations (if any) of system**

We got the error message like "tweepy.errors.TooManyRequests: 429 Too Many Requests" when we tried to convert all user-ids to usernames. The message means that our application has exceeded the rate limit set by the Twitter API, and the rate limit is the number of requests that an application is allowed to make to the API within a certain time period. We have over 1.4 million tweets which means we need to make over 1.4 million requests, so it seems impossible to convert every user-id to username when we are doing indexing. Thus, when we were testing the field of user-id, we chose to use a sample json that has only about 100 tweets.

We failed to solve the synonym problem. We implemented a class called "MyAnalyzer" and it has some functions such as "lowercase filter" and "stop word removal" just like StandardAnalyzer, but SynonymGraphFilter couldn’t be implemented because of some compile errors we failed to fix. We started with SynonymFilter for a long time to fix compile errors and later we found it deprecated, so we turned to SynonymGraphFilter and were bogged down by the compile errors too. Therefore, it didn’t work out until the end.

We have two versions of the solution to address the hashtag. Both of them are working properly while both of them have some limitations. The first version is an easy way which is just tokenize all the hashtags and then use each token as the key to build the index, which is similar to the strategy of addressing the text. The limitation of this way is that if “A B C D” and “D C B A” have different meanings, this way will eliminate the difference as it just tokenizes the hashtag. The hashtag is much more precise than the text so it needs some more precise solution.   
 The second solution is that we keep the original hashtag, which means we do not tokenize it to keep its precise original meaning. And then we use different text Analysers for hashtag and text. We use KeywordAnalyzer for the hashtag field when we process the query and StandardAnalyzer for the text field. As the KeywordAnalyzer will keep the original query precisely and combining with the strategy that we do not tokenize the hashtag when we build the index, we will get precise results when the user query matches the hashtag. Then we use PerFieldAnalyzerWrapper to merge these two Analyzers to implement the functionality that when addressing the hashtag we can get the precise result and when addressing the text we can follow the basic tokenized algorithm.



But the limitation of the second version is that we just address the precise case which means if a hashtag is “A B C” and the user just searches the query “A” then the user will not get the result of which the hashtag is “A B C”. We do not have time to combine the advantages of both the two versions to get a perfect hashtag solution. In the future if we have more time we will try it.

**5. Obstacles and solutions**

When collecting the runtime and number of documents, we need to save it to a csv file and then plot it. However, the operation of frequently saving data to files will affect the running time of the program and affect the statistics of the running time. Therefore, we first save the statistical data in a temporary list, and finally output it to the csv file, so as to reduce the impact of collecting data on the running time of the program.

At first we had no idea how to assign different weights to each field when retrieving tweets and how to determine the relative importance of each field. Half the team was assigned to solve the problem. After looking up a lot of documents, we finally have got two solutions and both of them work. One is to use HashMap and MultiFieldQueryParser while the other is to use BooleanQuery.Builder() to add BoostQuery. Both methods have the same effect.

**6. Instruction on how to deploy the crawler.**

1. Unzip the zip file to the location you prefer.
2. Before running the shell script, make sure you have the right requirement, then please modify the config.json file with your bearer\_token and other parameters you wish to change.
3. Run the shell script. It will generate a specific size of JSON file (in Mb).

**7. Instruction on how to build the Lucene index.**

1. Unzip the file to local.
2. Make sure the required software and extensions are installed.
3. Make sure the tweets data file is local and its format is in JSON.
4. Run the following code at the terminal where the first argument is the data file name and the second argument is the keyword that the user wants to search.

