Twitter Keyword Search

Shiza Ali and Mohammad Hammas Saeed

Abstract

In this project we build a Keyword search engine that accepts a text file containing a number of tweets as input (each line is one tweet), and provides a simple user interface for querying the list of tweets against keywords. The system replies to keyword queries with the top 10 tweets that are most relevant to user query keyword(s).

1. Introduction

A search engine is a software system designed to perform web searches. It identifies items in a database. The search results depend on a query (keywords or characters) inputted by a user. Search engines have three primary functions: 1) Crawl: Search the Internet for content, looking over the code/content for each URL they find. 2) Index: Store and organize the content found during the crawling process. 3) Rank: Provide the pieces of content that will best answer a searcher's query, which means that results are ordered by most relevant to least relevant [1].
In this project we build a keyword search engine for a subset of Twitter's data.
Summary of our work:

a) *Crawl Tweets:* We read tweets from a text file.
b) *Create a database:* Our database is in the form of an inverted index. We create this to speed up the searching and ranking process.
c) *Replying to Keyword Queries:* Our system then outputs the top 10 tweets relevant to the keywords inputted by the user.
d) *Design a UI:* The GUI takes in queries by the user and outputs the relevant tweets

The report is further organized as follows: Section 2 discusses the details of the data structures and algorithms used in the development of the database and in Section 3 we discuss searching and ranking with reference to our project. In Section 4 we give a brief performance analysis of the data structures and algorithms used, in Section 5 we discuss our GUI.

2. Development of the Database

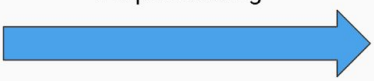
We use the following data structures and algorithms to construct the database

a) Inverted Index

An inverted index is a data structure popularly used in the creation of a search engine. It involves breaking down documents/data/text into a list of unique items called tokenization, and for each respective item, creating a reference to its source. For example:

ID	Tweet
tw1	I am Amazing!
tw2	Amazing food.
tw3	My cat ate food.

Preprocessing



Keywords	IDs
I	tw1
am	tw1
amazing	tw1, tw2
food	tw2, tw3
My	tw3
cat	tw3
ate	tw3

Figure 1: Creating an inverted index from a list of tweets

b) Hash map

An inverted index can be easily programmed using a hash map (Figure 3). Data structure that maps keys to values. The advantage of this searching method is its efficiency to handle vast amounts of data items in a given collection.

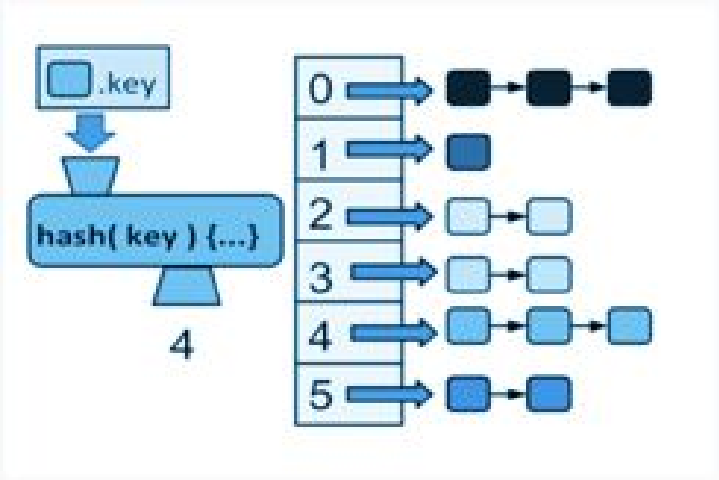


Figure 2: An image of a hash map.

We program our index table in C++ language and therefore we use STL library to help us in this task. This library provides two kinds of hash maps; **map** and **unordered_map**. The table below summarizes the differences between the two data structures:

	map	**unordered_map**
Ordering	Ascending Order	No order
Implementation	Self - balancing BST	Hash Table
Searching	O(log(n))	O(1) Average, O(n) Worst Case
Insertion	O(log(n)) + rebalance	O(1) Average, O(n) Worst Case
Deletion	O(log(n)) + rebalance	O(1) Average, O(n) Worst Case

Table 1: Comparison between map and unordered_map

Unordered_map internally uses a hash table. hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. In the hashing large keys are converted into small keys by using a hash function. The values are stored in a data structure called hash table [2].
Map internally uses red-black tree. Red-Black Tree is a self-balancing Binary Search with three rules. 1) each node has a color either red or black, 2) Root of the tree is always black, 3) there are two adjacent red nodes which means the red node cannot have a red parent or child [3].

c) Putting it all together: Development of the Database

The figure below explains how we create the database. After collecting the data, for every tweet we create a vector of unique keywords. Now for each of these keywords we check whether this keyword is in the hash map, if it is not there we create its place and store the corresponding tweet ID with it. If it's already in the hash map we simply store tweet ID in front of it.

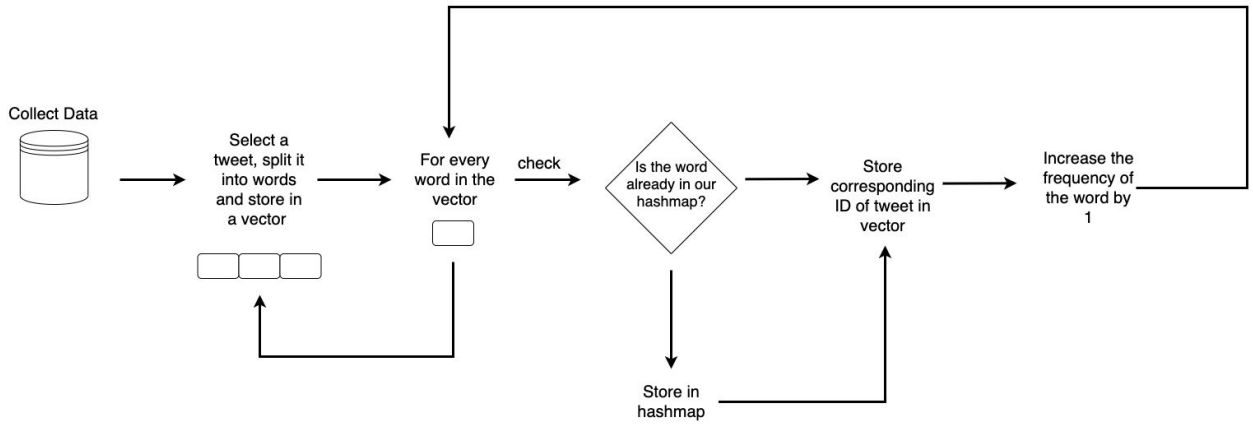


Figure 3: Flow chart identifying the steps involved in the creation of the data base

3. Searching and Ranking according to the Queries

For searching and ranking the queries we use priority queues. In a priority queue Every item has a priority associated with it. The high priority item is dequeued before an element with low priority, however, same priority elements are served according to order in the queue.
The table below shows the conversion of figure 2:

Keywords	Vector <tweet, occurrence>
I	<tw1, 1>
am	<tw1, 1>
amazing	<(tw1, 1),(tw2, 1)>

food	<(tw2, 1),(tw3, 1)>
My	<tw3, 1>
cat	<tw3, 1>
ate	<tw3, 1>

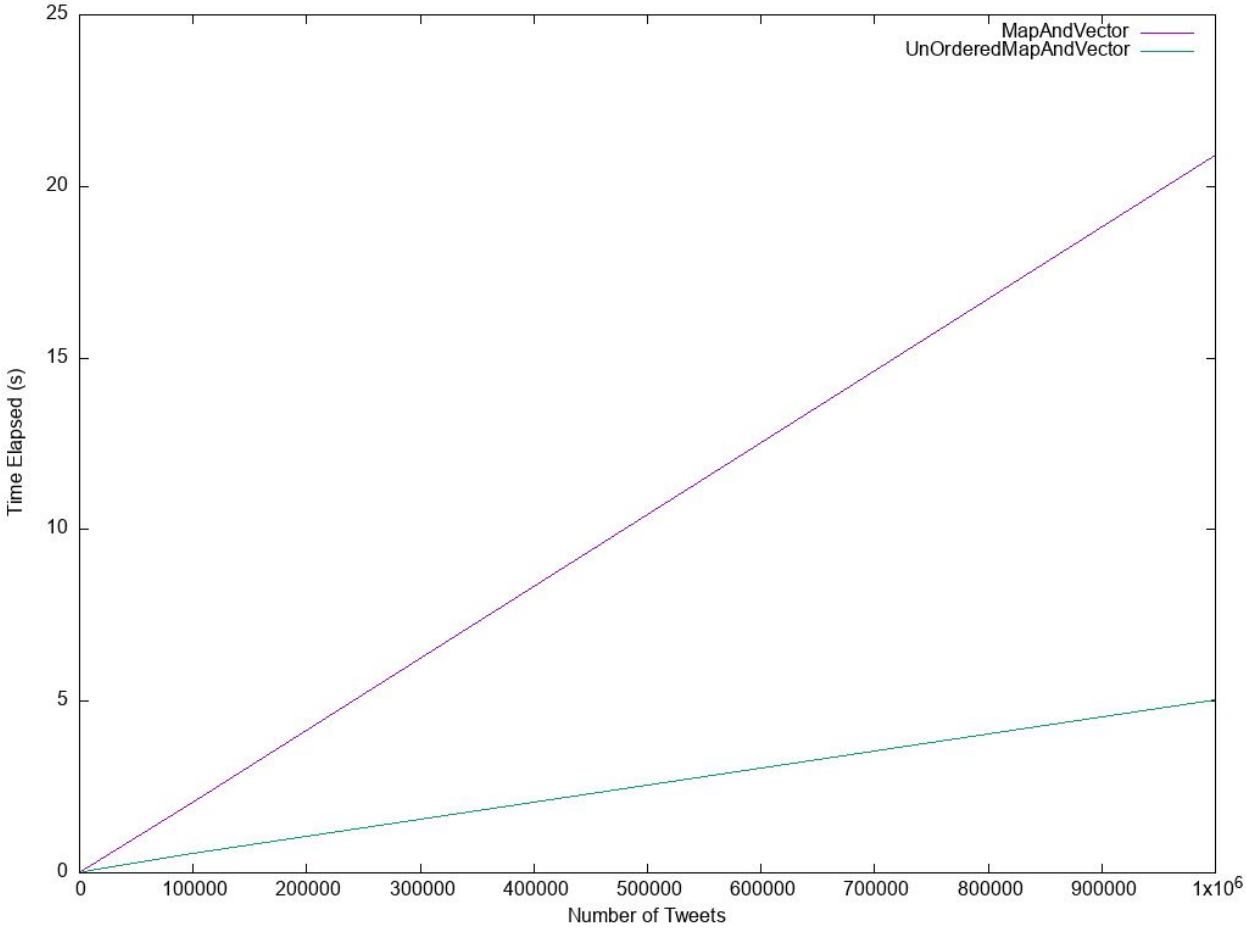
Table 2: Keywords and corresponding vectors

We maintain a min heap to implement priority queue.
In this process, once we find a keyword, we loop through the inner data structure which contains the number of the sentence and how much time it occurred in this sentence. Each time we know those data, we add the time it occurred to the weights of this sentence. After finding, the vector will go through a heap sort based on the weights. After that, the 10 sentences which contain largest weights is the result.

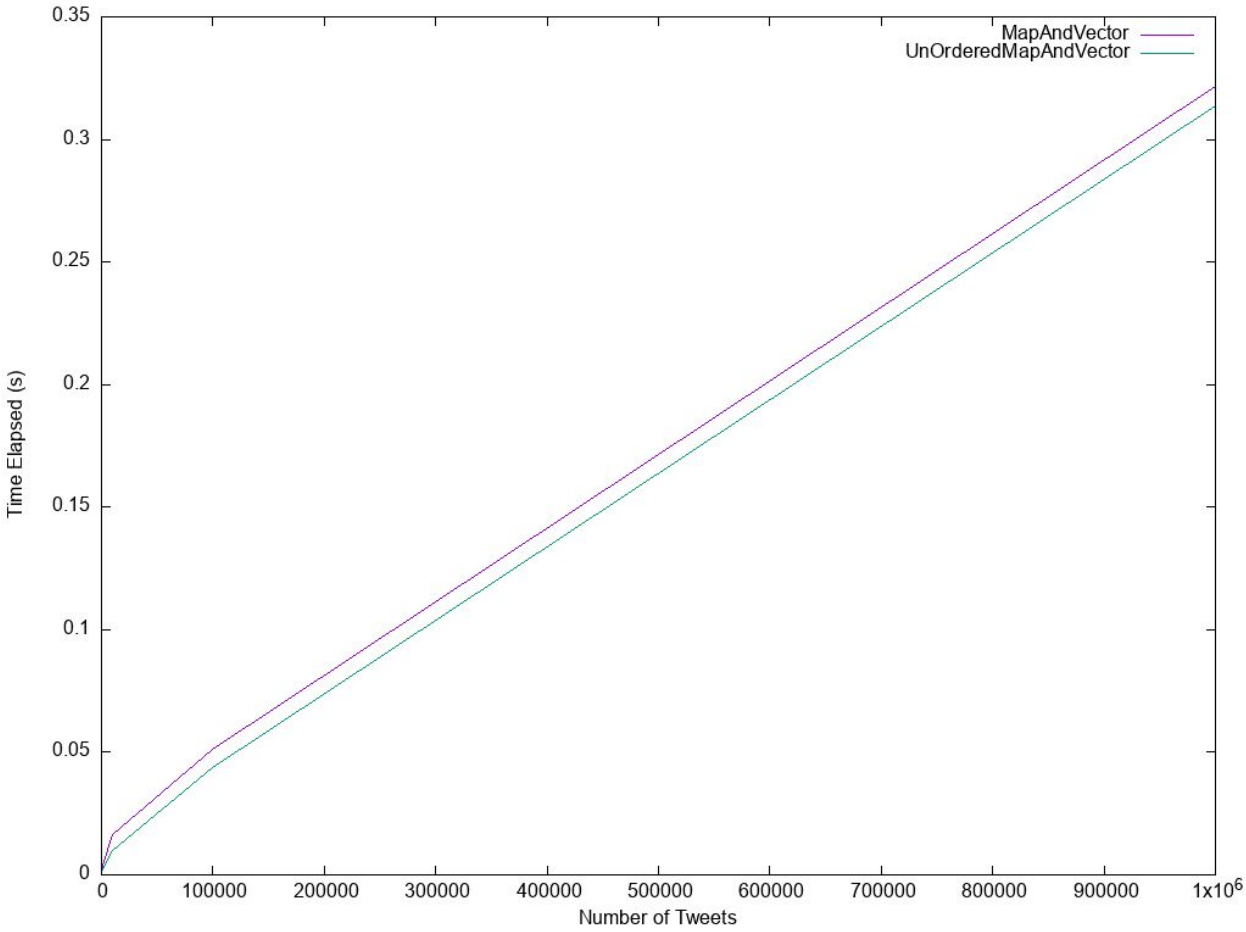
4. Performance Analysis of the Data structures

a) Comparison Analysis between different types of maps

The figure below shows the time taken to build the index inverted index when using the **unordered_map** and **map.** As expected, the unordered_map allowed the index to be built alot faster, especially as the number of tweets increase.



The figure below shows the time taken to search the index inverted index when using the **unordered_map** and **map.** Here, the difference is less but still, the unordered_map shows superiority over ordinary map.



The table 3 below shows the actual time taken to setup index, search and sort results for both kinds of maps we used.

Data structure	Set Up Index	Search	Sort Results
map + vector	~20s	~0.6s	~0.3s
unordered_map + vector	~5s	~0.3s	~0.3s

Table 3: Time Comparisons between map and unordered_map. Analysis performed on a huge dataset ~ 1 million tweets

By analyzing the figures and table above, we can have the following conclusions:

1. When building a database, unordered_map has a significant advantage over map. This is because map has O(logN) complexity while unordered_map has O(1) average complexity.

2. When searching, unordered_map's advantage over map is not clear because of higher possibility of collisions. Although unordered_map has an average complexity of O(1), its worst case complexity can be O(n) which makes it slower than map.

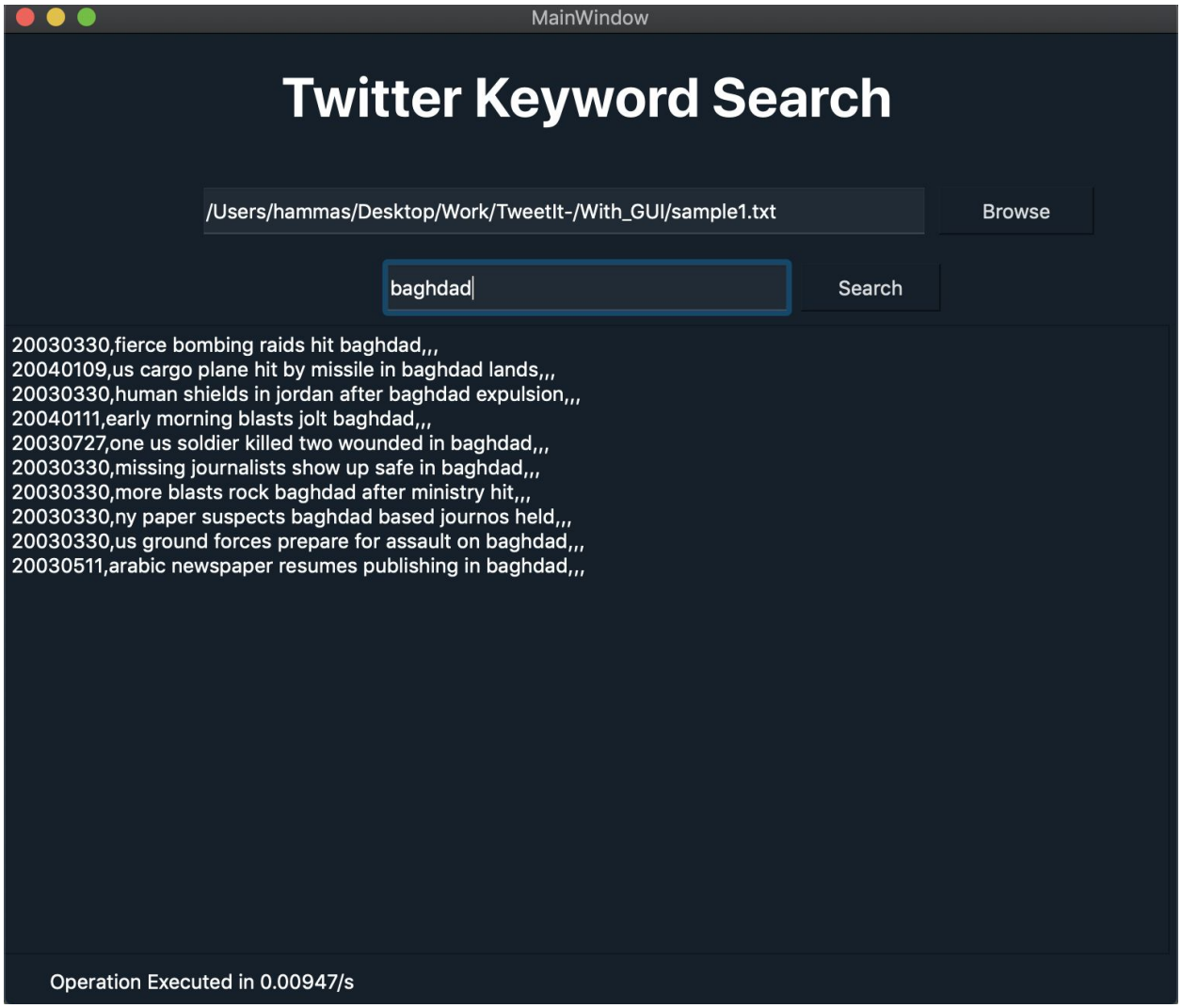
b) Analysis of Priority Queue

We implement the priority queue as a heap. Overall Complexity of insertion in a heap is O(log N), and searching top is O(1). We use this data structure because it is fairly fast and efficient.

As observed in Table 3 above, the sorting of results is very fast even when the number of tweets is large.

5. GUI

We used Qt software to create our GUI for this project. The image below shows the view of our GUI. The user can “Browse” and provide a file to read tweets from. Then user can enter a query to “Search” the inverted index. Our GUI also shows the time elapsed for each request at the bottom.



1. We need to define our GUI graph which includes the interface rectangular layout. We add the function “Text Edit”, “Text Line”, “Time Edit” and “Button” in our GUI.
2. Second, we need to define these functions one by one which well build the connection slot with their next step. For example, we defined the “Button Clicked” slot that connected to reading file. When you clicked the button, it will execute reading file function. The reading function will pick the file up from particular address.

3. Then, we also defined our “Text Edit” with the same function. This function will allow our algorithm file could read the content in “Text Edit”. Then we build the “Time Connection” which means that we transfer our time shown from terminal into our GUI.
4. Because we defined the timing shown in our algorithm, we just need to rebuild this function and transfer the function from our terminal to our GUI show.
5. Final step, it is something similar to step 4. We also finished our function to transfer our terminal results shown into our GUI result show.
6. When you click the “Open File” button, the program will let you choose one file which you want to read it. Then the reading file will be read by our algorithm file and the GUI will show the reading time at the right corner. Then, you type your keyword and click the search button. The result will be shown as evident in the image above.

6. Conclusion

In order to make setting up the database and searching keywords faster, for our Twitter Keyword Search Engine we designed different data structures based on inverted index to accelerate the inserting and searching process. After doing complexity calculation and testing on the data, we found that using the unordered_map is better. Also, we have used a Heap to store the weight of each sentence together with it’s sentence number, which not only accelerates the process of sorting the results but also accelerates the process of accessing the sentence. We have also developed a GUI.

7. References

[1] https://www.merriam-webster.com/dictionary/search%20engine
[2]https://www.hackerearth.com/zh/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/
[3] https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/