Twitter Search Application – Team 28

# Abstract

The Twitter Search Application is service that allows to automatically submit searches to Twitter and retrieve matching tweets. It can be used to gather tweets for social science and other research, although this is not its main purpose. In this we try to introduce a simple method to check the coverage of its results for narrowly focused topics.

# Introduction

Twitter users write over five hundred million messages every day. Users discuss topics such as music, television, sports, politics, and technology through these so-called tweets. Our attempt to search for tweets form such a wide range of data based on Users, Tweets, or Hashtags, efficiently and implement a cache mechanism and compare the execution time with and without it.

# Dataset

For this Application the data from ‘Corona-2’ & ‘Corona-3’ was loaded into MySQL Server and Elasticsearch which were used as Relational and Non-Relational Databases. From these Json Data necessary fields were selected and loaded into the databases. Technologies used.

1. MySQL server to store Users Data.

From the User object following fields were selected ‘name’, ‘screen\_name’, ‘User\_id\_str’, ‘follower\_count’, ‘friend\_count', 'status\_count’, ‘location’ & ‘verified’ where ‘User\_id\_str’ is selected as primary key.

1. Elasticsearch to store Tweets data.

Elasticsearch was chosen due to o its capacity to handle large volumes of data with low latency. It excels in searching through tweets by offering robust full-text search capabilities and efficient indexing, which is essential for quickly retrieving relevant tweets based on specific keywords or hashtags. Its inverted index structure allows for fast searches across the text content of tweets.

# Data Modeling

* For Relational datastore

Here a schema was defined as ‘users\_data’. While scanning through the Json data, first check if the user\_id exist in MySQL server then skip it and move on to next line in Json data, else commit the user details into the ‘users\_data’ table.

* For Non-relational Datastore

Here two mapping have been defined to maximize search efficiency for different drill down searches. In the first mapping we just mapped user\_id, name & screen\_name. this was done to improve search efficiency when searching Tweets by user’s name/ screen\_name.

For the second mapping we mapped the entire tweet object. From this only necessary fields like ‘tweet id’, ‘text’, ‘user id’, ‘name’, ‘hashtag’, ‘likes’, ‘retweet count’, ‘created at’, ‘retweeted status’, were committed to Elasticsearch to save memory. Apart from these an additional field ‘Engagement score ’was added to each tweet/retweet, which used to rank the relevance order of the search results of tweets.

# Preprocessing of Data

The Corona dataset contains a lot of useless data which needs to be identified and filtered out before loading it into the databases. Along with that it also has some inconsistencies which need to be fixed to ensure accurate search results. One such was inconsistency was the absence of original tweets for given retweet, in such a scenario a dummy original tweet would be created for the original tweet’s user\_id. this further enhances search results for search by tweets by user query and provide an accurate relevance order.

For deciding the relevance order a relevance matrix was created that calculates an engagement\_score for each tweet as it is loaded into the database on the following formula,

engagement\_score =

Another metric Combined\_score is Defined to get relevance order of the tweet and is defined as,

Combined\_score =

Here is a default score assigned by Elasticsearch to every entry in the database and is calculated on Term Frequency-Inverse Document Frequency i.e. it calculates the importance of a term within a document relative to its importance in the entire collection of documents. Terms that appear frequently in a document but infrequently in the entire collection receive higher scores. The combined\_score is calculated when the query is executed, and search results are ordered accordingly.

# Loading approach

A diagram of a software flowchart

Description automatically generated

Read a new line from Json file check if tweet present in Elasticsearch if present go to next line in Json file else get the previously discussed fields from User and tweet Objects, if the extracted object is a tweet, then insert the tweet in Elasticsearch and check and insert user details into MySQL server. If it’s a retweet, then insert the retweet details and mark retweet as true check if user who did the retweet is in MySQL and insert user’s details if not present. Also make sure to see if the original tweet is inserted into Elasticsearch.

# Caching

Caching helps improve the performance of search queries by storing frequently accessed data in memory. When a user performs a search query that has been executed before, the results can be retrieved from the cache much faster than fetching them from the underlying data store, reducing latency and improving the overall user experience.

For this application a Least Recently Used (LRU) Cache is implemented using an ordered dictionary with capacity of 7. It is a type of cache management strategy where the least recently accessed items are removed from the cache when it reaches its maximum capacity. To ensure durability of cache we checkpoint it every 60 minutes. The Cache mechanism is always running as a background process by daemon threading.  
The cache class has 4 functions.

Get(key) – If there is a key in the cache-> Move the key to end of dictionary and return the value else return none.

Put(key) - If there is a key in the cache-> Move the key to end of dictionary and update the value or add the key value pair. If Len (Cache) > capacity: Remove the first item (Least Recently Used)

Save\_on\_disk () & load\_form\_disk (): - To save and load the contents of cache to local disk for durability.

Cache key Generation.

* Function-Specific Prefix:

Start with a descriptive prefix related to the function to categorize the data, ensuring clarity and organization in cache management.

* Parameter Integration:

Append the function’s input parameters to the prefix, converting them into a string format to create a unique identifier for each function call.

Example:

For a function retrieving user tweets, the cache key is formed as cache\_key = f"user\_tweets\_{user\_id}", where user\_id is dynamically inserted based on the function's input. search\_user\_tweets(user\_id): cache\_key = f"user\_tweets\_{user\_id}” search\_tweets(query\_string): cache\_key = f"search\_{query\_string}"

# Search Application

1. Searching a user based on string.

Querying the MySQL users\_data table for the input string using contains keywords to all possible matches for name & screen\_name. The results are ordered by MySQL built in ranking algorithm.

1. Search tweets by String or Hashtags in a certain timeframe.

Only considering original tweets and no retweets are ignored. It is possible to search on just the text field or the hashtag of the Tweet. An additional filter of time range can be applied. The resulting tweets are ordered by descending values of combined\_score.

1. Searching all the Users who retweeted a particular Tweet.

Given a tweet\_id we can find all the retweets and details of the retweeters as well as original tweeter. The details about retweeters are ordered by followers count and verified.

This done by first finding user\_id for corresponding tweet\_id and getting the users details from MySQL server, this process for all matches of tweet\_id where retweet is true.

1. Top 10 Tweets

First consider only tweets i.e. retweet = False. Then we sort the result based on the engagement\_score and limit output by 10.

1. Top 10 Users

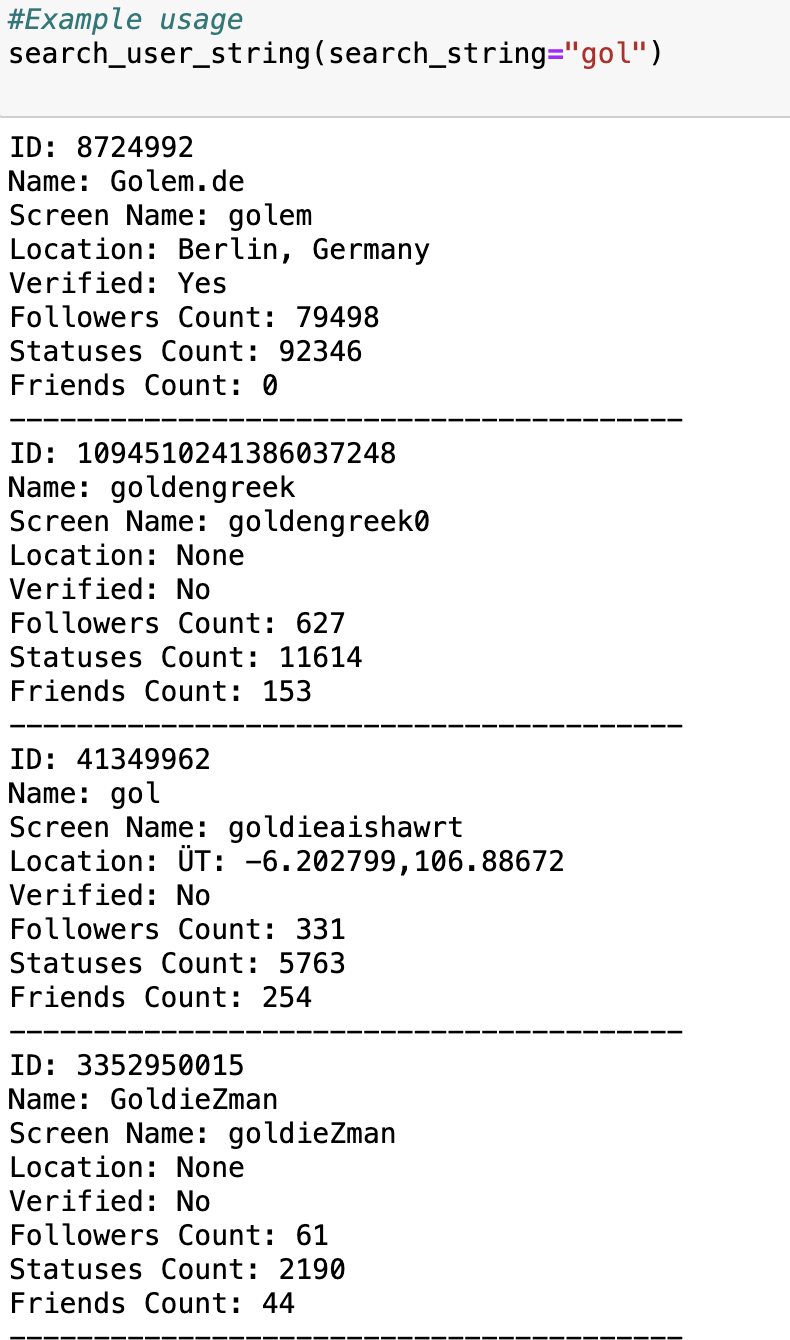
first take all the unique user\_id then for each user\_id find out all the original tweets aggregate all the engagement\_score2 for all the original tweets made by the user\_id now add followers count for user\_id to this agg\_engament\_score2.

Repeat the process for all user and sort based on above and get the top 10 user\_id.

For these top 10 user\_id get the User details form MySQL server.

# Results

Querying user details by passing string as input.



Total runtime of the program is 0.04676103591918945 seconds.

Querying tweets by passing string as input based on Tweet text Or Hashtags A screenshot of a computer

Description automatically generated

Total runtime of the program is 0.04271507263183594 seconds.

Querying users who retweeted a particular tweet

A screenshot of a computer

Description automatically generated

Total runtime of the program is 0.05240583419799805 seconds.

Top 10 Tweets

A screenshot of a computer

Description automatically generated

Total runtime of the program is 0.04349708557128906 seconds.

Top 10 Users

A screenshot of a computer screen

Description automatically generated

Total runtime of the program is 24.708147048950195 seconds.

Impact of Caching on execution time of queries

A table with text and numbers

Description automatically generated

# Conclusion

In this project, the impact of Cache is evident as it provides very data retrievals by 100 times on an average. Also having a good mapping strategy, reduces time taken to load data into database and preselecting the schema helps to save precious memory and storage space on both system and server.

# References

1. [Evaluating the comprehensiveness of Twitter Search API results](https://www.researchgate.net/publication/313442495_Evaluating_the_comprehensiveness_of_Twitter_Search_API_results_A_four_step_method)
2. [Twitter Discussion Application Programming Interface](https://ivypanda.com/essays/twitter-discussion-application-programming-interface/)
3. [An Exploratory Study of Twitter Messages about Software Applications](https://www.researchgate.net/publication/318511337_An_Exploratory_Study_of_Twitter_Messages_about_Software_Applications)
4. [Social Media Research](https://subjectguides.library.american.edu/c.php?g=1238130&p=9060337)
5. [Free and Low-Cost Twitter Research Software Tools for Social Science](https://journals.sagepub.com/doi/10.1177/0894439320904318)

# List what each team member did for the project

1. LayaKishore Reddy Desireddy (ld786) - Non Relational Datastore.
2. Bhavana Vemula (bv234) - Search Application.
3. Vikram Sriram Yasaswi Sagaram (vs865) - Caching.
4. Jayant Chaudhari (jc3303) - Relational Datastore, Caching.