TWEET SEARCH APPLICATION

USING THE TWITTER STREAMING API

By

Yaniv Bronshtein

A Research Project Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Statistics

Specialization in Data Science for MSDS

At

Rutgers, the State University of New Jersey

February 2022

Abstract

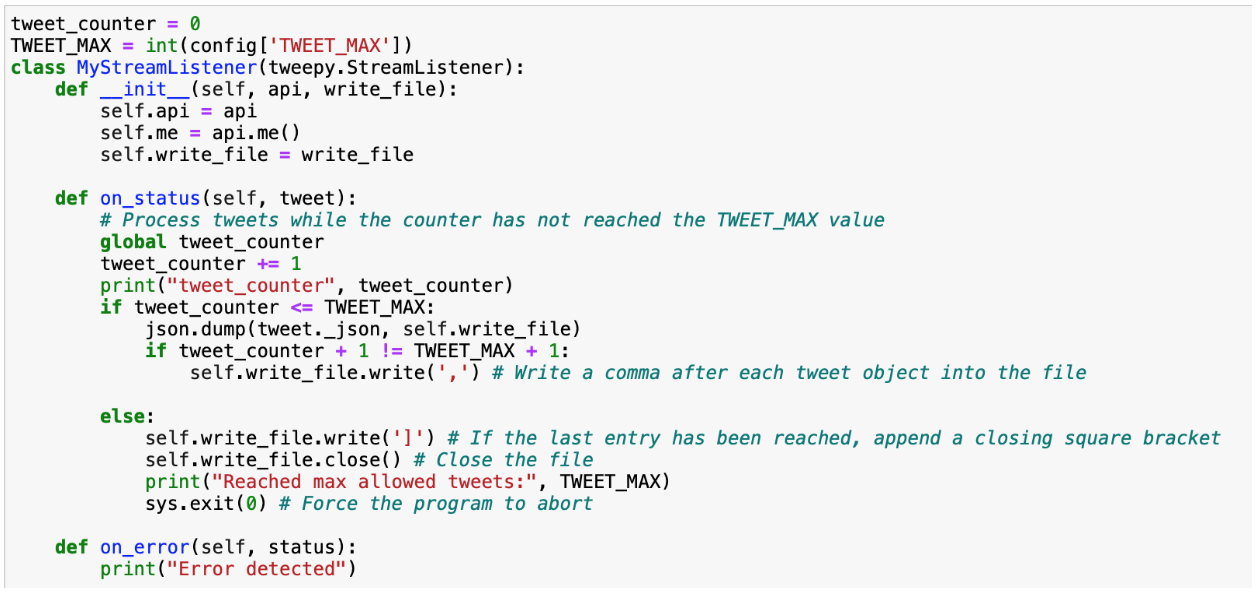
As the modern world becomes more data hungry, there is a growing demand for creating systems capable of harnessing that data. These systems must allow entities to efficiently parse data at will yet also maintain data integrity and security. This project serves as a proof of concept as it aims to create a product that processes large amounts of social media data from the popular Twitter platform, stores the data securely, and allows the user to retrieve information through searching by word, user, hashtag, and time range.

The product development was split into three separate phases: Data Collection, Data Storage, and the front-facing Data Search Application. The Twitter Streaming API was harnessed to pull 30,000 tweets for the following hashtags in April 25, 2021 over the course of 4 hours (12pm-4pm EST): #sundayvibes, #UFCVegas23, and #WrestleMania. The tweets were selected based on their trending count for that day. Special care was taken to choose tweets that would be relevant for at least a couple of weeks rather than single day events. Tweet Data was separated into two categories: user information and tweet metadata. The user information was stored in a local MySQL database while the tweet information was stored in a remote MongoDB cluster. The search application interfaced with a local instance of Redis DB that was used to cache previously searched data.

Table of Contents

I.Tweet Collection

In order to start the project, a connection needed to be established to the Twitter Streaming API provided by the social media platform to Developers. To handle the connection to the API, the user-friendly Python library Tweepy was used. Tweets were processed until the max count was reached, after which the program aborted itself.



```python
tweet_counter = 0
TWEET_MAX = int(config['TWEET_MAX'])
class MyStreamListener(tweepy.StreamListener):
    def __init__(self, api, write_file):
        self.api = api
        self.me = api.me()
        self.write_file = write_file

    def on_status(self, tweet):
        # Process tweets while the counter has not reached the TWEET_MAX value
        global tweet_counter
        tweet_counter += 1
        print("tweet_counter", tweet_counter)
        if tweet_counter <= TWEET_MAX:
            json.dump(tweet._json, self.write_file)
            if tweet_counter + 1 != TWEET_MAX + 1:
                self.write_file.write(',') # Write a comma after each tweet object into the file

        else:
            self.write_file.write(']') # If the last entry has been reached, append a closing square bracket
            self.write_file.close() # Close the file
            print("Reached max allowed tweets:", TWEET_MAX)
            sys.exit(0) # Force the program to abort

    def on_error(self, status):
        print("Error detected")
```

Figure 1. Tweepy tweet collection application

As per **Figure 1,** the response was in a JSON format, so it was beneficial to dump the results into a json file for constant time access as opposed to sequential access for text files.

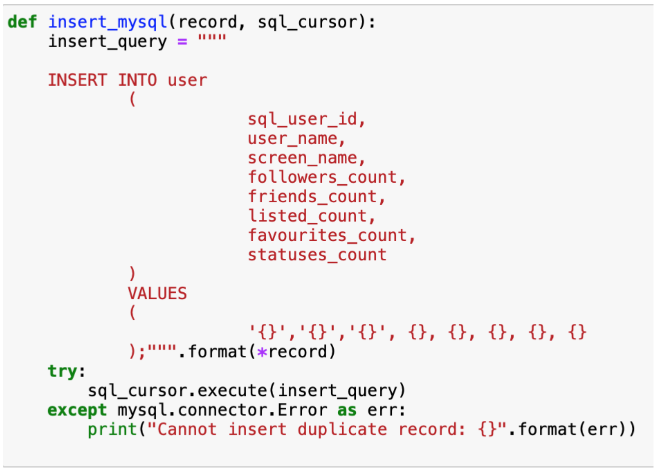
II. Storage Application

In designing the storage application, there were several important considerations. In the data streamed using tweepy, there are both tweets and retweets. Based on the code snippet in **Figure 2a** and the summary in **Figure 2b,** the Storage Application distinguishes tweets and retweets based on the presence of the substring “RT”. The entire JSON response is considered a status. If the status is in fact a retweet, there exists a nested status under the field “retweeted_status”. As such, the tweet metadata to be stored in MongoDB is extracted either from the “retweeted_status” or the top level. In terms of such metadata, the tweet_id, created_date, user_id, followers_count, favorite_count, hashtags/retweet hashtags, and the tweet/retweet text were collected. In terms of user information stored in the MySQL DB, the user_id, name, screen_name, followers_count, friends_count, listed_count, favourites_count, and statuses count were collected.

After selecting fields, it was important that small optimizations be made in the form of indexing. In both SQL and NoSQL database systems, manual indexation on certain fields allows for faster reads. Given the requirements from the Search Application of parsing through large amounts of data in MongoDB and MySQL, it was crucial to create indexes as summarized in **Figure 2c.**

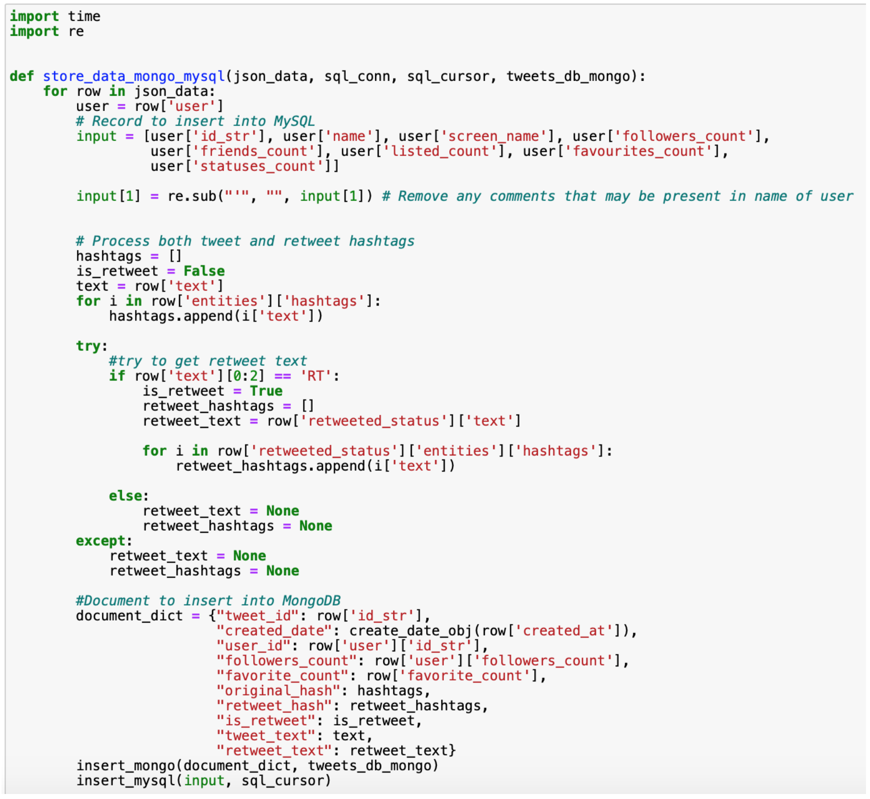
Based on the tweet structure, the tweet_id is a unique identifier of a tweet on the platform. However, in order to connect the user data in the MySQL database to the tweet data in the MongoDB database, it was imperative to include the user_id as a field in MongoDB. Next, the created_at field allowed the application to optimize the search based on time range functionality. Finally, the followers_count allowed for the implementation of sorting results by implementing the concept of relevance.

In the MySQL Database, the User table contained just 3 indexes: sql_user_id, screen_name, and followers_count. The sql_user_id did not require a special index as it was already designated as a primary key for the User table. However, the screen_name and followers_count were both necessary fields to index. The screen_name was necessary for the “Search by User” functionality and the followers count was necessary to implement the concept of relevance.



```
def insert_mysql(record, sql_cursor):
    insert_query = """

    INSERT INTO user
            (
                        sql_user_id,
                        user_name,
                        screen_name,
                        followers_count,
                        friends_count,
                        listed_count,
                        favourites_count,
                        statuses_count
            )
            VALUES
            (
                        '{}','{}','{}', {}, {}, {}, {}, {}
            );""".format(*record)
    try:
        sql_cursor.execute(insert_query)
    except mysql.connector.Error as err:
        print("Cannot insert duplicate record: {}".format(err))
```



```
import time
import re


def store_data_mongo_mysql(json_data, sql_conn, sql_cursor, tweets_db_mongo):
    for row in json_data:
        user = row['user']
        # Record to insert into MySQL
        input = [user['id_str'], user['name'], user['screen_name'], user['followers_count'],
                 user['friends_count'], user['listed_count'], user['favourites_count'],
                 user['statuses_count']]

        input[1] = re.sub("'", "", input[1]) # Remove any comments that may be present in name of user


        # Process both tweet and retweet hashtags
        hashtags = []
        is_retweet = False
        text = row['text']
        for i in row['entities']['hashtags']:
            hashtags.append(i['text'])

        try:
            #try to get retweet text
            if row['text'][0:2] == 'RT':
                is_retweet = True
                retweet_hashtags = []
                retweet_text = row['retweeted_status']['text']

                for i in row['retweeted_status']['entities']['hashtags']:
                    retweet_hashtags.append(i['text'])

            else:
                retweet_text = None
                retweet_hashtags = None
        except:
            retweet_text = None
            retweet_hashtags = None

        #Document to insert into MongoDB
        document_dict = {"tweet_id": row['id_str'],
                         "created_date": create_date_obj(row['created_at']),
                         "user_id": row['user']['id_str'],
                         "followers_count": row['user']['followers_count'],
                         "favorite_count": row['favorite_count'],
                         "original_hash": hashtags,
                         "retweet_hash": retweet_hashtags,
                         "is_retweet": is_retweet,
                         "tweet_text": text,
                         "retweet_text": retweet_text}
        insert_mongo(document_dict, tweets_db_mongo)
        insert_mysql(input, sql_cursor)
```

Figure 2a. Functions for storing data into the mysql and mongodb databases

MongoDB	MySQL
● tweet_id(String)	● sql_user_id (VARCHAR)
● created_date (dateTime)	● sql_tweet_id (VARCHAR)
● user_id(String)	● user_name (VARCHAR)
● followers_count(Integer)	● screen_name (VARCHAR)
● favorite_count(Integer)	● followers_count (BIGINT)
● original_hash(List of Strings)	● listed_count (BIGINT)
● retweet_hash(List of Strings)	● favourites_count (BIGINT)
● isRetweet(Boolean)	● statuses_count (BIGINT)
● tweet_text(String)	
● retweet_text(String)	

Figure 2b. Summary of fields in the two Databases

MongoDB	MySQL
● tweet_id	● sql_user_id(Primary Key)
● user_id	● screen_name
● created_at	● followers_count
● followers_count	

Figure 2c. Summary of Indexed Fields

III. Search Application

Figure 3 shows the final form of the search application. The native python Tkinter library was utilized to create the GUI. In the app, there are only four predefined operations available to the user: radio button selection, the option to quit the application, the option to clear the output to start a new query, and the option to execute the current query. If there is any error, a message highlighted in red provides the user with the verbose error message.

Search By Hashtag

Suppose the user selects the “Search By Hashtag” radio button. After entering text and clicking “Go”, a click event triggers a call to the *go()* method. The program enters the body of the first “if” condition, calling the *search_by_hashtag()* method. First a MongoDB query is generated to search for an $elemMatch on both hashtags and retweets. Next, a Redis key is created to store data in the cache. Finally, the method *search_helper()* is called with the MongoDB query, user entered text, and Redis key. If the key exists in the cache and the entry still has TTL property with a value greater than 0, the summary string is recycled. Otherwise, the MongoDB query is executed. An additional query to count the number of unique users that utilized the hashtag is executed. In the final summary, the total retweets, number of unique users, top 3 tweets sorted in descending order by follower count, and summary generation time are displayed to the end user.

Search By Word

Suppose the user selects the “Search by Word” radio button. After entering text and clicking “Go”, a click event triggers a call to the *go()* method. The program enters the body of the first ‘elif’ condition where the *search_by_word()* method is called. Just as in the previous use case, both a MongoDB query and redis key are created. In this method, the MongoDB query uses the $regex parameter to parse tweet/retweet text for an exact match. Just as in the previous use case, the *search_helper()* method is called with the MongoDB query, user entered text, and redis key. The summary object is produced according to the same business logic as with searching by hashtag.

Search By User

Suppose the user selects the “Search by User” radio button. After entering text and clicking “Go”, a click event triggers the execution of the go() method. The program enters the second ‘elif’ condition where the search_by_user() method is called. Unlike with the previous two use-cases, a MySQL query is used to pull user_id information corresponding to the screen_name collected by the user. It was a deliberate choice to use the screen_name instead of a combination of a first name and last name because the former is unique while the latter is not. The extracted user_id is used by the MongoDB query to pull all tweets and retweets authored by said user_id. While the search_helper() is used for searching by word and searching by hashtag, it is not used for searching by user or searching by time range due to the stark difference in error handling performed by the last two methods. As such, the last two use cases contain many lines of recycled code from search_helper().

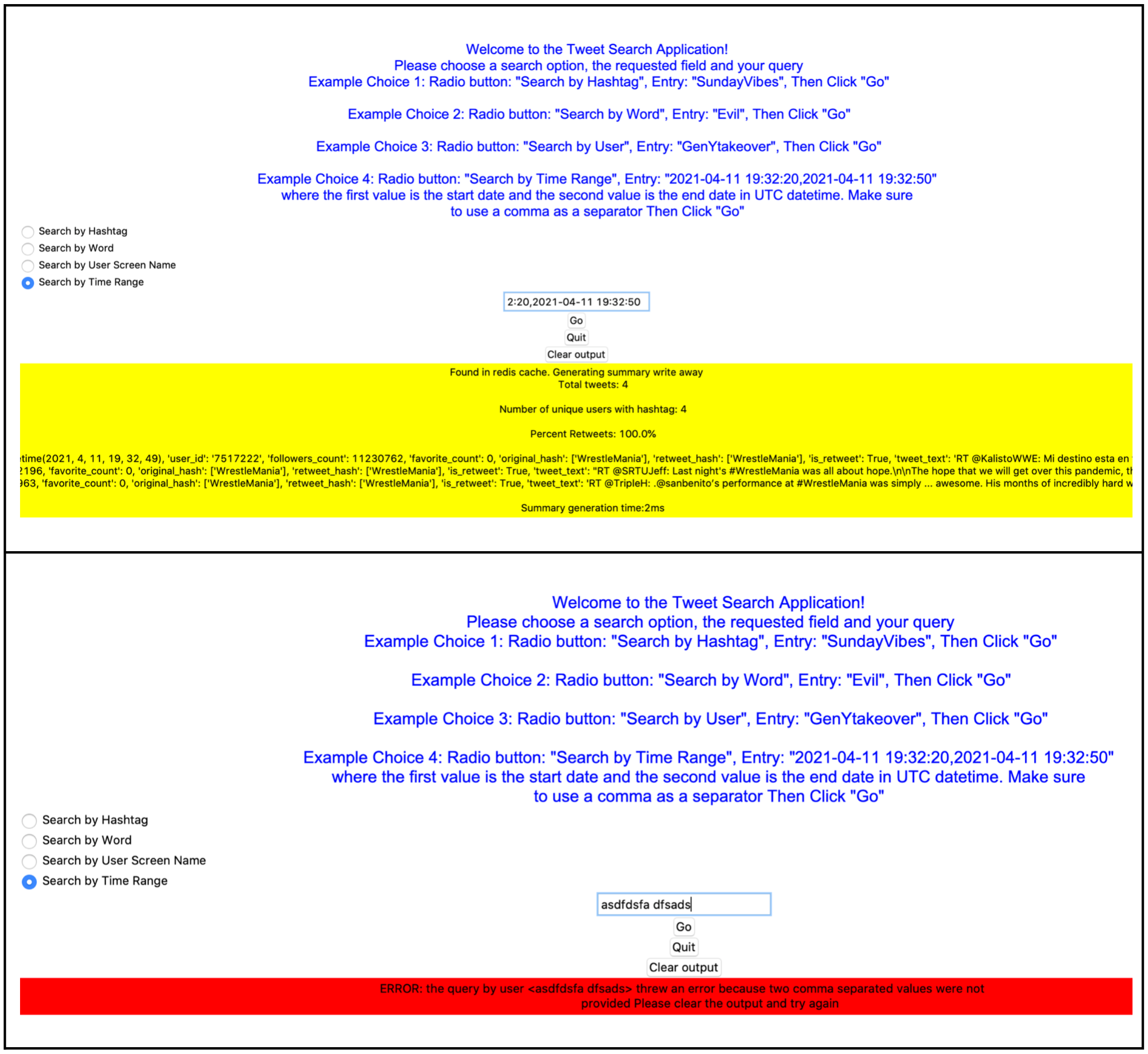
Search By Time Range

Suppose the user selects the “Search by Time Range” radio button. In the final “elif”, there is additional logic not present in the code for the previous three use cases. First, the text is split by the comma delimiter. Next, the program converts the two strings into datetime objects. In the future, it would be better to perform more rigorous checks on the user input including but not limited to verifying the dates, making sure that click event triggers a call to the search_by_time_range() method.

In this method, the $gte and $lt parameters are used to find the tweets in the MongoDB cluster that fall in the range of two Python datetime objects.The summary string is generated in the same way as the previous three cases.

Control Functions

At any point, the user can click on the “Quit” and “Clear” buttons. Clicking “Quit” generates a click event that destroys the main app frame and terminates the program. Clicking “Clear” generates a click event that calls the *clear()* method that destroys the main app frame and restarts the program.


Welcome to the Tweet Search Application!
Please choose a search option, the requested field and your query
Example Choice 1: Radio button: "Search by Hashtag", Entry: "SundayVibes", Then Click "Go"

Example Choice 2: Radio button: "Search by Word", Entry: "Evil", Then Click "Go"

Example Choice 3: Radio button: "Search by User", Entry: "GenYtakeover", Then Click "Go"

Example Choice 4: Radio button: "Search by Time Range", Entry: "2021-04-11 19:32:20,2021-04-11 19:32:50"
where the first value is the start date and the second value is the end date in UTC datetime. Make sure
to use a comma as a separator Then Click "Go"

- Search by Hashtag
- Search by Word
- Search by User Screen Name
- [✓ Search by Time Range]

2:20,2021-04-11 19:32:50

```
Found in redis cache. Generating summary write away
Total tweets: 4

Number of unique users with hashtag: 4

Percent Retweets: 100.0%

time(2021, 4, 11, 19, 32, 49), 'user_id': '7517222', 'followers_count': 11230762, 'favorite_count': 0, 'original_hash': ['WrestleMania'], 'retweet_hash': ['WrestleMania'], 'is_retweet': True, 'tweet_text': 'RT @KalistoWWE: Mi destino esta en
2196, 'favorite_count': 0, 'original_hash': ['WrestleMania'], 'retweet_hash': ['WrestleMania'], 'is_retweet': True, 'tweet_text': "RT @SRTUJeff: Last night's #WrestleMania was all about hope.\n\nThe hope that we will get over this pandemic, th
963, 'favorite_count': 0, 'original_hash': ['WrestleMania'], 'retweet_hash': ['WrestleMania'], 'is_retweet': True, 'tweet_text': 'RT @TripleH: .@sanbenito's performance at #WrestleMania was simply ... awesome. His months of incredibly hard w

Summary generation time:2ms
```

Welcome to the Tweet Search Application!
Please choose a search option, the requested field and your query
Example Choice 1: Radio button: "Search by Hashtag", Entry: "SundayVibes", Then Click "Go"

Example Choice 2: Radio button: "Search by Word", Entry: "Evil", Then Click "Go"

Example Choice 3: Radio button: "Search by User", Entry: "GenYtakeover", Then Click "Go"

Example Choice 4: Radio button: "Search by Time Range", Entry: "2021-04-11 19:32:20,2021-04-11 19:32:50"
where the first value is the start date and the second value is the end date in UTC datetime. Make sure
to use a comma as a separator Then Click "Go"

- Search by Hashtag
- Search by Word
- Search by User Screen Name
- [✓ Search by Time Range]

asdfdsfa dfsads

```
ERROR: the query by user <asdfdsfa dfsads> threw an error because two comma separated values were not
provided Please clear the output and try again
```

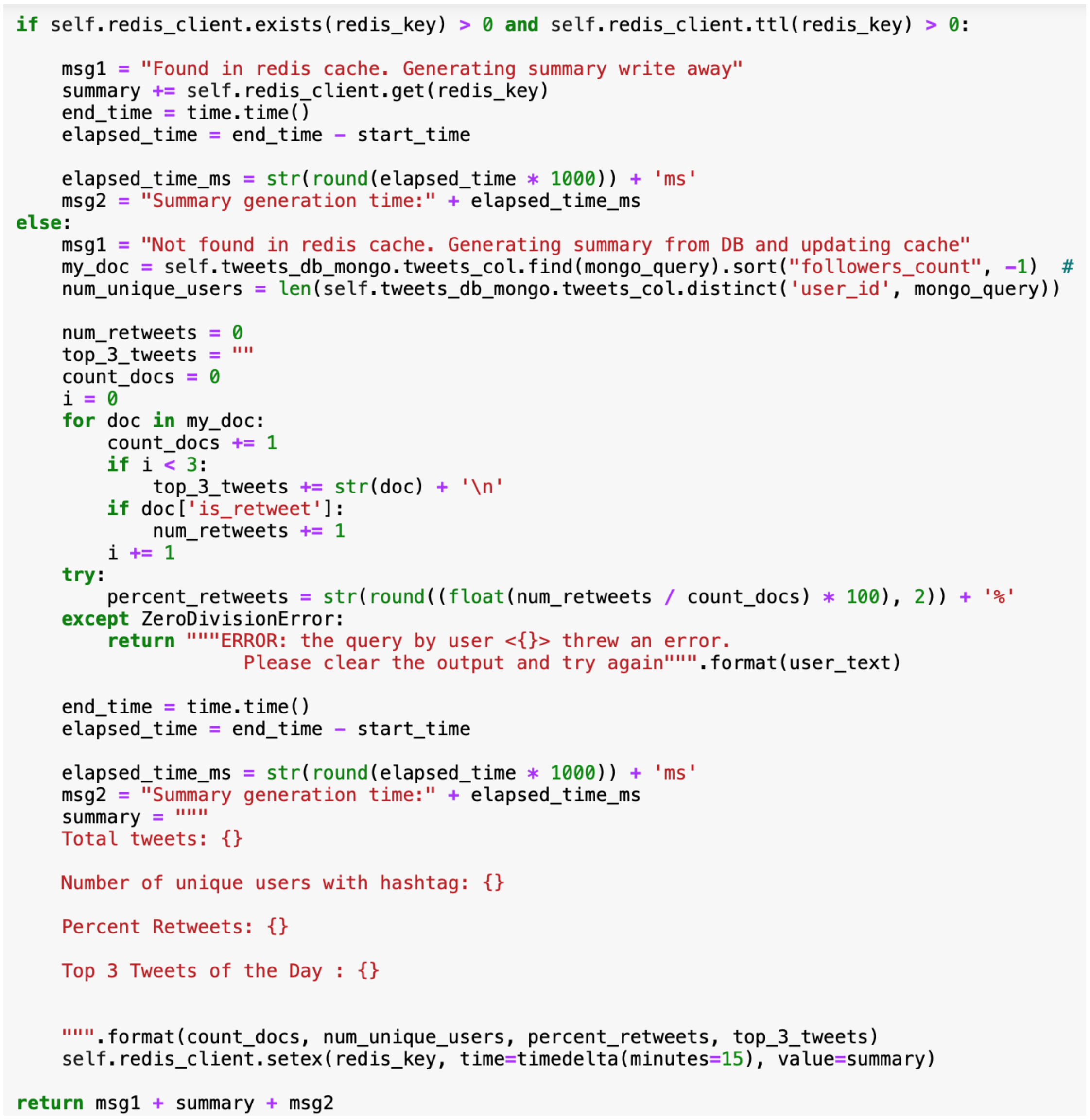
Figure 3: Screenshot of tkinter GUI containing use case of Example Choice 4 (a search by time range) with correct and incorrect input.

The simple design of the application allowed for the quick implementation of the REDIS cache. In the cache, the key was made up of the encoded value of the radio button selected (1=Search by Hashtag, 2=Search by Word, 3=Search by User, 4=Search by Time Range) and the entirety of the text entered by the user. The value includes the summary report string generated including the number of users, percent retweets, and the top three tweets sorted on the basis of follower count. **Figure 4** shows a large difference in timing between cached and uncached data. This is due to a combination of reasons:

1. Since this is not a large-scale application, the cache has less than 30 entries while MongoDB has 30000 entries (making it much easier to search in the cache).
2. The setex() and get() methods in REDIS are both O(1) operations. In addition, REDIS lives in memory whereas our version of MongoDB lives in a cluster on the cloud.
3. The following assumption was made: the execution time is the total time required to produce the report. This will always run in O(k) time where k is the size of the search result returned by MongoDB or MySQL whichever is bigger. **Figure 5** shows a code snippet of such a loop.

	Search by Hashtag	Search by Word	Search by User	Search by Time Range
Uncached	416ms	436ms	211ms	222ms
Cached	13ms	7ms	1ms	2ms

Figure 4: Timings for specific queries by hashtag, word, user, and time range.



```python
if self.redis_client.exists(redis_key) > 0 and self.redis_client.ttl(redis_key) > 0:

    msg1 = "Found in redis cache. Generating summary write away"
    summary += self.redis_client.get(redis_key)
    end_time = time.time()
    elapsed_time = end_time - start_time

    elapsed_time_ms = str(round(elapsed_time * 1000)) + 'ms'
    msg2 = "Summary generation time:" + elapsed_time_ms
else:
    msg1 = "Not found in redis cache. Generating summary from DB and updating cache"
    my_doc = self.tweets_db_mongo.tweets_col.find(mongo_query).sort("followers_count", -1)  #
    num_unique_users = len(self.tweets_db_mongo.tweets_col.distinct('user_id', mongo_query))

    num_retweets = 0
    top_3_tweets = ""
    count_docs = 0
    i = 0
    for doc in my_doc:
        count_docs += 1
        if i < 3:
            top_3_tweets += str(doc) + '\n'
        if doc['is_retweet']:
            num_retweets += 1
        i += 1
    try:
        percent_retweets = str(round((float(num_retweets / count_docs) * 100), 2)) + '%'
    except ZeroDivisionError:
        return """ERROR: the query by user <{}> threw an error.
                Please clear the output and try again""".format(user_text)

    end_time = time.time()
    elapsed_time = end_time - start_time

    elapsed_time_ms = str(round(elapsed_time * 1000)) + 'ms'
    msg2 = "Summary generation time:" + elapsed_time_ms
    summary = """
    Total tweets: {}

    Number of unique users with hashtag: {}

    Percent Retweets: {}

    Top 3 Tweets of the Day : {}


    """.format(count_docs, num_unique_users, percent_retweets, top_3_tweets)
    self.redis_client.setex(redis_key, time=timedelta(minutes=15), value=summary)

return msg1 + summary + msg2
```

Figure 5: Search application method snippet including logic for checking cache, querying the database and generating the summary report.

IV. Conclusion

The Tweet Search Application is a limited desktop application that runs in a Python Jupyter Notebook. When the notebook is closed, the app is terminated. If work on the app would continue, it would make sense to make it cross platform, giving the users the option to view it in a browser and both the Android and iOS platforms. This would be a beneficial update to the Tweet Search Application as Twitter itself is a social media app most commonly used on phones. If the user has multiple electronic devices, their app data and preferences should be synced across all their devices. Such an update would require the usage of a cloud tool such as AWS or Azure, and the conversion of the local MySQL database to a web hosted database. It is also crucial that the system implements a more complicated cache replacement policy to account for the rapidly changing trending tags. The ideal Tweet Search Application would thus most likely include a Machine Learning component to turn the system into both a recommendation and search engine.