FINAL REPORT

Team 6: Benjamin Barnett, Yaniv Bronshtein

**Data Collection Application (Benjamin Barnett)**

* Tweet Collection
* Hiding credentials in .env

**Data Storage Application (Benjamin Barnett)**

* Setting up and designing structure of MySQL database
* Setting up and designing structure of MongoDB database
* Commands and functions to insert data into MySQL and MongoDB
* Commands and functions to create MySQL table and MongoDB collection

**Search Application (Yaniv Bronshtein)**

* Creation and error handling of Tkinter GUI
* Setting up and designing the Redis cache
* Commands to query data from MySQL database
* Commands to query data from PyMongo database
* Commands to query the Redis cache
* Commands to insert into the Redis cache

# Data Collection Application

In the interim report, we mentioned that most of our time was spent on tweaking the data collection application. Since then, most of our total effort was used for other content. However, it is worth restating the process of our data collection. Due to constant internet interruptions, collecting the necessary tweets was an arduous task. We often struggled with the fact that even though certain tweets were trending, they were specific to just one day as opposed to tweets that could be updated. For example, since Sunday happens exactly once every week (versus once every year, for instance), a caching system that used such a hashtag would provide valuable data versus #NationalSiblingDay or #NationalPetDay. Likewise, sporting events such as wrestling and UFC were frequently promoted and lent themselves to other subtweets. After much consideration, we chose the following hashtags: #sundayvibes, #UFCVegas23, and #WrestleMania.

In terms of data collection, the challenge was to store the data in an efficient file structure as text files only offer sequential reads. Since the data needed to be stored in a dictionary, JSON file storage was a natural choice. But to utilize the powerful JSON library, we needed to figure out how to store the tweets in a valid JSON format. To solve this problem, we wrote a ‘[‘ into the file before processing any tweets, and then wrote ‘,’ to the file after each tweet (with the exception of the last tweet). Lastly, we closed the file with ‘]’ so that the JSON file contained the following structure: [{tweet1}, {tweet2}, ……, {tweetn}], after which the program aborted itself. See **Figure 1** in the appendix for the tweet and retweet structure provided by the tweepy API and their fields.

# Data Storage Application

## Field Selection

After collecting the tweets, we believed that the hard part was over. However, we found most of our time spent debugging and redesigning the structure of our two databases: MongoDB and MySQL. The stages of our progress can be split at the time of the interim report, final presentation, and code demonstration, which we will refer to by version 1, 2, and 3, respectively. In version 1, we included many features in the two databases without having a clear idea of overall applicability. MongoDB and MySQL were also not yet successfully set up, so we intended to fix this by the next part of the project. In version 2, we decided to replace the timestamp\_ms with the created\_date field of type dateTime for MongoDB. After trying many times to implement the search by TimeRange, we realized that PyMongo only recognizes dates that are dateTime objects. Although there were more elegant solutions, we decided to create a manual helper function to parse the “created\_at” String field provided by tweepy and return a valid python dateTime object. In terms of MySQL, there was no functional difference between version 1 and version 2 other than a name change (this was due to having several duplicate fields in the two databases and the removal of a timestamp field). The most drastic change occurred in version 3 two days before the demo when nothing was working. We went back to the drawing board and only kept the bare minimum features with a valid use case. **Figure 2** summarizes the overall field selection progress.

## Indexing/Optimization

Without the understanding of indexing and optimization, it may be hard to understand why we removed certain features. **Figure 3** shows the progress during the time of the interim report, final presentation, and code demo. In version 1, we created 3 indexes in MongoDB and 3 indexes along with a primary key in MySQL. But once we began the insertion process, we realized that the timestamp feature was completely unusable in PyMongo since there was no method in Python available to convert a user inputted date into the tweepy provided timestamp. Only after we had created and tested the function described in the previous section were we able to assign an index and move forward with the application. Between version 1 and version 2, we decided to make the user\_id not a unique field (and therefore needed the user table to contain a composite key). In addition, user\_name was not as unique as screen\_name so we opted to create an index on the screen\_name instead. However, we noticed that processing 30,000 tweets would mean that integrity constraints would always be broken anyway. The solution to that was to surround the SQL insertion in a try-except block as seen in **Figure 4.** After version 2, we realized that we needed to figure out a better way of implementing relevance. We came up with using followers\_count that was stored in both MongoDB and MySQL and thus created an index on that field in both databases. In version 3, we actually went back to a similar design to version 1 in that user\_id was the primary key. We reasoned that having the tweet\_id inside the user table would not have added value in the search application; it is enough to know the sql\_user\_id to search in MongoDB for the corresponding tweet information. Overall, the struggle with indexing was to include the bare minimum of indexes. If there were too many indexes, insertion time would grow.

# Search Application

Only after the Data Storage application was perfected were we able to move on to the Search Application depicted in **Figure 5**. We decided to create a GUI using the native Tkinter library as it allowed us to show error handling better. Using a for loop in Jupyter Notebook would have required constant scrolling and detracted from the user experience. The app thus simplifies the work required from the end-user by having only one variable operation (entering the query). Other than that, 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.

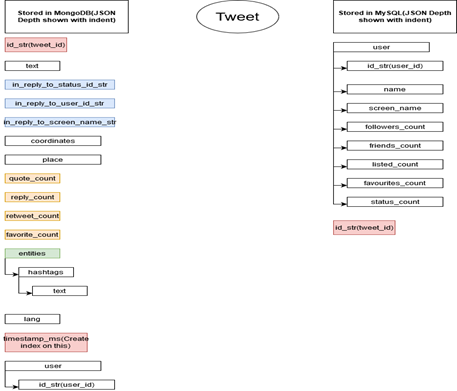
The simple design of the application allowed us to quickly and naturally design the REDIS cache. In the cache, the key was made up with the encoded value of the radio button selected (1=Search by Hashtag, 2=Search by Word, 3=Search by User, 4=Search by Time Range) and the entirety of the text entered 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 6** shows a large difference in timings between cached and uncached data. This is due to a combination of reasons:

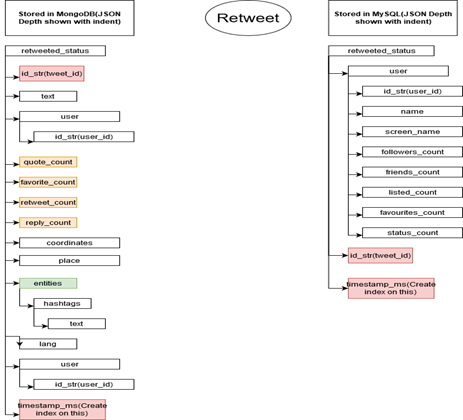
1. Since this is not a large scale application, the cache has less than 30 entries while MongoDB has 30000 entries (making it much easier to search in the cache).
2. The setex() and get() methods in REDIS are both O(1) operations and REDIS lives in memory whereas our version of MongoDB lives in a cluster on the cloud.
3. We took the liberty of defining the execution time as not the time taken to execute the database query but rather 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 7** shows a code snippet of such a loop.

# Concluding Remarks

In this project, we learned the valuable lesson that while getting a piece of code to work on a small dataset is important, it will most likely break down with a large dataset. For example, our application worked perfectly with an 11 tweet sample in version 2. As soon as we tried to insert the entire dataset consisting of 30,000 entries, the application crashed due to an integrity constraint error that led to a last minute rewriting of the entire application. In order to prevent a similar situation from occurring in a future project, we agreed to evaluate mission critical code on datasets of small, medium and large sizes.

# Appendix





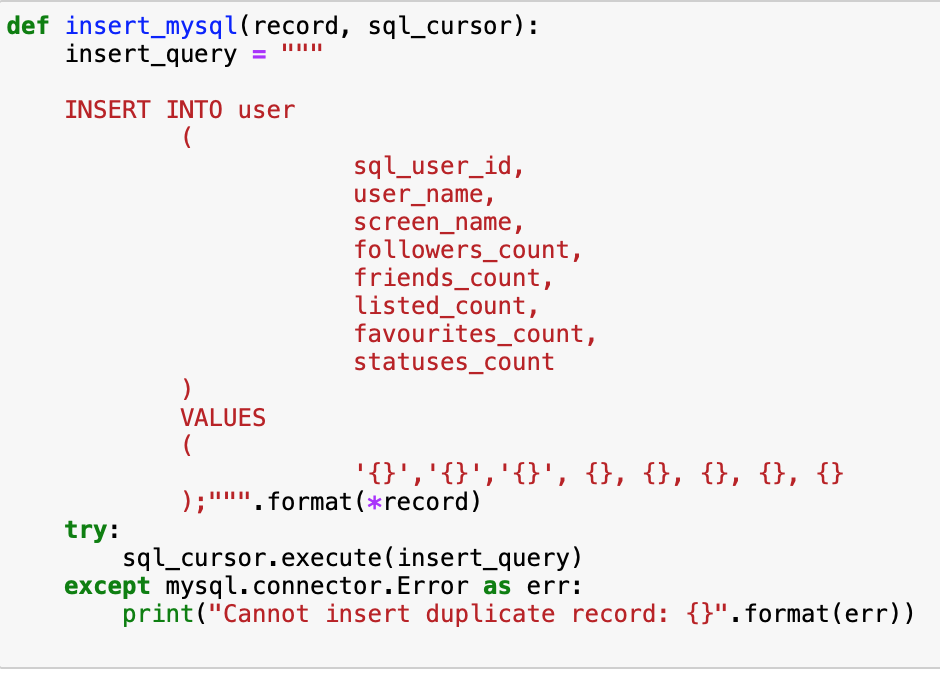
**Figure 1:** The tweet and retweet structures provided by the tweepy API.

| Version | MongoDB | MySQL |
| --- | --- | --- |
| Interim Report | * tweet\_id(String) * user\_id (String) * tweet\_text (String) * quote\_count (Integer) * favorite\_count(Integer) * retweet\_count (Integer) * reply\_count (Integer) * coordinates(?) * place (String) * hashtags(List of Strings) * lang(String) * timestamp\_ms(String) * isRetweet(Boolean) | * user\_id (VARCHAR) * tweet\_id (VARCHAR) * name (VARCHAR) * screen\_name (VARCHAR) * friends\_count (BIGINT) * listed\_count (BIGINT) * favourites\_count (BIGINT) * statuses\_count (BIGINT) * timestamp\_ms (VARCHAR) |
| Final Presentation | * mongo\_tweet\_id(String) * mongo\_user\_id (String) * tweet\_text (List of Strings) * quote\_count (Integer) * favorite\_count(Integer) * retweet\_count (Integer) * reply\_count (Integer) * coordinates(?) * place (String) * hashtags(List of Strings) * lang(String) * created\_date(dateTime) * isRetweet(Boolean) | * sql\_user\_id (VARCHAR) * sql\_tweet\_id (VARCHAR) * name (VARCHAR) * screen\_name (VARCHAR) * friends\_count (BIGINT) * listed\_count (BIGINT) * favourites\_count (BIGINT) * statuses\_count (BIGINT) |
| Code Demo | * tweet\_id(String) * created\_date (dateTime) * user\_id(String) * followers\_count(Integer) * favorite\_count(Integer) * original\_hash(List of Strings) * retweet\_hash(List of Strings) * isRetweet(Boolean) * tweet\_text(String) * retweet\_text(String) | * sql\_user\_id (VARCHAR) * sql\_tweet\_id (VARCHAR) * user\_name (VARCHAR) * screen\_name (VARCHAR) * followers\_count (BIGINT) * listed\_count (BIGINT) * favourites\_count (BIGINT) * statuses\_count (BIGINT) |

**Figure 2:** Fields used in database design at the time of interim report, final presentation, and code demo.

| Version | MongoDB | MySQL |
| --- | --- | --- |
| Interim Report | * tweet\_id * User\_id * Timestamp\_ms | * User\_id (Primary Key) * Tweet\_id * user\_name * Timestamp\_ms |
| Final Presentation | * mongo\_tweet\_id * mongo\_user\_id * created\_at | * (sql\_user\_id , sql\_tweet\_id) Composite Primary Key * screen\_name |
| Code Demo | * tweet\_id * user\_id * created\_at * followers\_count | * sql\_user\_id(Primary Key) * screen\_name * followers\_count |

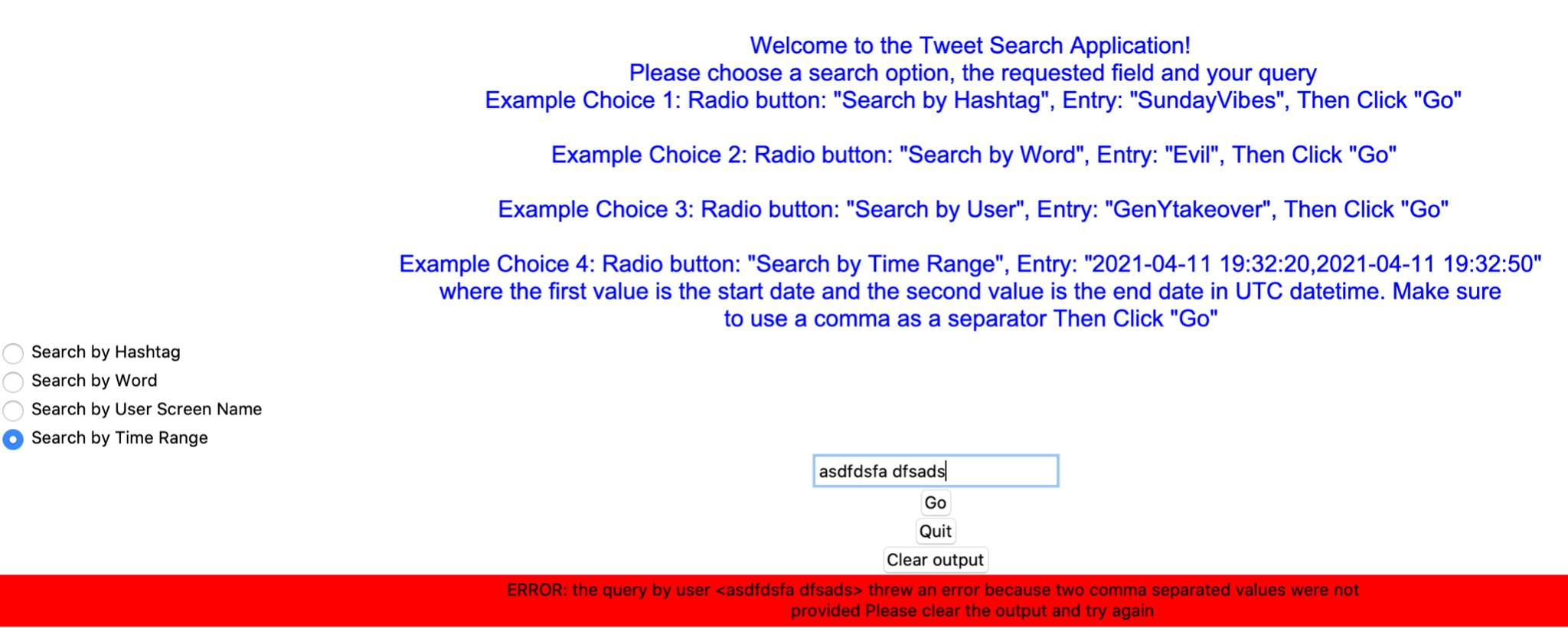
**Figure 3:** Index design used at the time of interim report, final presentation, and code demo.



**Figure 4:** MySQL record insertion function.



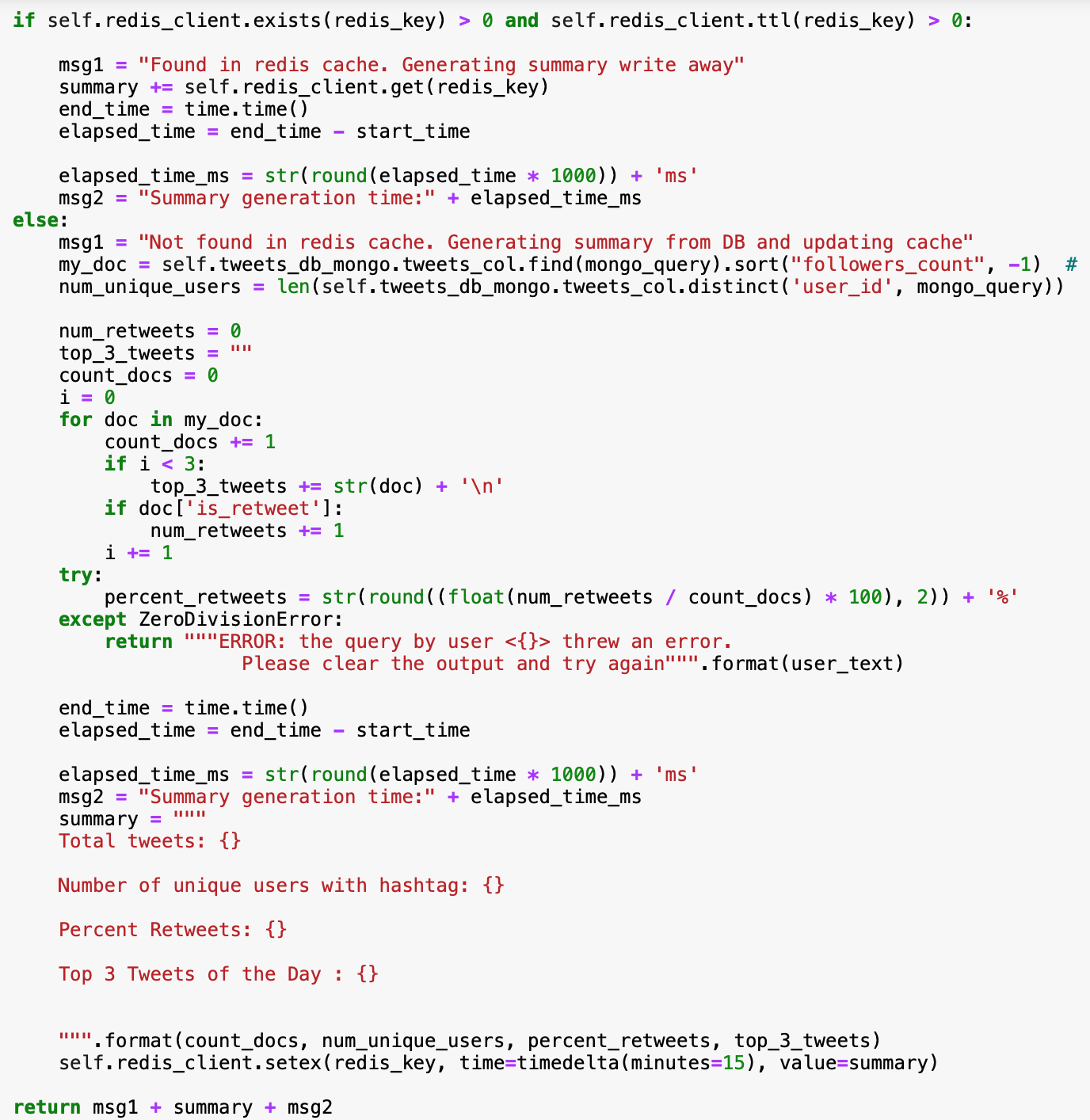
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



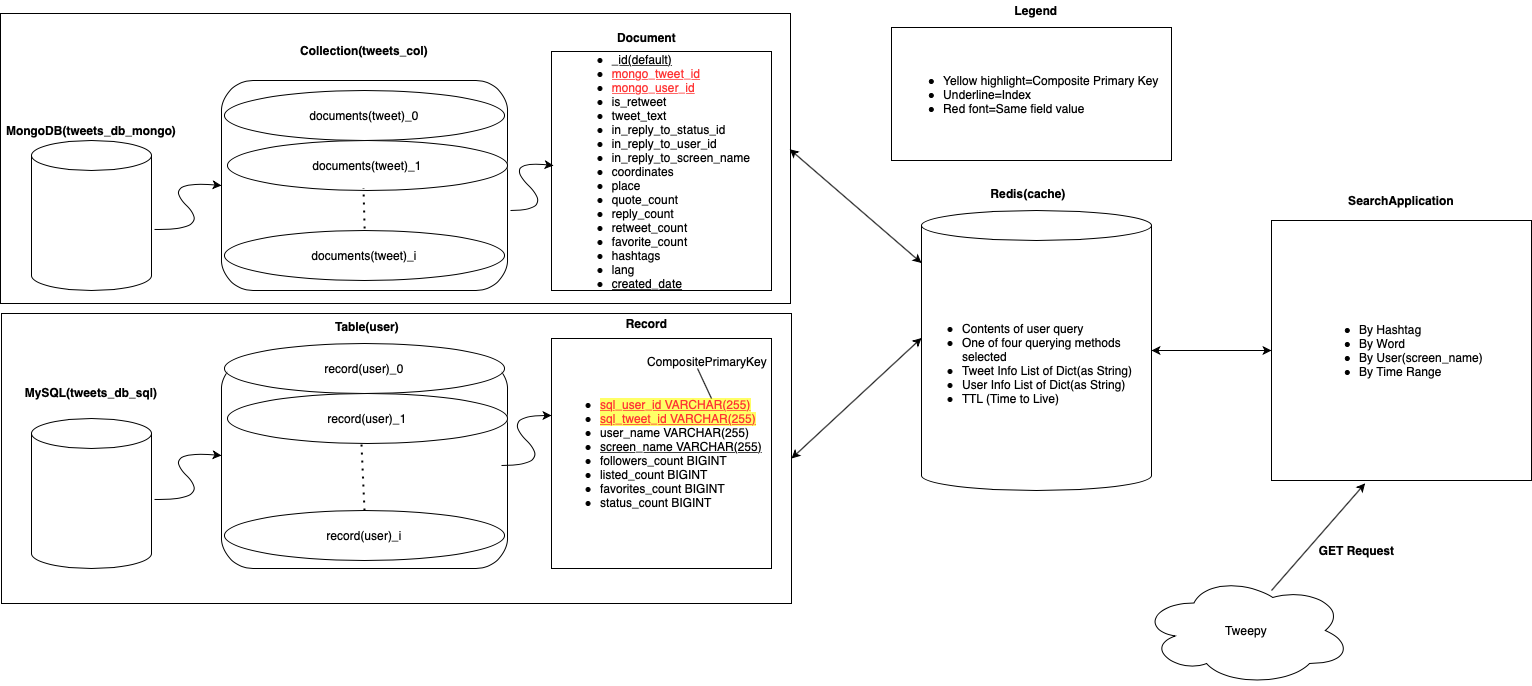
**Figure 5:** Screenshot of tkinter GUI containing use case of Example Choice 4 (a search by time range) with correct and incorrect input.

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

**Figure 6:** Timings for specific queries by hashtag, word, user, and time range.



**Figure 7:** Search application method snippet including logic for checking cache, querying the database and generating the summary report.

**Figure 8:** Application design used in the final presentation.