Twitter Search Application

# **Team 27**

**Team Members:**

Nikitha Sunku Prashanth - ns1590

Priyanka Nagasuri - pn296

Nancy Soni - ns1583

Shruthi Nanditha Ganesh - sg2057

# Abstract

This project aims to design and implement a search application tailored for a tweet store database. The application allows users to search tweets based on string, hashtag, and user, while also providing drill-down features for deeper exploration of search results. The system employs a combination of MongoDB and PostgreSQL databases for efficient storage and retrieval of tweet and user data. The architecture, functionality, and implementation details of the search application are discussed, along with its key features and performance considerations. The project will also incorporate a user interface that allows users to input their search queries and browse through the search results. The proposed search engine could be useful for researchers, marketers, journalists, and anyone who wants to search through a specific dataset of tweets for information on a particular topic or trend.

# Introduction

With the proliferation of social media platforms like Twitter, the volume of user-generated content, particularly tweets, has grown exponentially. Managing and searching through this vast amount of data poses significant challenges. The proposed search application is designed to address these challenges by providing efficient search and exploration capabilities for tweet data.

# Implementation

The implementation of the search application involves several components:

**Database Setup:**

**MongoDB Storage for Tweets:**

MongoDB, an open-source NoSQL database was used, which simplifies tweet storage, search and recall without the need to write a tweet parser since it effectively parses the data when we retrieve it. Also, unlike other NoSQL databases, MongoDB offers strong consistency, an expressive query language, and secondary indexes.

Tweets were stored in the tweets\_data collection within the twitter\_db database in the MongoDB cluster. The collection is indexed in text, name, and media fields. Hashtags and date to optimize search and retrieval

operations.

**PostgreSQL Storage for Tweets:**

PostgreSQL was utilized for its robust relational database capabilities, ensuring data integrity, and supporting complex queries with a structured approach. To obtain the connection to PostgreSQL and Python, psycopg2 was installed. Tables have been created for user's metadata with relationships maintained through foreign keys. This setup facilitates efficient data management and analysis, particularly for operations that require joining data across tables, such as aggregating user interactions or analyzing tweet threads.

**MongoDB Data Preprocessing:**

Here is the schema used to store tweet data in the MongoDB cluster as a non-relational database after exploring the data. First, it checks if the tweet is a retweet, if it is then it is added to the retweets\_final collection and a quoted retweet is added to the tweets\_final collection. If the tweet is not a retweet, it adds it to the tweets\_final collection, and a quoted tweet and a retweet are added to the tweets\_final collection. Some columns were renamed for simplicity and the time the tweet was created was transformed into Unix timestamp format. A new column named “priority” was added considering to order of relevance of the search results during the search and can be used as popular tweets. It was counted by adding counts of columns favourite\_count, retweet\_count, quote\_count, and reply\_count and dividing it by followers count. Loading all this data took around 5 hours, with each tweet processed individually to make it a streaming app. After the data was loaded, it was indexed on both the tweets\_final and retweets\_final collections. Tweets\_final collection was indexed on user-id, post-id, and timestamp descending and on hashtags on text. While retweets\_final collection was indexed on tweet id ascending, user-id descending, and the compound index was used by tweeted and timestamp.

Figure 1 shows the example of the code used to load the data and Figure 2 & Figure 3 shows the schema of tweets and retweet collections. Figure 4 shows the code used to index the data.

A screenshot of a computer code

Description automatically generated

Figure 1 : Code used for data loading and preprocessing in MongoDB.

A screenshot of a computer code

Description automatically generated A screenshot of a computer code

Description automatically generated

Figure 2: Tweets\_final collection Schema. Figure 3: Retweets\_final collection Schema.

A screenshot of a computer code

Description automatically generated

Figure 4: Code used to index non-relational data.

**Postgre Data Preprocessing**

A series of sophisticated data management techniques are employed within a PostgreSQL database environment, leveraging both SQL and Python to optimize and streamline database operations. By initializing a connection to the PostgreSQL database using the psycopg2 library. This setup is critical for enabling the execution of SQL commands directly from Python. The connection is established by specifying detailed parameters such as host, database, user, and password, which are essential for accessing the PostgreSQL server securely and efficiently.

Upon establishing a stable database connection, the code likely configures the database schema to enhance performance and manageability. This includes defining partitioned tables, a strategy that significantly improves query performance and data management for large datasets by segmenting tables into discrete, manageable subsets based on specific key fields. Alongside, strategic indexing is implemented on columns identified as critical for query performance. Using SQL to create index statements, these indexes are designed to optimize retrieval times and enhance the efficiency of query operations by minimizing the need for full-table scans.

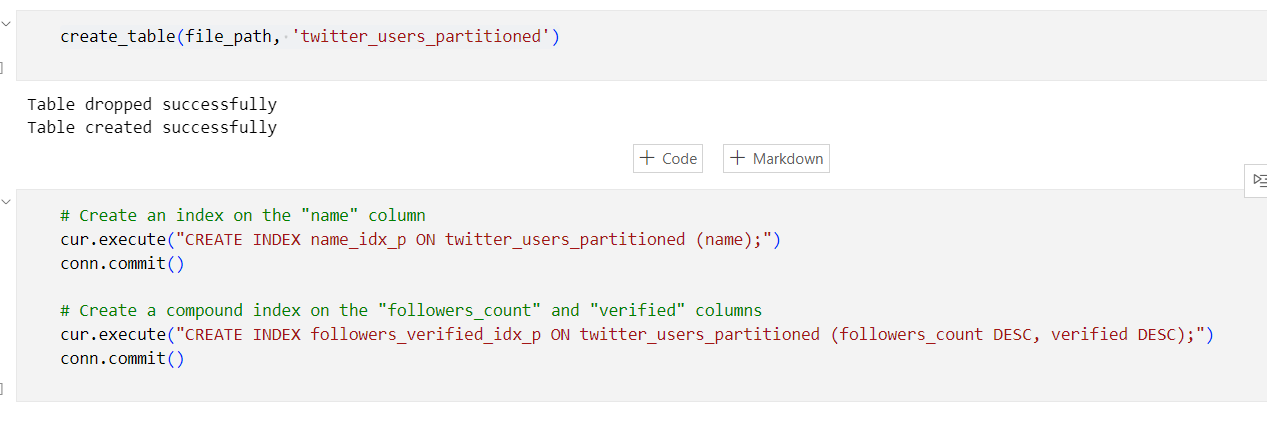


Figure 5 : Indexing.



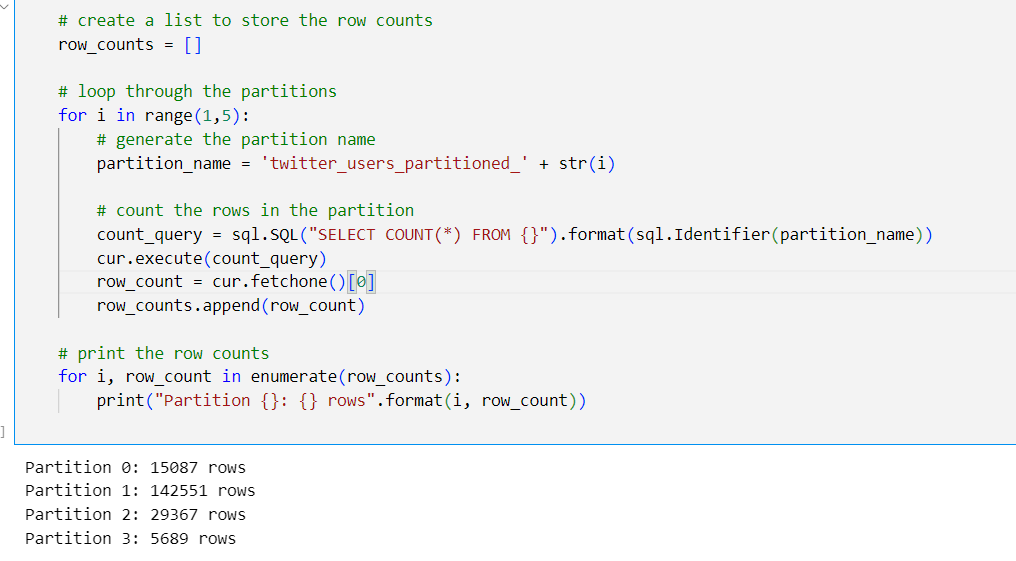


Figure 6 : Partitioning

Further database interactions might include complex SQL queries for data analysis, updates, or deletions, demonstrating the notebook's capability to handle various data management tasks dynamically. This includes tuning queries for performance, utilizing PostgreSQL’s advanced features like window functions, and maintaining data integrity through transactional controls.

This workflow exemplifies a robust integration of Python and SQL within a PostgreSQL context, optimizing data handling and processing capabilities for large-scale and complex data environments. The technical sophistication of the operations ensures that data is managed efficiently, securely, and ready for advanced analytical applications.

# Caching

To decrease database requests and increase performance, the search application uses caching. Certain searches are the bottleneck while doing operations on our search application that connects to PostgreSQL and MongoDB databases. Some of the queries in our MongoDB database may become quite slow as it grows. When there is a problem with the connection between the database server and the search application server, this problem gets worse. Caching is therefore crucial to resolving these problems.

The cache holds frequently accessed data, like top tweets, well-known users, and hashtags. To keep the most recent information, the cache uses a Least Recently Used (LRU) algorithm with a 100-item cache size. An Ordered Dictionary has been implemented to store the data of the cache. Therefore, when the cache exceeds its maximum size then the least recently used key will be removed from the dictionary.

A Time to Live (TTL) of 3600 seconds is set to guarantee freshness; the cache is then refreshed from the source. If the TTL of an entry elapses without being accessed or updated, the entry becomes stale as it no longer represents the most current state. A cache lookup is initiated when an application retrieves a stale entry from the cache. The cache deletes the stale entry if the entry's TTL has passed.

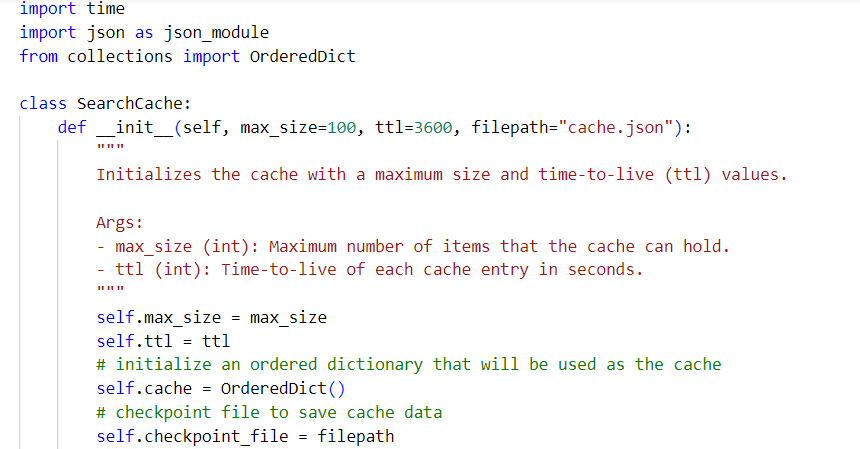


Figure 7 : Initialization of Cache

The program updates the cache with the most recent data upon each subsequent access by retrieving the fresh data from the source. If an entry is found to be stale based on its TTL, it is removed from the cache before returning the value. This ensures that only fresh and up-to-date data is served from the cache. The expiry Mechanism with Time-To-Live (TTL) is initiated by checking each entry’s TTL against the current time. If the difference between the current time and the entry's timestamp exceeds the TTL, the entry is considered stale and is removed from the cache. Additionally, when setting a new cache entry, the timestamp of the entry is recorded along with its value. This timestamp is used to determine the elapsed time since the entry was last accessed or updated, allowing for TTL-based expiry.

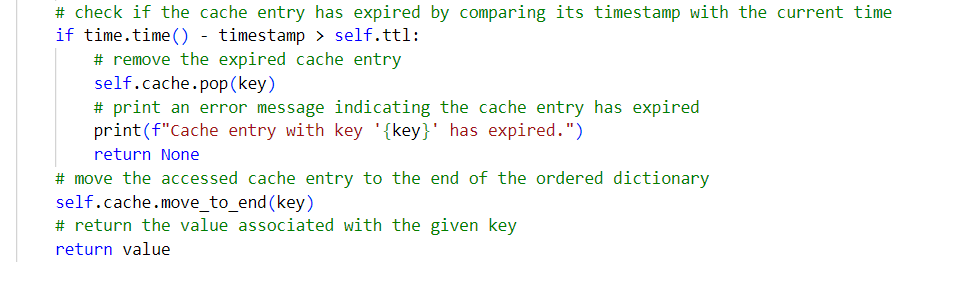
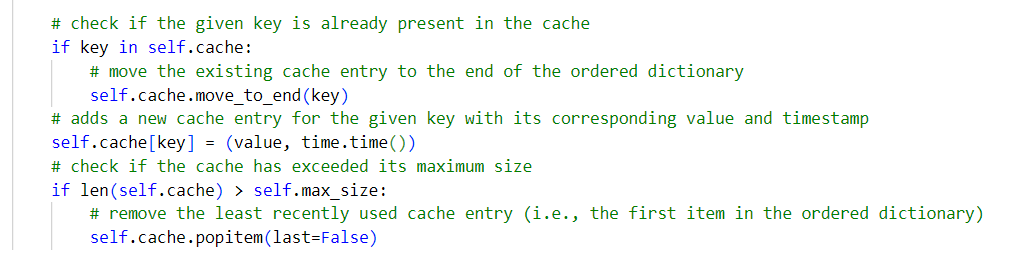
 

Figure 8 : Check for stale entries Figure 9 : Timestamp association with each key

To save the current state of the cache to a file on disk. Checkpointing occurs periodically, ensuring that the cache state is persisted to disk at regular intervals. This process helps in preserving the cache state across application restarts or failures.

This caching strategy, with its combination of LRU eviction policy, TTL-based expiration, and checkpointing for persistence, significantly enhances search responsiveness by reducing latency and database load. It ensures that frequently accessed data remains readily available, offering users fast and up-to-date search results.

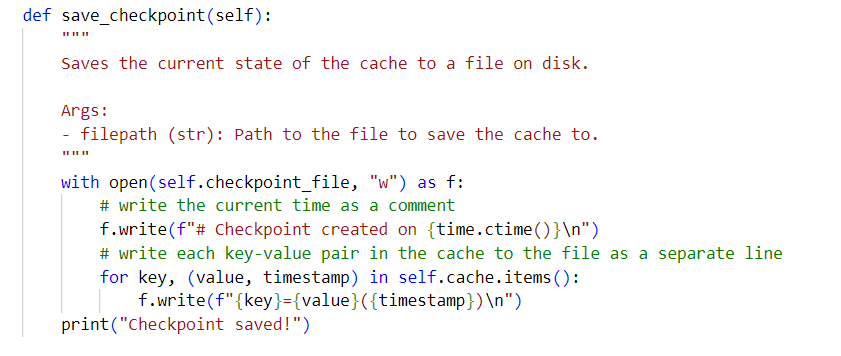


Figure 10 : Checkpoint Implementation

Caching results**:** Below Table 1 shows the execution of time with and without cache.

|  |  |  |
| --- | --- | --- |
| **Query** | **Timing without cache** | **Timing with cache** |
| Search by string | 0.5115 | 0.0003 |
| Most Popular users | 1.9375 | 0.0012 |
| Search by hashtag | 0.0575 | 0.0031 |
| Search by username | 0.0588 | 0.0001 |
| Most Popular Hashtags | 0.0656 | 0.0003 |

Table 1 : Execution of time with and without cache

# Web User Interface

Python flask is the micro-web framework for connecting Python with the user interface. For the UI we used technologies like HTML and CSS. Route definitions are created using the @app.route() decorator, specifying the URL path and associated view function for handling requests. Each route's view function handles incoming requests, processes data, and returns an appropriate response, often by rendering HTML templates using render\_template().

# **Search Application**

For the search application, an object-oriented approach has been utilized, encapsulating the search functionality for PostgreSQL and MongoDB databases within the SearchEngine\_postgre and SearchEngine\_mongodb classes. Both classes leverage caching to improve performance and connect to both MongoDB and PostgreSQL databases to fetch data. Constructor (\_\_init\_\_) Initializes the search engine object with specified cache settings (cache\_size and cache\_ttl).

In the search application, users are provided with the ability to search by:

|  |  |
| --- | --- |
| * Top 10 Most Popular Users | * String |
| * User\_name | * Hashtag |

Additionally, users are presented with other options such as:

* Top 20 Tweet with maximum retweet\_count
* Top 10 hashtags

**most\_popular\_users**: This method retrieves the Top 10 most popular Twitter users from the database along with their attributes, it utilizes a SQL query to fetch the required data. The query selects users based on their follower count, ranks them, and then retrieves the top 10 users as shown in Figure 11 Results are fetched using the cursor and converted into a list of dictionaries, each representing a user. Figure 12 shows the execution time without cache and with cache.

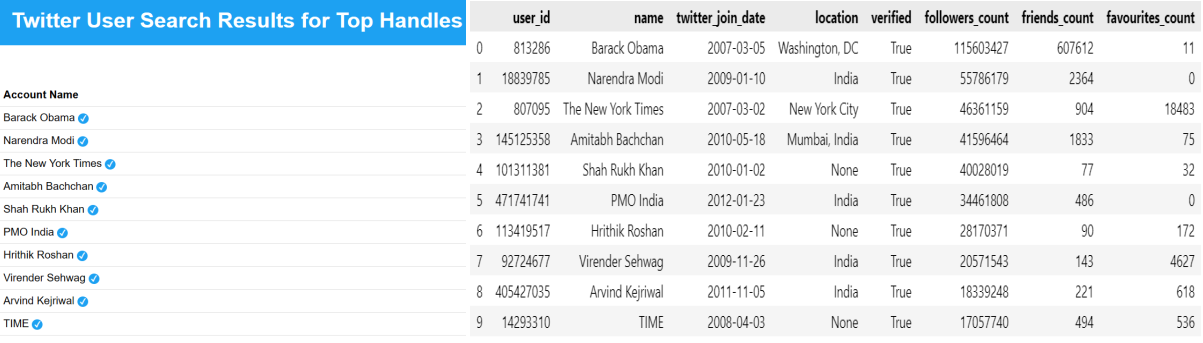


Figure 11: Results of the Top 10 Most Popular Users

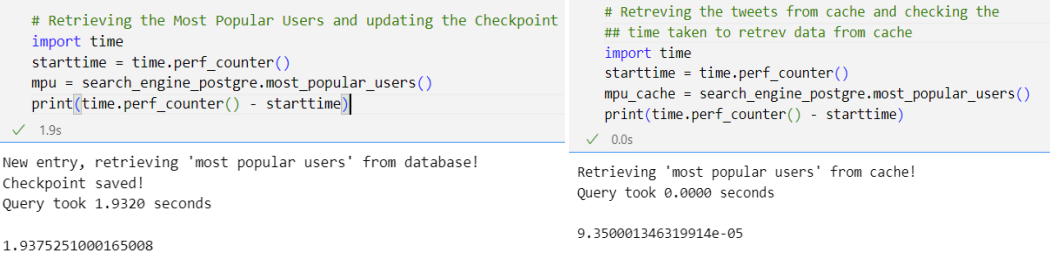


Figure 12 : Execution time With and Without Cache

**search\_user**: This method searches for users whose names match a provided string. It constructs an SQL query using a wildcard pattern to search for similar usernames. Results are fetched using the cursor and converted into a list of dictionaries containing usernames and screen names as shown in Figure 13 below. The Figure 14 shows the results of the execution time of caching.

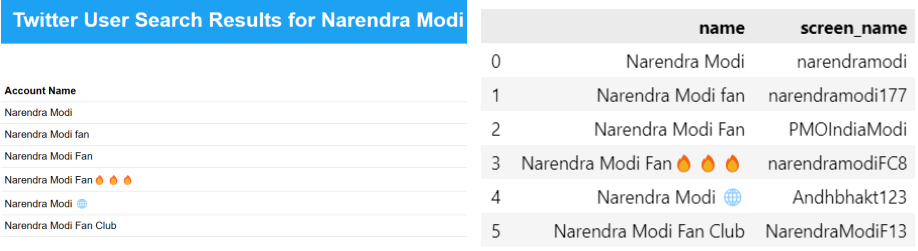


Figure 13 : Search\_user Results.

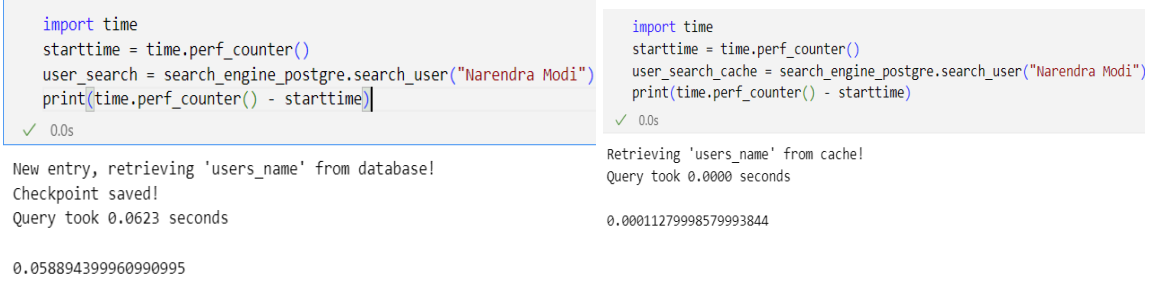


Figure 14 : Execution time taken to search\_user With and Without Cache

The SearchEngine\_mongodb class serves as a robust tool for searching and retrieving tweets stored in a MongoDB database. With a focus on enhancing user experience and query efficiency, the class offers four main methods: search\_by\_string, search\_by\_hashtag, get\_top\_hashtags, and top\_tweets. The search\_by\_string method allows users to find tweets containing a specific string, leveraging MongoDB's $search aggregation pipeline stage for full-text search functionality, The Results are shown in Figure 15. Similarly, the search\_by\_hashtag method enables users to locate tweets associated with a particular hashtag by utilizing the same aggregation pipeline stage as shown in Figure 16. The get\_top\_hashtags method retrieves the top hashtags based on their frequency of occurrence within the tweets, employing MongoDB's aggregation pipeline to aggregate and sort the data effectively as outlined in Figure 17. Additionally, the top\_tweets method retrieves the top tweets based on the number of retweets, providing users with valuable insights into trending content as portrayed in Figure 18. Performance metrics, such as query execution time and caching effectiveness, underscore the efficiency of the search engine, enhancing user satisfaction and system performance.

A screenshot of a computer

Description automatically generated

Figure 15 : Searching the tweets by String.

A screenshot of a computer

Description automatically generated

Figure 16 : Searching the tweets by Hashtags.

A screenshot of a computer

Description automatically generated

Figure 17 : Searching Top 10 Hashtags

A screenshot of a computer

Description automatically generated

Figure 18 : Searching Top 20 Tweets by retweets\_count.

# Conclusion

Compared to other social media platforms like Facebook and Instagram, Twitter data is more complex. Retweets, responses (which are tweets on their own but are comments), and quoted tweets are all available on Twitter. Because of the complexity of Twitter data, designing storage databases is crucial to making it easier to handle and query the data that has been stored. We came to understand the significance of database structural elements like indexes (which affect query time), the necessity of dividing data into multiple tables (which facilitates easy querying and conserves database disk space), and the necessity of properly establishing all relationships between the databases and tables. As an illustration, our common parameter is the user ID, which we connect the data from PostgreSQL and MongoDB, and it was crucial to have both columns with the same data types.

Overall, working on this project helped us learn the basics of database management systems and in particular the structure and usage of MongoDB, PostgreSQL databases, Least recently used caching, and Flask WebApp. We have also learned the importance of teamwork and coordination to build a successful DBMS and the need for applications/tools like GitHub.

# References

1. [https://www.tutorialspoint.com/postgresql/index.htm](file:///C:\Users\91987\AppData\Local\Microsoft\Windows\INetCache\IE\OE425QNS\Twitter-Search-Application-main) - Postgre installation, configuration, SQL commands, and advanced features.
2. [https://www.geeksforgeeks.org/python-mongodb-tutorial/](file:///C:\Users\91987\AppData\Local\Microsoft\Windows\INetCache\IE\OE425QNS\Twitter-Search-Application-main) - Mongodb installation, indexing and utilisation
3. [https://developer.twitter.com/en/docs/twitter-api/enterprise/search-api/overview](file:///C:\Users\91987\AppData\Local\Microsoft\Windows\INetCache\IE\OE425QNS\Twitter-Search-Application-main) - Real-Time Tweet Retrieval and Analysis Using the Twitter Search API, S. Asur, B. A. Huberman, ICWSM, 2010.
4. [https://www.codecademy.com/article/cache-eviction-policies](file:///C:\Users\91987\AppData\Local\Microsoft\Windows\INetCache\IE\OE425QNS\Twitter-Search-Application-main) - Enhanced Cache Management Policy with Eviction Based on Least Recently Used (LRU) Algorithm for Performance Improvement in Cloud Storage,T. Senthilkumar, T. Muthuramalingam, Journal of Cloud Computing: Advances, Systems and Applications, 2017
5. [https://github.com/priyankanagasuri/694\_2024\_Team27\_Project](file:///C:\Users\91987\AppData\Local\Microsoft\Windows\INetCache\IE\OE425QNS\Twitter-Search-Application-main) - Github link of the project code uploaded.

# Contribution

|  |  |
| --- | --- |
| **Name** | **Work** |
| Nancy Soni - ns1583 | Data Downloading and Processing the Queries in MongoDB |
| Shruthi Nanditha Ganesh - sg2057 | Processing Pymongo Queries, Postgre SQL Queries |
| Nikitha Sunku Prashanth - ns1590 | WebApp using Flask and caching |
| Priyanka Nagasuri - pn296 | Search Engine modules for MongoDB and Postgre SQL, Search Queries |

Table 2 : Each team member's Contribution to the project.