FINAL PROJECT - 694

DATA MANAGEMENT FOR ADVANCED DATA SCIENCE APPLICATIONS

Abhinav Saxena*(as1431),* ***Atharva Bhusari****(ab2414),* ***Nikhil Mishra****(nm1116),* ***Urjit Patil****(up63)*

Github Ids: iamabhinav13, buzzi0204. nikhilm21, patilurjit

https://github.com/buzzi0204/694_2023_9_Final_Project

1. Introduction

Twitter is a popular social media platform that generates a lot of data. This data can be analyzed to gain insights into user behavior, popular topics, and trends. In this project, two popular data stores, namely MongoDB and MySQL, were used to store and query the Twitter data efficiently. MongoDB, a NoSQL database providing high performance and scalability, and MySQL, a relational database with robust transactional support, were leveraged to combine their strengths and improve the performance of data storage and retrieval.

To enable quick retrieval of data, indexing was implemented in both databases, and creating sorted lists of values for a particular field allowed the databases to efficiently search through large datasets, reducing the time required to retrieve data. The *createIndex()* method was used in MongoDB, while the *CREATE INDEX* statement was used in MySQL to create indexes on the required columns.

Multiple search functionalities were implemented to search for specific data in the Twitter dataset, such as keyword search, hashtag search, and user search. Specialized queries were created to search through specific fields in the Twitter dataset, making searching for particular data more manageable. To improve the performance of the search application, a cache was developed for frequently accessed data. By caching frequently accessed data, the number of queries to the database was reduced, improving the application's overall performance.

2. Dataset

A Python helper function was written to transform the data files into JSON format for convenient loading. Relevant data parameters were selected from the JSON data using the Twitter API documentation based on information and completeness and inserted into MongoDB and MySQL databases. This prepared the data for an efficient keyword, hashtag, and user searches. The transformed and selected data enabled quick and accurate retrieval of the queried information from the databases.

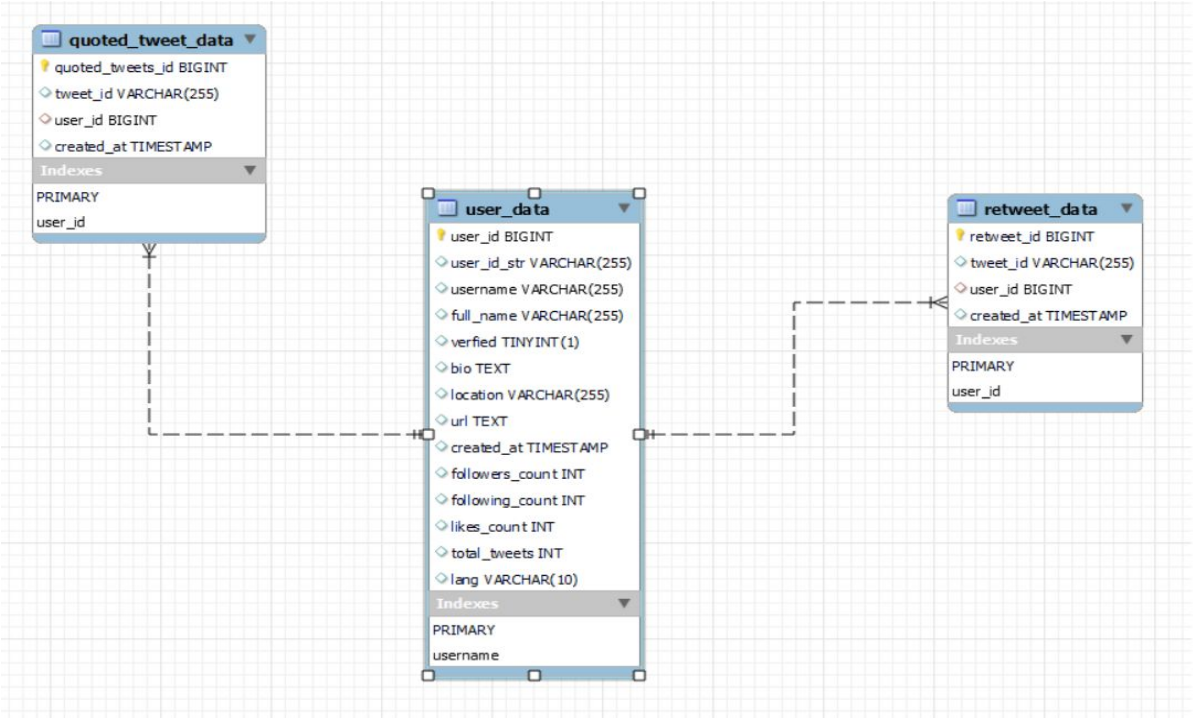


Fig 2.1: Relational Database Tables

Fig 2.2: Non-Relational Database Tables

The distribution of keys in the relational database tables and their primary keys and indexes can be observed in Figure 2.1, and Figure 2.2 displays the data that has been inserted into the non-relational database, where a new attribute called 'popularity' has been created as the sum of quote_count, reply_count, and favourite_count. This attribute was created for the ordering of the search results.

3. Data Loading

A meticulous analysis of the Twitter API documentation was performed to design an appropriate schema and determine the data to be utilized. Each key in the extracted data was scrutinized and cross-referenced with the documentation to understand its purpose and usage. Careful notes were taken on the use and meaning of each key-value pair to identify the significance and relevance of each one. This process helped make informed decisions on the data elements to be included in the final schema and the design of both the relational and non-relational schemas.

3.1 Preprocessing the data:

To make the data more accessible and amenable to processing in Python, dedicated Python scripts were written for specific tasks. This script converted the raw data provided into a structured JSON format, where the data was organized hierarchically and readable. This transformation enabled easy manipulation, querying, and analysis of the data using Python's extensive data processing capabilities.

3.2 Relational Database *(MySQL)*

MySQLdb library was utilized to execute SQL commands in Python. The functions in this library were used to create connections with the database and execute SQL commands to create tables and load the data.

```
db = MySQLdb.connect(host="localhost",
                     user="root",
                     passwd="root",
                     db="twitter_db")
cur = db.cursor()

################################################################################
####### Creating tables for tweets, retweets, quoted_tweets
################################################################################

create_table_query = "CREATE TABLE IF NOT EXISTS user_data(\
    user_id BIGINT PRIMARY KEY NOT NULL,\
    user_id_str VARCHAR(255),\
    username VARCHAR(255),\
    full_name VARCHAR(255),\
    verfied BOOLEAN,\
    bio TEXT,\
    location VARCHAR(255),\
    url TEXT(255),\
    created_at TIMESTAMP,\
    followers_count INTEGER,\
    following_count INTEGER,\
    likes_count INTEGER,\
    total_tweets INTEGER,\
    lang VARCHAR(10)\
);"
```

Fig 3.1: User data table creation

```
cur.execute(create_table_query)


create_table_query = "CREATE TABLE IF NOT EXISTS retweets(\
    retweet_id BIGINT PRIMARY KEY,\
    tweet_id VARCHAR(255),\
    user_id BIGINT,\
    created_at TIMESTAMP,\
    FOREIGN KEY (user_id) REFERENCES user_data(user_id)\
);"

cur.execute(create_table_query)

create_table_query = "CREATE TABLE IF NOT EXISTS quoted_tweets(\
    quoted_tweets_id BIGINT PRIMARY KEY,\
    tweet_id VARCHAR(255),\
    user_id BIGINT,\
    created_at TIMESTAMP,\
    FOREIGN KEY (user_id) REFERENCES user_data(user_id)\
);"

cur.execute(create_table_query)
```

Fig 3.2: Tweet and Retweet data table creation

The user_id, which acted as the primary key in the user_data table, was used as a foreign key in the retweets table and quoted_tweets table.

```
query_insert = "INSERT INTO user_data VALUES(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,\
%s,%s,%s);"

val_dict = {}
rt_val_dict = {}
qt_val_dict = {}

for i in range(len(data)):
    val_list = []

    for key in key_list:
        if key == 'created_at':
            data[i]['user'][key] = pd.to_datetime(data[i]['user'][key])
        val_list.append(data[i]['user'][key])

    val_dict[i] = tuple(val_list)

    # SQL Query
    try:
        cur.execute(query_insert, val_list)
    except Exception as e:
        print(e)
        pass

    if 'retweeted_status' in data[i]:
        rt_val_list = []
        for key in key_list:
            if key == 'created_at':
                data[i]['retweeted_status']['user'][key] = pd.to_datetime(data[i]['retweeted_status']['user'][key])
            rt_val_list.append(data[i]['retweeted_status']['user'][key])
    else:
        continue

    rt_val_dict[i] = tuple(rt_val_list)
```

Fig 3.3: Inserting data into user_data table

```
    try:
        cur.execute(query_insert, rt_val_list)
    except Exception as e:
        print(e)
        pass

    if 'quoted_status' in data[i]:
        qt_val_list = []
        for key in key_list:
            if key == 'created_at':
                data[i]['quoted_status']['user'][key] = pd.to_datetime(data[i]['quoted_status']['user'][key])
            qt_val_list.append(data[i]['quoted_status']['user'][key])
    else:
        continue

    qt_val_dict[i] = tuple(qt_val_list)

    try:
        cur.execute(query_insert, qt_val_list)
    except Exception as e:
        print(e)
        pass

db.commit()
```

Fig 3.4: Inserting data into user_data table (MySQL)

All the user data within the regular tweets, retweets, and quoted tweets was also inserted into the databases. Duplicates were also handled in the code, as only one unique user_id should exist. Using user_id as a foreign key in the retweets and quoted_retweets table, information about the user and his retweets and quoted tweets can be retrieved.

3.3 Non-Relational Database *(MongoDB)*

The MongoClient class from the Pymongo library connects Python to a MongoDB database. It was then used to read data from the JSON file, extract specific keys, and create MongoDB documents using a custom function. The completed documents were inserted into a MongoDB collection. The function also handled retweets and quoted tweets by inserting their data as separate documents.

```
########################################################################
########## Inserting tweets in MongoDB
########################################################################
try:
    conn = MongoClient()
    print("Connected successfully")
except:
    print("Could not connect to MongoDB")

# database
db = conn.twitter_db
collection = db.tweets_data

keys = ['id', 'id_str', 'text', 'created_at', 'truncated',
        'is_quote_status','qoute_count', 'reply_count', 'entities',
        'retweet_count', 'favorite_count', 'lang', 'timestamp_ms', 'geo']

def extract_source(input_string):
    sources = ['iPhone', 'Android', 'WebApp', 'Instagram']

    for source in sources:
        if source in input_string:
            extracted_source = source
            return extracted_source
```

Fig 3.5: Inserting data into Tweet data collection (MongoDB)

```
def mongo_insertor(index, keys):
    """

    Args:
        index ([type]): [description]
        keys ([type]): [description]

    Returns:
        [type]: [description]
    """
    obj = {
        "_id": index['id'],
        "source": extract_source(index['source'])
        }

    for key in keys:
        try:
            obj[key] = index[key]
        except:
            pass

    obj['user_id'] = index['user']['id']
    return obj
```

Fig 3.6: Inserting data into Tweet data collection (MongoDB)

```
for index in data:
    if 'retweeted_status' in index.keys():
        obj = mongo_insertor(index['retweeted_status'], keys)
        try:
            collection.insert_one(obj)
        except Exception as e:
            print(e)
            pass

    if 'quoted_status' in index.keys():
        obj = mongo_insertor(index['quoted_status'], keys)
        try:
            collection.insert_one(obj)
        except Exception as e:
            print(e)
            pass


    obj = mongo_insertor(index, keys)
    try:
        collection.insert_one(obj)
    except Exception as e:
        print(e)
        pass
```

Fig 3.7: Inserting data into Tweet data collection (MongoDB)

4. Caching

4.1 Implementation

The Cache class comprises two primary methods: get and set. The get method retrieves values from the cache if they exist in the cache, thus avoiding costly database queries. While if the requested key is not present in the cache, the get method returns None, requiring a database query to fetch the value. The set method stores key-value pairs in the cache and ensures that the cache size does not exceed the maximum size. It employs a Least Recently Used (LRU) algorithm to remove the oldest pair when the cache size surpasses the limit. The cache size plays a crucial role as it impacts the cache hit rate, which measures the number of times the application discovers the requested data in the cache. The application has set the cache size limit to ten. Figure 4.1 displays the cache initialization in this application.

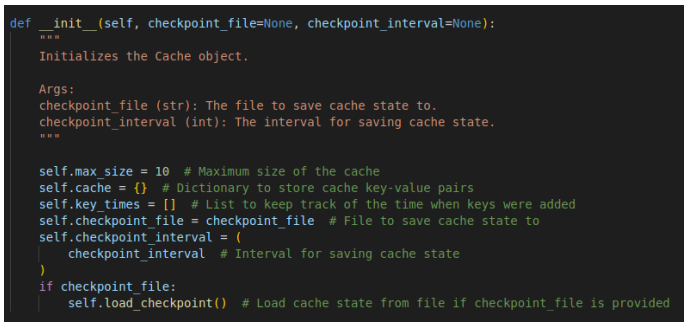


Fig 4.1: Initialization of Cache

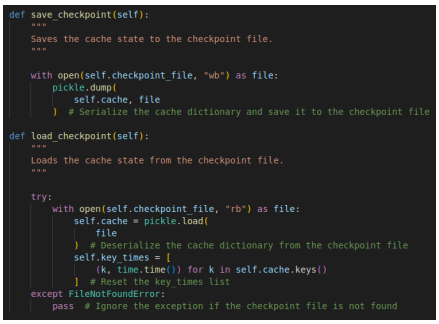


Fig 4.2: Checkpoint

4.2 Checkpoint & Eviction Strategy

The Cache class includes a checkpoint_file parameter to store and retrieve cache data from a file if it exists. It consists of the save_checkpoint and load_checkpoint methods. These methods allow the serialization and deserialization of the cache dictionary to and from the checkpoint file. In the save_checkpoint method, the cache dictionary is serialized using the pickle module and saved to the checkpoint file. On the other hand, the load_checkpoint method deserializes the cache dictionary from the checkpoint file using the pickle module. The method also resets the key_times list to the current time for all the keys in the cache dictionary. The load_checkpoint method ignores the exception and returns None if the checkpoint file is not

found. These methods work with the checkpoint_file parameter to ensure that the cache data is stored and retrieved correctly from the file. This implementation can be seen in Fig 4.2.

5. Search Application

5.1 Functions:

1. *get_username/full_name:*

The 'get_username' function takes a username as input and searches for the tweets from users whose username or full name matches the input. It retrieves data from a MySQL database using an SQL query and then retrieves tweets from a MongoDB database using the retrieved user ids. The resulting data is then combined and sorted by popularity. The function uses a cache to store the results of previous queries and avoid repeating the exact search. If the cache does not contain the requested data, the function executes the search and stores the result in the cache. If no tweets are found, the function prints a message indicating the same.

2. *get_hashtag:*

The 'get_hashtag' function searches for tweets that contain a given hashtag and returns information about those tweets and the users who posted them. It first checks if the requested data is already in the cache and returns it directly if yes. Otherwise, it queries MongoDB for all tweets that contain the hashtag using a regular expression. It extracts the necessary data into two data frames - one for user information and another for tweet information. It then uses the extracted user ids to query a MySQL database for user names and joins the two data frames to obtain the final result. The result is then sorted by a popularity metric based on the tweet's number of likes, retweets, followers, and quotes. Finally, the result is cached for future use.

3. *get_keyword:*

The 'get_keyword' function retrieves tweets from the MongoDB database that contain a specific keyword using a regular expression to search for the keyword within the tweet text. The function then extracts relevant information such as the user id, tweet id, and popularity metric

from the retrieved documents and performs an SQL query on a MySQL database to retrieve the users' usernames associated with the tweets. The extracted data is then merged and sorted based on the popularity metric. Finally, the function caches the result. If the requested keyword is not found in tweets, the function prints a message indicating the same.

4. *get_top_10_tweets/users:*

The function 'get_top_10_tweets' retrieves the top 10 tweets based on the popularity metric. It pulls data from MongoDB and MySQL, where it first retrieves the required tweet data from MongoDB and then retrieves the corresponding user data from MySQL. The results are then merged and returned as the final output. This function also utilizes caching to improve the application's performance by storing the results in the cache and retrieving them if the same query is repeated.

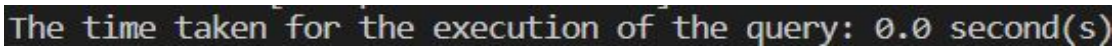
The function 'get_top_10_users' retrieves the top 10 users based on the popularity metric. It returns the result and stores it in the cache.

6. Results:

Implementing the above functions involved connecting them to the databases and user interface (UI) using the FlaskAPI, a Python microframework for building web applications. FlaskAPI provides a simple and flexible way to create RESTful APIs, allowing data communication and transfer between different software systems.

The implementation utilizes FlaskAPI to create endpoints for each function, allowing them to be called via HTTP requests. These endpoints were then integrated into the user interface, allowing users to interact with the data and retrieve the search results via a web browser.

(i) First Search (Before Caching)



(ii) Second Search (Caching occurred)

Fig 6.1: Execution of the keyword Query

Figure 6.1 depicts a search for the keyword 'covid' using the implemented functions. The search results are shown on the right side of the figure in the form of a table. The first query for the keyword took approximately 26 seconds, while subsequent queries for the same keyword took significantly less time. This demonstrated a successful implementation of the caching mechanism, which reduces the response time of the search operation.

The cache stores the search results for the first query, and the subsequent queries for the same keyword fetch the results from the cache instead of making a new query to the database, resulting in a faster response time. This demonstrates the effectiveness of the caching mechanism in reducing the load on the database and improving the application's performance.

7. Conclusion:

This project's combined use of MySQL and MongoDB offers a versatile and adaptable data management and processing solution. Integrating the two databases enables efficient retrieval and querying of data by leveraging their strengths. Moreover, the project's Cache class optimizes performance by storing frequently accessed data in memory and facilitating the saving and loading of cache data from a file, ensuring reliability in case of application crashes or restarts. This approach showcases the potential and flexibility of utilizing multiple database technologies for efficient data management and processing.

8. References:

1. https://developer.twitter.com/en/docs/twitter-api
2. https://flask.palletsprojects.com/en/2.3.x/
3. https://www.mongodb.com/docs/
4. Advanced Python: How To Implement Caching In Python Application | by Farhad Malik | FinTechExplained | Medium

9. Contributions:

1. Data Preparation and Loading - Atharva Bhusari, Nikhil Mishra, Urjit Patil
2. SQL implementation - Atharva Bhusari, Urjit Patil
3. MongoDB implementation - Nikhil Mishra
4. Search Function development - Atharva Bhusari, Nikhil Mishra, Urjit Patil
5. Cache implementation - Atharva Bhusari, Urjit Patil
6. Search Function class for UI - Urjit Patil
7. Flask UI implementation - Atharva Bhusari, Urjit Patil
8. Flask UI design and connection - Nikhil Mishra
9. Report and Presentation - Nikhil Mishra
10. Python Docstring - Abhinav Saxena