DATA 230: Lab 1 ChatGPT Analyzer

SJSU ChatGPT Tweet Analysis

Anusmriti Sikdar
Data Analytics
SJSU
San Jose, USA
anusmriti.sikdar@sjsu.edu

Akanksha Tyagi
Data Analytics
SJSU
San Jose, USA
akanksha.tyagi@sjsu.edu

Anjali Mishra
Data Analytics
SJSU
San Jose, USA
anjalihimanshu.ojha@sjsu.edu

Priyanka Bhyregowda
Data Analytics
SJSU
San Jose, USA
priyanka.bhyregowda@sjsu.edu

Sahana Thoravalli Prabhuswamy
Data Analytics
SJSU
San Jose, USA
sahana.thoravalliprabhuswamy@sjsu.edu

***Abstract*—**Recently, ChatGPT has been one of the most discussed topics. Twitter is a popular social media platform for people to connect and share their thoughts. Analyzing Twitter data provides deep insights into the way people think. By analyzing Twitter data on ChatGPT we can obtain valuable information. Hence, there is a need for a ChatGPT tweet analyzer application. A relational database provides many advantages like access, security, and integrity of the data compared to other earlier storage systems like files. Many users can concurrently access the database, the data is secured as authorization is required for data access. Any RDBMS promises data integrity with its ACID properties. MySQL is an open-source RDBMS that has all the required features for storing application-related data**.**

Keywords—ChatGPT, AWS RDS, RDBMS

I. Introduction

Analysis of tweet data has been a major research trend. ChatGPT which is an AI-based chatbot is found to bring revolutionary changes to the field of Artificial Intelligence and Machine Learning technology. Developing a cloud-native application is the modern approach for every software solution. AWS RDS provides a cloud-based RDBMS environment for any data storage needs. Many useful metrics can be obtained by analyzing tweet data related to ChatGPT. A client-server model application for the extraction and visualization of these results is one of the solutions. The ChatGPT analyzer app can be run on any machine with a command line interface, python, and access to the internet, which will be available in most of the systems.

A. Functional Requirements of our application from the given dataset:

1) Most active users from the dataset
2) Most Discussed tweet with respect to replies based on conversation ID in the given data
3) Most used language to tweet on chatGPT
4) Most Liked Tweet about ChatGPT
5) Most Used Media Type
6) Users with more number of retweets
7) Number of Tweets on 'ChatGPT'
8) Most retweeted tweet with the comments
9) Conversations with the most number of media shares.
10) Most retweeted tweet in each language
11) Top 10 Users who have more outsource for reference on the tweets
12) Number of tweets each day on chatGPT
13) Most viral tweets
14) Most popular hashtags used
15) Most used sources to tweet about ChatGPT
16) Most mentioned users in the tweets about ChatGPT
17) Handling the deletion of records in the tweets table gracefully
18) Restricting tweet length to 280
19) Auditing the deleted tweets

B. Limitations of our application from the given dataset:

1. Provides a fixed number of metrics
2. The metrics are based on a small set of data
3. Static queries

C. Deployment and Usage:

The database of the application is deployed in the cloud and the users need to have the required files in their system to run the application. The user can run the application in the command line and obtain the required metrics. The user can load his tweet dataset into the AWS RDS DB instance using the scripts. Only a DB Admin connects to the AWS RDS DB instance and makes changes.

II. FUNCTIONAL COMPONENT ANALYSIS

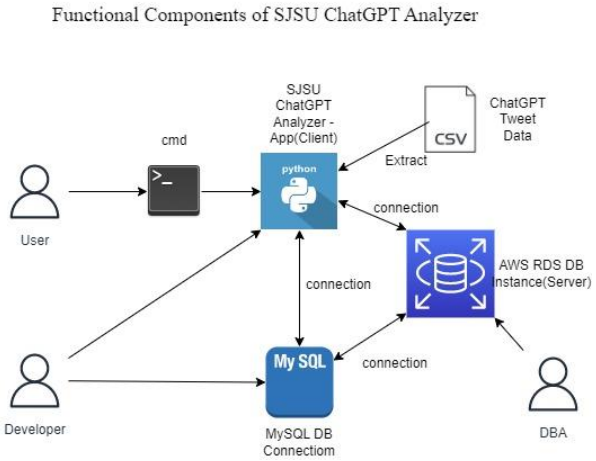
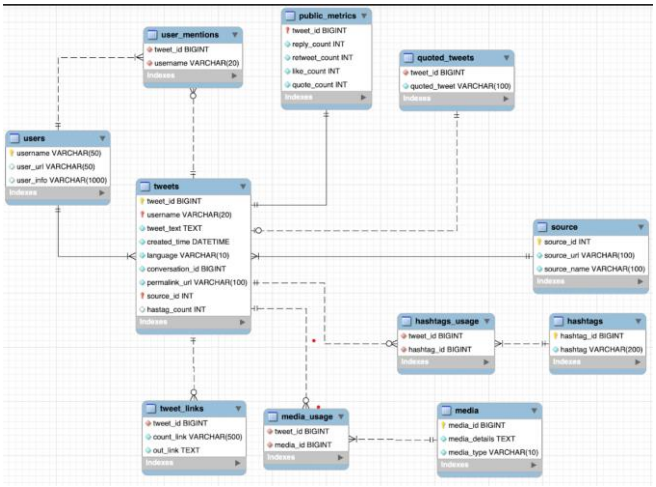


Fig. 1. Functional component analysis

SJSU ChatGPT Analyzer application has a client-server architecture. The Analyzer application acts as a client to the AWS RDS MySQL Instance which acts as a server. AWS RDS provides a cloud-based database environment that caters to the data need of our application. The raw data which is in the form of a .csv file is extracted using a python program and then cleaned and transformed into the required format and then loaded into the 'twitter' database in AWS RDS MySQL Instance. The application runs on a command line interface and the user of our application can select options to view a few of the interesting metrics that can be obtained from the application. The developer can connect to AWS RDS MySQL Instance endpoints either through a MySQL client or through the python application. Only a DB Admin can connect to the AWS RDS MySQL Instance and can do performance tuning and handles access and privileges of the DB.

III. ER DIAGRAM



For the given data set we indentfied the entities like "User", "Tweet", "Media", "HashTag", "Out_Links" and "Source" and their corresponding relations. The chatgpt data has the above mentioned components. There are following relationship we captured between entities as "User_Mentions", "Media_Usage" and "Out_Links". The main relation is "Tweets" relation which kinds of ties everything together. For the sources we see very high redundancy so keep it out in a different table which have only 843 diffenent entires so keeping in out in a different table minimize the table size also.

While designing the database we keep the read and write operations in mind. Like "Public_Metrics" which will be keep updated after every impression so keeping those things out from "tweets" table will minimize the unnecesasary updates to the "tweet" table. Once a tweet is saved there are no updates regarding that tweet.

To capture all the "hashtags" in the system, we keep them in a sepaerate table and there is relation "hashtag_usage" will keep record which tweet uses which hashtags. If someone delete their tweet, it will not delete the hashtag, because they may be used in other tweets. We used the similar design for the media also which is again reused many time.

"User_mentions" table will keep which user mentioned in which tweet, since it captures the only users who are already in the system, so there is relationship between "users" table and "user_mentions" table.

For the conversations, its associated with the parent tweet so we keep the conversationId within the "tweets" table and there are no other attributes associated with the converation whitin the dataset which require it to be a seperate entity.

Our database is 3NF normalized and we did not keep any paritial and transtive dependencies. Our design keep in mind the kind of use case we are solving and make those queries as fast as possible with very minimal joins.

IV. METHODS

Methods implemented in this Lab is summarized in this section.

A. *Stored Procedures*

- Stored Procedure 1: get_most_viral_tweet()

The stored procedure get_most_viral_tweet will return the 15 most viral tweets from the tweets table.

```
SQL Code:
        DROP PROCEDURE IF EXISTS get_most_viral_tweet;
        DELIMITER //
            CREATE PROCEDURE get_most_viral_tweet()
             BEGIN
                    SELECT
                            t.tweetId, t.username, t.tweetText, t.createdTime, t.lang,t.conversationId, t.permalinkUrl,
                            t.sourceId,
                            COALESCE(pm.retweetCount, 0) + COALESCE(pm.quoteCount, 0) +
                            COALESCE(pm.likeCount, 0) AS  viralCount
                     FROM
                            tweets t LEFT JOIN public_metrics pm ON t.tweetId = pm.tweetID
                     ORDER BY viralCount DESC
                      LIMIT 15;
                CALL get_most_viral_tweet();
```

Fig. 2. get_most_viral_tweet()

Explanation- This stored procedure first joins the tweets table with the public_metrics table using a left join, so that all tweets will be returned even if they have no associated public metrics yet. The COALESCE function is used to handle cases where the public metrics are NULL (i.e. if a tweet has no retweets, likes, or quotes). We then calculate the viral count by adding up the retweet, quote, and like counts. The results are sorted in descending order by viral count. LIMIT 15 will return 15 rows, which is the most viral tweets based on the number of retweets, quotes, and likes.

We can call the procedure with call statement and execute it like: CALL get_most_viral_tweet();

- Stored Procedure 2: get_hashtag_count()

This procedure named GET_HASHTAG_COUNT processes data from hash_tags table to count and return.

```
SQL Code:
  DROP PROCEDURE IF EXISTS GET_HASHTAG_COUNT;
  DELIMITER //
  CREATE PROCEDURE GET_HASHTAG_COUNT()
  BEGIN
       SELECT
                                    h.hashtag, count(tweetId) as
total_count
                          FROM
                                  hashtags h join hashtags_usage hu
using (hashtagId)
                          group by
                                    h.hashtagId
                          order by
                                     total_count desc;
```

Fig. 2. get_hashtag_count()

The stored procedure named GET_HASHTAG_COUNT processes data from hash_tags table to count and return the most frequently used hashtags. It joins the hashtags table with the hashtag_usage table using hashtag ID and retrieve the count of the Hashtags which is returned in DESCENDING order.

We can call the procedure with call statement and execute it like: CALL GET_HASHTAG_COUNT ();

- Stored Procedure 3: most_used_source()

This stored procedure most_used_source will return the top 10 most used sources for tweet.

```
SQL Code:

drop procedure if exists most_used_source;
DELIMITER //
      CREATE PROCEDURE most_used_source()
      BEGIN
            SELECT
                  s.SourceName as Source_Name, count(t.tweetId) as Total_Count
            FROM
                  tweets t JOIN `source` s using (sourceId)
        GROUP BY s.SourceName
        ORDER BY Total_Count desc
        LIMIT 10;
```

The stored procedure most_used_source selects and returns the top 10 most used sources for tweets and return the most used sources. The procedure starts by selecting the SourceName column from the source table and counting the number of tweets associated with each source using the COUNT() function. The tweets table is joined with the source table using the sourceId foreign key.

We can call the procedure with call statement and execute it like : CALL most_used_source ();

- Stored Procedure 4: most_mentioned_user()

The stored procedure most_mentioned_user aims to identify and return the top 10 most frequently mentioned Twitter users in the tweets table.

```
SQL Code:

drop procedure if exists most_mentioned_user ;
DELIMITER //
      CREATE PROCEDURE most_mentioned_user()
      BEGIN
            select
                  username, count(tweetId) as Total_Mentions
            from
                  user_mentions
            group by username
        order by Total_Mentions desc
        limit 10;
```

Fig. 4. most_mentioned_user()

The procedure selects the username column from the user_mentions table and counting the number of tweets in which each user is mentioned using the COUNT() function. The results are grouped by the username using the GROUP BY clause and sorted in descending order by the total count of mentions for each user using the ORDER BY clause. Finally, the top 10 results are returned using the LIMIT clause. We can call the procedure with call statement and execute it like: most_mentioned_user ();

B. *Triggers*

- Trigger 1: prepare_before_deleting_tweet

The trigger prepare_before_deleting_tweet is executed before a delete operation is performed on the tweets table for each row being deleted.

```
SQL Code:
DROP TRIGGER IF EXISTS prepare_before_deleting_tweet;
DELIMITER //
      CREATE TRIGGER prepare_before_deleting_tweet
      Before delete ON tweets
      FOR EACH ROW
      BEGIN
            -- Delete from user mentions table
        DELETE from user_mentions where tweetId = old.tweetId;
        -- Delte from hashtag usage table
        DELETE from hashtags_usage where tweetId = old.tweetId;
        -- Delete from media usage table
        DELETE from media_usage where tweetId = old.tweetId;
        -- Delete from media usage table
        DELETE from public_metrics where tweetId = old.tweetId;
        -- Delete from media usage table
        DELETE from tweet_links where tweetId = old.tweetId;
      END;//
DELIMITER ;;

delete from tweets where tweetId = '1617156404137295878';
select * from tweets where tweetId = 1617156404137295878;
```

This trigger will delete any rows from the user_mentions table that reference the tweet being deleted using the DELETE statement with a WHERE clause that matches the tweetId column to the old.tweetId value. Then, it deletes any rows from the hashtags_usage table, media_usage table, public_metrics table, and tweet_links table that reference the tweet being deleted in the same way. This trigger ensures that all associated data is deleted before the actual tweet is deleted, so that there are no foreign key constraints that prevent the deletion.

- Trigger 2: trigger_tweet_length

The trigger trigger_tweet_length is created and is set to execute BEFORE INSERT (for each row) on the

tweets table.

```
SQL Code:
Drop trigger if exists trigger_tweet_length;
DELIMITER //
CREATE TRIGGER trigger_tweet_length
BEFORE insert ON tweets
FOR EACH ROW
BEGIN
    IF LENGTH(NEW.tweettext) > 280 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Tweet text cannot be longer than 280 characters';
    END IF;
END;//
insert into tweets (tweetid,tweettext) values
(1610500572050200000,'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaajjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaajjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjaaaaaaaaaaaaaaaaaaaaaaaaaaaaajjjjjjjjjjjjjjjjjjjjj
jjjjjjjjjjjaaaaaaaaaaaaaaaaaaaaaaaaaaaaaajjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjaaaaaaaaaaaaaaaaaaaaaaa
aaaaaajjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjaaaaaaaaaaaaaaaaaaaaaaaaaaaajjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
jjj');
```

The trigger checks whether the new value of the tweet text column is longer than 280 characters. If the text is too long, the trigger raises an exception using the SIGNAL statement with a custom error message.

- Trigger 3: log_before_delete

This trigger is designed to log information about deleted tweets, which could be useful for auditing or tracking purposes.

```
SQL Code:

CREATE TABLE log (
    delete_timestamp datetime primary key,
    db_user varchar(50),
    tweet_id bigint not null,
    username varchar(50) not null
);
DELIMITER //
      CREATE TRIGGER log_before_delete
      Before delete ON tweets
      FOR EACH ROW
      BEGIN
            -- Delete from user mentions table
        DELETE from user_mentions where tweetId = old.tweetId;
        -- Delte from hashtag usage table
        DELETE from hashtag_usage where tweetId = old.tweetId;
        -- Delete from media usage table
        DELETE from media_usage where tweetId = old.tweetId;
        INSERT INTO log values (now(), user(), old.tweetid,old.username);
      END;//
DELIMITER ;

delete from tweets where username='group7';
```

C. *Query*

- Query 1 : Active Users tweeted in a given time in descending order

```
SELECT  username as active_users, COUNT(tweetid) as tweet_count
FROM tweets
GROUP BY username
ORDER BY COUNT(tweetid) DESC limit 10;
```

- Query 2: Most Discussed tweet with respect to replies based on conversation id in the given data

```
Select
        tweets.tweetid, tweets.tweettext, tweets.username, reply_to_tweet
from
        tweets join
        (
                SELECT
                        tweetId, username, conversationId,
                        COUNT(tweetId) over (partition by conversationId) as reply_to_tweet
                FROM tweets
                ORDER BY reply_to_tweet DESC
    ) converation using(tweetId)
where tweets.tweetId = tweets.conversationId;
```

- Query 3:Tweets on ChatGPT based on Used Language:

```
select
        lang, count(*)as total_count
from
        tweets
group by lang
order by total_count desc limit 10;
```

- Query 4: Most Liked Tweet about ChatGP:

```
select tweetid,username,tweettext,createdtime,likecount from tweets join public_metrics using
(tweetid)
order by likecount desc;
```

- Query 5: Most Used Media Type

```
select mediatype, count(*)as media_used from media
group by mediatype
order by media_used desc;
```

- Query 6: Users with more number of retweets

```
select username,tweetid,retweetcount from tweets join public_metrics
using (tweetid)  order by retweetcount desc;
```

- Query 7: number of Tweets on Chat GPT

```
select count(*) as number_of_tweets_on_Chatgpt from tweets
where tweettext like '%chatgpt%';
```

- Query 8: Most retweeted tweet with the comments

```
select username,tweetid,quotecount as retweet_with_comments
 from tweets join public_metrics
using (tweetid)  order by retweet_with_comments desc;
```

- Query 9: Conversations with most number of media share

```
select conversationid, mediatype, count(mediaid) as `number of media shared`
from tweets join media_usage using (tweetid) join media using (mediaid)
group by conversationid, mediatype
order by count(mediaid) desc
limit 10;
```

- Query 10 : Most retweeted tweet in each language

```
with langquery as (
select tweetid, `lang`, username, tweettext, retweetcount,
row_number() over (partition by `lang` order by retweetcount desc) as rownum
from tweets join public_metrics using (tweetid))
Select * from langquery
where rownum =1
order by retweetcount desc;
```

- Query 11: top 10 Users who have more outsource for reference on the tweet

```
select username, count(outlink) as `Number of links`
from tweets join tweet_links using (tweetid)
group by username
order by count(outlink) desc
limit 10;
```

- Query 12: top 10 Users who have more outsource for reference on the tweet

```
select date(createdtime) as Tweeted_date, count(tweetid) as tweet_count
from tweets
group by Tweeted_date;
```

-

D. *Table Structure*

- Table : hashtags

```
-- -----------------------------------------------------
-- Table `twitter`.`hashtags`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `twitter`.`hashtags` ;


CREATE TABLE IF NOT EXISTS `twitter`.`hashtags` (
  `hashtagId` BIGINT NOT NULL,
  `hashtag` VARCHAR(200) NOT NULL,
  PRIMARY KEY (`hashtagId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;
```

- Table: users

```
-- -----------------------------------------------------
-- Table `twitter`.`users`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `twitter`.`users` ;

CREATE TABLE IF NOT EXISTS `twitter`.`users` (
  `username` VARCHAR(50) NOT NULL,
  `User_url` VARCHAR(50) NULL DEFAULT NULL,
  `userInfo` VARCHAR(1000) NULL DEFAULT NULL,
  PRIMARY KEY (`username`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;

CREATE UNIQUE INDEX `username` ON `twitter`.`users` (`username` ASC) VISIBLE;
```

- Table: source

```
-- -----------------------------------------------------
-- Table `twitter`.`source`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `twitter`.`source` ;

CREATE TABLE IF NOT EXISTS `twitter`.`source` (
  `sourceId` BIGINT NOT NULL,
  `Source_url` VARCHAR(4000) NOT NULL,
  `sourceName` VARCHAR(100) NOT NULL,
  PRIMARY KEY (`sourceId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;
```

- Table : tweets

```
DROP TABLE IF EXISTS `twitter`.`tweets` ;
CREATE TABLE IF NOT EXISTS `twitter`.`tweets` (
  `tweetId` BIGINT NOT NULL,
  `username` VARCHAR(20) NOT NULL,
  `tweetText` TEXT NOT NULL,
  `createdTime` DATETIME NOT NULL,
  `lang` VARCHAR(10) NOT NULL,
  `conversationId` BIGINT NOT NULL,
  `permalinkUrl` VARCHAR(100) NOT NULL,
  `sourceId` BIGINT NOT NULL,
  `hastagCounts` INT NULL DEFAULT '0',
  PRIMARY KEY (`tweetId`, `username`, `sourceId`),
  CONSTRAINT `tweets_ibfk_1`
    FOREIGN KEY (`username`)
    REFERENCES `twitter`.`users` (`username`),
  CONSTRAINT `tweets_ibfk_2`
    FOREIGN KEY (`sourceId`)
    REFERENCES `twitter`.`source` (`sourceId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;
CREATE INDEX `username` ON `twitter`.`tweets` (`username` ASC) VISIBLE;
CREATE INDEX `sourceId` ON `twitter`.`tweets` (`sourceId` ASC) VISIBLE;
```

- Table: hashtag_usage

```
DROP TABLE IF EXISTS `twitter`.`hashtags_usage` ;

CREATE TABLE IF NOT EXISTS `twitter`.`hashtags_usage` (
  `tweetId` BIGINT NOT NULL,
  `hashtagId` BIGINT NOT NULL,
  CONSTRAINT `hashtags_usage_ibfk_1`
    FOREIGN KEY (`tweetId`)
    REFERENCES `twitter`.`tweets` (`tweetId`),
  CONSTRAINT `hashtags_usage_ibfk_2`
    FOREIGN KEY (`hashtagId`)
    REFERENCES `twitter`.`hashtags` (`hashtagId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;

CREATE INDEX `tweetId` ON `twitter`.`hashtags_usage` (`tweetId` ASC) VISIBLE;
```

- Table: media

```
DROP TABLE IF EXISTS `twitter`.`media` ;

CREATE TABLE IF NOT EXISTS `twitter`.`media` (
  `mediaId` BIGINT NOT NULL,
  `mediaDetails` TEXT NOT NULL,
  `mediaType` VARCHAR(10) NOT NULL,
  PRIMARY KEY (`mediaId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;
```

- Table: media_usage

```
DROP TABLE IF EXISTS `twitter`.`media_usage` ;

CREATE TABLE IF NOT EXISTS `twitter`.`media_usage` (
  `tweetId` BIGINT NOT NULL,
  `mediaId` BIGINT NOT NULL,
  CONSTRAINT `media_usage_ibfk_1`
    FOREIGN KEY (`tweetId`)
    REFERENCES `twitter`.`tweets` (`tweetId`),
  CONSTRAINT `media_usage_ibfk_2`
    FOREIGN KEY (`mediaId`)
    REFERENCES `twitter`.`media` (`mediaId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;

CREATE INDEX `tweetId` ON `twitter`.`media_usage` (`tweetId` ASC) VISIBLE;

CREATE INDEX `mediaId` ON `twitter`.`media_usage` (`mediaId` ASC) VISIBLE;
```

- Table: public_metrics

```
DROP TABLE IF EXISTS `twitter`.`public_metrics` ;

CREATE TABLE IF NOT EXISTS `twitter`.`public_metrics` (
  `tweetId` BIGINT NOT NULL,
  `replyCount` INT NOT NULL DEFAULT '0',
  `retweetCount` INT NOT NULL DEFAULT '0',
  `likeCount` INT NOT NULL DEFAULT '0',
  `quoteCount` INT NOT NULL DEFAULT '0',
  PRIMARY KEY (`tweetId`),
  CONSTRAINT `public_metrics_ibfk_1`
    FOREIGN KEY (`tweetId`)
    REFERENCES `twitter`.`tweets` (`tweetId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;
```

- Table : quoted_tweets

```
DROP TABLE IF EXISTS `twitter`.`quoted_tweets` ;

CREATE TABLE IF NOT EXISTS `twitter`.`quoted_tweets` (
  `tweetId` BIGINT NOT NULL,
  `quotedTweet` VARCHAR(100) NOT NULL,
  CONSTRAINT `quotedTweets_ibfk_1`
    FOREIGN KEY (`tweetId`)
    REFERENCES `twitter`.`tweets` (`tweetId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;

CREATE INDEX `tweetId` ON `twitter`.`quoted_tweets` (`tweetId` ASC) VISIBLE;
```

- Table : tweet_link

```
DROP TABLE IF EXISTS `twitter`.`tweet_links` ;

CREATE TABLE IF NOT EXISTS `twitter`.`tweet_links` (
  `tweetId` BIGINT NOT NULL,
  `countLink` VARCHAR(500) NOT NULL,
  `outLink` TEXT NOT NULL,
  CONSTRAINT `tweet_links_ibfk_1`
    FOREIGN KEY (`tweetId`)
    REFERENCES `twitter`.`tweets` (`tweetId`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8mb4
COLLATE = utf8mb4_0900_ai_ci;

CREATE INDEX `tweetId` ON `twitter`.`tweet_links` (`tweetId` ASC) VISIBLE;
```

E. *Access Priviledge*

RDBMS systems provide concurrent access to data and data security. A DBA or database administrator can create users and grant access to various levels of access to the users. In the application, there are three levels of access privileges. We have totally three users for our DB:

1) *Admin - Has all access to all DBs*
2) *Developer - Has all access to twitter db*
3) *Analyst1 - Has SELECT access on twitter db*

```
sql = '''SELECT user FROM mysql. user'''
cursor.execute(sql)
cursor.fetchall()
```

```
(('Analyst1',),
 ('Developer',),
 ('admin',),
 ('mysql.infoschema',),
 ('mysql.session',),
 ('mysql.sys',),
 ('rdsadmin',))
```

```
15] sql = '''SHOW GRANTS FOR Developer;'''
cursor.execute(sql)
cursor.fetchall()
```

```
(('GRANT USAGE ON *.* TO `Developer`@`%`',),
 ('GRANT ALL PRIVILEGES ON `twitter`.* TO `Developer`@`%`',))
```

```
sql = '''SHOW GRANTS FOR Analyst1;'''
cursor.execute(sql)
cursor.fetchall()
```

```
(('GRANT USAGE ON *.* TO `Analyst1`@`%`',),
 ('GRANT SELECT ON `twitter`.* TO `Analyst1`@`%`',))
```

F. *Logging*

Log files are stored in the form of tables, error logs are enabled by default and slow query logs and general logs are enabled for the general maintenance of the database.

1) *Slow Query log-*

```
Select * from mysql.slow_log
Select * from mysql.general_log
```

start_time	user_host	query_time	lock_time	rows_sent	rows_examined	db	last_insert_id	insert_id	server_id	sql_text	thread_id
2023-03-20 02:02:25.422633	admin[admin] @ 99-102-93-68.lightspeed.sntcc...	00:02:42.715684	00:00:00.000005	10	87577	twitter	0	0	1548915789	BLOB	15
2023-03-20 18:35:58.632031	admin[admin] @ 99-102-93-68.lightspeed.sntcc...	00:00:22.324997	00:00:00.000005	35246	100004	twitter	0	0	1548915789	BLOB	253
2023-03-20 22:31:56.001976	admin[admin] @ 99-102-93-68.lightspeed.sntcc...	00:00:25.801182	00:00:00.000000	35246	100004	twitter	0	0	1548915789	BLOB	14

2) *General Query Log-*

```
Select * from mysql.slow_log
Select * from mysql.general_log
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

event_time	user_host	thread_id	server_id	command_type	argument
2023-03-20 01:37:15.521248	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB
2023-03-20 01:37:15.522574	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB
2023-03-20 01:37:15.523263	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB
2023-03-20 01:37:15.523686	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB
2023-03-20 01:37:16.396553	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:17.396665	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:18.396768	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:19.396852	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:20.396903	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:21.396978	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:22.397098	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:23.202854	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Query	BLOB
2023-03-20 01:37:23.397372	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:24.397240	rdsadmin[rdsadmin] @ localhost []	10	1548915789	Execute	BLOB
2023-03-20 01:37:24.600777	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB
2023-03-20 01:37:24.601420	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB
2023-03-20 01:37:24.602256	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB
2023-03-20 01:37:24.602741	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB
2023-03-20 01:37:24.604040	rdsadmin[rdsadmin] @ localhost [127.0.0.1]	8	1548915789	Query	BLOB

eneral_log 35 ×

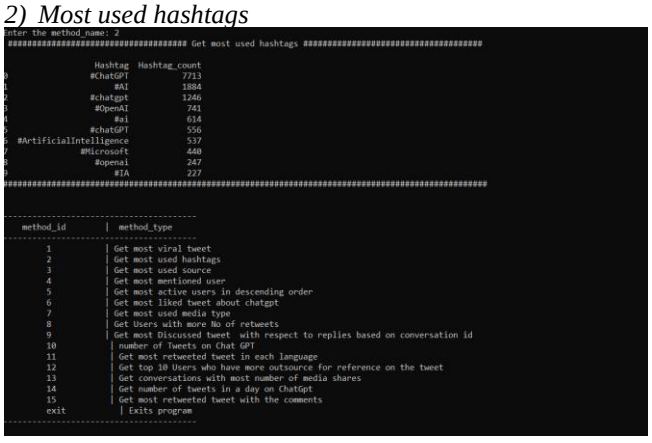
V. RESULTS

A. *User Interface output:*

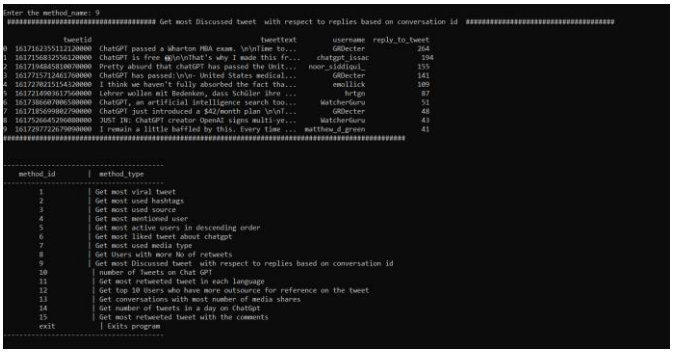
User Interface will have a list of options that are discussed in the functional requirements. A few sample output is shown here:

1) *Most viral tweets*

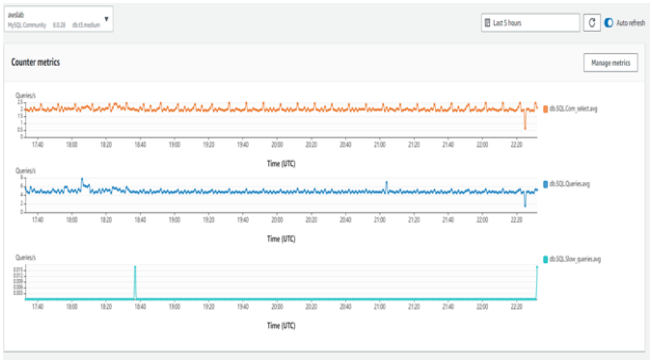
2) *Most used hashtags*

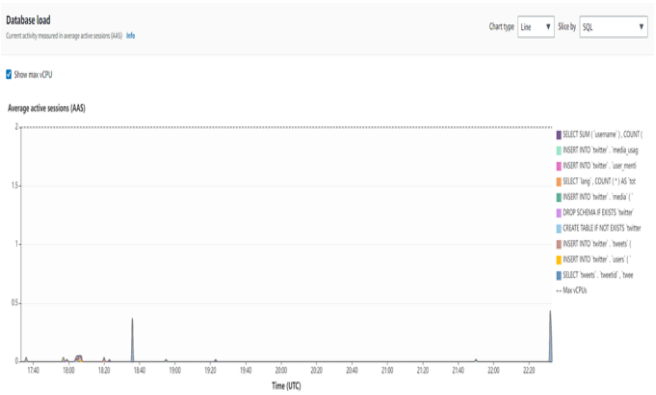


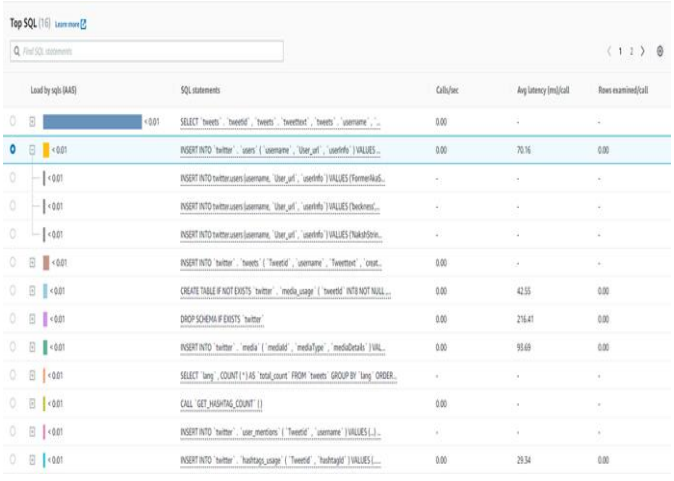
3) Most discussed tweet wrt replies based on conversationID



B. Performance Insights Output:

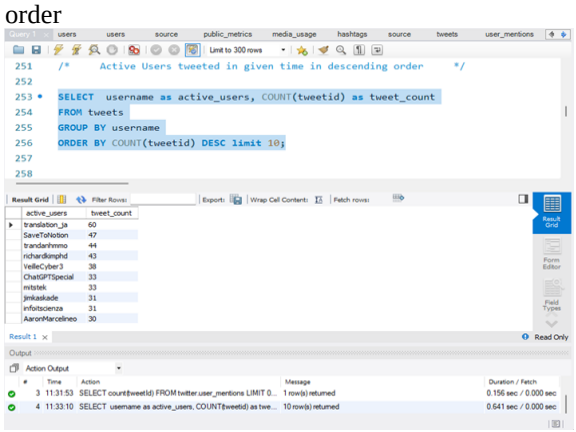




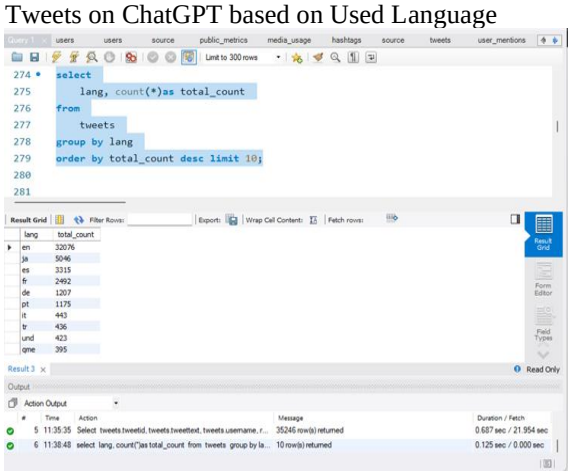


C. Output from queries, stored procedure, and triggers

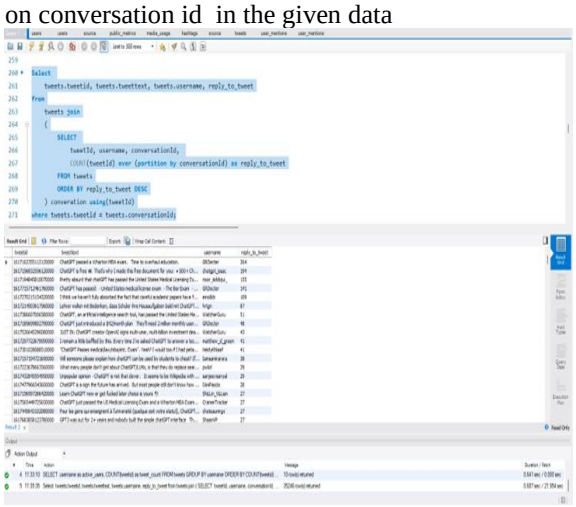
1. Active Users tweeted in given time in descending order



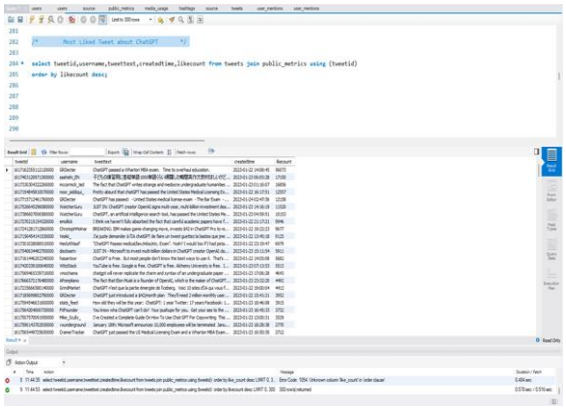
2. Tweets on ChatGPT based on Used Language



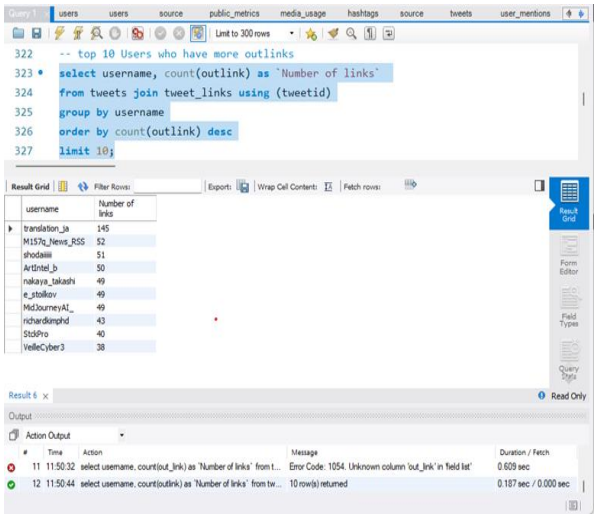
3. Most Discussed tweet with respect to replies based on conversation id in the given data



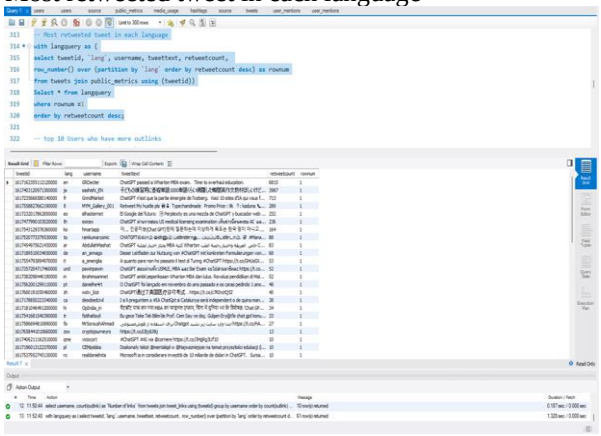
4. Most Liked Tweet about ChatGPT



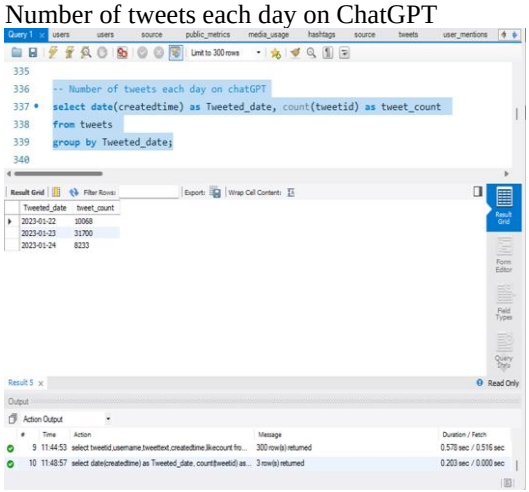
5. Top 10 Users who have more outsource for reference on the tweet



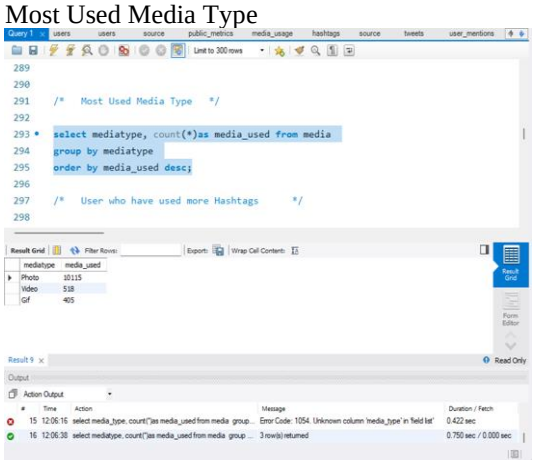
6. Most retweeted tweet in each language



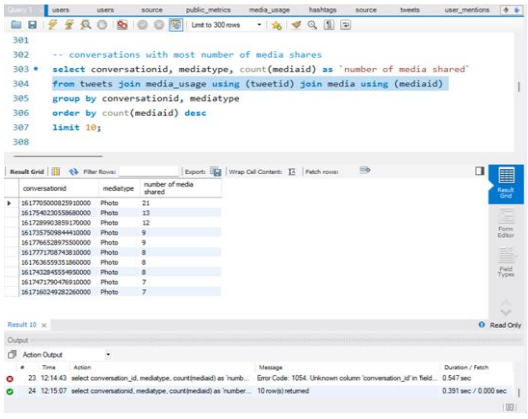
7. Number of tweets each day on ChatGPT



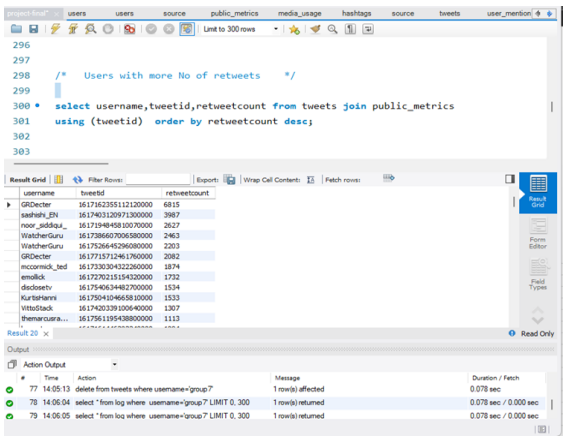
8. Most Used Media Type



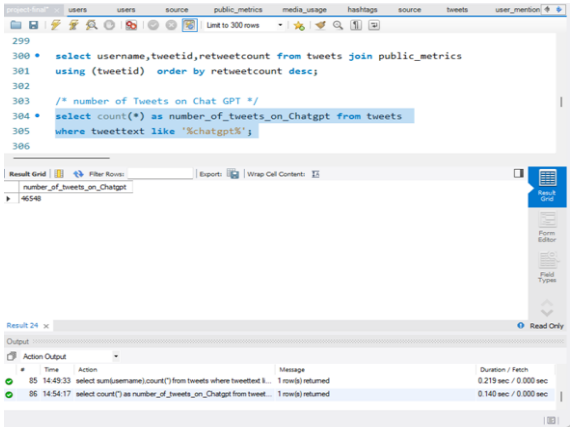
9. Conversations with Most Number of Media Shares



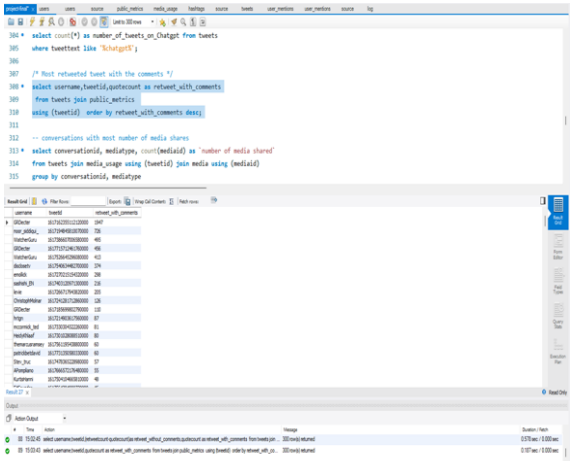
10. Users with higher retweets



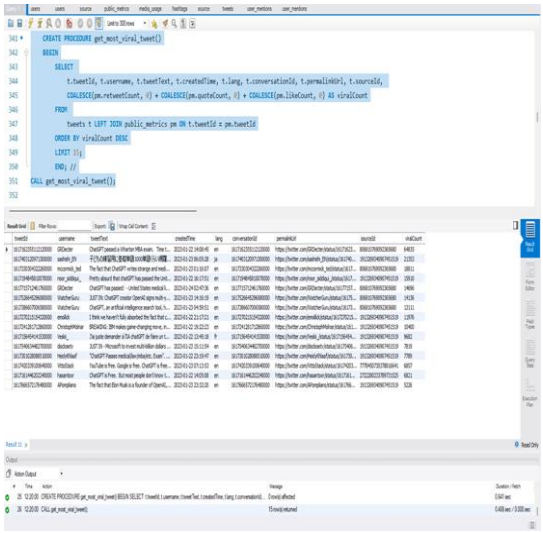
11. Number of Tweets on Chat GPT



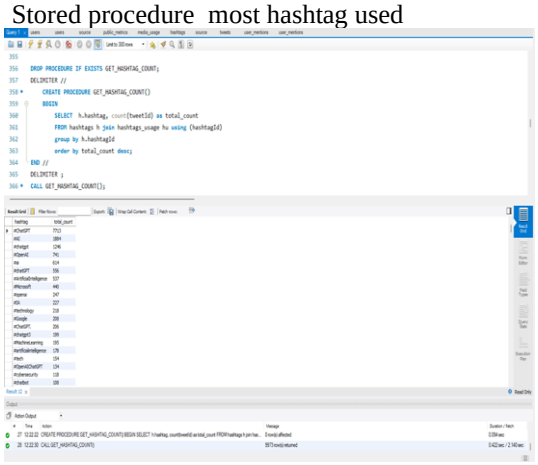
12. Most retweeted tweet with the comments



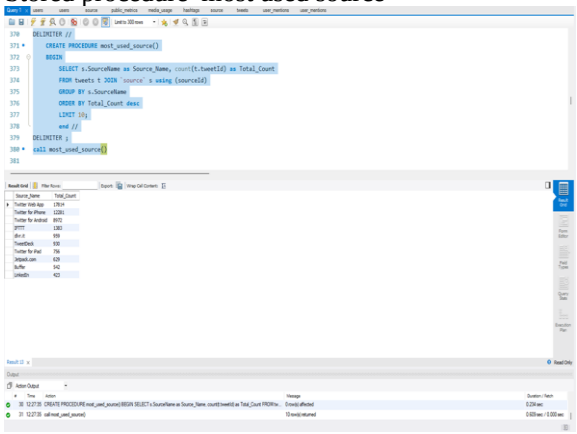
13. Stored Procedure Most Viral Tweet



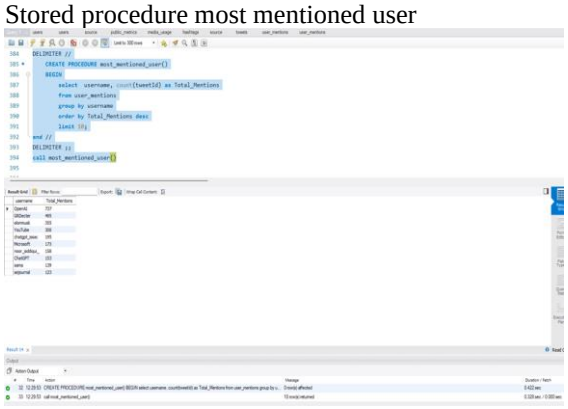
14. Stored procedure most hashtag used

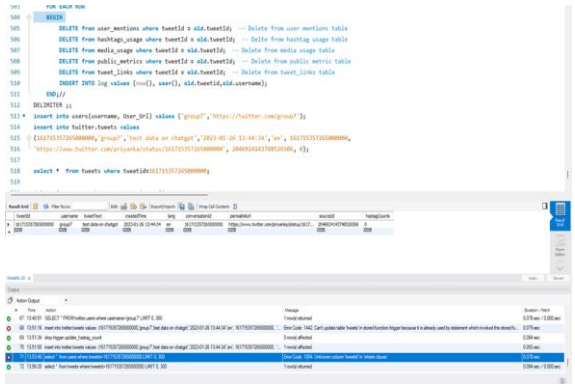


15. Stored procedure most used source



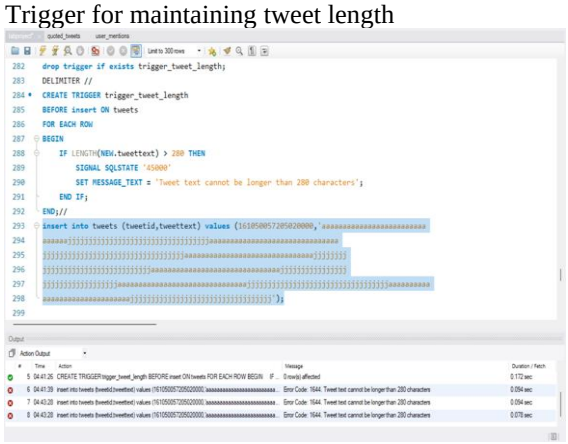
16. Stored procedure most mentioned user

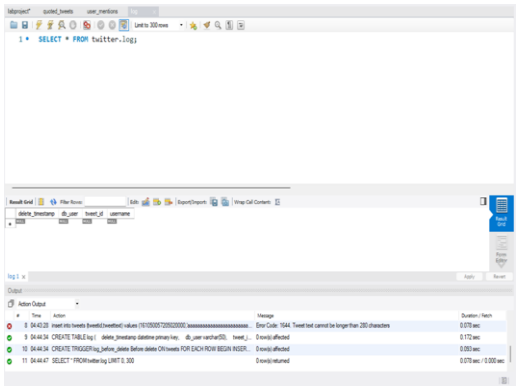




Log Table Before data was deleted

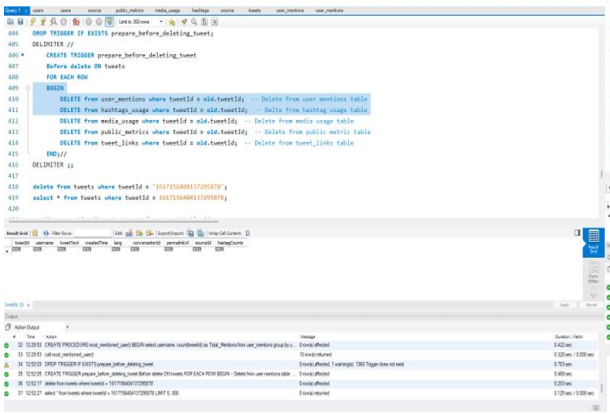
17. Trigger for maintaining tweet length

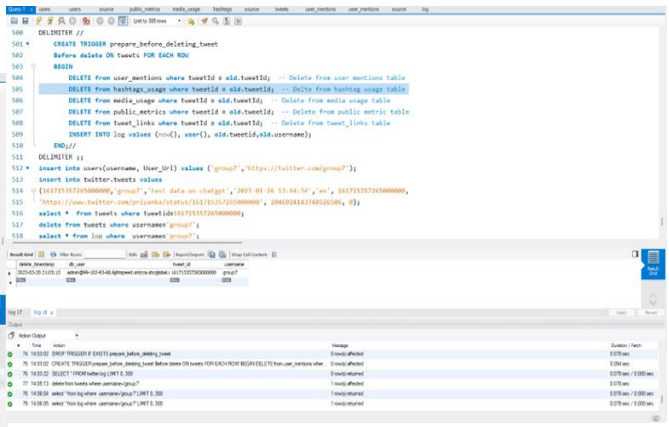




Log Table After data was deleted

18. Trigger for before delete





19. Logging Before Delete Trigger

Before delete

VI. CONCLUSION

ChatGPT App is a cool app that provides many metrics, but a user can only access static queries, in the future it can be made into a dynamic query application. When more data is available the data loading part can be automated to execute at the scheduled time and push data into the cloud AWS RDS Database Instance on a regular basis. The next round of analysis can involve machine learning algorithms to make predictions and perform sentiment analysis on the tweet text.