CS5855 Final Coursework

- **Ramachandran Murugesh**

Design Flaws of the twitter table (bad_giant_table):

As we can see from the structure of the bad_giant_table, the table is a result of merging more than 2 table for the sole purpose of data visibility: it helps with analysing the data without having much trouble to analyse more than 1 table to draw a result. This kind of data is dirty since it has a lot of redundant data, a single primary key and empty values in a few columns of the table. These types of data are purely used for visualization, i.e. to get a broader view of how the data is, i.e. the resultant of joining more than 2 tables. Tables like this (bad_giant_table) can't be used for query processing, since these are poorly normalized and unoptimized. For a small dataset, these might work perfectly, but when it comes to dealing with a large dataset, it has a negative impact on the performance of the database.

- Having a clear idea of the purpose of the data:
  The purpose of switching from file-system to a database system is to efficiently store and retrieve data. To do this effectively, a database designer must know what the data is going to represent, how the data is going to be collected and the volume of the data we are going to dealing with. Because the volume of the data and purpose of the data is going to directly affect the levels of normalization, database design and in general the implementation of the entire database system. None of these pre-requisites have been kept in mind while creating the bad_giant_table.
- Poor normalization:
  Every table in a database should be normalized to the third normal form 3NF, since this will have a direct impact on the basic *CRUD* operations – *Create*, *Read*, *Update* and *Delete* operations in a database. Although the bad_giant_table is in 1NF (all the column values are atomic, no columns in the bad_giant_table has a column which holds more than one value), the table has a lot of redundant data which have to normalized. The following are the flaws in the bad_giant_table:
  i) The table is not in 2NF, this can be explained with an example. Consider the following record-
     Tweet_id – 260213890372211000, user_id – 502387030
     This user has only tweeted once which corresponds the tweet_id above. When we delete this record, we not only lose the details of the tweet, we lose the information pertaining to the particular user.
  ii) The column user_id column is redundant; a particular user can have 'n' number of tweets corresponding to him/her.
  iii) The columns in_reply_to_screen_name and in_reply_to_user_id is directly dependent on the column in_reply_to_status_id which is already mapped to a tweet. Hence, the screen_name and user_id can be retrieved that way.
  iv) Some of the Hashtag columns 1-6, have all null values from 1 through 6. If there aren't any hashtags in a tweet, there is no reason to store null values in it.
  v) Functional dependency to represent the bad_giant_table:
     Since tweet_id is unique, it can independently identify each column of the table.
     Tweet_id -> created_at
     Tweet_id -> text varchar
     Tweet_id -> in_reply_to_screen_name,
     Tweet_id -> in_reply_to_status_id,

Tweet_id -> in_reply_to_user_id,
Tweet_id -> retweet_count,
Tweet_id -> tweet_source,
Tweet_id -> retweet_of_tweet_id,
Tweet_id -> hashtag1,
Tweet_id -> hashtag2,
Tweet_id -> hashtag3,
Tweet_id -> hashtag4,
Tweet_id -> hashtag5,
Tweet_id -> hashtag6,
Tweet_id -> user_id,
Tweet_id -> user_name,
Tweet_id -> user_screen_name,
Tweet_id -> user_location,
Tweet_id -> user_utc_offset,
Tweet_id -> user_time_zone,
Tweet_id -> user_followers_count,
Tweet_id -> user_friends_count,
Tweet_id -> user_lang,
Tweet_id -> user_description,
Tweet_id -> user_status_count,
Tweet_id -> user_created_at

- Bad Referential Integrity Constraints:
  This table (bad_giant_table) has only one Primary Key constraint. If very few constraints or no constraints are implemented from the database design stage, the data processing completely relies on business logic.
- Poor Indexing:
  As the number of records in the table increase, the time taken to process each query increases exponentially. Having as many columns as we have in the bad_giant_table will result in poor retrieving times on *SELECT* statements. The next thing on our mind is to index every column in the table, but this will have an adverse effect on operations like *CREATE*, *UPDATE* and *DELETE* statements.
- Failing to use Database Engine Features:
  The database engine offers features such as creation of views, indexing, aggregator functions, transactions, procedures and triggers. In the bad_giant_table, we have a column by the name user_status_count which holds the total number of tweets of a particular user(including retweets). This result can be easily obtained by using the aggregator function *SUM*.

These are the design flaws that directly affect the bad_giant_table.

Improved and Normalized design for Twitter Table (bad_giant_table):

- Normalization of the table:
  Since the table has no columns that have a group of values, i.e. the table satisfies the property of atomicity. This means that the table is in 1NF.
  Since the table is not in 2NF as discussed from the flaws above, splitting the tables into smaller independent tables is a solution.

  bad_giant_table(created_at, text, tweet_id PRIMARY KEY, in_reply_to_screen_name, in_reply_to_status_id, in_reply_to_user_id, retweet_count, tweet_source, retweet_of_tweet_id, hashtag1, hashtag2, hashtag3, hashtag4, hashtag5, hashtag6, user_id, user_name, user_screen_name, user_location, user_utc_offset, user_time_zone, user_followers_count, user_friends_count, user_lang, user_description, user_status_count, user_created_at)

  i) Splitting the user columns from the bad_giant_table:

     USER_TABLE(user_id, user_name, user_screen_name)
     User_id -> user_name, user_screen_name (Each user_id will have a user_name and user_screen_name, combination of user_name and user_screen_name is going to yield the user_id.)

  ii) Splitting the region and language settings from the bad_giant_table:

     LOCALE_SETTINGS(user_id, user_lang, user_location, user_utc_offset, user_time_zone)
     user_id -> user_lang, user_location, user_utc_offset, user_time_zone (Each user_id will have a user_lang, user_location, user_utc_offset, user_time_zone)
     In this table user_id is a foreign key referencing the user_id in *USER_TABLE*. User_id is also added with a *UNIQUE* constraint to avoid duplicate values in this table. Because every user is going to have only one user_lang, user_location, user_utc_offset and user_time_zone.

  iii) Splitting the profile of the user from the bad_giant_table:

     USER_PROFILE(user_id, user_followers_count, user_friends_count, user_description, user_status_count, user_created_at)
     user_id -> user_followers_count, user_friends_count, user_description, user_status_count, user_created_at
     In this table user_id is a foreign key referencing the user_id in *USER_TABLE*. User_id is also added with a *UNIQUE* constraint to avoid duplicate values in this table.

  iv) Splitting the reply related fields from the bad_giant_table:

     REPLY(in_reply_to_user_id, in_reply_to_screen_name, in_reply_to_status_id)
     in_reply_to_user_id -> in_reply_to_screen_name, in_reply_to_status_id
     (in_reply_to_user_id can uniquely identify in_reply_to_screen_name, in_reply_to_status_id)

Since multiple people can reply to a person's tweet, the columns in_reply_to_screen_name and in_reply_to_user_id can be stored once in the *REPLY* table and retrieved easily.

v) Splitting the hashtag fields from the bad_giant_table:

HASHTAG(tweet_id, hashtag1, hashtag2, hashtag3, hashtag4, hashtag5, hashtag6)
Tweet_id -> hashtag1, hashtag2, hashtag3, hashtag4, hashtag5, hashtag6
Each tweet_id corresponds to hashtags from 1 to 6. The purpose of this table to limit the number of empty/null values in hashtag columns 1-6. If there exists no hashtags for a particular tweet_id, there won't exist a record in hashtag table.

vi) ***TWEET***(tweet_id, text, user_id, retweet_of_tweet_id, retweet_count, in_reply_to_status_id, tweet_source, created_at)
Tweet_id -> text, user_id, retweet_of_tweet_id, retweet_count, in_reply_to_status_id, tweet_source, created_at
Each tweet_id can uniquely identify the columns above. In this table in_reply_to_status_id is a foreign key to *REPLY* table and user_id is a foreign key to *USER_TABLE*.

Note: The tables *LOCALE_SETTINGS* and *USER_PROFILE* are created to reduce the number of columns for each user in the *USER_TABLE*. The redundancy with user_lang could not be reduced due to the inconsistency in the data. If each user_utc_offset and user_time_zone can uniquely identify a user_lang, a separate table could have been drawn from this. Apart from these, the table has been normalized to the level 3NF.

Query Results:

Tweets, users and languages:

1) Total Number of tweets in total: (Since each tweet_id is unique in *TWEET* table)

```
Terminal - mjac207@cim-ts-node-02:~
File Edit View Terminal Tabs Help
CS5855J/mjac207=> SELECT COUNT(*) AS TWEETS_COUNT
CS5855J/mjac207-> FROM TWEET;
 tweets_count
--------------
       110574
(1 row)

CS5855J/mjac207=>
```

2) For every language the number of tweets in that language: Joining *TWEET* table with *LOCALE_SETTINGS ON* user_id and applying group by function to calculate the count.

```
File Edit View Terminal Tabs Help
 user_lang | tweets_count_per_language
-----------+--------------------------
 ar        |                     1405
 ca        |                        7
 cs        |                        2
 de        |                       75
 el        |                        2
 en        |                    74077
 es        |                    17910
 eu        |                        1
 fi        |                        1
 fil       |                        9
 fr        |                      231
 hu        |                        1
 id        |                     1423
 it        |                       26
 ja        |                     9861
 ko        |                      691
 msa       |                       14
 nl        |                       61
 no        |                        1
 pl        |                        4
 pt        |                     3977
 ru        |                      396
 sv        |                        2
 th        |                      233
 tr        |                      123
 ur        |                        1
 zh-cn     |                       26
 zh-tw     |                       14
(28 rows)
```

Retweeting habits:

1) Fraction of tweets that are retweets. (Calculating the count of retweet_of_tweet_id and dividing it by the total number of tweets present in the table)

```
Terminal - mjac207@cim-ts-node-02:~
File  Edit  View  Terminal  Tabs  Help
CS5855J/mjac207=> SELECT
CS5855J/mjac207-> CAST((SELECT COUNT(retweet_of_tweet_id) FROM TWEET
CS5855J/mjac207(> WHERE retweet_of_tweet_id IS NOT NULL) AS float)/CAST((SELECT
COUNT(*) FROM TWEET) AS float) AS fraction_of_tweets_that_are_retweets;
 fraction_of_tweets_that_are_retweets
--------------------------------------
                  0.19387921211134626
(1 row)

CS5855J/mjac207=> 
```

2) Computing average number of retweets per tweet. (Assumption: Calculating the average of column – retweet_count where retweet_of_tweet_id is NOT NULL)

```
Terminal - mjac207@cim-ts-node-02:~
File  Edit  View  Terminal  Tabs  Help
CS5855J/mjac207=> SELECT AVG(retweet_count) AS average_retweets_per_tweet FROM T
WEET
CS5855J/mjac207-> WHERE retweet_of_tweet_id IS NOT NULL;
 average_retweets_per_tweet
----------------------------
      400.2789439313368784
(1 row)

CS5855J/mjac207=> 
```

3) Computing fraction of tweets that are never retweeted. (Since we already calculated the fraction of tweets that are retweets, we subtract 1 from it to get the fraction of tweets that are never retweeted.

```
Terminal - mjac207@cim-ts-node-02:~
File Edit View Terminal Tabs Help
CS5855J/mjac207=> SELECT
CS5855J/mjac207-> (1-CAST((SELECT COUNT(retweet_of_tweet_id) FROM TWEET
CS5855J/mjac207(> WHERE retweet_of_tweet_id IS NOT NULL) AS float)/CAST((SELECT
COUNT(*) FROM TWEET) AS float)) AS fraction_of_tweets_not_retweeted;
 fraction_of_tweets_not_retweeted
----------------------------------
               0.8061207878886537
(1 row)

CS5855J/mjac207=>
```

4) Computing fraction of tweets which are retweeted fewer times than the average number of retweets. (Assumption: Calculating the number of retweets which have a retweet_count that is less than the average number of retweets – already calculated above. This is divided by the total number of tweets. The distribution of retweet_count is not the best way to derive the average number of retweets per tweet. Since popular users have a very high value of retweet_count which will affect the distribution of the retweet_count.

```
Terminal - mjac207@cim-ts-node-02:~
File Edit View Terminal Tabs Help
CS5855J/mjac207=> SELECT
CS5855J/mjac207-> CAST((SELECT COUNT(retweet_of_tweet_id) FROM TWEET
CS5855J/mjac207(> WHERE retweet_count < (SELECT AVG(retweet_count) AS average_re
tweets_per_tweet FROM TWEET
CS5855J/mjac207(> WHERE retweet_of_tweet_id IS NOT NULL) AND retweet_of_tweet_id
 IS NOT NULL) AS float)/CAST((SELECT COUNT(*) FROM TWEET) AS float) AS fraction_
of_fewer_retweets;
 fraction_of_fewer_retweets
----------------------------
        0.16928030097491273
(1 row)

CS5855J/mjac207=>
```

Hashtags:

1) Computing the number of distinct hashtags in the tweets.

```
CS5855J/mjac207=> SELECT count(*) AS COUNT_DISTINCT_HASTAGS FROM
CS5855J/mjac207-> ((select hashtag1 AS distinct_hashtag FROM HASHTAG WHERE hasht
ag1 IS NOT NULL)
CS5855J/mjac207(> UNION
CS5855J/mjac207(> (select hashtag2 AS distinct_hashtag FROM HASHTAG WHERE hashta
g2 IS NOT NULL)
CS5855J/mjac207(> UNION
CS5855J/mjac207(> (select hashtag3 AS distinct_hashtag FROM HASHTAG WHERE hashta
g3 IS NOT NULL)
CS5855J/mjac207(> UNION
CS5855J/mjac207(> (SELECT hashtag4 AS distinct_hashtag FROM HASHTAG WHERE hashta
g4 IS NOT NULL)
CS5855J/mjac207(> UNION
CS5855J/mjac207(> (SELECT hashtag5 AS distinct_hashtag FROM HASHTAG WHERE hashta
g5 IS NOT NULL)
CS5855J/mjac207(> UNION
CS5855J/mjac207(> (SELECT hashtag6 AS distinct_hashtag FROM HASHTAG WHERE hashta
g6 IS NOT NULL))count_distinct;
 count_distinct_hastags
------------------------
                  10158
(1 row)

CS5855J/mjac207=>
```

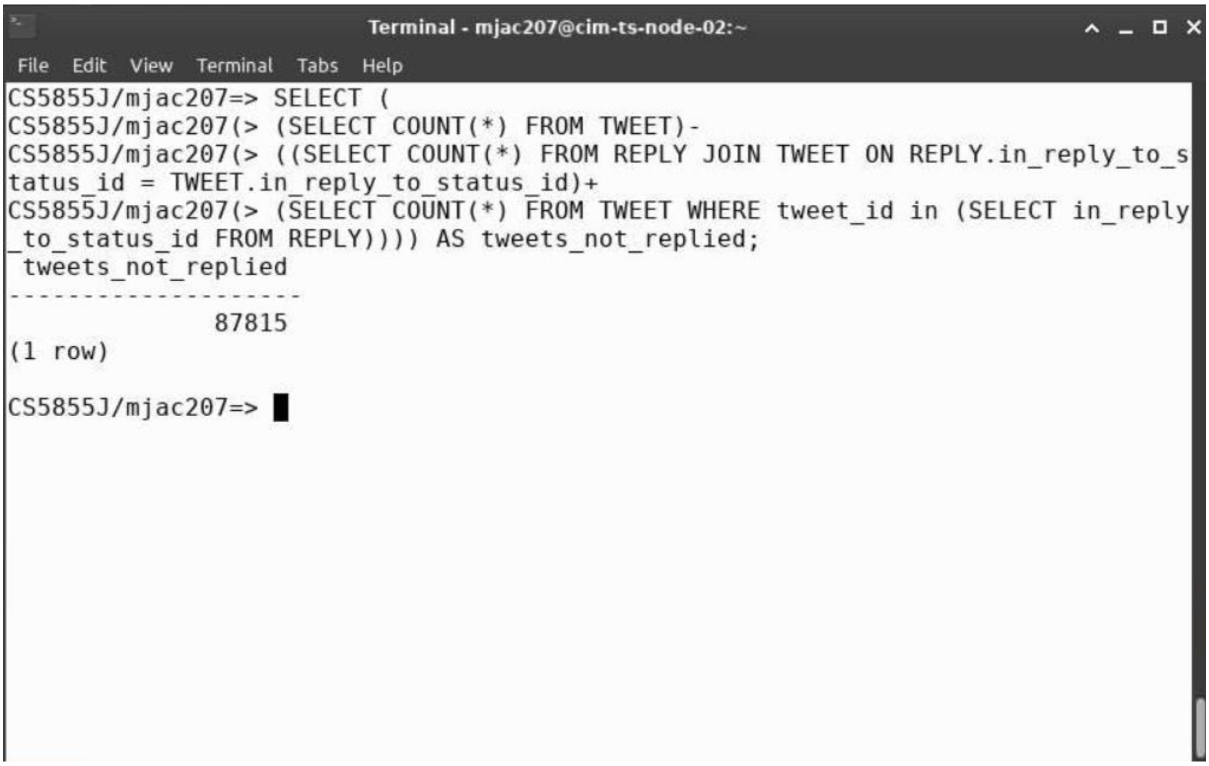
2) Computing the top ten most popular hashtags in the table.

```
CS5855J/mjac207(> UNION ALL
CS5855J/mjac207(> (SELECT hashtag5 AS popular_hashtag FROM HASHTAG WHERE hashtag
5 IS NOT NULL)
CS5855J/mjac207(> UNION ALL
CS5855J/mjac207(> (SELECT hashtag6 AS popular_hashtag FROM HASHTAG WHERE hashtag
6 IS NOT NULL))hashtag_count_popular
CS5855J/mjac207-> GROUP BY popular_hashtag
CS5855J/mjac207-> ORDER BY hashtag_count DESC
CS5855J/mjac207-> LIMIT 10;
     popular_hashtag       | hashtag_count
---------------------------+---------------
 ReasonsIFailAtBeingAGirl  |           467
 RED                       |           240
 oomf                      |           190
 HonestyHour               |           172
 TeamFollowBack            |           139
 EresGuapaSi               |           130
 10PeopleYouTrulyLove      |           126
 TweetLikeAGirl            |            98
 ImSingleBecause           |            97
 WeAllGotThatOneFriend     |            96
(10 rows)

CS5855J/mjac207=>
```

Replies:

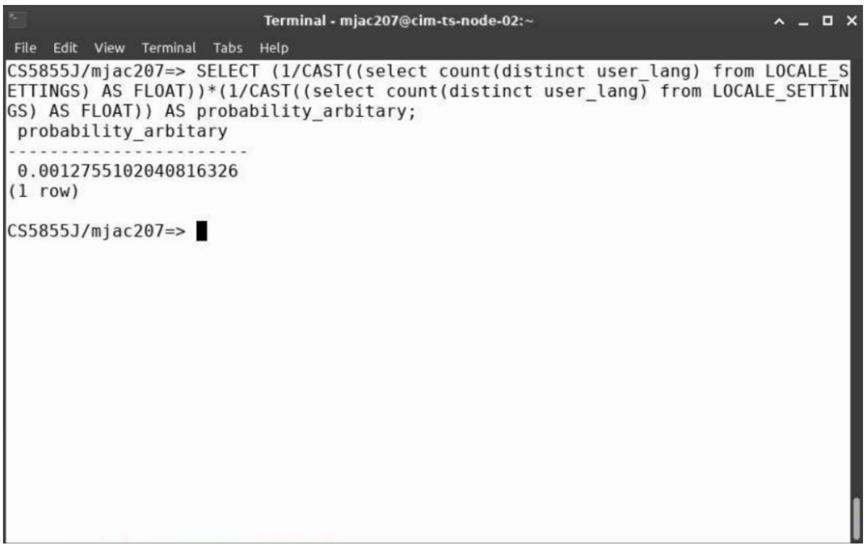
1) Computing the tweets that are neither replies nor replied to. (Assumption: the tweets that have a value in the column in_reply_to_status_id are the ones that have replied to a tweet. The value in the column in_reply_to_status_id holds the value of the id of the original tweet. We add these two counts and subtract it from the total number of tweets in the table. This will result in a set of tweets that are neither replies nor replied to.



```
Terminal - mjac207@cim-ts-node-02:~
File Edit View Terminal Tabs Help
CS5855J/mjac207=> SELECT (
CS5855J/mjac207(> (SELECT COUNT(*) FROM TWEET)-
CS5855J/mjac207(> ((SELECT COUNT(*) FROM REPLY JOIN TWEET ON REPLY.in_reply_to_s
tatus_id = TWEET.in_reply_to_status_id)+
CS5855J/mjac207(> (SELECT COUNT(*) FROM TWEET WHERE tweet_id in (SELECT in_reply
_to_status_id FROM REPLY)))) AS tweets_not_replied;
 tweets_not_replied
--------------------
              87815
(1 row)

CS5855J/mjac207=>
```

2) Computing the probability of two arbitrary users having the same language setting. (Assumption: Since this is a completely random probability of two users having the same language setting. There are a total of 28 distinct languages. The probability of two users having the same language setting is 1/28 * 1/28.



```
Terminal - mjac207@cim-ts-node-02:~
File Edit View Terminal Tabs Help
CS5855J/mjac207=> SELECT (1/CAST((select count(distinct user_lang) from LOCALE_S
ETTINGS) AS FLOAT))*(1/CAST((select count(distinct user_lang) from LOCALE_SETTIN
GS) AS FLOAT)) AS probability_arbitary;
 probability_arbitary
----------------------
 0.0012755102040816326
(1 row)

CS5855J/mjac207=>
```