DBMS Project Report

PES University

Database Management Systems

UE18CS252

Submitted By

|  |  |  |
| --- | --- | --- |
| **SRN :**  **PES1201801145** | **Name** :  GAURAV PANDEY | **Evaluation date and time:** |

**Functional Dependencies 2**

**Identifying Keys based on FDs 2**

[**Normalization**](#_wc0v2611q54c) **& testing for lossless join property 2+2**

[**DDL**](#_pn441yprtk2f)**: Table creation with all constraints 2+2**

[**Triggers**](#_a9bswny9ffj) **2**

[**SQL Queries**](#_exvna2br67rz) **2**

**Viva / modifications (Unit III/ IV concepts) 2+2**

**INSTAGRAM DATABASE**

|  |
| --- |
| **Problem statement:**  A social media website database needs to store information about users (identified by user\_id with username, password and created\_at as attribute), photos(identified by id  with image\_url, user\_id, created\_at as attribute), comments(identified by id with comment\_text, photo\_id, user\_id, created\_at as attributes), likes(identified by user\_id, photo\_id with created\_at as attribute), follows(identified by follower\_id, followee\_id and created\_at as attribute), tags(identified by id with tag\_name and created\_at as attributes, photo\_id).  Users may/may not have photos, likes, comments, photo tags. A user must be uniquely identified by user\_id. We are interested to store the time of creation in each table for legal reasons. A user must not be allowed to follow himself/herself. A user must not be able to like a photo more than once. A user must be allowed to pick from only a standard set of photo tags defined in the tags table. Historical information about who followed whom should be stored even after the person unfollows the other. |

# **Database Schema: (show all the tables and the constraints)**

**Users Table**

Uses an auto incremented integer(id) as primary key so that its faster to search for individual usernames instead of matching the usernames. Username which is unique and of the type varchar(255). Every user must have username and password therefore they are NOT NULL.

As per the problem statement it also has created\_at attribute which store the time at the point of insertion.

**Photos Table**

Uses an auto incremented integer(id) as primary key. All photos have image\_url attribute which store the url of the location of the image, user\_id which stores the id of the user which uploaded the photo and created\_at attribute which store the time at the point of insertion.

The user\_id attribute has a foreign key reference to id attribute in users table.

**Comments Table**

Uses an auto incremented integer(id) as primary key. Every comment should have a comment\_text therefore it is marked as NOT NULL. Also every commented should have the user\_id of the user who wrote the comment and photo\_id of the photo on which the comment was written. There is a and created\_at attribute which store the time at the point of insertion. Photo\_id and user\_id have foreign key references to photos(id) and users(id) respectively.

**LikesTable**Uses user\_id and photo\_id which are both integers as the primary key. Every data entry should have user\_id and photo\_id therefore they are set to NOT NULL. There is a and created\_at attribute which store the time at the point of insertion. Photo\_id and user\_id have foreign key references to photos(id) and users(id) respectively.

**Follows Table**

Uses follower\_id and followee\_id which are both integers as the primary key. Every data entry should have follower\_id and followee\_id therefore they are set to NOT NULL. There is a and created\_at attribute which store the time at the point of insertion. Data entry is only allowed if the follower\_id and followee\_id exist in the users table as id therefore there is a foreign key reference from both of them to users(id).

**Unfollows Table**

The attributes are the same as the follows table. As per the problem statement this table is used to store the historical data of who followed whom when the users unfollows someone.

Therefore, a trigger is used for this which initiates when a data entry has been deleted from the follows table which adds that data to this table.

**Tags Table**

Stores the list of valid tags which can be used by the user. Uses an auto incremented integer(id) as primary key for faster retrieval. All tag\_name attributes are UNIQUE and of the type varchar(255). There is a and created\_at attribute which store the time at the point of insertion.

**Photo\_tags Table**

Uses photo\_d and tag\_id which are integers as the primary key. As every tag used is for a photo both photo\_id and tag\_id are NOT NULL. As users can add tags to only existing photos there is a foreign key reference from photo\_id to photos(id) table. As every tag mentioned should be an valid tag there is a foreign key refernce from tag\_id to tags(id) table.

# **Functional Dependencies: (List based on your application constraints)**

**Users Table**

**id** -> username, password, created\_at

**Photos Table**

**id** -> image\_url, user\_id, created\_at

user\_id -> users(id)

**Comments Table**

**id** -> comment\_text, photo\_id, user\_id, created\_at

photo\_id -> photos(id)

user\_id -> users(id)

**Likes Table**

user\_id, photo\_id -> created\_at

user\_id -> users(id)

photo\_id -> photos(id)

**Tags Table**

**id** -> tag\_name, created\_at

# **Candidate keys: (Justify how did you get these as keys)**

In the likes table user can have multiple likes on different photos, therefore it is the combination of userid and photoid which makes it a key. **Also as user\_id and photo\_id are the candidate key a user can never like a photo more than once which is in accordance to the problem statement.**(follows, unfollows,photo\_tags table is similar to this).

Most of the tables(users, photos, comments, tags) have an auto incremented integer(NOT NULL) id which is used as candidate key.The key is made as integer for faster matching of things like tag name, username etc. Eg: Instead of selecting username as a key id is selected due to performance reasons.

# **Normalization and testing for lossless join property:**

**a)**

USERS(id, username, password, user\_created\_at, {PHOTOS(photo\_id, image\_url, photo\_created\_at),{COMMENT(comment\_id, comment\_text, comment\_created\_at)}})

**1NF** normalisation of the above gives 3 tables

USERS(id, username, password, created\_at)

PHOTOS(id, user\_id, image\_url, created\_at)

COMMENTS(id, user\_id, photo\_id, comment\_text, created\_at)

**b)**

TAGS(id, image\_url, caption, tag\_name)

The problem with the above one is that all the tags for a particular photo will be stored in a array inside the tag\_name attribute.

Therefore, after **1NF** normalisation this gets separated to 2 tables (one for photos and one for photo\_tags which has foreign key reference to the tags table which stores all the valid tags at a single place)

PHOTOS(id, user\_id, image\_url, created\_at)

TAGS(id, tag\_name, created\_at)

PHOTO\_TAGS(photo\_id, tag\_id)

**2NF normalisation explanation**

Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency X → Y is a full functional dependency if removal of any attribute A from X means that the dependency does not hold anymore.

A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.

Eg:

USERS(**id**, username, password, user\_created\_at, **photo\_id**, image\_url, photo\_created\_at, **comment\_id**, comment\_text, comment\_created\_at)}})

The above table is not 2NF normalised as **username, password , user\_created\_at** is functionally determined by only **id.**

Similarly, image\_url, photo\_created\_at are functionally determined by photo\_id and comment\_text, comment\_created\_at are functionally determined by comment\_id

After 2NF normalisation

Table union

USERS(id, username, password, created\_at)

PHOTOS(id, user\_id, image\_url, created\_at)

COMMENTS(id, user\_id, photo\_id, comment\_text, created\_at)

**The tables used are 3NF normalised.**

**Testing for lossless join property**

Lossless join property ensures that no spurious tuples are generated when a NATURAL

JOIN operation is applied to the relations resulting from the decomposition.

The idea behind lossless join property is that on decomposition of tables no functional dependencies should be lost in the process.

Eg:

In table ABC

A,B -> C

The above table cannot be normalised to A,B and B,C as the functional dependencies get lost in this process. Therefore, the above normalisation is not lossless.

**The tables used have been tested for this condition and none of the decompositions made cause the loss of functional dependencies thus they follow lossless join property.**

Example which **is** lossless

When this

USERS(id, username, password, user\_created\_at, {PHOTOS(photo\_id, image\_url, photo\_created\_at),{COMMENT(comment\_id, comment\_text, comment\_created\_at)}})

gets normalised to

USERS(id, username, password, created\_at)

PHOTOS(id, user\_id, image\_url, created\_at)

COMMENTS(id, user\_id, photo\_id, comment\_text, created\_at)

None of the functional dependencies mentioned above are lost in the process. Therefore the above normalisation is lossless.

Example which **is** **not** lossless.

**Likes Table**

user\_id, photo\_id -> created\_at

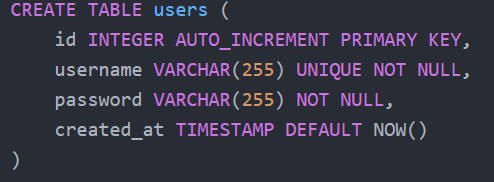
user\_id -> users(id)

photo\_id -> photos(id)

The above table cannot be normalised to user\_id, photo\_id and photo\_id, created\_id as the above mentioned functional dependency is lost. Therefore if such a normalisation is not then it wont be lossless.

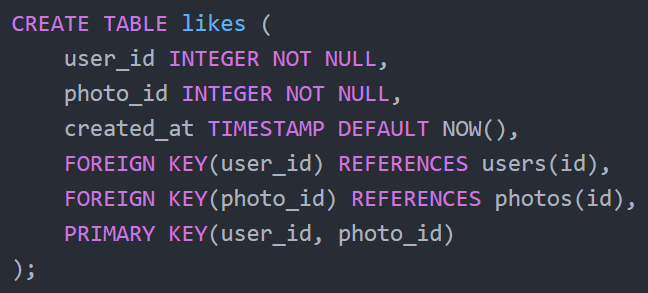
**DDL:**

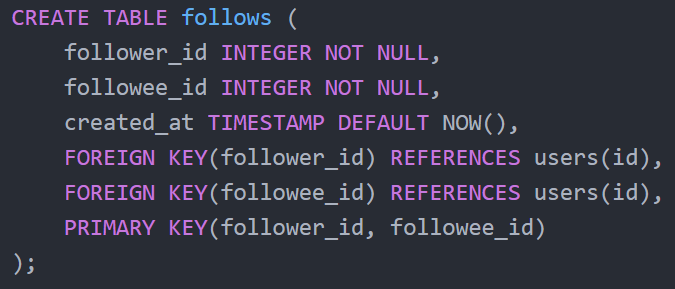
Create table scripts here. Ensure integrity constraints are defined. Add sample insert statements as well, that you would be using for demo.

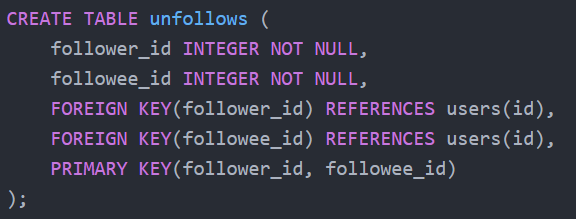


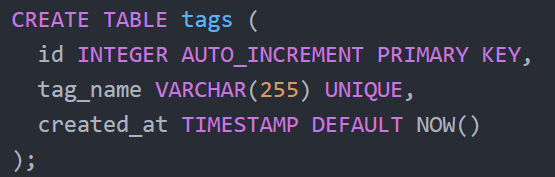
# 

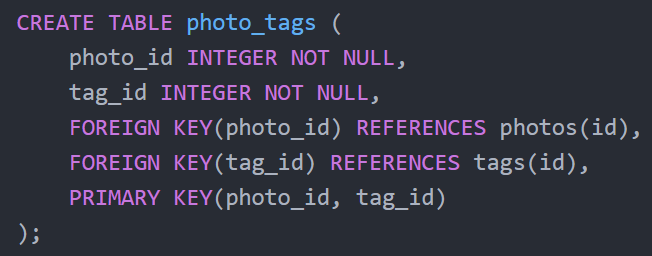
# 









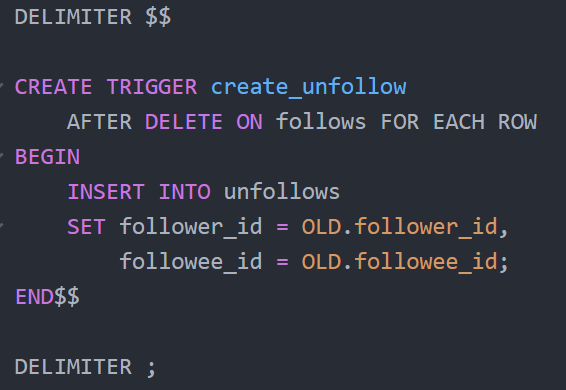


# **Triggers:**

1. Identify a constraint to implement as a trigger and write the English statement for that.
2. Write the trigger creation statement along with any stored procedures/functions involved.

**Ans**

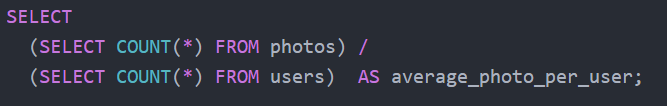
As per the problem statement the historical data of who followed who needs to be saved for analytical reasons. Therefore, when a user unfollows someone that entry should be deleted from the follows table and stored in the unfollows table.



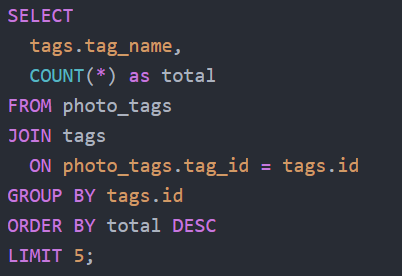
# **SQL Queries:**

<Write a few english sentences and SQL queries for them. Ensure at least 2 correlated-nested Advanced and 2 aggregate queries. >

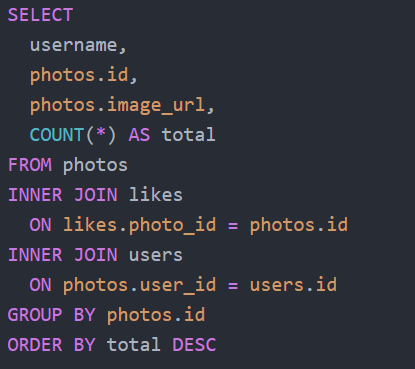
1) Calculate the average number of photos per user **(aggregate query)**



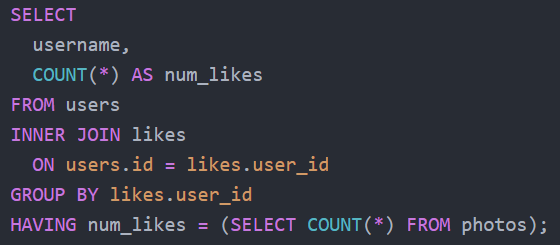
2) What are the top 5 most commonly used hashtags **(aggregate query)**



3) Which username and the photo which has the greatest number of likes along with the number of likes**. (nested query)**



4) Which users have liked every single photo**. (nested query)**



5) Which users have never posted a photo

