Abstract

The Image Sharing Platform Database is a database that has been modelled from scratch. This database is set to emulate real-life databases of photo sharing online platforms; and how they store data in a concise manner.
The data model is set to include the fundamental information required regarding the posts that users put up on said Image Sharing Platforms, and the basic features that these users may be provided with. A few extremely popular services that are image sharing platforms at their core, include Pinterest, Instagram, 9GAG, and the list goes on.
Ensuring that the database design follows a normalized structure is critical to ensuring the security and integrity of the data being stored. Through the database model designed, this goal of providing data security and ensuring data integrity thorough normalization is also hoped to be achieved. The database has been broken down into modular, and compact relations and tables, providing the necessary and fundamental features expected by users.
Since there is no denying that there is room for improvement, as is the case with nearly all software developed, scalability has been kept in mind while formulating the database design, and has served as a motivating factor to make the make the design as unambiguous, uncomplicated, predictable, and controllable as possible.

Table of Contents

Introduction

The Image Sharing Platform Database, as it sounds, is a database that aims to manage the data for an online social media platform that could be used for image sharing. Although rudimentary in comparison with the database models adopted by the wide array of commercial image sharing platforms available, the Image Sharing Platform Database aims to provide a system that can be used readily and securely, and easily scaled.

The fundamental principles of this image sharing platform revolves around providing robust database support for users who wish to share images, with tags of their choice, from the variety locations they find themselves in.

A good part of the experience also involves indulging in and appreciating what the community, i.e., other users have to offer. Users can follow, other users, and unfollow as per their wishes, as well.

Adding tags and swanky captions to the images uploaded as posts can help find more like-minded individuals, and this only enriches the experience provided by the image sharing platform even more.

One of the greatest common issues with such platforms however, are bots. Bots can ruin the user experience, cause unnecessary wastage of database storage, and also can be critically harmful to the platform's survival, since they may be used to launch attacks like "51% attack". However, through effective monitoring of the database, the patterns of the bots can be detected as well, and dealt with. Queries have been implemented that work in concordance with the flexible database design to detect suspicious activity (such as user accounts liking all the photos present on the platform). Queries have been created for a variety of questions that the administrators of the platform might have, pertaining to user outreach and usage statistics as well. These queries help provide meaningful insight into data, and practices adopted by the target audience. Monitoring outreach, user reactions and interactions, the users' perspectives, strategies to adopt for better user engagement, and so on can be determined by running queries (along the lines of few of the queries implemented), and analysing them thoroughly.

Since it is a database, however, outlooks from a practical viewpoint have been presented as well, in order to ensure optimization and quicker responses. The database is essentially a support system in the backend to an application, and is generally regarded as the slowest component of the entire infrastructure. Thus, the database and database model cannot be expected to perform tedious tasks such as input validation for every input, and other complex operations, since it can prove to be quite cumbersome and also severely affect latency and the user experience. Thus, this database model provides provisions for performing input validations, but not on an extravagant scale, but those that involve operations that are not too exhaustive and enforces constraints, to a good extent with this perspective also in mind. The extensive input validation mechanisms are left to the imagination of the developers using the database for their platform.

The database design has been created almost entirely from the standpoint of serving as a backend-database; providing essential functionalities, while managing the trade-off between functionality, and speed and scalability, such that the users can have a seamless, and satisfying experience; and the database can be scaled for more features.

Mini World Description

The entities that can be used to accurately describe the mini world, while ensuring that fundamental requirements are taken care of, have been mentioned below.

1. User:

Users is representative of the details regarding the users who have an account.

2. Follower and Unfollower:

Follower characterizes all the followers that a respective user has. It has been ensured that every user will be reflected as the follower of another user only once, and vice versa.

3. Unfollower:

Unfollower characterizes all the instances in which a user unfollowed another user.

4. Photo:

Photo is characterizes the data that corresponds to the photos that would be uploaded by the users.

5. Like:

Like is representative of all the likes that users have given on respective photos that have been uploaded. It is ensured that a user cannot like a photo more than once. Likes can be used for determining if bots have employed, so users that show suspicious activity may be put under review, or removed at once.

6. Comment:

Comment is used to represent all the details of the content that users have commented on respective existing pictures. Comments can also be used for determining if bots have employed; so accounts that show suspicious activity may be put under review, or removed at once.

7. Tag:

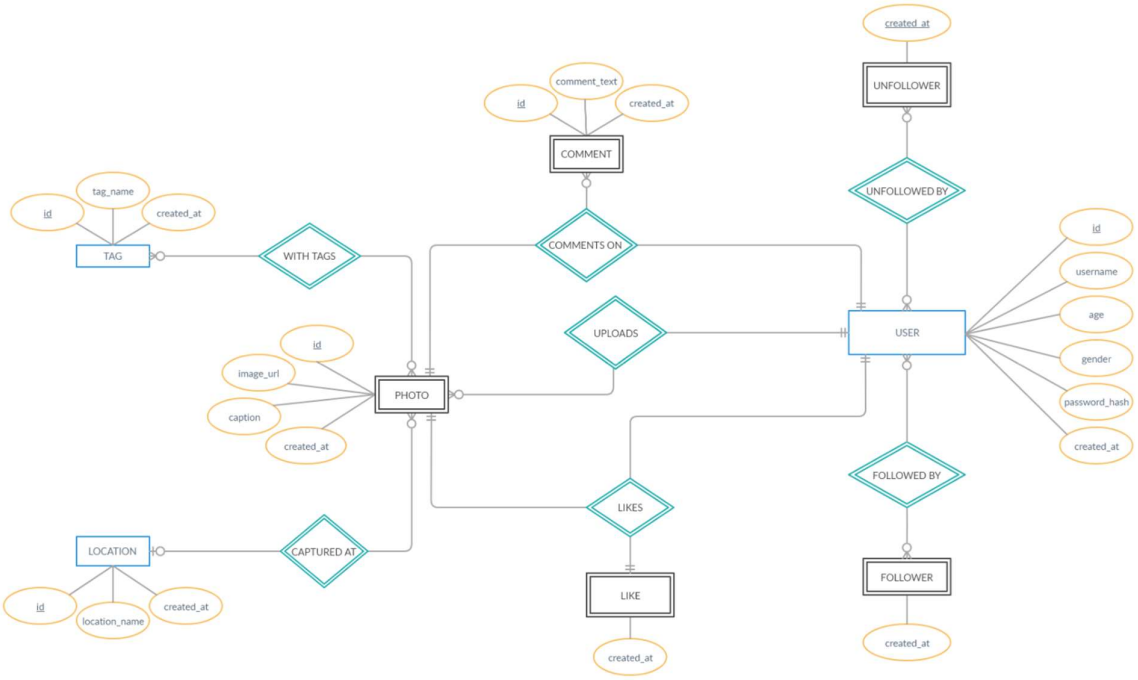
Tag is used to represent all the tags that have been used throughout the span of the user base. This can be used for analyzing outreach metrics, and for moderation of interaction for users with their followers.

8. Location:

Location is used to represent all the location from which photos have been posted by the users. A photo can be have only one location assigned to it.

The entities described in the mini world, and their respective relationships will be detailed upon in the ER Diagram and the Data Model.

ER Diagram



Description of Relationships

The ER Diagram has been constructed following Crow's foot notation. We will now be analyzing all the relationships, with reference to their respective entities.

1. UPLOADS:
    - Each USER can upload 0 or many PHOTOs.
    - Each PHOTO that has been stored, corresponds to 1 and exactly 1 USER.

2. COMMENTS ON:
    - Each COMMENT made by a USER on a PHOTO belongs to only that particular PHOTO.
    - Each USER who has made a COMMENT on a PHOTO can COMMENT many more times on the same photo.
    - Each COMMENT on every PHOTO, is made by a single, particular USER.

3. LIKES:
    - Each LIKE expressed by a USER on a PHOTO belongs to only that particular PHOTO.
    - Each USER who has LIKED a PHOTO can LIKE the same photo only once.
    - Each LIKE on every PHOTO, is made by a single, particular USER.

4. WITH TAGS:
    - Each PHOTO can have 0 or multiple TAGs corresponding to it.
    - Each TAG can referred to by 0 or more PHOTOs.

5. CAPTURED AT:
    - Each PHOTO can have 0 or 1 LOCATION where it was captured mentioned.
    - 0 or more PHOTOs can be captured in the same LOCATION.

6. FOLLOWED BY:
    - Each USER can be followed 0 or more FOLLOWERs.
    - Each FOLLOWER can follow 0 or more USERs.

7. UNFOLLOWED BY:
    - Each USER can be unfollowed 0 or more UNFOLLOWERs.
    - Each UNFOLLOWER can unfollow 0 or more USERs.

Entity Strength

Weak Entities:
Weak entities are entities that wouldn't exist without the presence of a related strong entity.
The weak entities in this model are as follows.

- PHOTO:

A PHOTO cannot exist if a USER does not exist.

- COMMENT:

A COMMENT cannot exist if a USER, or a PHOTO does not exist.

- LIKE:

A LIKE cannot exist if a USER, or a PHOTO does not exist.

- FOLLOWER

A FOLLOWER cannot exist if a USER does not exist.

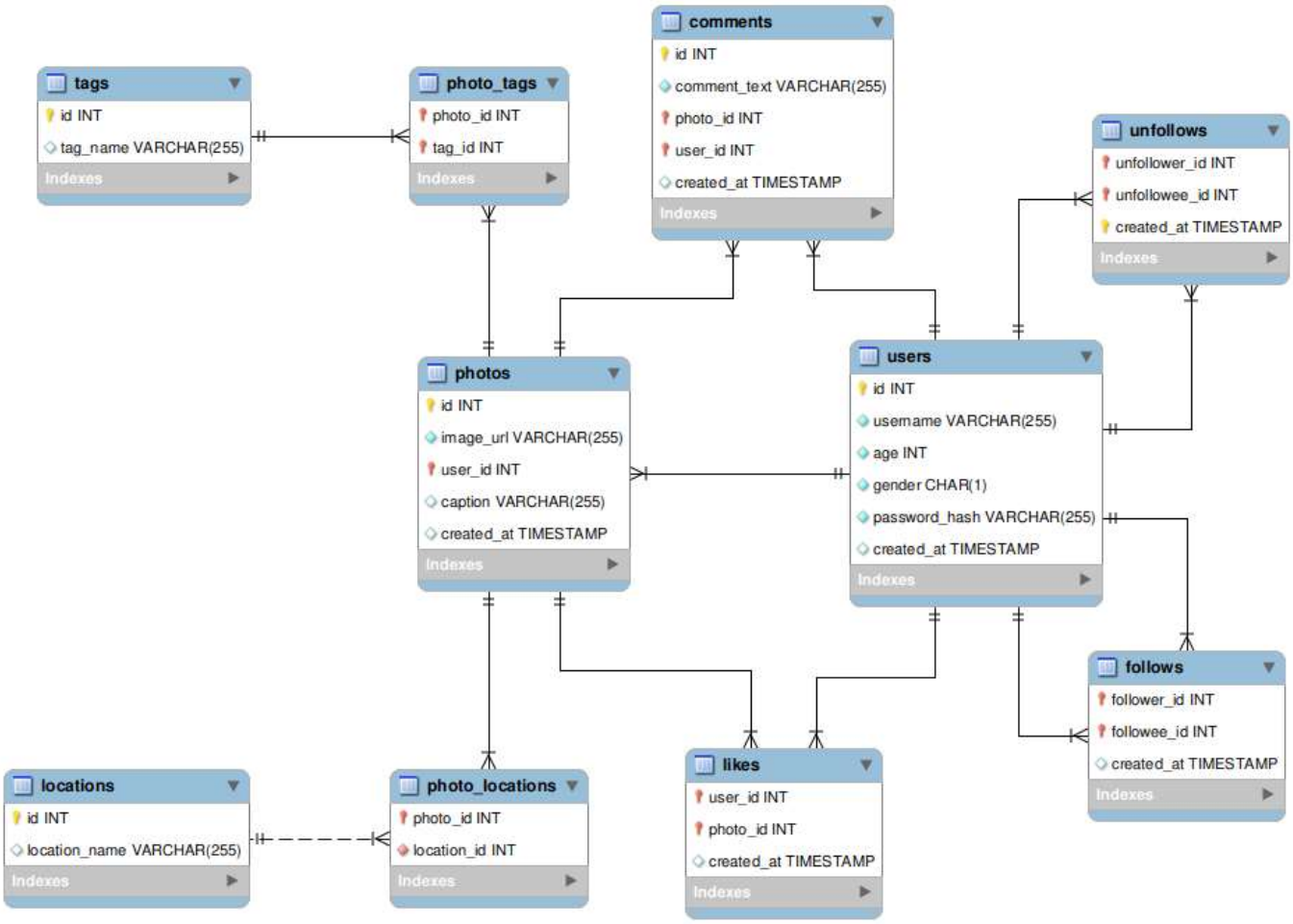
- UNFOLLOWER

An UNFOLLOWER cannot exist if a USER does not exist.

Strong Entities:
Strong entities are entities that will continue exist with or without the presence of any other entity.
The strong entities in this model are as follows.

- USER
- LOCATION
- TAGS

Data Model – Schema



Schema Description

The schema of the database designed can be depicted and elaborated upon as follows. The data types of all the attributes have been depicted alongside the respective attributes in the schema.

- The relations *users, locations, and tags* do not depend on values from any other relations.

- The 'users' relation has a primary key, id. It also has the attributes username, age, gender, password_hash, and created_at.

The attributes *username* and *password_hash* are stored in order to provide authentication services; as it is standard practice for storing the value of obtained from applying hash functions to passwords in the database. Conventionally input validation is

done on the client side itself, before attempting to insert values into the database. Thus *username* does not necessarily need to be constrained to be unique. This also helps normalize the data better, and also make it scalable to applications that may support/require non-unique usernames as well.
'gender' has constraints configured in order to ensure that only 'M','F', and 'O' (signifying Male, Female, and Other respectively) can be stored.
'age' has been rigged with constraints to ensure that only those above the age of 18 can have an account.
'created_at' stores the time when the new user entry was created in the *'users'* relation.

- The '*photos'* relation has a composite primary key, comprising of '*id'* and '*user_id'*, which is a foreign key from the table '*users'*.
  '*photos'* relation also has the attributes '*image_url', 'caption'* and *'created_at'*. These attributes describe the URL which can be used for accessing the respective photo, the caption that the user has chosen to provide with the photo, and the time when the entry was inserted into the relation, which is automatically generated. In case no *caption* has been provided, then the *caption* is set to be **NULL** by default.

- The *'comments'* relation has a composite primary key that is composed of and '*id*' , and '*photo_id'* and '*user_id'*; the latter two being foreign keys from '*photos'* and '*users*' relations respectively representing their respective '*id*'s. There also exists an attribute named '*comment_text'* which is used to store the comment made by the specified user on the respective photo. The '*created_at'* attribute records the timestamp of when the abovementioned comment was made.

- The *'likes'* relation has a composite primary key that is composed of '*user_id'* and '*photo_id'*; foreign keys from '*users'* and '*photos*' relations respectively representing their respective '*id*'s.

- The '*created_at'* attribute records the timestamp of when the respective user liked the photo being referred to. Each photo can be liked by a user only once.

- The *'locations'* relation has a primary key '*id'* and another attribute '*location_name*', which signifies the name of a unique location.

- The *'tags'* relation has a primary key '*id'* and another attribute '*tag_name*', which signifies the name of a unique tag.

- The *'photo_locations'* relation has a composite primary key that is composed of the '*photo_id'* and '*location_id'*; foreign keys from '*photos'* and '*location*s' relations respectively, representing their respective '*id'*s. Each photo can have only one location mentioned. This relation is introduced in order to ensure that there is normalization of data.

- The *'photo_tags'* relation has a composite primary key that is composed of the

'*photo_id'* and '*tag_id'*; foreign keys from '*photos'* and '*tags'* relations respectively, representing their respective '*id'*s. Each photo can have each tag mentioned only once. This relation is introduced in order to ensure that there is normalization of data.

- The *'follows'* relation is symbolic of the follower entity. It has a composite primary key that is composed of the '*follower_id'* and '*followee_id'*; both foreign keys from the '*users'* relation, representing the '*id'*s of the user who is following, and the user being followed, respectively. There also exists another attribute, '*created_at'* which is used to store the time at which a user followed the other user. Each user can follow another user only once.

- The *'unfollows'* relation is symbolic of the follower entity. It has a composite primary key that is composed of the '*created_at*' 'un*follower_id'* and 'un*followee_id'*; both foreign keys from the '*users'* relation, representing the '*id'*s of the user who is unfollowing, and the user being unfollowed, respectively. '*created_at'* is used to store the time at which a user followed the other user. Due to '*created_at'* being a part of the primary key, even if a user were to unfollow another user once again (unfollowing after re-following), then that instance would be recorded as well. The '*unfollows'* relation has been modelled similar to the '*follows'* relation because the data removed from the '*follows*' relation will be recorded in the '*unfollows*' relation, with the help of triggers, as we will soon come to discover.

- In order to provide data integrity, and reduce storing of irrelevant data, the database model employs usage of ON DELETE and ON UPDATE constraints, at all points where foreign keys have been employed. In the majority of cases, cascading is allowed for cases where a parent entry is deleted or updated.

Functional Dependencies and Normalization

Functional Dependencies

users

- `id → {username, age, gender, password_hash, created_at}`

tags

- `id → {tag_name}`
- `tag_name → {id}`

locations

- `id →  {location_name}`
- `location_name → {id}`

photos

- `{id, user_id} → {image_url, caption, created_at}`

comments

- `{id, user_id, photo_id} → {comment_text, created_at}`

First Normal Form (1NF)

The First Normal Form expects a scalable design of tables to facilitate easier retrieval of data.

- The ER-to-Relational Mapping Algorithm ensures that all multivalued attributes will be converted into a new set of independent tables. Thus, the domain of all the attributes, in all the relation will contain only atomic values.
- Each attribute, in each relation has been customized to have a meaningful name with respect to the data stored in it, and in an attempt to improve understandability of the relations. Every relation thus has attributes unique with respect to each other.
- It has been ensured, and enforced using integrity constraints, that values that belong to an attribute are all of the same data type.
- The order in which the data is stored in the tables is not of any significance either.

Thus, it can be perceived that all the relations are in First Normal Form (1NF).

Second Normal Form (2NF)

- It has already been established that all the conditions for first normal form are satisfied.
- Next, criteria required for the second normal form to be observed is that all the relations must be devoid of partial dependencies. Partial dependencies arise when an attribute in a relation is functionally dependent on part of a composite primary key, or candidate key.

It has been ensured that no partial dependencies arise in all the relations that effectuate composite primary keys. Thus, it can be observed that all the relations are in Second Normal Form (2NF).

Third Normal Form (2NF)

- It has already been established that all the conditions for the second normal form have been satisfied as well.
- 3NF can be said to be observed, only when it is ensured that all the relations are devoid of transitive dependencies. Transitive dependencies are observed, when an attribute in a relation depends upon a non-prime attribute and not on the prime attributes.

It has been made certain that such dependencies are not present. The relation have been made very compact, yet significant in order to ensure no transitive dependencies. Thus, it can be inferred that all the relations are in Third Normal Form (3NF).

Boyce-Codd Normal Form (BCNF)

- It has already been established that 3NF is expected to be satisfied.
- All the determinants that constitute the functional dependencies for the relations with more than 2 attributes in the schema are prime attributes. Thus, no non-prime attribute can derive the prime attribute, unless, the total number of attributes in that relation equals 2.

Thus, Boyce-Codd Normal Form (BCNF) is expected to be satisfied for all the relations.

Fourth Normal Form (4NF)

- It has already been established that BCNF form is expected to be satisfied.
- For the Fourth Normal Form to be observed, the relations must be devoid of multi-value dependencies. Multi-value dependencies are said to arise in relations that have more than three attributes, with at least 2 independent attributes. The compactness of this model has ensured that the majority of the relations do not satisfy these preliminaries. For the relations that can be classified under these conditions, however, it is understandable that multi-valued dependencies will not arise in any relation, i.e., for one value of an attribute, the self-independent attributes will not exhibit multiple values.

Thus, all the relations are said to be in the Fourth Normal Form (4NF).

Fifth Normal Form (5NF)

- It has already been established how 4NF seems to be satisfied.
- All the relations are structured in a manner, and have been decomposed to the maximum extent possible, thus making the relations as normalized as possible, such that there is no loss of data, or scope for inference of incoherent/incorrect data from performing any operations on the relations.

Thus, it can be inferred that the Fifth Normal Form (5NF) is preserved.

Potential Violations of the Normal Forms

- If new attributes named '*ethnicity'* and *'population_distribution'* are introduced in the *'locations'* relation, then transitive dependencies might arise, since '*ethnicity'* depends upon '*population_distribution'* more than it does on '*locations*'.
- The relations *location* and *tags* have a primary key '*id'*, and a unique key each. If new attributes were introduced in either of these relations, then a non-prime attribute could possibly derive the values that correspond to a prime attribute. Thus, BCNF form would be violated. This is because a non-prime attribute (the attribute for with unique values) could derive the records that correspond to a prime attribute.
- If either the relations '*location'* or '*tag'* and the corresponding '*photo_location'* and '*photo_tag'* were removed, and either their corresponding attributes was added as attributes in '*photos*', then both 4NF would be violated .
- If '*id*' wasn't a primary key in '*comments'*, then multi-valued dependency would be observed.
- If the relations '*photo_locations*' or '*photo_locations*' didn't exist, and '*photo_id'* was directly being stored in '*locations*' or '*tags*' instead, then multi-valued dependency would arise.
- If the '*photos'* relation is decomposed incorrectly, lossy decomposition could possibly arise, especially since it is involved in a ternary relationship.

Lossless Decomposition

Lossless Decomposition can be observed to be present in this database model. In an effort to provide thorough normalization, and a compact structure, the relationship between '*photos',* and '*locations'* and *'tags'* respectively has been broken down to give rise to a new relation called, '*photo_locations*' and '*photo_tags*' respectively. This enforces normalization better. If join is performed with respect to these newly formed tables, and either of the parent tables, then the resultant relation would be the same as how a relation formed would look like, if it wasn't divided in such a manner. Further, since neither of these tables have any attributes distinct from the attributes present in '*locations*'/'*tags*' and *'photos'*, there is no extra incoherent data that gets added either, if a join is performed. Thus, lossless decomposition is said to be explicitly enforced, at two instances in this model.

DDL

```
--Create database ‘image_share’
DROP DATABASE IF EXISTS image_share;
CREATE DATABASE image_share;
USE image_share;


--Create table ‘users’
CREATE TABLE users (
    id INTEGER AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL,
    age INT NOT NULL CHECK (age>18),
    gender char(1) NOT NULL CHECK (gender in ('M','F','O')),
    password_hash varchar(255) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW()
);

--Create table ‘tags’
CREATE TABLE tags (
  id INTEGER AUTO_INCREMENT PRIMARY KEY,
  tag_name VARCHAR(255) UNIQUE,
  created_at TIMESTAMP DEFAULT NOW()
);

--Create table ‘locations’
CREATE TABLE locations (
  id INTEGER AUTO_INCREMENT PRIMARY KEY,
  location_name VARCHAR(255) UNIQUE,
  created_at TIMESTAMP DEFAULT NOW()
);

--Create table ‘photos’
CREATE TABLE photos (
    id INTEGER AUTO_INCREMENT,
    image_url VARCHAR(255) NOT NULL,
    user_id INTEGER NOT NULL,
    caption VARCHAR(255) DEFAULT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    FOREIGN KEY(user_id) REFERENCES users(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    PRIMARY KEY(id, user_id)
);

--Create table ‘comments’
CREATE TABLE comments (
    id INTEGER AUTO_INCREMENT PRIMARY KEY,
    comment_text VARCHAR(255) NOT NULL,
    photo_id INTEGER NOT NULL,
```

```
    user_id INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    FOREIGN KEY(photo_id) REFERENCES photos(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    FOREIGN KEY(user_id) REFERENCES users(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE
);



--Create table 'likes'
CREATE TABLE likes (
    user_id INTEGER NOT NULL,
    photo_id INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    FOREIGN KEY(user_id) REFERENCES users(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    FOREIGN KEY(photo_id) REFERENCES photos(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    PRIMARY KEY(user_id, photo_id)
);

--Create table 'follows'
CREATE TABLE follows (
    follower_id INTEGER NOT NULL,
    followee_id INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    FOREIGN KEY(follower_id) REFERENCES users(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    FOREIGN KEY(followee_id) REFERENCES users(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    PRIMARY KEY(follower_id, followee_id)
);

--Create table 'unfollows'
CREATE TABLE unfollows (
    unfollower_id INTEGER NOT NULL,
    unfollowee_id INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    FOREIGN KEY(unfollower_id) REFERENCES users(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    FOREIGN KEY(unfollowee_id) REFERENCES users(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
```

```
    PRIMARY KEY(unfollower_id, unfollowee_id, created_at)
);

--Create table 'photo_tags'
CREATE TABLE photo_tags (
    photo_id INTEGER NOT NULL,
    tag_id INTEGER NOT NULL,
    FOREIGN KEY(photo_id) REFERENCES photos(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    FOREIGN KEY(tag_id) REFERENCES tags(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    PRIMARY KEY(photo_id, tag_id)
);

--Create table 'photo_locations'
CREATE TABLE photo_locations (
    photo_id INTEGER NOT NULL PRIMARY KEY,
    location_id INTEGER NOT NULL,
    FOREIGN KEY(photo_id) REFERENCES photos(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    FOREIGN KEY(location_id) REFERENCES locations(id)
        ON UPDATE CASCADE
        ON DELETE CASCADE
);
```

Queries

The queries created are of veritable significance, and can be used for analytics of the image sharing platform. These analytics will further help boost the user experience provided.

1. **Determine the 5 oldest users present on the platform.**

```
-- Requirement: To determine the 5 oldest users present.

SELECT *
FROM users
ORDER BY created_at
LIMIT 5;
```

```
mysql> SELECT *
    -> FROM users
    -> ORDER BY created_at
    -> LIMIT 5;
+----+------------------+-----+--------+---------------------+---------------------+
| id | username         | age | gender | password_hash       | created_at          |
+----+------------------+-----+--------+---------------------+---------------------+
| 80 | Darby_Herzog     |  43 | M      | ZqxU3lyZeWjM1dzgIocO | 2016-05-06 00:14:21 |
| 67 | Emilio_Bernier52 |  69 | F      | mgfCBiHuUUzzhZE4pNAQ | 2016-05-06 13:04:30 |
| 63 | Elenor88         |  25 | M      | kzrdcIF1JPa6pSRYG7FM | 2016-05-08 01:30:41 |
| 95 | Nicole71         |  32 | M      | M4VHJlrKtTsVMsiz4YjN | 2016-05-09 17:30:22 |
| 38 | Jordyn.Jacobson2 |  54 | F      | e8RBZH8eCCNrKRp0kNtM | 2016-05-14 07:56:26 |
+----+------------------+-----+--------+---------------------+---------------------+
5 rows in set (0.00 sec)
```

2. **Determine on which days of the week, the image sharing platform observes the most registrations.**

```
-- Requirement: On what days do the users register onto the
platform?

SELECT
    DAYNAME(created_at) AS
day,
    COUNT(*) AS total
FROM users
GROUP BY day
ORDER BY total DESC;
```

```
mysql> SELECT
    ->     DAYNAME(created_at) AS day,
    ->     COUNT(*) AS total
    -> FROM users
    -> GROUP BY day
    -> ORDER BY total DESC;
+-----------+-------+
| day       | total |
+-----------+-------+
| Thursday  |    16 |
| Sunday    |    16 |
| Friday    |    15 |
| Tuesday   |    14 |
| Monday    |    14 |
| Wednesday |    13 |
| Saturday  |    12 |
+-----------+-------+
7 rows in set (0.00 sec)
```

3. **Identify all the users who haven't ever posted a photo on the image sharing platform.**

```
-- Requirement: Identify all the users who have never posted
a photo.

SELECT
    users.id,
    username
FROM users
LEFT JOIN photos
    ON users.id = photos.user_id
WHERE photos.id IS NULL;
```

```
mysql> SELECT
    ->     users.id,
    ->     username
    -> FROM users
    -> LEFT JOIN photos
    ->     ON users.id = photos.user_id
    -> WHERE photos.id IS NULL;
+----+--------------------+
| id | username           |
+----+--------------------+
|  5 | Aniya_Hackett      |
|  7 | Kasandra_Homenick  |
| 14 | Jaclyn81           |
| 21 | Rocio33            |
| 24 | Maxwell.Halvorson  |
| 25 | Tierra.Trantow     |
| 34 | Pearl7             |
| 36 | Ollie_Ledner37     |
| 41 | Mckenna17          |
| 45 | David.Osinski47    |
| 49 | Morgan.Kassulke    |
| 53 | Linnea59           |
| 54 | Duane60            |
| 57 | Julien_Schmidt     |
| 66 | Mike.Auer39        |
| 68 | Franco_Keebler64   |
| 71 | Nia_Haag           |
| 74 | Hulda.Macejkovic   |
| 75 | Leslie67           |
| 76 | Janelle.Nikolaus81 |
| 80 | Darby_Herzog       |
| 81 | Esther.Zulauf61    |
| 83 | Bartholome.Bernhard |
| 89 | Jessyca_West       |
| 90 | Esmeralda.Mraz57   |
| 91 | Bethany20          |
+----+--------------------+
26 rows in set (0.00 sec)
```

4. **Determine user and the image that seems to be the most popular on the image sharing platform**

```
-- Requirement: Determine the user and the image that has got
the most likes.
SELECT
    username, photos.id as "photo id",
    photos.image_url,
    COUNT(*) AS likes
FROM photos
INNER JOIN likes
    ON likes.photo_id = photos.id
INNER JOIN users
    ON photos.user_id = users.id
GROUP BY photos.id
ORDER BY likes DESC
LIMIT 1;
```

```
mysql> SELECT
    ->     username, photos.id as "photo id",
    ->     photos.image_url,
    ->     COUNT(*) AS likes
    -> FROM photos
    -> INNER JOIN likes
    ->     ON likes.photo_id = photos.id
    -> INNER JOIN users
    ->     ON photos.user_id = users.id
    -> GROUP BY photos.id
    -> ORDER BY likes DESC
    -> LIMIT 1;
+-----------------+----------+-----------------+-------+
| username        | photo id | image_url       | likes |
+-----------------+----------+-----------------+-------+
| Harrison.Wells12 |      145 | http://buddy.com |    48 |
+-----------------+----------+-----------------+-------+
1 row in set (0.00 sec)
```

5. **Identify dead accounts, i.e., accounts belonging to users who have never posted a photo, or liked/commented on another photo.**

```
-- Requirement: Identify all the users who have never posted a
photo, or liked/commented on another photo. Detecting and
deleting such accounts periodically could help eliminate the
entries corresponding to highly inactive users, thus saving up
storage space in the database.
SELECT users.id,
    username
FROM users
    LEFT JOIN photos
        ON users.id = photos.user_id
    LEFT JOIN comments
        ON users.id = comments.user_id
    LEFT JOIN likes
        ON users.id = likes.user_id
WHERE
        photos.id IS NULL
    AND
        comments.id IS NULL
    AND
        likes.created_at IS NULL;
```

```
mysql> SELECT users.id,
    ->     username
    -> FROM users
    ->     LEFT JOIN photos
    ->         ON users.id = photos.user_id
    ->     LEFT JOIN comments
    ->         ON users.id = comments.user_id
    ->     LEFT JOIN likes
    ->         ON users.id = likes.user_id
    -> WHERE
    ->         photos.id IS NULL
    ->     AND
    ->         comments.id IS NULL
    ->     AND
    ->         likes.created_at IS NULL;
+----+------------------+
| id | username         |
+----+------------------+
|  7 | Kasandra_Homenick |
| 25 | Tierra.Trantow    |
| 34 | Pearl7            |
| 45 | David.Osinski47   |
| 49 | Morgan.Kassulke   |
| 53 | Linnea59          |
| 74 | Hulda.Macejkovic  |
| 80 | Darby_Herzog      |
| 89 | Jessyca_West      |
| 90 | Esmeralda.Mraz57  |
+----+------------------+
10 rows in set (0.00 sec)
```

6. **Determine the average posts per user**

```
-- Requirement: Determine the average posts per user using
the services.
SELECT
      (SELECT COUNT(*) from photos)/(SELECT COUNT(*) from
users) AS avg;
```

```
mysql> SELECT
    ->     (SELECT COUNT(*) from photos)/(SELECT COUNT(*) from users) AS avg;
+--------+
| avg    |
+--------+
| 2.5700 |
+--------+
1 row in set (0.02 sec)
```

7. **Find out all the total number of photos posted by every gender of the userbase.**

```
-- Requirement: Return the total number of
photos a gender has posted.
SELECT
    users.gender, COUNT(*) as
total_photos_posted
FROM users
INNER JOIN photos
    ON photos.user_id=users.id
GROUP BY users.gender;
```

```
mysql> source ./queries/9.sql
+--------+---------------------+
| gender | total_photos_posted |
+--------+---------------------+
| M      |                 130 |
| F      |                 116 |
| O      |                  11 |
+--------+---------------------+
3 rows in set (0.00 sec)
```

8. **Find the 5 tags that photos are most popularly used.**

```
-- Requirement: List out the 5 most frequently
used tags.
SELECT
    tags.tag_name,
    COUNT(*) AS total
from photo_tags
JOIN tags
    ON photo_tags.tag_id = tags.id
GROUP BY tags.id
ORDER BY total DESC
LIMIT 5;
```

```
mysql> SELECT
    ->     tags.tag_name,
    ->     COUNT(*) AS total
    -> from photo_tags
    -> JOIN tags
    ->     ON photo_tags.tag_id = tags.id
    -> GROUP BY tags.id
    -> ORDER BY total DESC
    -> LIMIT 5;
+----------+-------+
| tag_name | total |
+----------+-------+
| smile    |    59 |
| beach    |    42 |
| party    |    39 |
| fun      |    37 |
| food     |    25 |
+----------+-------+
5 rows in set (0.01 sec)
```

9. **Find out all the users who have liked every single post on the image sharing platform.**

```
-- Requirement: Find out all the users that
have liked every single photo.
SELECT
    username,
    user_id,
    COUNT(*) AS total
FROM users
   INNER JOIN likes
    ON likes.user_id = users.id
GROUP BY likes.user_id
HAVING total = (SELECT COUNT(*) FROM photos);
```

```
mysql> SELECT
    ->     username,
    ->     user_id,
    ->     COUNT(*) AS total
    -> FROM users
    -> INNER JOIN likes
    ->     ON likes.user_id = users.id
    -> GROUP BY likes.user_id
    -> HAVING total = (SELECT COUNT(*) FROM photos);
+------------------+---------+-------+
| username         | user_id | total |
+------------------+---------+-------+
| Aniya_Hackett    |       5 |   257 |
| Jaclyn81         |      14 |   257 |
| Rocio33          |      21 |   257 |
| Maxwell.Halvorson|      24 |   257 |
| Ollie_Ledner37   |      36 |   257 |
| Mckenna17        |      41 |   257 |
| Duane60          |      54 |   257 |
| Julien_Schmidt   |      57 |   257 |
| Mike.Auer39      |      66 |   257 |
| Nia_Haag         |      71 |   257 |
| Leslie67         |      75 |   257 |
| Janelle.Nikolaus81|     76 |   257 |
| Bethany20        |      91 |   257 |
+------------------+---------+-------+
13 rows in set (0.02 sec)
```

10. **Display the details of all users who are in their twenties, i.e., between ages 20-29, and constitute the youth of the userbase.**

```
-- Requirement: Obtain the details of all
the users who are in their twenties.

    SELECT
        username,
        id,
        age,
        gender,
    FROM users
        WHERE (age<30 AND age>19);
```

```
mysql> source ./queries/10.sql
+----------------------+----+-----+--------+
| username             | id | age | gender |
+----------------------+----+-----+--------+
| Kenton_Kirlin        |  1 |  29 | M      |
| Arely_Bogan63        |  4 |  24 | F      |
| Aniya_Hackett        |  5 |  21 | F      |
| Jaclyn81             | 14 |  25 | M      |
| Norbert_Carroll35    | 17 |  22 | M      |
| Kenneth64            | 22 |  22 | M      |
| Irwin.Larson         | 32 |  27 | M      |
| Aurelie71            | 58 |  24 | F      |
| Ressie_Stanton46     | 62 |  21 | F      |
| Elenor88             | 63 |  25 | M      |
| Mike.Auer39          | 66 |  25 | F      |
| Erick5               | 70 |  25 | M      |
| Jaylan.Lakin         | 73 |  25 | F      |
| Leslie67             | 75 |  21 | F      |
| Jessyca_West         | 89 |  27 | M      |
| Frederik_Rice        | 92 |  25 | M      |
| Keenan.Schamberger60 | 96 |  26 | M      |
+----------------------+----+-----+--------+
17 rows in set (0.00 sec)
```

Triggers

1. prevent_self_follows

Trigger:

```
DELIMITER $$

CREATE TRIGGER prevent_self_follows
        BEFORE INSERT ON follows for EACH ROW
    BEGIN
                        IF NEW.follower_id = NEW.followee_id
             THEN
                            SIGNAL SQLSTATE '45000'
                 SET MESSAGE_TEXT = 'You cannot follow yourself.';
                        END IF;
    END;
$$
DELIMITER ;
```

Description:

The prevent_self_follows trigger is used for ensuring that users are not allowed to follow themselves. Since following oneself isn't a logically sound concept, this trigger has been implemented. When insertion of a new record in *follows* is attempted, it is first checked if the *follower_id* is the same as the *followee_id*. If they are found to be the same, then the insertion is not permitted, and a message stating 'You cannot follow yourself.' is displayed.

Execution:

Before Activation of Trigger	After Activation of Trigger

```
mysql> INSERT INTO follows(follower_id, followee_id) VALUES (1,1);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM follows WHERE follows.follower_id = follows.followee_id;
+-------------+-------------+---------------------+
| follower_id | followee_id | created_at          |
+-------------+-------------+---------------------+
|           1 |           1 | 2020-06-02 00:29:48 |
+-------------+-------------+---------------------+
1 row in set (0.00 sec)

mysql> DELETE FROM follows WHERE follows.follower_id = 1 AND follows.followee_id = 1;
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM follows WHERE follows.follower_id = follows.followee_id;
Empty set (0.00 sec)
```

```
mysql> source ./triggers/prevent_self_follows.sql
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM follows WHERE follows.follower_id = follows.followee_id;
Empty set (0.00 sec)

mysql> INSERT INTO follows(follower_id, followee_id) VALUES (1,1);
ERROR 1644 (45000): You cannot follow yourself.
mysql> SELECT * FROM follows WHERE follows.follower_id = follows.followee_id;
Empty set (0.00 sec)

mysql>
```

2. capture_unfollow

Trigger:

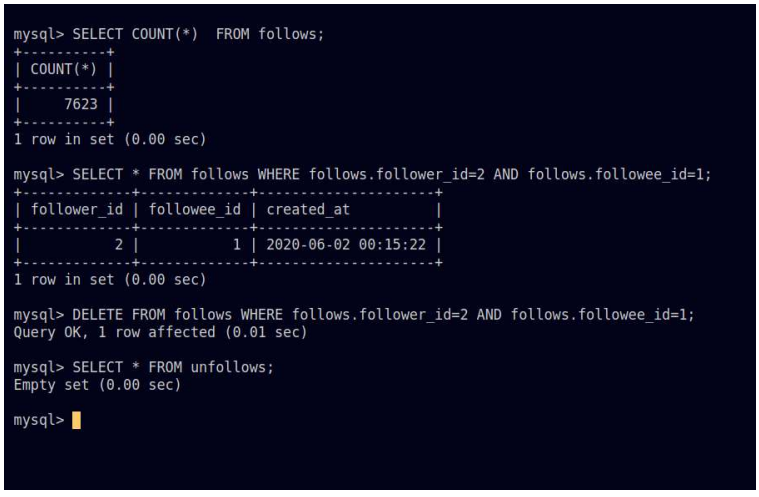
```
DELIMITER $$

CREATE TRIGGER capture_unfollow
AFTER DELETE ON follows for EACH ROW
BEGIN
INSERT INTO unfollows(unfollower_id, unfollowee_id)
VALUES (OLD.follower_id,OLD.followee_id);
END;
$$
DELIMITER ;
```

Description:

The capture_unfollow trigger is used for keeping track of users unfollowing other users. When a user unfollows another user, the corresponding tuple is removed from the *follows* relation. This record is stored into the *unfollows* relation. The *follower_id* and *followee_id* attributes that were deleted from the *follow* relation will be inserted into *unfollows* relation as *unfollower_id* and *unfollowee_id* respectively.
Having an option like this provides an immense advantage in measuring the impact of marketing and outreach schemes, and judging its reception from the target audience/general masses as well. This data if made available, can prove to be extremely insightful and help provide users new outlooks, through which they can engage their followers and adjust outreach better.

Execution:

Before Activation of Trigger	After Activation of Trigger



```
mysql> SELECT COUNT(*)  FROM follows;
+----------+
| COUNT(*) |
+----------+
|     7623 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT * FROM follows WHERE follows.follower_id=2 AND follows.followee_id=1;
+-------------+-------------+---------------------+
| follower_id | followee_id | created_at          |
+-------------+-------------+---------------------+
|           2 |           1 | 2020-06-02 00:15:22 |
+-------------+-------------+---------------------+
1 row in set (0.00 sec)

mysql> DELETE FROM follows WHERE follows.follower_id=2 AND follows.followee_id=1;
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM unfollows;
Empty set (0.00 sec)

mysql>
```

```
mysql> source ./triggers/capture_unfollow.sql
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT COUNT(*)  FROM follows;
+----------+
| COUNT(*) |
+----------+
|     7622 |
+----------+
1 row in set (0.00 sec)

mysql> DELETE FROM follows WHERE follows.follower_id=3 AND follows.followee_id=1;
Query OK, 1 row affected (0.01 sec)

mysql> SELECT COUNT(*)  FROM follows;
+----------+
| COUNT(*) |
+----------+
|     7621 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT * FROM unfollows;
+--------------+--------------+---------------------+
| unfollower_id | unfollowee_id | created_at          |
+--------------+--------------+---------------------+
|            3 |            1 | 2020-06-02 00:21:25 |
+--------------+--------------+---------------------+
1 row in set (0.00 sec)
```

Conclusion

The primary goals that was set for this project have been fulfilled. Through this project, a scalable and robust database model, that efficiently and effectively handles data has been built. The database has proven to be highly normalized, effective, and compact. The database models, and the relations as well as relationships have been explained as well, in accordance with convention. Users can easily create their accounts, follow or unfollow other users, plan and test and experience new ways to engage with the community formed by the userbase, upload photos with metadata such as tags, and locations, and also like and comment the content they find appealing to their heart's content, without having to worry about incorrect storage mechanisms for their data. It was initially mentioned, that this database was built with the perspective from serving as a backend-database for storing and processing information when required. It is hoped that the reader is convinced of this motivation, and would agree that the set goals have been achieved, along the lines of this very motivation.

Future Work and Scope

As previously mentioned, nearly all software developed has scope for more features to be made available. This holds true in the context of this database design as well, For starters, the database can provide more features such as storing entire images itself, either as base64 strings, or making use of blobs. The database could be scaled to support posting entire videos. Making provisions for organizing images into albums could also be a welcome feature.